# Lightweight Support for Fine-Grained Persistence on Stock Hardware

*Antony L. Hosking*

Technical Report 95-02
February 1995

Computer Science Department
University of Massachusetts at Amherst

# LIGHTWEIGHT SUPPORT FOR FINE-GRAINED PERSISTENCE ON STOCK HARDWARE

A Dissertation Presented

by

ANTONY LLOYD HOSKING

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

February 1995

Department of Computer Science

© Copyright by Antony Lloyd Hosking 1995

All Rights Reserved

*To Carla*

## ACKNOWLEDGEMENTS

One cannot survive the Ph.D. experience without the support and guidance of a great many people. To those individuals I neglect to mention here by name (and there are many) I still offer my deepest thanks.

Upon my arrival at a new university in a strange country I was fortunate to land in the office of Eliot Moss. As my advisor throughout my time at UMass, Eliot has been a constant source of support, encouragement, and inspiration. His intellectual and scholarly integrity has been the primary role model in my own academic development. It is an honor and a privilege that I am now able to count him as a colleague.

Special thanks must also go to my comrades in arms, the members of the Object Systems Laboratory, whose companionship (both academic and otherwise) I have greatly enjoyed. I thank Amer Diwan for his enthusiasm in educating me as to the intricacies of cache memory simulation, Eric Brown for his tireless support and development of the Mneme persistent object store, Darko Stefanović for his assistance debugging and improving the garbage collector toolkit, Rick Hudson for our many technical discussions, and Norm Walsh for his advice and assistance with TEX and LATEX (see also Walsh [1994]). Thanks also to the many members of the OOS lab over the years for their part in making it such a vital and interesting workplace.

This dissertation has been greatly improved by the comments of my examining committee, Lori Clarke, Jack Stankovic, Dave Stemple and Wayne Burleson. Their advice and interest have helped me immensely.

Lastly, I thank my family for their guidance, encouragement, and confidence. I thank my parents for nurturing me in an environment where the pursuit of knowledge was always

such incredible fun, and striving towards excellence a noble goal in itself. To my wife, Carla Brodley, I am forever indebted for her sage advice and unflagging support through the ups and downs of postgraduate student life. When obstacles appeared to block my way she saw the way around them if I did not.

——

This research has been supported by the National Science Foundation under grants CCR-9211272, CCR-8658074 and DCR-8500332, and by the following companies and corporations: Sun Microsystems, Digital Equipment, Apple Computer, GTE Laboratories, Eastman Kodak, General Electric, ParcPlace Systems, Xerox and Tektronix.

# ABSTRACT

LIGHTWEIGHT SUPPORT FOR FINE-GRAINED PERSISTENCE
ON STOCK HARDWARE

FEBRUARY 1995

ANTONY LLOYD HOSKING

B.Sc., UNIVERSITY OF ADELAIDE

M.Sc., UNIVERSITY OF WAIKATO

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor J. Eliot B. Moss

Persistent programming languages combine the features of database systems and programming languages to allow the seamless manipulation of both short- and long-term data, thus relieving programmers of the burden of distinguishing between data that is transient (temporarily allocated in main memory) or persistent (residing permanently on disk). Secondary storage concerns, including the representation and management of persistent data, are directly handled by the programming language implementation, rather than the programmer. Moreover, unlike traditional database systems, persistent programming languages extend to persistent data all the data structuring features supported by the language, not just those imposed by the underlying database system.

Prototype persistent languages have until now focused more on functionality than performance. In contrast, this dissertation addresses performance issues in the language implementation. It presents an architecture and framework for persistence which allows programming language implementation techniques to be brought to bear on the problem of performance. Building on this framework, a prototype persistent programming language is

implemented, and submitted to performance evaluation to obtain direct comparisons of the performance of several implementation alternatives for different aspects of persistence. The results of these performance evaluations, which use established benchmarks, indicate that persistence can be implemented on general-purpose machines without imposing significant overhead above and beyond the fundamental costs of data transfer to and from secondary storage. Moreover, the results show that software-mediated implementation techniques can be a competitive alternative to techniques that rely on low-level support from the operating system and hardware.

# TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS . . . . . . . . . . v

ABSTRACT . . . . . . . . . . vii

LIST OF TABLES . . . . . . . . . . xiii

LIST OF FIGURES . . . . . . . . . . xv

LIST OF ABBREVIATIONS . . . . . . . . . . xvii

Chapter

1. INTRODUCTION . . . . . . . . . . 1

1.1 Persistence . . . . . . . . . . 2
1.2 Scope of the Dissertation . . . . . . . . . . 3

1.2.1 The Thesis . . . . . . . . . . 3
1.2.2 Performance . . . . . . . . . . 4
1.2.3 Contributions . . . . . . . . . . 4

1.2.3.1 Model of Persistence . . . . . . . . . . 4
1.2.3.2 Prototype and Experimental Framework . . . . . . . . . . 5
1.2.3.3 Performance Evaluation . . . . . . . . . . 6

1.3 Outline of the Dissertation . . . . . . . . . . 7

2. A SURVEY OF PERSISTENCE . . . . . . . . . . 9

2.1 Languages . . . . . . . . . . 9

2.1.1 Database Programming Languages . . . . . . . . . . 10

2.1.1.1 Language Extensions . . . . . . . . . . 10
2.1.1.2 Languages with Advanced Data Models . . . . . . . . . . 14
2.1.1.3 Languages with Object-Oriented Data Models . . . . . . . . . . 17

2.1.2 Persistent Programming Languages . . . . . . . . . . 19

2.1.2.1 PS-Algol . . . . . . . . . . 20

2.1.2.2 Napier88 . . . . . . . . . . . . . . . . . . . . . . . . . 21
2.1.2.3 LOOM . . . . . . . . . . . . . . . . . . . . . . . . . . 22
2.1.2.4 Alltalk . . . . . . . . . . . . . . . . . . . . . . . . . . 23
2.1.2.5 Avalon/C++ . . . . . . . . . . . . . . . . . . . . . . . 23

2.2 Storage Management . . . . . . . . . . . . . . . . . . . . . . . . 24

2.2.1 Persistent Object Stores . . . . . . . . . . . . . . . . . . . . 24

2.2.1.1 ObServer . . . . . . . . . . . . . . . . . . . . . . . . . 24
2.2.1.2 Mneme . . . . . . . . . . . . . . . . . . . . . . . . . . 25
2.2.1.3 Texas . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
2.2.1.4 ObjectStore . . . . . . . . . . . . . . . . . . . . . . . 28
2.2.1.5 EXODUS . . . . . . . . . . . . . . . . . . . . . . . . . 28
2.2.1.6 Other Commercial Ventures . . . . . . . . . . . . . . . 31

2.2.2 Object Oriented Database Systems . . . . . . . . . . . . . . . 31

2.2.2.1 GemStone . . . . . . . . . . . . . . . . . . . . . . . . 32
2.2.2.2 ORION . . . . . . . . . . . . . . . . . . . . . . . . . . 33
2.2.2.3 $O_2$ . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

2.3 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

2.3.1 Essential Persistence Mechanisms . . . . . . . . . . . . . . . . 35

2.3.1.1 The Read Barrier: Object Faulting . . . . . . . . . . . 36
2.3.1.2 The Write Barrier: Detecting and Logging Updates . . . 36

2.3.2 Fine-Grained Persistence and the Performance Problem . . . . . . . 37

3. ARCHITECTURE . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

3.1 Object Faulting . . . . . . . . . . . . . . . . . . . . . . . . . . 40
3.2 Swizzling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
3.3 Resilience . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
3.4 Extensions to the Basic Architecture . . . . . . . . . . . . . . . . 43

3.4.1 Buffer Management . . . . . . . . . . . . . . . . . . . . . . . 44
3.4.2 Concurrency Control . . . . . . . . . . . . . . . . . . . . . . 44

4. IMPLEMENTATION . . . . . . . . . . . . . . . . . . . . . . . . . . 47

4.1 Smalltalk . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47

4.1.1 Object-Oriented Method Invocation . . . . . . . . . . . . . . . 48

4.1.1.1 Method Lookup . . . . . . . . . . . . . . . . . . . . . 48

4.1.2 The Virtual Machine . . . . . . . . . . . . . . . . . . . . . . 50

4.1.2.1 Object Pointers . . . . . . . . . . . . . . . . . . . . . . . 51
4.1.2.2 Threaded Code . . . . . . . . . . . . . . . . . . . . . . . . 52

4.1.3 The Virtual Image . . . . . . . . . . . . . . . . . . . . . . . . . . . 54

4.2 Persistent Smalltalk . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54

4.2.1 The Read Barrier: Object Faulting . . . . . . . . . . . . . . . . . . . 54

4.2.1.1 Residency Constraints . . . . . . . . . . . . . . . . . . . . 55
4.2.1.2 Edge Marking . . . . . . . . . . . . . . . . . . . . . . . . 58
4.2.1.3 Node Marking . . . . . . . . . . . . . . . . . . . . . . . . 59
4.2.1.4 Object Faulting Summary . . . . . . . . . . . . . . . . . . 60

4.2.2 Swizzling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
4.2.3 The Write Barrier: Detecting and Logging Updates . . . . . . . . . 62

4.2.3.1 Object-Based Schemes . . . . . . . . . . . . . . . . . . . . 62
4.2.3.2 Card-Based Schemes . . . . . . . . . . . . . . . . . . . . . 63
4.2.3.3 Page-Protection Schemes . . . . . . . . . . . . . . . . . . . 65

5. PERFORMANCE EVALUATION . . . . . . . . . . . . . . . . . . . . . . . . 67

5.1 Benchmarks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67

5.1.1 Benchmark Database . . . . . . . . . . . . . . . . . . . . . . . . . . 68
5.1.2 Benchmark Operations . . . . . . . . . . . . . . . . . . . . . . . . . 69

5.2 Experimental Setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70

5.2.1 The Read Barrier: Object Faulting . . . . . . . . . . . . . . . . . . . 73
5.2.2 The Write Barrier: Detecting and Logging Updates . . . . . . . . . 74

5.3 Results: Object Faulting . . . . . . . . . . . . . . . . . . . . . . . . . . . 75

5.3.1 Lookup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76

5.3.1.1 Cold Lookup . . . . . . . . . . . . . . . . . . . . . . . . . 78
5.3.1.2 Warm Lookup . . . . . . . . . . . . . . . . . . . . . . . . 80
5.3.1.3 Hot Lookup . . . . . . . . . . . . . . . . . . . . . . . . . . 80
5.3.1.4 Running Time Overheads . . . . . . . . . . . . . . . . . . 82

5.3.2 Traversal . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84

5.3.2.1 Cold Traversal . . . . . . . . . . . . . . . . . . . . . . . . 84
5.3.2.2 Warm Traversal . . . . . . . . . . . . . . . . . . . . . . . . 86
5.3.2.3 Hot Traversal . . . . . . . . . . . . . . . . . . . . . . . . . 86
5.3.2.4 Running Time Overheads . . . . . . . . . . . . . . . . . . 88

5.3.3 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88

5.4 Results: Detecting and Logging Updates . . . . . 89

5.4.1 Insert . . . . . 89
5.4.2 Update . . . . . 92

5.4.2.1 Cold Update . . . . . 93
5.4.2.2 Warm Update . . . . . 93
5.4.2.3 Hot Update . . . . . 93
5.4.2.4 Long Transactions . . . . . 99

5.4.3 Conclusions . . . . . 104

5.5 Related Performance Studies . . . . . 105

6. CONCLUSIONS . . . . . 107

6.1 Implications . . . . . 107
6.2 Future Work . . . . . 108

6.2.1 Compilation Versus Interpretation . . . . . 108
6.2.2 Other Architectures . . . . . 109
6.2.3 Higher-Level Functionality . . . . . 110

6.3 Closing Remarks . . . . . 111

APPENDICES

A. SMALLTALK SOURCE LISTING OF OO1 BENCHMARKS . . . . . 113
B. SUPPLEMENTAL RESULTS: OBJECT FAULTING . . . . . 133

BIBLIOGRAPHY . . . . . 141

# LIST OF TABLES


Table Page

4.1 Tagging object pointers . . . 53

5.1 Schemes compared in object faulting experiments . . . 74

5.2 Schemes compared in resilience experiments . . . 75

5.3 Lookup: key cold/warm/hot results . . . 79

5.4 Lookup: fault detection overheads . . . 83

5.5 Traversal: key cold/warm/hot results . . . 87

5.6 Traversal: fault detection overheads . . . 88

5.7 Insert . . . 90

5.8 Long-running Update: traversal overheads (per part modified) . . . 103

5.9 Long-running Update: intrinsic run-time overheads (per update) . . . 103

5.10 Long-running Update: break-even points for cards-4096 versus pages . . . 103

# LIST OF FIGURES


Figure | Page

3.1 System architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40

3.2 Node marking . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

4.1 Objects, classes, inheritance, and object-oriented method invocation . . . . 49

4.2 Method lookup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50

4.3 Object pointers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52

4.4 Edge marking . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58

4.5 Node marking: fault blocks . . . . . . . . . . . . . . . . . . . . . . . . 59

4.6 Node marking: indirect blocks . . . . . . . . . . . . . . . . . . . . . . . 60

5.1 Lookup, by object faulting scheme . . . . . . . . . . . . . . . . . . . . . 77

5.2 Traversal, by object faulting scheme . . . . . . . . . . . . . . . . . . . . 85

5.3 Insert . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90

5.4 Insert: Hot breakdown . . . . . . . . . . . . . . . . . . . . . . . . . . . 92

5.5 Cold Update . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94

5.6 Cold Update (expanded scale) . . . . . . . . . . . . . . . . . . . . . . . 94

5.7 Warm Update . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95

5.8 Warm Update (expanded scale) . . . . . . . . . . . . . . . . . . . . . . . 95

5.9 Hot Update . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97

5.10 Hot Update (expanded scale) . . . . . . . . . . . . . . . . . . . . . . . 97

5.11 Update: Hot breakdown . . . . . . . . . . . . . . . . . . . . . . . . . . 98

5.12 Long-running Update: checkpoint overhead per transaction . . . . . . . . . 99

5.13 Traversal overheads . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101

B.1 Lookup, swizzling one object per fault . . . . . . . . . . . . . . . . . . . . . . 134
B.2 Lookup, swizzling one LSEG per fault . . . . . . . . . . . . . . . . . . . . . . 135
B.3 Lookup, swizzling one PSEG per fault . . . . . . . . . . . . . . . . . . . . . . 136
B.4 Traversal, swizzling one object per fault . . . . . . . . . . . . . . . . . . . . . 137
B.5 Traversal, swizzling one LSEG per fault . . . . . . . . . . . . . . . . . . . . . 138
B.6 Traversal, swizzling one PSEG per fault . . . . . . . . . . . . . . . . . . . . . 139

# LIST OF ABBREVIATIONS

| | |
|---|---|
| FB | fault block |
| KB | kilobyte(s): 1KB = $2^{10}$ bytes |
| LSEG | logical segment |
| MB | megabyte(s): 1MB = $2^{20}$ bytes |
| PID | persistent identifier |
| PSEG | physical segment |
| RISC | reduced instruction set computer |
| VM | virtual machine |

# CHAPTER 1

# INTRODUCTION

The problem of storing, retrieving, and manipulating large amounts of highly-structured information, while sharing that information among multiple, cooperating users, attracts considerable research interest. New classes of applications including computer-aided design (CAD), computer-aided software engineering (CASE), document preparation, and office automation, demand capabilities that far exceed those of traditional programming languages and database systems. These applications highlight a need for unified programming environments that integrate the concepts of data manipulation and long-term storage management.

While traditional programming languages have successfully solved the data manipulation problem, their support for long-term storage management is inadequate. Typically, they provide only a very weak notion of long-term storage in terms of "files", onto which programmers must map their long-term data. This requires considerable effort to translate from a highly-structured in-memory format to the "flat" format required for transfer to and from a file. Not only is this inconvenient and a source of programming errors, it is also potentially dangerous, since the translation process may bypass type safety.

On the other hand, traditional database systems, having been designed to solve the storage management problem, typically restrict the model of data in the database. For example, relational database systems provide just one long-term data type — the relation — along with a limited set of operators for manipulating relations. The resulting lack of computational completeness severely limits the kinds of applications that can be programmed directly using just relations and relational operators, requiring application writers to step outside the database language into some general-purpose programming language when

they desire the expressiveness that the database language lacks. There is then the attendant problem of converting data between the relational format and the format expected by the general-purpose programming language. For example, graphical user interface front-ends to traditional database systems are typically programmed in some high-level programming language, in which database language code fragments are embedded to access and manipulate the underlying database. Thus, the application writer must be familiar with two different programming paradigms and their (not necessarily straightforward) integration.

This lack of integration of data manipulation language with storage management, requiring programmers to be explicit in converting data between internal and external formats, has been characterized as an *impedance mismatch* [Copeland and Maier, 1984] between programming languages and database systems. In an attempt to overcome this mismatch, researchers have devoted considerable effort to forging a new technology, integrating many of the features of languages and databases.

## 1.1 Persistence

The notion of *persistence* — the quality or state of continuing to exist — is the driving force behind the convergence of programming languages with databases. A *persistent system* is a software system which maintains data whose lifetime is independent of the programs that create and manipulate that data. The data are maintained in a *persistent store*, which programmers view as a stable extension of volatile memory, in which they can dynamically allocate new data, but which persists from one program invocation to the next. Programs to create and manipulate the data are written in a *persistent programming language*, which allows traversal and modification of the data structures in the persistent store to be programmed *transparently*, without explicit calls to read and write the data. Rather, the language implementation and run-time system contrive to make persistent data resident in memory on demand, much as non-resident pages are automatically made resident by a paged virtual memory system. Moreover, a persistent program can modify persistent

data and commit the modifications so that their effects are permanently recorded in the persistent store.

## 1.2 Scope of the Dissertation

This dissertation explores several dimensions of the implementation design space for persistent programming languages, focusing on alternative implementation strategies for the two fundamental mechanisms of persistent systems: the reading and writing of data that reside only temporarily in memory, but permanently on disk. It evaluates the performance of these alternatives within a prototype persistent programming language, to determine if a clean model of persistence is intrinsically expensive, or can be supported efficiently on stock hardware.

### 1.2.1 The Thesis

We contend that problems of efficiency are best attacked directly in the language implementation and run-time system, where persistence mechanisms can be implemented with very low cost, and the language implementor can exploit semantic-level properties of the language to eliminate unnecessary overhead. Our thesis is summed up as follows:

> *Transparent persistence can be efficiently supported by the language implementation and run-time system, without resorting to specialized hardware or relying on specific features of the underlying operating system and hardware architecture.*

That persistent systems need not rely on specialized hardware and operating system features is important, for it means that persistence can be implemented on virtually any stock hardware, putting persistent systems within the reach of all computer users. Moreover, it has been demonstrated time and again that reliance on specialized features of hardware and operating systems leaves an implementation vulnerable to being left behind by rapid advances in computer technology. Witness the fate of Lisp machines and microcoded

Smalltalk implementations. In contrast, implementations that are independent of both hardware and operating system can often be migrated to new computer systems with ease.

### 1.2.2 Performance

The claim to efficiency requires some definition of *performance*, of which persistent systems have two main aspects. The first is the overhead to manage main memory as a volatile cache of resident persistent data: detecting the need to retrieve data into memory, and flushing updates back to secondary storage. The goal is to minimize this overhead, so that in the case that all data needed by a persistent program is resident, then its performance will approach that of its non-persistent counterpart.

The second aspect of performance pertains to the transfer of data between memory and secondary storage, including the conversion of data from its disk format to the memory format expected by the persistent program. Efficiency in this aspect of performance is more difficult to define, but is naturally characterized as that which makes the best use of system resources (both space and time).

### 1.2.3 Contributions

This dissertation makes several contributions:

- a practical model of persistence based on *object faulting* that can be supported efficiently within the programming language implementation, allowing *new techniques* to manage the *retrieval*, *manipulation*, and *modification* of persistent data
- a *prototype system* and *novel experimental framework* for meaningful direct comparison of alternative implementations of persistence mechanisms
- *performance results* comparing the different implementations, showing several new techniques are robust over a broad range of program behaviors

#### 1.2.3.1 Model of Persistence

The model of persistence, based on *object faulting*, is flexible enough to admit many different implementation approaches for the fundamental mechanisms of persistence. These

mechanisms mediate the retrieval of persistent data (stored on disk) into main memory for direct manipulation by the executing program, and the propagation of updates to that data back to stable storage:

- An object fault occurs when the program attempts to access a data object that is not yet resident in memory. This dissertation considers several implementations for the detection of object faults, including explicit checks in software and hardware-assisted approaches that take advantage of the virtual memory page protection primitives supported by the operating system and hardware.
- Once an object is made resident there is no reason why the executing program should not refer to it directly in memory, so that accessing a resident persistent object incurs no additional overhead. We consider several representations for references to *resident* persistent data that circumvent the otherwise high overhead of persistent references.
- Updates occurring to (memory-resident) persistent data must ultimately be logged to stable storage to ensure their persistence. We consider a number of schemes for tracking updates for efficient generation of the log, including new schemes that detect updates explicitly in software, as well as the use of virtual memory primitives to maintain per-page dirty bits.

These mechanisms constitute the minimal functionality any persistent system must offer, allowing the safe, efficient, and reliable management of persistent data. More complex functionality (e.g., concurrency control, distribution, interoperability) is not absolutely necessary, although it may certainly be desirable. The mechanisms can be supported directly by the language implementation, or as mentioned, by relying on virtual memory primitives.

#### 1.2.3.2 Prototype and Experimental Framework

The prototype implements our model of persistence for an efficient implementation of the Smalltalk programming language and system. The object-oriented execution paradigm of Smalltalk allows us to minimize the overheads of residency checks for object faulting by imposing residency constraints on objects crucial to the forward progress of computation. The result is that residency checks occur only on the receiving object at time of method lookup (the process by which Smalltalk implementations effect class-based polymorphism).

Method lookup caching means residency checks occur only when a method lookup cache miss occurs (see Chapter 4 for further details).

The prototype persistent Smalltalk implementation can be tailored so that different instantiations of the prototype use different implementations of the low-level persistence mechanisms. All other aspects of the implementation are kept constant across all instances of the prototype. This allows a head-to-head comparison of alternative implementations where only the particular persistence mechanism under study varies across all instances of the prototype. That is, the generic prototype system represents a fixed frame of reference for comparison of different points on several dimensions of the design space (e.g., object fault detection, object retrieval, update detection), by allowing variables along each dimension to be controlled separately. In this respect, the prototype is a unique experimental test-bed for the exploration of persistent systems implementation.

In addition, the prototype implementation is able to take advantage of the object-oriented execution paradigm of Smalltalk to restrict the need for software-mediated residency checks to method invocation, and then only on the receiver of the invocation. This comes as a result of careful consideration of the object residency requirements for executing Smalltalk programs to make forward progress.

#### 1.2.3.3 Performance Evaluation

The performance evaluation represents the first performance study of implementation alternatives for persistent programming languages, within a single prototype system. The results, which measure the absolute cost of alternative object faulting and update detection mechanisms, demonstrate that persistent programming languages can be both *practical* and *efficient*, where previous work has explored only functionality rather than performance. In particular, the prototype exhibits performance near optimal for resident objects (with variations attributable only to underlying hardware cache effects). Moreover, our newly-devised software-mediated persistence mechanisms have performance as good, if not significantly

better than, equivalent approaches that rely on virtual memory primitives supported by the operating system and hardware. The results demonstrate the extreme importance of our residency constraints in the near elimination of residency checking overheads.

The results also have implications with respect to the performance of operating systems in their support of memory management for persistence and other applications. In particular, software-only methods can outperform approaches that rely on virtual memory page protection primitives. Much of the problem is attributable to the large page granularity of modern operating systems, where the objects being managed in memory are much smaller than a page. Nevertheless, the overheads to manage page protections, and to field page protection violations at user level, are high enough to significantly degrade the performance of programs that rely on virtual memory page protection to detect anything but the most exceptional conditions.

The performance results stand on their own in demonstrating that persistence can offer useful functionality while being efficiently realizable on stock hardware, making it a mature feasible technology that is deserving of serious consideration by database system and language designers alike.

## 1.3 Outline of the Dissertation

The rest of the dissertation is outlined as follows. Chapter 2 surveys literature relevant to persistence, focusing on the convergence of programming language and database technology to support persistence. Given this background it then describes our notion of the key mechanisms crucial for high-performance persistent systems. Chapter 3 presents our architecture for persistence and its rationale, which serves as the basis for the implementation of persistent Smalltalk described in Chapter 4. In Chapter 5, we look at the performance of several alternatives for implementation of persistence mechanisms, by subjecting several versions of the prototype persistent Smalltalk implementation to a range of benchmarks.

Finally, Chapter 6 summarizes the key results and their implications, and describes possible research directions for future exploration.

# CHAPTER 2

# A SURVEY OF PERSISTENCE

Persistence should be understood in the context of a fundamental convergence of the research directions of the historically distinct fields of programming languages and database systems. This requires that we familiarize ourselves with the relevant culture and history of both fields in relation to several aspects of persistence: languages for manipulation of persistent data and storage management for persistent data. We begin by considering the development of database and persistent programming languages, by examining their characteristics and key features. Storage management solutions break down into approaches that are minimal *persistent object stores*, versus *object-oriented databases*, which support a higher degree of database functionality. Having surveyed the relevant literature we then refine our definition of persistence to focus on its key mechanisms in order to frame the performance problem that we address in this dissertation.

## 2.1 Languages

The database world has approached the problem of impedance mismatch, which results from embedding data definition and query language constructs in a traditional programming language, by developing *database programming languages* that make database constructs a part of the programming language. In programming languages, the effort has been in the other direction, making the full type system of the programming language available for use in defining and manipulating persistent data. This distinction is somewhat fuzzy, but may be summed up by treating languages that provide some sort of *bulk* long-term data type, along with operations for manipulating it, as database programming languages, and others,

in which arbitrarily typed data can persist, as persistent programming languages. Atkinson and Buneman [1987] give a comprehensive survey on the issues of types, persistence, and database features in programming languages.

### 2.1.1 Database Programming Languages

We group the database programming languages by their defining characteristics, as follows. The earliest database programming languages simply extended an existing programming language to include a relation data type. More recently, there has been a trend towards languages that incorporate advanced *semantic data models*; these are intended to support a more natural definition of the user-level semantics of the database than can be captured in a pure relational model. Still other database languages have been designed for programming applications in object-oriented database systems, so they incorporate an object-oriented data model.

#### 2.1.1.1 Language Extensions

The following languages all add relations to an existing programming language. In so doing they place certain restrictions on the attribute values that may be stored in the tuples of relations. In particular, a tuple may not contain pointers; they justify this restriction not only because of the implementation challenges required to support persistent pointers, and the sharing semantics this implies, but also because it is difficult to express such semantics within the pure relational data model.

**Pascal/R** The first of the extended languages was Pascal/R [Schmidt, 1977; Schmidt and Mall, 1980]. It extends Pascal with a relation type constructor `relation of`, similar to `set of`. This constructor takes two parameters: a `record` type, corresponding to the type of the tuples of the relation; and a subset of the record's fields as the primary key of the relation.

The tuples of a relation are restricted to "flat" records — all tuple fields must be scalar or string values.

A relation may be altered by the elementary operations of insertion, deletion, and replacement of tuples in one relation based on the tuples in another relation along with their primary keys. For example, the statement:

$$\mathcal{R}_1 \mathrel{:+} \mathcal{R}_2;$$

(where $\mathcal{R}_1$ and $\mathcal{R}_2$ are relation variables of the same type) inserts into $\mathcal{R}_1$ copies of those tuples of $\mathcal{R}_2$ whose key values do not already occur in any tuple of $\mathcal{R}_1$. $\mathcal{R}_2$ remains unchanged.

Low-level, tuple-wise iteration over a relation, from the tuple with the lowest key value to the tuple with the highest key value, is provided through the elementary retrieval operations of low (initialize iteration) and next (get the tuple with next highest key value), along with the boolean function aor (all of relation). These are used in conjunction with the standard Pascal iteration statements. In many cases, ordering the access based on the key is unnecessary and may prevent a more efficient access order. For this reason, a new high-level iteration statement for each is provided to allow repetition of a computation for each tuple of the relation, in some arbitrary order.

Two other extensions realize the full power of the relational calculus. First, existential and universal quantifiers are provided for use in Boolean expressions to construct predicates over relations; these quantifiers may be nested. Second, a relation constructor permits the expression of an arbitrary relational calculus query with one Pascal/R statement.

Finally, the program must be bound to a database in some way. This is achieved by specifying a database variable in the parameters to the program (cf. Pascal's file parameters), and declaring its type using the database type constructor to specify the relations that constitute the program's view of the database. Type safety is guaranteed by checking that the type of the database variable is an acceptable view of the actual database when the database is bound to the program parameter.

**DBPL** Following on from Pascal/R, Matthes and Schmidt [1989] designed its successor, DBPL, by integrating a set- and predicate-oriented view of relations into Modula-2. The bulk data type is the set, which is an extension of the relation type of Pascal/R to include elements of arbitrary type, with operators and constructors similar to those of Pascal/R. Like Pascal/R relations, a set may be keyed on a list of components of its element type — no two elements of the set can have the same key value.[1] Unlike Pascal/R relations, sets need not be keyed.

Low-level access to the elements of a keyed set is supported through associative access based on the key, much as array elements are indexed, or record components are named. High-level access is provided through *access expressions*. An access expression denotes those elements of a set that satisfy some selection predicate. The elements denoted by an access expression may be formed into a new set using the set constructor. There is also a for each statement, similar to that of Pascal/R, which iteratively performs some computation for each of the elements denoted by a given access expression.

Any variable declared (in the usual way) at the outermost scope of a database module will be a persistent database variable. DBPL retains pointer types for compatibility with Modula-2, although they cannot persist, so database variables cannot contain pointers. All other values (e.g., scalars, records, variant records, arrays, etc.) can persist by being stored in database variables, in contrast with Pascal/R where only relations can persist. Since pointers cannot persist and recursive data structures are prohibited in DBPL, recursive structures and persistent data with shared sub-components must be modelled using keyed sets.

DBPL also supports a transaction model for concurrent access. Access to database variables is restricted to transaction procedures, whose effects on the database are atomic. Nested and recursive calls to transaction procedures are prohibited.

[1] A key component cannot be part of a variant section of a record type or be of type set.

**PLAIN** Wasserman [1979] designed PLAIN (Programming LAnguage for INteraction) to support the construction of interactive information systems, drawing on the procedural control structures, type system, and variable declarations of Pascal. Relations in PLAIN are much the same as those in Pascal/R, with the same restrictions on their attribute types.

Data access is provided in a number of ways. Individual tuples can be accessed associatively by their key, as in DBPL. Tuples may be inserted into and deleted from relations. There is also a generalized iterator applicable to relations as well as other composite objects such as sets.

Unlike Pascal/R, the high-level operations on relations are the relational algebra operators select, project, and join, along with set intersection, union, and difference. Intermediate results, obtained through the evaluation of algebraic expressions, may be assigned to variables of type marking. These are similar to relations, except they are unkeyed (permitting duplicate tuples), cannot persist, and cannot be updated; as such, they are merely "snapshots" of relations or other markings.

**RIGEL** Yet another procedural Pascal-like language that incorporates relations, RIGEL [Rowe and Shoens, 1979] is similar to Pascal/R in that its expressions include the relational calculus. Like PLAIN it provides an iterator that is a generalized version of the Pascal for statement, which iterates over each element of a sequence of values produced by a *generator* expression. The generator expression for relations is the where statement, which generates a selected sequence of tuples from a relation, much like an access expression in DBPL.

An interesting feature of RIGEL is its extension of data abstraction into the database, including support for modules and abstract data types. Modules allow the definition of alternative interfaces to a database, with client programs importing only those interfaces they need. The view type constructor extends the power of abstract data types to relations. A view is an abstract relation, defined by a mapping from the tuples of the base relations of the database to the view's tuples. Views may be updated, with certain restrictions: the

view programmer must specify update operations in terms of updates to the underlying base relations.[2] Combined with modules, views allow the definition of interfaces that are very different from those embodied by the base relations.

**Theseus** Lastly, Theseus [Shopiro, 1979] was designed as a language for use in a test-bed for exploring program optimization issues. Unfortunately, Theseus was never implemented, so its practical value remains unproven. Theseus is an extension of Euclid (itself a Pascal extension) with a relation type constructor and primitive operations which make for relational completeness: any expression in the relational algebra can be computed by a Theseus program.

A distinguishing feature of Theseus is its use of association sets as the elements of the relations. Essentially a generalized record, an association set is a set of name-value pairs. The names are declared globally along with their type, to allow static type checking — a name can only be associated with values of its type. Pointers, association sets, and names are not permitted as values in association sets.

#### 2.1.1.2 Languages with Advanced Data Models

The relational extensions of the previous section require all bulk data to be cast in terms of relations. To overcome this inflexibility, semantic data models have been developed to allow a more natural definition of the database in terms of user level abstractions. The languages have several features in common, including *inheritance*; some also interpret *classes* extensionally as bulk data.

**ADAPLEX** Smith et al. [1983] took the *functional data model* of DAPLEX [Shipman, 1981], which views the database as a collection of functions mapping entities to other values,

[2] This avoids the problems of *undefined* and *ambiguous* updates: an undefined update is one for which no sequence of operations on the base relations will produce a correct update when the data is accessed through the view; an ambiguous update is one for which more than one sequence of operations will produce a correct view update but each produces a very different meaning on the underlying data.

and merged it with with the Ada programming language to create ADAPLEX. ADAPLEX supports subtyping with (single) inheritance, allowing an *entity* type to be defined as a subtype of another, inheriting its attributes. Each entity type has an associated implicit *extent* (class), comprising all entities of that type currently in the database. The language provides iterative access to all the elements of an extent that satisfy some predicate.

**TAXIS** The conceptual modelling language TAXIS [Mylopoulos et al., 1980; Nixon et al., 1987] is designed to support the high-level modelling of interactive information systems. Classes and (multiple) inheritance are the chief means for structuring the world. Every entity in the conceptual schema is represented as an instance of some class. In true object-oriented fashion, everything in TAXIS is an entity, including numbers, strings, and even classes (instances of some *metaclass*) and running programs (instances of some transaction class). Inheritance is embodied in the *is-a* relationship, which allows the definition of one class as a derivation of a number of other classes.

Like ADAPLEX, TAXIS takes an extensional view of classes: a class is a collection of entities that share common properties. Note that the extent of a subclass is a subset of the extent(s) of its superclass(es). TAXIS supports a number of mechanisms for defining extents. *Variable* classes permit updates to their extents through the operations of insertion and deletion, as well as other operations including the iterative retrieval of selected instances from their extent — variable classes may be keyed and roughly correspond to relations. An *aggregate* class defines its extent as the cross product of the instances (i.e., the extents) of a group of classes. *Finitely defined* classes have their extents defined once and for all when they are first defined (cf. enumerated types in Pascal). All of these are generalized by *test-defined* classes, which define their extent procedurally in terms of a transaction that generates the instances. These are particularly interesting since they allow the expression of arbitrarily complex constraints on the instances of a class.

**Galileo** Another conceptual modelling language, Galileo [Albano et al., 1985; Albano et al., 1989], takes a slightly different approach by separating the concepts of class (extent) and type.[3] Subtyping deals with the *intensional* aspects of inheritance (for static checking of type compatibility among all the possible values of a type and those of its supertypes), while subclasses deal with the *extensional* aspects of inheritance.

**ADABTPL** The last of the advanced data models is found in ADABTPL (Abstract DAtaBase Type Programming Language) [Stemple et al., 1988; Fegaras et al., 1989; Sheard and Stemple, 1989], a high-level functional database programming language based on sets. It is much more than a conceptual modelling language since its goals far exceed mere conceptual modelling; rather, it is intended as a language for *formally* specifying and implementing database systems. ADABTPL is also distinguished by its omission of any notion of extent or class; in this regard it is more closely related to DBPL with its provision for sets as bulk data types.

The composite types of ADABTPL include tuples (labelled cross products), lists, and sets. Explicit subtyping is introduced by several of its type constructors:

- where restricts a type by admitting only those values that satisfy a given predicate, allowing the expression of database integrity constraints, such as indicating that a set is keyed
- discriminated union (tagged disjoint sum) types (similar to the union types of C, except that for type safety they must be qualified by their tag) are supertypes of their component types
- inherits defines a subtype having all the operations of its supertype(s)
- with adds tuple components to any given type, to create a subtype
- parameterized types are supertypes of their instantiations

There are also implicit subtyping rules for tuples, sets, and functions.

ADABTPL is a functional language. This has a number of consequences. First, it treats functions as first-class values. Second, it is side-effect free, so there are no variables, only

[3] Actually, since every class must be defined in terms of the abstract type of its elements, each class must have a corresponding unique type, so class and type are not totally separated in Galileo.

immutable values. There is one exception to this, arising out of the need to view a database as a mutable entity, with a value that changes over time. Every ADABTPL schema defines a named database variable, along with its type. This is an implicit input to all transactions, which are written in an imperative style. While the effect of a transaction is to compute values and update the database, its *semantics* is given by a function that takes the database and inputs, and returns a new database. Furthermore, the pure recursive function semantics of the ADABTPL type system have been axiomatically formalized. This formulation drives the automatic verification of transactions, providing feedback to database designers about transactions that violate integrity constraints.

#### 2.1.1.3 Languages with Object-Oriented Data Models

Object-oriented database programming languages incorporate the salient features of the object-oriented data model: objects, methods, classes, inheritance, and *identity*. Object identity is the main feature distinguishing object-oriented languages from the value-oriented languages we have seen so far. While value-oriented languages treat two entities as identical if they have the same value, object-oriented languages regard an object as having a unique identity for all time, regardless of its value — objects can have the same value and still not be identical. Object identity can be characterized in terms of pointer semantics: every object has a unique *identifier* somewhat like a memory pointer, by which it can be referenced. Object-oriented databases support such pointer semantics, so that references between persistent objects are retained, allowing explicit sharing of subcomponents among different objects in the database. This semantics is reflected in the following programming languages, which all use generalized pointers to refer to both in-memory and persistent objects.

**E** The "persistent systems implementation language" E [Richardson and Carey, 1987; Richardson and Carey, 1990] augments C++ with *database* types, which are expressed by

adding the prefix db to the standard C++ type names and type constructors; thus, any C++ type can be analogously defined as a db type. Only values of db type can persist, so persistence is not orthogonal. Any variable of db type can be qualified with the persistent storage class, indicating that the variable is to be bound to a value in the database — assignments to the variable are reflected as updates to the database.

E qualifies as a database programming language through its provision for bulk data in the form of *collections*, which are introduced via another extension to C++: *generator classes*. A generator class is a parameterized C++ class, and must be instantiated before it can be used. Instantiations of the built-in generator class collection define an unordered collection of elements of a given db type. Items may be entered into a collection only when they are created, with a special form of the new statement. When an item is destroyed (with the delete operator) it is also removed from its collection; destroying a collection has the effect of destroying all its elements first. E also extends C++ with *iterators*, which are a high-level generalization of looping control structures, similar to those of Pascal/R, PLAIN, and RIGEL. An iterator for scanning the elements of a collection is a built-in member function of the collection generator class.

The net result is that data can persist in one of two ways in E: either by assignment to a statically allocated persistent variable, or by dynamic allocation in a persistent collection. Database objects are deallocated explicitly, just as heap-allocated volatile data in C++ is deallocated explicitly. There is no restriction on the values that may persist or be stored in a database collection. In particular, pointers to db types can persist, so that sharing of subcomponents between objects is automatically preserved in the database.

**O++** The object-oriented database programming language O++ [Agrawal and Gehani, 1989a; Agrawal and Gehani, 1989b] is similar to E in that it is based on C++. It differs from E in collapsing the distinction between database types and volatile types, with persistence being an attribute of object instances rather than object types. Just as volatile objects are

allocated with the new operator, there is a corresponding operator, pnew for allocating persistent objects in the database; similarly, deletion is explicit using the pdelete operator. The keyword persistent is a type qualifier (like const), so that pointer types can be declared to refer to persistent objects. The dual type qualifier allows the declaration of pointers that can refer to both persistent and volatile data, with run-time checking to determine whether the referenced object is persistent or volatile.

Bulk data are provided by the *cluster* and *set* data types. Clusters are extents: all persistent objects of the same type are placed in the cluster corresponding to that type; before a persistent object can be allocated, the cluster for its type must be created. Sets are actually multisets (they can contain duplicates), are declared like arrays, and have the operations of assignment, union, difference, insertion, and deletion. Selective, ordered access to the elements of sets and clusters is supported by an iterator that is optionally qualified by a predicate that each element must satisfy, and an expression by which element retrieval should be ordered. A similar iterator allows iteration over all the elements of a type's cluster as well as the elements of the clusters of its subtypes. Arbitrary joins are supported by allowing multiple loop variables for an iterator.

### 2.1.2 Persistent Programming Languages

In contrast with the database programming languages, persistent programming languages have focussed on placing the full type system of the language at the programmer's disposal in defining persistent data. Early on, Atkinson et al. [1983a] characterized persistence as "an orthogonal property of data, independent of data type and the way in which data is manipulated". This particular characterization has important ramifications for the design of persistent programming languages, since it encourages the view that a language can be extended to support persistence with minimal disturbance of its existing syntax. With the exception of Avalon/C++, the following persistent languages achieve this admirably.

All persistent programming languages represent an attempt to extend the address space of programs beyond that which can be addressed directly by the available hardware, just as virtual memory represents the extension of the memory address space of a program beyond that of physical memory (virtual address translation allows transparent access to data regardless of its physical memory location; the operating system and hardware cooperate to trap references to pages that are not yet resident in physical memory). All provide an abstraction of persistent storage in terms of a persistent dynamic allocation heap: objects in the heap are referred to by language-supported pointers. If the entire heap can fit in virtual memory, then pointers can be represented as direct virtual memory addresses. However, this limits the size of the heap to that of the virtual address space. Extended addressability requires pointers that are not necessarily virtual memory addresses, as well as a mechanism to perform translation of pointers to virtual addresses to allow the program to manipulate the data.

#### 2.1.2.1 PS-Algol

PS-Algol [Atkinson et al., 1982; Atkinson et al., 1983b; Atkinson et al., 1983a] can lay claim to being the first true persistent language, since it was envisioned as a minimal extension to the heap-oriented programming language S-Algol. Persistence was incorporated with no change to the syntax of the programming language by modifying the language run-time system to allow heap structures to persist, and providing a small number of new predefined procedures for binding programs to these preserved heap structures. A PS-Algol database consists of an associative table in which the root objects of heap structures are associated with a string or integer key. A PS-Algol program is able to open any number of databases for read or exclusive write access, enter and look up objects in a database, scan through the entries in a database executing some function for each entry, commit any changes made to a database open for writing (saving all objects reachable from the root entries), and close a database to make it available for write access to other users. The

overall effect is that *any* value in a PS-Algol program can be made to persist, no matter what its type, so persistence is fully orthogonal. Atkinson and Morrison [1985] indicate that this notion of persistence has profound implications for the way one writes programs, particularly if procedures are first-class data values in the programming language.

Because pointers in PS-Algol are untyped, every PS-Algol object must carry its type with it, so that run-time type checking can be performed whenever pointers to it are dereferenced. Such type checking also includes a residency check to ensure that the target object is resident. These overheads have a significant impact on program performance.

#### 2.1.2.2 Napier88

Napier88 [Morrison et al., 1989; Dearle et al., 1989] is the successor to PS-Algol. As a new language, it was designed in the light of hindsight gained from experience with its predecessor. In contrast with PS-Algol, Napier88 employs strong typing, so that static type checking can be performed in certain circumstances to eliminate some of the run-time overhead that was so costly in PS-Algol. Nevertheless, there are situations that require dynamic type checking, so Napier88 objects must still carry their type with them.

The associative nature of the database in PS-Algol is generalized by the Napier88 *environment*. This is a collection of associations between names, values, and their types. Environments are first-class values in Napier88: they can be assigned, returned by functions, and manipulated just like other values; name-value pairs can be dynamically inserted and deleted from them. All expressions are viewed as being evaluated in the context of some environment (the names in the expression denote values in the current environment) whether that environment is bound to the expression statically or dynamically. In the case that binding is static, then type checking can also be performed statically. If binding is dynamic, then dynamic type checking occurs at the moment of binding to ensure that the values in the environment are of the same type as expected by the expression. This allows the expression itself to be statically type checked in the knowledge that the binding will

be performed safely. To provide for persistence there is a top-level persistent environment PS (for Persistent Store), which binds names to values in the database. As in PS-Algol, all objects reachable from this persistent root will persist.

#### 2.1.2.3 LOOM

LOOM [Kaehler and Krasner, 1983; Kaehler, 1986] represents another early attempt to extend the size of the heap beyond that addressable by a machine word. Its goal was to provide extended virtual memory support for Smalltalk systems on machines with a narrow (16-bit) word width. Object pointers are stored in 32 bits on disk, and an object table is used to translate between the short and long forms. When an object is brought into memory, its 32-bit persistent pointer is hashed to find an entry for it in the object table. All in-memory references to the object are then indirected through its object table entry. If the object contains references to other objects then they must be converted to short form, in a process which has since been termed *swizzling*: references to resident objects are converted to their in-memory short object pointer; references to non-resident objects are represented either as an in-memory pointer to a *leaf* or as a *lambda*. A leaf is a resident proxy object that represents an object on disk, containing sufficient information to locate that object. A lambda is a place-holder (actually the null pointer with value zero) for a pointer to an object that has not yet been assigned a short pointer.

Object faults in LOOM are triggered via explicit checks, isolated to certain operations in the Smalltalk interpreter. A leaf is distinguished by a bit in its object table entry. Objects containing lambdas are also specially marked. Handling an object fault means retrieving the object from disk and converting it to its in-memory format. This object-at-a-time transfer of objects between memory and disk is the major downfall of the LOOM approach, since Smalltalk objects are too small a unit for cost-effective transfer, resulting in thrashing — I/O overheads dominate total execution time. LOOM's implementors speculate on refinements to

their system that would group objects together for more efficient transfer between memory and disk.

#### 2.1.2.4 Alltalk

Alltalk [Straw et al., 1989] takes a similar approach to LOOM in its implementation of a *persistent* Smalltalk system. However, Alltalk does not swizzle objects between disk and memory formats. Thus, object pointers are always external identifiers that must be translated whenever they are dereferenced. Although this saves overhead in writing and reading data to and from disk, it does appear to severely curtail the execution speed of Alltalk programs — the implementors report a factor of four slowdown in the running time of programs running under Alltalk versus a non-persistent Smalltalk implementation.

Alltalk records updates by setting a flag in objects as they are modified. The flags are consulted by the transaction manager at time of commit to determine which objects have been modified.

#### 2.1.2.5 Avalon/C++

Avalon/C++ [Detlefs et al., 1988] is a language for programming reliable distributed systems, with provision for transactions and persistence of distributed data. It supports these features by extending C++ with three built-in classes: recoverable, atomic, and subatomic. Class recoverable has two primitive operations for ensuring persistence: pin causes the pages containing an object to be pinned in memory, whereupon the object can be modified; unpin logs the modifications to stable storage, and unpins the object's pages. Any object whose type inherits from the recoverable class will be persistent; changes that are made to it between calls to pin and unpin are guaranteed to be reflected in the stable store. The atomic class provides read/write locking, while subatomic allows for more fine-grained synchronization.

Because atomicity, recoverability, and persistence are attributes of classes, orthogonality is violated, since only the instances of classes inheriting those attributes can have those properties.

## 2.2 Storage Management

Architectures for persistence typically have one component in common: a *storage manager*, responsible for maintaining data in some inexpensive stable storage medium such as magnetic disk and for fielding requests to retrieve and save specified data. *Object-oriented* storage managers allow retrieval of a data *object* based on its *identifier* (ID). Systems that support identifier-based retrieval have been called *persistent object stores*. Object-oriented database systems also support this style of access, but they are distinguished from the persistent object stores by their provision of full database functionality, including concurrency control, recovery, transactions, distribution, data access via associative queries, and languages for data definition and manipulation.

### 2.2.1 Persistent Object Stores

Persistent object stores all have some primitive notion of object and identity. They vary in the level of semantics they give objects: some see objects as "bags of bytes", while others impose a more structured view. They also vary in their support of different aspects of database functionality.

#### 2.2.1.1 ObServer

ObServer [Skarra et al., 1986; Hornick and Zdonik, 1987] uses a client/server model, in which a centralized object server (the server and its data reside on the same node) manages data for (possibly remote) client processes. Client requests on the server follow a mailbox interprocess communication model. ObServer objects are simple byte vectors.

For efficiency, ObServer clusters related objects together for retrieval into *segments* of varying size; a segment is the unit of transfer for objects between client and server, and from secondary storage to main memory. When a client receives a segment it extracts the individual objects into a hash table, and then discards the segment. A client *finalizes* any updates it has made to objects by communicating just the changes to the server, which modifies the objects in its master copy of the segment correspondingly. This approach reduces communication traffic with the centralized server in an environment where many processes may be accessing the same segment at the server.

ObServer also allows segments to be grouped together for retrieval. Segment groups contain at least one segment, and every segment is a member of at least one group containing only that segment. A segment group is identified with a name supplied by the client when it is created. An object's identifier is assigned by the server when the object is created, and remains the same for all time; the identifier is *not* recycled when the object is deleted. A single client request can be for a segment group or for a particular object. When a client requests an object, it receives the entire segment in which the object resides. In addition, the client can specify a segment group as the context in which it is working; in this case it will receive the other members of the segment group asynchronously, while it is working on the initial segment.

There are a number of other interesting features of ObServer. Objects can be replicated in different segments — the server is responsible for maintaining the coherence of duplicate objects. Also, several types of primitive locks, along with a low-level tailorable transaction model, support many different styles of concurrency control.

#### 2.2.1.2 Mneme

Mneme [Moss, 1990] is intended for tight integration with persistent programming languages through a procedural interface. The primary abstraction of Mneme is the persistent store as a persistent heap: objects persist so long as they are reachable from designated

root objects. Mneme treats an object as slightly more than just a sequence of uninterpreted bytes: it is able to determine the positions of object identifiers contained within objects via call-backs to the client programming language. Thus, Mneme is able to garbage collect the store, rearranging the space of object identifiers and recycling the identifiers of deleted objects. One implication of such rearrangement is that an object's identifier does not necessarily remain the same for all time (although it *is* guaranteed always to be unique); this is not a problem, since a program must initiate all access to objects through one of the root objects, and object identifiers are guaranteed not to change for the duration of a program's session of access to the store.

Mneme groups objects together into files. Each file has a root object from which all the objects in the file can be reached. Access to objects is supported by procedures that retrieve an object (and its containing segment), returning a pointer to the object in memory; clients must explicitly notify Mneme of any updates they make to objects via these pointers for those updates to be made permanent.

Within a file, objects are clustered into segments for retrieval, similarly to ObServer. The unit of transfer between the disk and Mneme's buffers is the *physical* segment (PSEG), which may have arbitrary size (up to some large system-defined limit); a PSEG can contain any number of objects. Objects within a PSEG are further grouped into *logical* segments. A logical segment (LSEG) can contain at most 255 objects. LSEGs represent a partitioning of the object identifier space into efficiently managed units of 255 consecutive numerical object IDs.

#### 2.2.1.3 Texas

The Texas persistent store [Singhal et al., 1992; Wilson and Kakkad, 1992] takes a dramatically different approach to supporting persistence. Instead of representing object references to the application as abstract object IDs, which must be presented to the storage manager via a call interface to gain access to the object, Texas uses a virtual memory page

mapping approach to fault objects, so that an application never uses anything but virtual memory addresses to refer to persistent objects. *Within* the persistent store, object references are represented as symbolic persistent IDs (PIDs), which encode a persistent page number and the offset within that page of their target. Thus, Texas allows persistent address spaces significantly larger than virtual memory, since PIDs can be arbitrarily large. However, at any one time an application can address only as much of the persistent store as can be mapped into virtual memory.

Before an application can refer to an object, Texas must generate a virtual address for it, by reserving an access-protected page of virtual memory for the corresponding persistent page containing the object. Because the offset of the object in the persistent page is known, Texas is able to calculate the corresponding virtual address of the object in the reserved virtual memory page. When the application actually attempts to access the object in its access-protected virtual memory page a page trap occurs, and is handled by Texas, which reads in the persistent page from the store and copies it into the reserved virtual page. As the page is copied, all persistent object references within the page are swizzled to virtual memory pointers, by reserving virtual memory pages for the objects to which they refer (assuming the referenced pages are not already mapped into virtual memory). Swizzling relies on support from the application language to distinguish object references from scalar data. The newly-resident virtual memory page is then unprotected, and execution resumes. As execution proceeds, virtual pages are thus reserved in a “wave-front” just ahead of the most recently faulted and swizzled pages, guaranteeing that the application will only ever see virtual memory addresses.

Texas tracks updates to persistent objects by protecting virtual memory pages that may be updated from write access. On the first write to a protected page an access violation occurs, which Texas handles by unprotecting the page and making a copy of it in a *clean version buffer*. At transaction commit, a modified page will have a clean copy in the clean

version buffer, which is compared with the modified page to generate a log entry recording the differences between the clean and modified versions.

The beauty of Texas is that it requires little or no modification to an existing language to support persistence, so long as object layout information is readily determined for use in swizzling. Wilson and Kakkad [1992] report promising preliminary performance results using Texas to support transparent persistence for C++ — ingeniously, they extract object layout information from the tables generated by the GNU C++ compiler for use by symbolic debuggers.

#### 2.2.1.4 ObjectStore

ObjectStore [Lamb et al., 1991; ObjectStore, 1990] is a commercial product[4] which uses a page mapping scheme similar to Texas, but differs from Texas by storing objects on disk in their virtual memory format. When a persistent page is to be allocated in virtual memory, ObjectStore first tries to assign the page to the same virtual address as when the page was last resident, in which case there is no need to swizzle pointers that refer to the page. If there is a conflict with some other page, then swizzling becomes necessary to adjust pointers that refer to objects on the page. Note that this direct-mapped virtual memory approach means that the store size can grow only as large as is directly addressable by a virtual memory pointer. ObjectStore generates recovery information by logging entire dirty pages.

#### 2.2.1.5 EXODUS

The EXODUS Storage Manager (ESM) [Carey et al., 1986; Carey et al., 1989] is the storage component of the EXODUS extensible database system. A *storage object* in EXODUS is an uninterpreted byte sequence of almost unlimited size that can grow and shrink, with growth and shrinkage possible even in the middle of the object. Every storage object resides in a *file object*, which is a collection of storage objects. ESM also features concurrency

[4] Object Design, Inc., One New England Executive Park, Burlington, MA 01803.

control (with two-phase locking of byte ranges and whole storage objects), recovery (via a combination of shadowing for concurrency among multiple users and logging of updates within transactions), and support for versioning.

A procedural interface supports the creation and deletion of file objects and storage objects within a given file. Files may be scanned: there are procedures to open a file for scanning, to get the object identifier of the next object in the file, and to close the file. Object access is supported by a procedure to get a pointer to a range of bytes within a given storage object; the desired range is read into the client's buffers, and a pointer to the now-resident byte range is returned to the caller. The byte range is "pinned" in memory, until some subsequent call indicates that the bytes are no longer needed, whereupon the byte range is "unpinned" and becomes subject to replacement by the buffer manager. Pinned bytes can be updated in place in the client buffer pool, though EXODUS must be informed of the subrange of bytes that have been modified so that it can generate a log record based on the changes.

ESM is the underlying storage subsystem for the E language described above. Over the years, there have been several different architectures supporting persistence in E based on ESM [Richardson, 1990; Schuh et al., 1990]. Some required object updates to be carried out via the call interface, with updates made to the pinned objects in the client buffer pool, and logged at the same time. To avoid this call overhead White and DeWitt [1992] introduced a new architecture (EPVM 2.0) using *object caching*. Objects are still retrieved into the client buffer pool using the ESM interface, but they are then copied into the virtual memory of the application, while originals in the buffer pool are unpinned. A descriptor for the copied object is entered into a hash table based on the object's identifier. This descriptor contains a pointer to the object in memory along with a pair of values indicating the range of modified bytes in the object. Updates are applied directly to the object copies in the virtual memory space of the application and noted by adjusting the byte range.

At transaction commit, the hash table of cached objects is scanned. For each modified object, ESM is called to pin and update the object in the buffer pool and to generate a log record based on the range of modified bytes. White and DeWitt [1992] describe two versions of this caching scheme: the first copies objects one at a time from the buffer pool into virtual memory as they are accessed by the application; the second copies all of the objects on a given page of the buffer pool when the first object on the page is accessed.

The object caching scheme of EPVM 2.0 also performs some pointer swizzling, in which references to objects that are resident in the cache are converted to direct memory pointers. Each object includes a bit table indicating which of its slots contain direct pointers and which contain unswizzled IDs. Translating an ID means probing a hash table containing pointers for all cached objects, and caching the object if it is not already resident. EPVM 2.0 performs swizzling upon *discovery*: when a location is discovered to contain an unswizzled reference to a persistent object (usually as a result of loading the reference to perform some operation on it) the location is updated with a direct pointer to the object. Unfortunately, this imposes unnecessary overhead on every load, even if the reference is never used to access the object. It may also result in the unnecessary fetching of objects that are never accessed.

White and DeWitt compared their object caching implementation's performance with several other systems, including ObjectStore. Their results indicate that object caching is an attractive architecture for persistent programming languages. For small databases, in which the entire database can fit in main memory, caching objects a page at a time seems best, since there is little extra overhead in copying pages versus objects, with fewer copying operations being needed. However, for larger databases that do not fit in main memory, page caching will copy some objects unnecessarily, which leads to *double paging*: pages are first cached in virtual memory by the object caching mechanism, and then paged out by the virtual memory manager.

The comparison with ObjectStore produced mixed results. Cold database performance (obtained by running benchmarks against a database that starts out entirely on disk at the possibly remote database server) was worse for ObjectStore than for the architectures based on ESM. White and DeWitt suggest that these results indicate the cost of mapping data into a process's address space, including the high overhead of fielding page protection traps from the operating system. For a small database ObjectStore exhibited the best warm performance, since it allows access to objects using direct memory pointers; for the large database its performance was the worst, due to double paging.

#### 2.2.1.6 Other Commercial Ventures

There are a number of other commercial ventures marketing persistent storage systems to support some form of persistence for C++. Most use a preprocessor approach to extract object layout information, for compiler independence. They range from simple persistent object stores to minimal database systems with support for associative access. Unfortunately, their implementation details are proprietary and hence not readily disseminated. They include Objectivity/DB,[5] Versant ODBMS,[6] and Ontos (formerly VBASE [Andrews and Harris, 1987]),[7] among others.

### 2.2.2 Object Oriented Database Systems

While persistent object stores provide a spectrum of functionality for *storage* of objects, object-oriented database systems provide full *database* functionality as well. The following systems all provide a unified object model of the database which consists of objects, classes, and methods. Methods are defined on classes, and constitute the behavior of the instances of a class; they are invoked by sending messages to objects.

[5] Objectivity, Inc., 800 El Camino Real, 4th Floor, Menlo Park, CA 94025.
[6] Versant Object Technology Corporation, 4500 Bohannon Drive, Menlo Park, CA 94025.
[7] Ontologic, Inc., Three Burlington Woods, Burlington, MA 01803.

#### 2.2.2.1 GemStone

GemStone [Copeland and Maier, 1984; Purdy et al., 1987; Bretl et al., 1989] is a commercial database system incorporating the data model of Smalltalk (consisting of objects, messages, and classes) and the programming language OPAL,[8] with syntax much like that of Smalltalk, for data definition and manipulation. To support sharing, the database is partitioned into multiple name spaces; users can only see those objects in their allotted name spaces. Since name spaces can be shared, objects can also be shared. Each user works in his or her own private (i.e., unshared) workspace; any changes to shared objects are actually made to shadow copies in the workspace. Transactions provide for atomic update, with shared objects being replaced by their shadow copies at transaction commit; both optimistic and pessimistic concurrency control are supported. Name spaces also serve as the persistent roots of the database, since GemStone maintains persistence based on reachability; objects are deleted by a garbage collector.

The GemStone database is centralized, residing on a single node running a GemStone server. Access to the database can be either remote or local, from a Smalltalk or OPAL environment running on a workstation, through a C call interface, or using an interactive command-line interface. The Smalltalk-like execution model of GemStone allows execution to take place in the server, so that users have the option of copying objects to their own workspace for manipulation, or of invoking methods on the objects at the server — transaction semantics for these two modes of execution are quite different: server-executed operations run as single transactions, whereas operations in the workspace are bundled together as one transaction terminated by transaction commit.

Unlike some object-oriented databases, classes in GemStone are not extensional: they do not support associative queries over their instances. Rather, associative access is provided

[8] GemStone and OPAL are trademarks of Servio Corporation, 1420 Harbor Bay Parkway, Suite 100, Alameda, CA 94501

over *non-sequenceable collections*, such as bags and sets. Indexes can be maintained for individual collections, to support efficient associative access [Maier and Stein, 1986].

Finally, GemStone supports *schema evolution*: classes can be added, deleted, and modified, subject to certain constraints to ensure that instances can be converted to bring them into line with their modified class; e.g., classes cannot be deleted if they have any instances.

#### 2.2.2.2 ORION

ORION[9] [Banerjee et al., 1987; Kim et al., 1988; Kim et al., 1989; Kim et al., 1990] is a research project which produced a number of prototype object-oriented database systems, addressing database issues such as transaction management, queries and automatic query optimization, version control, change notification, dynamic schema evolution, and distribution.

The ORION data model allows for objects, messages, and classes, similar to Smalltalk. Objects must be explicitly deleted in ORION — there is no garbage collection. Classes are treated as extents, so that high-level queries may be posed against a single target class. Queries are optimized in a manner similar to that of relational databases.

Like GemStone, ORION supports schema evolution through controlled modification of the class hierarchy. However, instances are not converted immediately when their class changes, but only as they are retrieved from the database. This incremental approach has the advantage of allowing schema changes to be processed quickly, since the instances do not need to be updated immediately, although it does mean that the database can contain obsolete data.

[9] A commercial version of ORION is now available from Itasca Systems, Inc., 2850 Metro Drive, Suite 300, Minneapolis, MN 55425.

#### 2.2.2.3 $O_2$

$O_2$ [Bancilhon et al., 1988; Lécluse et al., 1988; Lécluse and Richard, 1989; Deux et al, 1990] is another prototype object-oriented database system. The $O_2$ data model provides objects, classes, methods, and values. Values are instances of types, constructed recursively from atomic types (e.g., integer, float, string) using type constructors (e.g., set, list, and tuple). Objects are instances of classes, encapsulating data and behavior. Every class has a type describing the structure of its instances, and may optionally be given an extent, which is the set of all its instances. Persistence is achieved by *naming* objects or values: a named object or value will persist, along with all objects or values reachable from it. As in GemStone, there is no explicit deletion, and the named objects and values are treated as persistent roots by the garbage collector.

Data manipulation in $O_2$ is supported by extending existing programming languages with an $O_2$ layer to handle object and value declaration, method invocation, and value manipulation, as well as selective iteration over the elements of sets and lists. Two languages have been extended so far: C ($CO_2$) and Basic ($BasicO_2$).

## 2.3 Discussion

The foregoing survey is a heady concoction of different characterizations of persistence, from the *ad hoc* relational extensions to existing languages, to the holistic approach of orthogonally persistent languages. From this, we now distill what we consider to be useful defining characteristics of persistence for a study of performance. Our bias is towards the computationally complete persistent programming languages, which see the persistent store as a stable dynamic allocation heap. There are several reasons for this bias. First, computational completeness permits the full range of possible computations over all types of persistent data. Second, the notion of persistent storage as a stable extension of the dynamic allocation heap allows a uniform and transparent treatment of both transient and persistent data, with persistence being orthogonal to the way in which data is defined,

allocated, and manipulated. This characterization of persistence allows us to identify the fundamental mechanisms that any such persistent system must support, as the basis for our study of the *performance* of persistent systems.

None of the above persistent programming languages have specifically addressed the issue of performance in their implementation. The language designers have been interested in exploring the bounds of expressiveness and functionality in their languages, viewing implementation more as a means of validating the feasibility of the language design than of showing that programs written in the language can be made to run efficiently. Now that persistence is better understood, its performance aspects have become more important. To be widely accepted, persistence must exhibit sufficiently good performance to justify its inclusion as an important feature of any good programming language. Ideally, persistence ought not to require any unusual support from the underlying hardware or operating system, so that persistent programming can be extended to the widest possible community.

### 2.3.1 Essential Persistence Mechanisms

As in traditional database systems, a persistent system must cache frequently-accessed data values in memory for efficient manipulation. Because memory is a relatively scarce resource, it is likely that there will be much more persistent data than can be cached at once. Thus, the persistent system must arrange to make resident just those persistent values needed by the program for execution. Without knowing in advance which data is needed, the system must decide dynamically when to retrieve needed data from secondary storage into memory (although any advance knowledge that *is* available should be used to guide prefetching). Updates can be made in place, in memory, but ultimately must be propagated back to stable storage. Thus, a persistent system must provide mechanisms for the detection and handling of references to persistent data and for the propagation of any modified data back to stable storage. Efficient implementation of these mechanisms is

the key to implementing a high-performance persistent programming language, since they provide the fundamental database functionality of retrieval and update.

#### 2.3.1.1 The Read Barrier: Object Faulting

The first mechanism mediates retrieval of data from stable storage into memory for manipulation by the program. Any operation that directly accesses a data value whose residency is in doubt must first check that the value is available in memory. Such residency checks constitute a *read barrier* to any operation that accesses persistent data: before the operation can read (or write) the data it must first make sure it is resident.

#### 2.3.1.2 The Write Barrier: Detecting and Logging Updates

The second key persistence mechanism ensures that updates to cached (i.e., volatile) persistent data are reflected in the database. Making updates permanent means propagating them to stable storage. Thus, every operation that modifies persistent data requires some immediate or subsequent action to commit the modification to disk. A persistent system might write the modifications straight through to disk on every update, but this is likely to be very expensive if updates are frequent or otherwise incur very little overhead. Instead, updating the stable store is typically held over to some later time, usually at the instigation of the programmer through the invocation of a *transaction commit* or a *checkpoint* primitive operation. Either way, any operation that modifies persistent data in memory must arrange for the update to be made permanent, directly by writing the modified data to disk, or indirectly by asking the system to remember that the update was made. Recording updates in this way constitutes a *write barrier* that must be imposed on every operation that modifies persistent data; writes require additional overhead to record the update.

Implementation of the write barrier is intimately tied up in assurances of database *resilience* in the face of system failures. When a user commits to a set of updates they want to be certain that all of the updates are permanently reflected in the database. Unfortunately,

a crash results in the loss of the volatile part of the database (the cached persistent data), including all updates to persistent data that have not yet been propagated to stable storage. Recovering from such a failure involves restoring the database to some consistent state from which processing can resume. Any implementation of the write barrier must take into account the mechanism by which programmers specify database consistency and the information that must be generated for recovery.

### 2.3.2 Fine-Grained Persistence and the Performance Problem

Conventional non-persistent programming languages, such as those in the Algol family (including Pascal, C, Modula-2, and their object-oriented cousins C++, Modula-3 and even Smalltalk) have a fine-grained view of data — they provide fundamental data types and operations that correspond very closely to the ubiquitous primitive types and operations supported by all machines based on the von Neumann model of computation. Such a close correspondence means that many operations supported in the language can be implemented directly with as little as one instruction of the target machine.

Orthogonality mandates that even data values as fine-grained as a single byte (the smallest value typically addressable on current machines) may persist independently. Clearly, this situation is significantly different from that of relational database systems, where the unit of persistence is the record, usually consisting of many tens, if not hundreds, of bytes. Where a relational system can spend hundreds or thousands of instructions implementing relational operators, an Algol-like persistent programming language must take an approach to persistence that does not swamp otherwise low-overhead and frequently executed operations.

Thus, implementations of the read and write barrier for such languages must be sufficiently lightweight as to represent only marginal overhead to frequently executed operations on fine-grained persistent data.

# CHAPTER 3

# ARCHITECTURE

This chapter describes an architecture for persistence and its rationale, and constitutes the framework for our approach to implementing persistent programming languages. The architecture itself is not unique and bears a close resemblance to the object caching architecture of White and DeWitt [1992]; in many respects it reflects the general consensus of the technical community as to how to structure persistent systems. However, the architecture's realization is unique in that it allows the language implementation maximum control over all objects being manipulated by a program, without having to pass through a restrictive interface to the underlying storage manager. Much of this flexibility comes as a result of using the Mneme persistent object store (described in Chapter 2) to manage the storage and retrieval of objects from disk, since Mneme allows direct access to the objects in its buffers via virtual memory pointers.

The architecture is illustrated in Figure 3.1. As mentioned earlier, Mneme groups objects into segments for efficient transfer to and from secondary storage, buffering the segments in main memory as necessary. Objects are copied on demand from the client buffer pool into the virtual memory address space of the program. This copying includes any translation needed to convert the objects into a form acceptable to the program. In particular, since Mneme uses object identifiers to refer to objects while the program uses virtual memory pointers, object references may be swizzled to direct memory pointers for manipulation by the program. The architecture permits standard programming language techniques for memory management, including those of garbage collection, to manage the objects resident in the program's virtual address space.

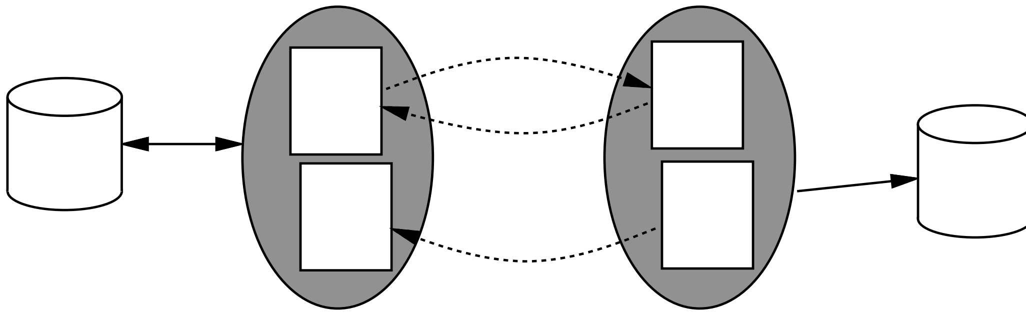


**Figure 3.1** System architecture

## 3.1 Object Faulting

Object faulting requires some mechanism to distinguish between references to resident and non-resident objects. These mechanisms may be loosely divided into two categories, depending on the strategy they adopt. For the purposes of this discussion we view the persistent heap as a *directed graph* where the objects are the *nodes* and the references between the objects are the *edges*.

*Edge marking* schemes take the approach of tagging the references between the objects. If tagged as *swizzled*, then a reference is a direct pointer to the corresponding object in memory; if *non-swizzled* then the reference contains an ID. An apparent disadvantage of edge marking is that IDs can be fetched from the pointer fields of objects, passed around, and stored, without accessing the target object. When the target object finally is accessed the origin of the reference may no longer be known.

*Node marking* schemes require that all object references in resident objects be converted to pointers. ObjectStore and Texas achieve this by reserving (although not necessarily fetching) virtual pages for the objects referred to by the pointers and by protecting those pages to trap all access. Another approach, similar to LOOM's leaf objects, is to have small proxy objects (we call them *fault blocks*) stand in for non-resident objects, as illustrated in Figure 3.2(a). A fault block contains the ID of the target object and is distinguishable

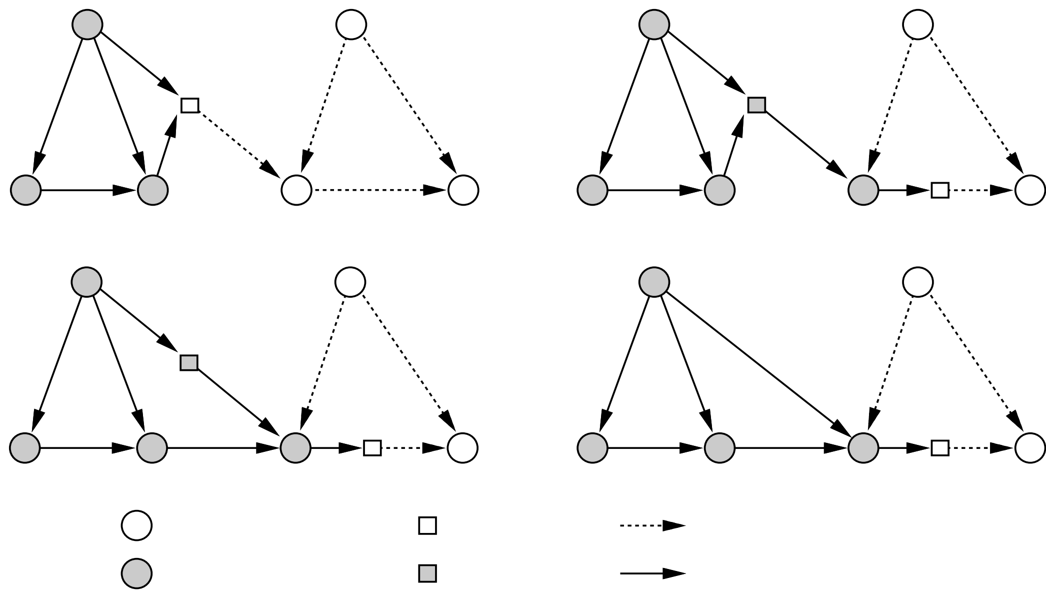


**Figure 3.2** Node marking

from an ordinary object. Whenever a reference is followed, if it refers to a fault block, then the target object is made resident (copied and swizzled as necessary). The fault block is changed to point to the now-resident object (see Figure 3.2(b)); we call the updated fault block an *indirect block*. If a reference to be followed refers to an indirect block then the target object can be located at the cost of an indirection. The indirection can be eliminated in several ways. First, if the source of the faulting reference is known then it can be updated to point directly to the resident object (Figure 3.2(c)). Remaining indirections can be removed directly via scanning, by the garbage collector, or when discovered by the application (Figure 3.2(d)).

References to tagged IDs and fault blocks may be detected via explicit checks upon pointer dereference. Alternatively, fault blocks can be allocated in protected virtual memory pages, so that dereferencing a pointer to a fault block is trapped, and handled by making the target object available. Another approach is to exploit the indirection implicit in the method invocation schemes of object-oriented programming languages, folding residency checks into the overhead of method invocation.

## 3.2 Swizzling

When an object is made resident its pointer fields are swizzled according to the mechanism being employed for fault detection. All fields referring to objects that are already resident can be converted to point directly to those objects — Mneme supports this mapping efficiently with a hash table. Otherwise, for edge marking we convert the reference to a tagged ID; for node marking, the reference is converted to point to a fault block for the non-resident object (a fault block is allocated if one does not yet exist for the target object).

The architecture leaves open the possibility of copying and swizzling any number of objects at one time from the Mneme buffer pool into memory. For this study we consider the swizzling granularities naturally inherent in this architecture: individual objects, logical segments, and physical segments. Swizzling just one object at a time has the advantage of copying and swizzling only those objects needed immediately by the program for it to continue execution. This will serve to minimize object fault latencies (including swizzling), as well as memory consumption.

Swizzling a logical or physical segment at a time can take advantage of any clustering present in the physical layout of objects in the database. Since all the objects in a segment are mapped before they are swizzled, any *intra*-segment references can be converted to direct pointers. If the static clustering is a good approximation to the dynamic locality of access by the program then the speed of program execution will improve since fewer object faults will occur.

## 3.3 Resilience

Our notion of recovery is dictated by certain assumptions about the behavior of persistent programs. We assume that a program will invoke a checkpoint operation at certain points throughout its execution to make permanent all modifications it has made to persistent objects. In the event of a system crash the database recovery mechanism must restore the

state of the database to that of the most recent checkpoint. Moreover, we assume that checkpoint latencies should be minimized so as to have the smallest possible impact on the running time of the program. This last point is important in interactive environments where checkpoints may noticeably delay response times.

A checkpoint operation consists of copying and unswizzling modified and newly-created objects (or modified subranges of objects) back to the client buffer pool and generating log records describing the range and values of the modified regions of the objects. Log records are generated only if there are differences between an object and its original in the client buffer pool. Persistence in our system is based on *reachability*, so the unswizzling operation may encounter pointers to objects newly created since the last checkpoint. Each such object is assigned a persistent identifier and unswizzled in turn, perhaps dragging further newly-created objects into the database, and a log record describing the new object is generated.

The precise format of the log records is not relevant to this study, since we are interested only in the mechanisms used to detect and log updates. However, we note that each log record is tagged by the persistent identifier of the modified object and encodes a range of modified bytes. Recovery involves applying these log records to the objects to which they pertain, in the order in which they occur in the log. Although alternative log-record formats might yield a slightly more compact log, or allow more efficient recovery, our log is *minimal* in the sense that it records just enough information to reconstruct each modified object.

## 3.4 Extensions to the Basic Architecture

As described so far, the architecture supports single-user access with recovery. We now argue that the architecture can be extended to provided additional functionality such as buffer management and concurrency control.

### 3.4.1 Buffer Management

To integrate buffer management with the recovery model, we guarantee that a modified segment is flushed to the database only after the log records associated with those modifications have been written. Outside of that constraint, the buffer manager is free to use any appropriate buffer replacement policy. Management of swizzled objects in the application's virtual memory must rely on techniques similar to garbage collection to determine which objects are subject to replacement [Hosking, 1991]; this is compatible with the recovery model, since modified objects that have been selected for replacement will be unswizzled and logged in a checkpoint fashion.

### 3.4.2 Concurrency Control

The recovery model is indifferent to concurrency, which can be introduced to the architecture in two ways. First, separate applications can share the same database, arbitrated by a server. Locking is managed by the server and the application's view of recovery is unchanged, modulo some additional information required in a log entry to identify its owner. Second, a single application may be multi-threaded. Additional locks must be managed within the application if data is shared among threads. Again, the recovery model remains essentially unchanged, modulo some additional log entry information to identify the owner of the entry.

The recovery model and support for concurrency provide the foundation for any transaction model. The incorporation of transaction models in persistent programming languages remains an open topic of research. We are not directly concerned with that issue here, and merely remark that our recovery model is based on that of the *database cache* [Elhardt and Bayer, 1984], for which several transaction models exist. The database cache was designed for fast transaction commit and rapid recovery after a crash. Modifications are always applied to copies of original database pages in main memory (the *cache*) so that transaction abort merely requires deletion of the copies. Transactions commit by flushing

dirty copies to the *safe*: a log of updated pages on stable storage. Once a dirty copy has been flushed to the safe, it becomes a changed original. Dirty copies are never flushed to the permanent database. Thus the permanent database contains only the effects of committed transactions. Similarly, the buffer manager may only select unpinned originals for replacement in the cache, flushing any changed originals to the database as necessary. Recovery involves reconstructing the cache from the safe. To keep the size of the safe manageable, it is periodically cleaned by removing log entries that are no longer necessary for recovery.

Elhardt and Bayer require locking and logging at the granularity of a page. Moss et al. [1987] extend the database cache algorithms to allow locking and logging at a finer granularity. The goal of their extension is to increase concurrency and, ultimately, performance. These extensions are certainly possible within our framework. Indeed, Moss et al. [1987] propose an investigation of the effects of different lock and log granularities on system performance, which we partially address here. Our system is not an exact implementation of the database cache, but we use a similar buffer management protocol and our checkpoints have semantics similar to the database cache transaction commit. Here we go beyond a comparison of logging granularities and investigate different methods for noting modifications.

In more general terms, the database cache and the approach to logging described here are nothing more than extended versions of the *write ahead log* used in traditional database systems [Gray, 1978; Verhofstad, 1978; Haerder and Reuter, 1983]; the extensions are for non-traditional database applications in which memory-residence is important. Thus, we could just as easily consider other approaches to concurrency control and recovery that are compatible with write-ahead logging, while retaining our own mechanisms for detecting and recording updates.

# CHAPTER 4

# IMPLEMENTATION

Chapter 3 described a basic architecture for persistence, and the mechanisms required for any implementation of a persistent programming language built on that architecture. This chapter delves into the specifics of the persistent implementation of Smalltalk used in the performance evaluation experiments of Chapter 5 to compare alternative implementations of the persistence mechanisms. Section 4.1 summarizes the features of the non-persistent Smalltalk implementation used as the starting point for the persistent implementation. Section 4.2 then goes on to describe how we extend that implementation for persistence, giving details of the alternative implementations of object faulting, using both node and edge marking, and mechanisms to detect updates for resilience.

## 4.1 Smalltalk

The Smalltalk implementation is based on the original specification of the Smalltalk-80[1] programming language and environment [Goldberg and Robson, 1983], consisting of two components: a *virtual machine*, which executes the bytecode instruction set to which Smalltalk source code is compiled, and a *virtual image*, which implements (in Smalltalk) the Smalltalk-80 programming environment. Several aspects of the implementation of the virtual machine are novel and are described below. We begin with a brief summary of the salient features of Smalltalk's object-oriented execution paradigm, to enlighten the uninitiated reader, before revealing the details of the implementation of Persistent Smalltalk.

[1] Smalltalk-80 is a trademark of ParcPlace Systems.

### 4.1.1 Object-Oriented Method Invocation

A Smalltalk *object* (see Figure 4.1) is an encapsulation of some private state and a set of operations called its *interface*. The private state consists of a number of data fields, called *instance variables*, directly accessible only from the code implementing the object's operations. Every object is an *instance* of some *class* object, which implements the common behavior of all its instances; a class object is itself an instance of its *metaclass*. Classes are arranged in a hierarchy, such that a *subclass* will *inherit* instance behavior from its *superclass*. Thus, an instance of the subclass will behave as an instance of the superclass, except where the subclass overrides or extends that behavior.

Computation in Smalltalk proceeds through the sending of *messages* to objects. A message consists of a *message selector* (e.g., at:put:) and a number of arguments, and represents a request to an object to carry out one of its operations. The effect of sending a message is to invoke one of the *methods* of the object receiving the message (the *receiver*). Invoking a method may be thought of as a procedure call, with the receiver being the first argument to the procedure, preceding the arguments specified in the message. The particular method to execute is determined dynamically, using the message selector and the class of the receiver. Each class object contains a reference to a *method dictionary*, associating message selectors with *compiled methods*. A compiled method consists of the virtual machine bytecode instructions that implement the method, along with a *literal frame*, containing the shared variables,[2] constants, and message selectors referred to by the method's bytecodes.

#### 4.1.1.1 Method Lookup

Determining which method to execute in response to the sending of a message proceeds as follows. If the method dictionary of the receiver's class contains the message selector,

[2] A shared variable is an object that encapsulates a reference to another object. If the contents of the variable are changed, then the change is visible to all other compiled methods holding references to that shared variable.

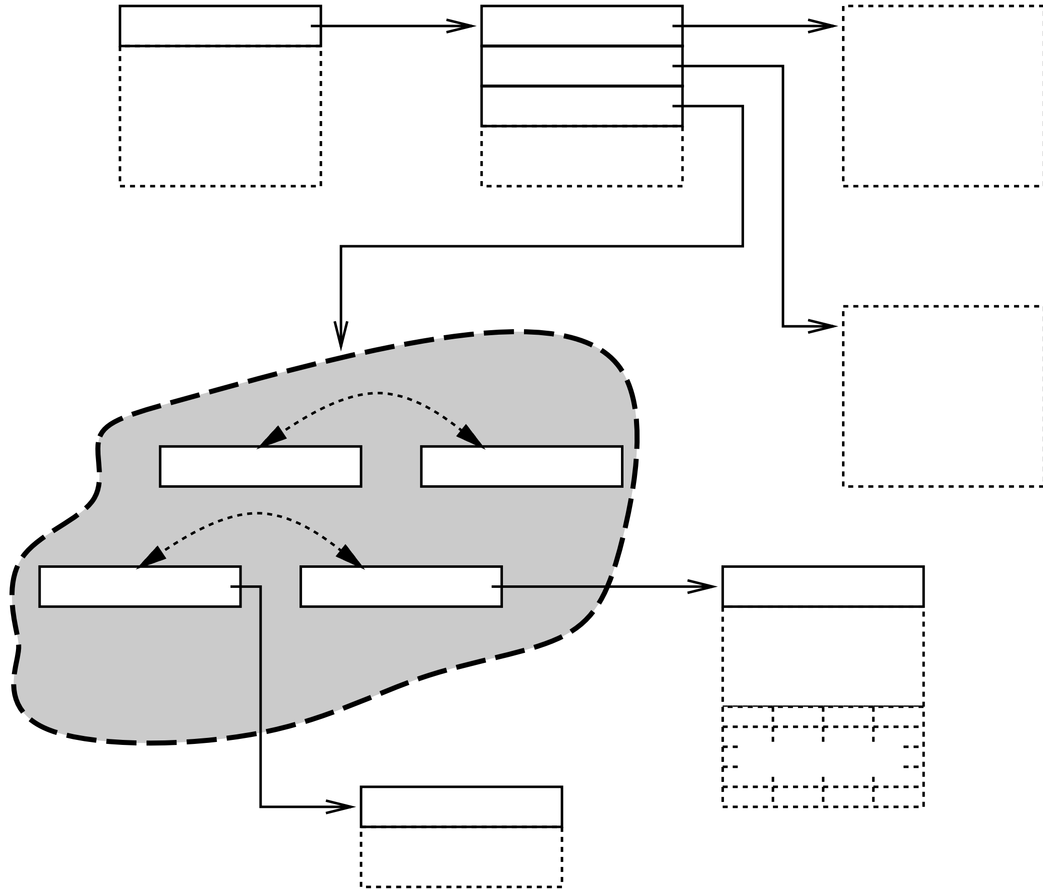


**Figure 4.1** Objects, classes, inheritance, and object-oriented method invocation

then its associated method is invoked. Otherwise, the search continues in the superclass of the object, and so on, up the class hierarchy. If there is no matching selector in any of the method dictionaries in the hierarchy then a run-time error occurs.

As described so far, the method lookup process would be very expensive, especially since a given message may be implemented by a method in a class that is high up in the superclass hierarchy, far removed from the class of the receiver. A *method lookup cache* can reduce this lookup cost significantly. A valid entry in the cache contains object references for a selector, a class, and a compiled method. Message sends first consult the method lookup cache, by hashing the object references of the selector and the receiver's class to

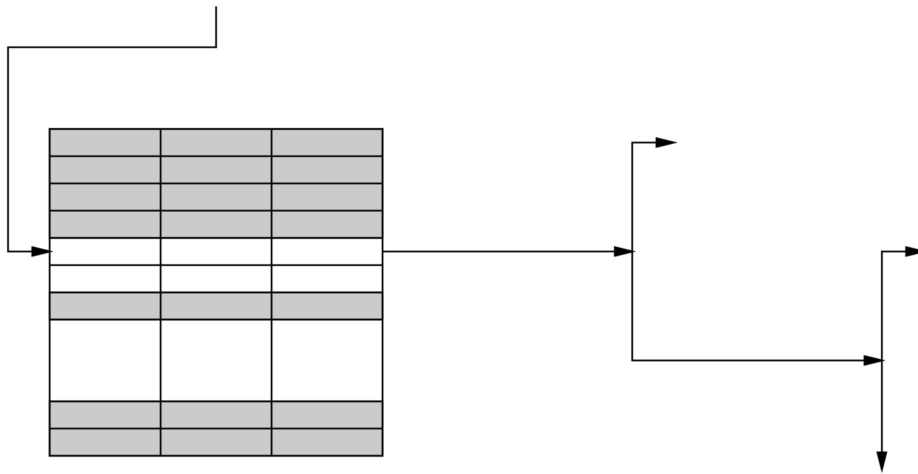


**Figure 4.2** Method lookup. The method cache is probed by hashing the message selector and receiver's class. A valid entry occurs either in its natural hash slot or the immediately following slot.

index an entry in the cache. If the selector and class of the cache entry match those of the message, then the cached compiled method is invoked directly. Otherwise, the full method lookup locates the compiled method and loads the cache entry with the selector, class and method, before invoking the method. Figure 4.2 illustrates the method lookup process. Our implementation uses a linearly-hashed method lookup cache; a valid entry may occur in its natural hash slot, or in the immediate successor. The cache replacement policy is to promote the second entry to the first slot, if that is its natural hash slot, and allocate the new entry into the second slot; otherwise, the new entry is allocated in the first slot.

### 4.1.2 The Virtual Machine

The virtual machine (VM) implements the bytecode instruction set to which Smalltalk source code is compiled, along with certain *primitive methods* whose functionality is built directly into the VM. A primitive method is invoked in exactly the same way as a method expressed as a sequence of Smalltalk expressions, but its implementation is not a compiled

method. Rather, the VM performs the primitive directly, without the need for a separate Smalltalk activation record. Primitive methods typically provide low-level access to the underlying hardware and operating system on which the VM is implemented. For example, low-level floating point and integer arithmetic, indexed access to the fields of objects, and object allocation, are all supported by primitive methods.

We retain the standard VM bytecode instruction set defined by Goldberg and Robson [1983]. It is subdivided by functionality as follows:

- *stack bytecodes* move object references between the evaluation stack of the current activation and:
    1. the named instance variables of the receiver for that activation
    2. the temporary variables local to that activation
    3. the shared variables of the literal frame of the active method
- *jump bytecodes* change the instruction pointer of the current activation
- *send bytecodes* invoke compiled or primitive methods
- *return bytecodes* terminate execution of the current activation, and return control to the calling activation

Although we retain the standard bytecode instruction set of Goldberg and Robson [1983], the VM implementation differs markedly from their description. Because the VM is implemented entirely in GNU C (a dialect of ANSI C with non-standard extensions), it ports straightforwardly to other machines on which GNU C is available. Notable features of the implementation include the use of direct 32-bit object pointers instead of 16-bit indexes into an object table, an improved scheme for managing Smalltalk stack frames (i.e., activation records) [Moss, 1987], generation scavenging garbage collection [Ungar, 1984; Ungar, 1987], and dynamic translation of compiled methods from bytecodes to *threaded code* [Bell, 1973].

#### 4.1.2.1 Object Pointers

Each object in the Smalltalk system has an associated unique identifier called its *object pointer*. For an object allocated in memory, its object pointer is simply the address of the

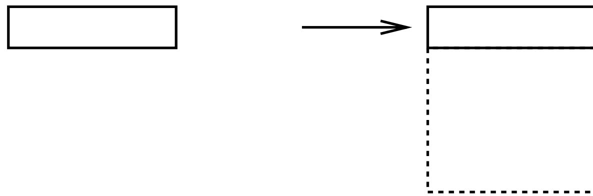


**Figure 4.3** Object pointers. An object allocated in memory is referred to directly by its address.

first word of the object, which here happens to contain the object pointer for its class (see Figure 4.3).

Memory objects are allocated on 32-bit word boundaries, so the two low bits of a memory object's address are guaranteed always to be zero for standard byte-addressed architectures. These "spare" bits in an object pointer allow a space-efficient *immediate* representation of scalar values such as integers and characters: the value is stored directly in the object pointer, rather than being allocated in memory. For efficiency, the VM must easily distinguish memory object pointers from immediate values. The low bit of the object pointer serves this purpose: a 0 indicates an address, while a 1 indicates some immediate value. Similarly, the VM efficiently discriminates immediate integer values using the second low-order bit: a 1 indicates a 30-bit signed immediate integer (SmallInteger), while a 0 indicates some other immediate value (true, false, nil, or an instance of Character). We extend the tag to differentiate the latter. These encodings are summarized in Table 4.1.

#### 4.1.2.2 Threaded Code

Rather than interpreting the bytecodes of compiled methods directly, the VM dynamically translates each compiled method into *threaded code* [Bell, 1973] the first time the method is invoked. Threaded code consists of a sequence of branch targets to each of the machine-code fragments implementing the bytecodes of the compiled method. Instead of expensively

**Table 4.1** Tagging object pointers. A 0 in the low bit indicates a memory pointer; a 1 in the second low-order bit indicates a SmallInteger; the remaining encodings extend the tag.

| Object pointer contents | Bit encoding |
|---|---|
| memory object address | ...00 |
| *unassigned* | ...0001 |
| *unassigned* | ...0101 |
| instances of Character | ...1001 |
| nil | 0...0001101 |
| false | 0...0011101 |
| true | 0...0111101 |
| *disallowed* | ...10 |
| instances of SmallInteger | ...11 |

decoding each bytecode on every VM instruction cycle, bytecode dispatch is thus reduced to indexing from the VM's program counter (maintained in a hardware register) to load the address of the next bytecode fragment, and performing an indirect branch to that target. On a modern RISC architecture this consists of just three instructions: one to load the branch target, one to increment the (simulated) program counter, and one for the indirect branch.

Generating threaded code is cheap; translation consists simply of indexing a static array mapping bytecodes to their machine-code targets. The up-front cost of translation is mitigated by faster cycle times for bytecode interpretation, resulting in faster execution of compiled methods. Subsequent repeat invocations of a given method incur no further translation overhead. Since it is common for a method in the Smalltalk system to be invoked more than once (see Krasner [1983]), the cost of translation is quickly repaid with faster bytecode execution. A similar technique is used to improve performance in the commercial ParcPlace implementation of Smalltalk-80, except that bytecoded methods are dynamically translated to native code on first invocation for direct execution on the host machine, instead of threaded code for interpretation.

### 4.1.3 The Virtual Image

The virtual image is derived from Xerox PARC's Smalltalk-80 image, version 2.1, of April 1, 1983, with minor modifications. It implements (in Smalltalk) all the functionality of a Smalltalk development environment, including editors, browsers, a debugger, the bytecode compiler, class libraries, etc. — all are first-class objects in the Smalltalk sense. Bootstrapping a (non-persistent) Smalltalk environment entails loading the entire virtual image into memory for execution by the VM.

## 4.2 Persistent Smalltalk

The persistent implementation of Smalltalk places the virtual image in the Mneme persistent object store, and the Smalltalk environment is bootstrapped by loading just that subset of the objects in the image sufficient to resume execution by the VM. We retain the original bytecode instruction set and make only minor modifications to the virtual image. Rather, our efforts are focussed on the VM, which is carefully augmented to make persistent objects resident as they are needed by the executing image.

### 4.2.1 The Read Barrier: Object Faulting

The discussion of Smalltalk method invocation in Section 4.1.1 illustrates just how many objects the VM must access for computation to proceed. Object faulting means that objects become resident only when needed by the executing Smalltalk virtual image, as a result of faults triggered by residency checks in the VM. To avoid burdening bytecode instruction execution with unnecessary residency checks, it is essential that the persistent VM perform residency checks only where absolutely necessary to ensure the availability of objects critical to the forward progress of computation. We are able to keep bytecode dispatch and execution completely free of residency checks by imposing *residency constraints* on these critical objects, as follows.

#### 4.2.1.1 Residency Constraints

Residency constraints represent a contract between the VM and the run-time system to ensure that certain references are always represented as direct pointers to their (resident) target objects. Residency constraints may be expressed in terms of a *co-residency* relation:

> Object $b$ is said to be *co-resident* with object $a$, written $a \rightarrow b$, if whenever $a$ is resident, $b$ must also be resident.

Note that co-residency is not symmetric: $a \rightarrow b \not\Rightarrow b \rightarrow a$. For swizzling purposes, if $a$ contains a reference to $b$, then that reference can always be represented as a direct pointer to $b$.

The residency constraints are derived by considering the objects whose residency is critical to execution of each category of bytecode:

- **Stack Bytecodes:** *The stack bytecodes directly manipulate the current activation's stack, named instance variables of the active receiver, temporary variables local to the current activation, and the shared variable literals of the active method.*

  A method activation consists of the active stack frame (i.e., the activation record) holding the state of the activation, the active receiver, and the active compiled method. Processes and their stack frames are first-class objects in the Smalltalk system, and so may be persistent. An active byte-compiled method must be co-resident with its stack frame, so that execution of the method can proceed. Threaded code for the method must also be generated when the stack frame is made resident.[3]

  Similarly, the receiver must be co-resident with its stack frame and active compiled method, so that the method's stack bytecodes can refer directly to the receiver. The stack and temporary variables are allocated in the stack frame, so they are always directly accessible.

[3]Threaded code is not made persistent, but is regenerated in each run, since it consists of the absolute virtual address branch targets for each of its method's bytecodes, which might change if the VM were to change. If threaded code were persistent, then changing the VM (such that the target addresses changed) would require all persistent threaded code to be regenerated. In contrast, the bytecodes stored in compiled methods are independent of the VM implementation.

Lastly, literals must be co-resident with their byte-compiled method, so that they too are directly accessible by the active method's bytecodes.

- **Jump Bytecodes:** *The jump bytecodes directly manipulate only the active stack frame and the instruction pointer.*

  This invariant imposes no additional residency constraints — the stack frame and its (threaded) compiled method are already guaranteed to be resident.

- **Return Bytecodes:** *The return bytecodes directly manipulate the active stack frame and the (calling) activation to which control is being returned.*

  To avoid a residency check on the calling activation, all activations in a process stack are made resident together. The constraint that a calling activation must be co-resident with the activation it calls achieves this transitively — when an active stack frame is made resident (usually because its process is being resumed), its caller, its caller's caller, and so on up the process stack, are made resident along with it.

- **Send Bytecodes:** *The send bytecodes directly access the receiver (to obtain its class) and the literal frame (for sends to literal selectors) when probing the method lookup cache.*

  Because computation is driven by the sending of messages, most objects need only to be made resident when first sent a message, since the send bytecodes must load the receiver's class for method lookup. By requiring that a class object be co-resident with any one of its instances, every object's class field can always be swizzled to contain a direct pointer. Thus, the send bytecodes need not perform a residency check on the receiver's class when probing the method lookup cache.

  Since literal selectors are always made resident along with their literal frame, there is no need for a residency check on the selector in the send bytecodes, so long as the selector is a literal. However, certain of the send bytecodes directly invoke *special* selectors recognized by the compiler, allowing the compiler to save space by omitting these selectors from the literal frames of compiled methods that use them. Instead, the special selectors are stored in a global variable known to the virtual machine,

which is indexed by the special send bytecodes to load the corresponding selector. By requiring these special selectors always to be resident, and because method literal selectors are always resident, there is no need for residency checks on the message selector in any send bytecode.

The receiver must still be resident before a send can proceed. Depending on the representation of references to non-persistent objects (edge marking versus node marking), we achieve this in one of two ways, as described below. Both approaches rely on the fact that probing the method lookup cache can go ahead even if the "class pointer" used in the probe is not a real pointer to memory, since the probe is based solely on the value of the pointer, not the object it refers to. So long as an appropriate "class pointer" value can be derived from *any* given reference, without resorting to special case code, cache probes can be implemented in exactly the same way as for non-persistent Smalltalk.

Only the full method lookup code need check for the exceptional case of a message being sent to a non-resident object. In this situation, the lookup yields a cache entry initialized to pass control to a primitive method whose only purpose is to fault the target object into memory, before forwarding the original message to the now-resident object. In effect, references to non-resident objects may be thought of as responding to all messages by executing this special primitive. Thus, for normal case execution, when the cache contains an entry matching the message selector and the receiver's class, message sends proceed without extra overhead.

**Figure 4.4** Edge marking. A non-resident object is referred to by a tagged immediate value containing its persistent identifier.

The co-residency constraints are summarized as follows:

- For any given object $o$, the class of $o$, $\mathcal{C}(o)$, is always co-resident with $o$:

$$o \rightarrow \mathcal{C}(o)$$

- For any stack frame object $s$, the class of $s$, $\mathcal{C}(s)$, the receiver for the activation represented by $s$, $\mathcal{R}(s)$, the stack frame representing the sender activation that activated $s$, $\mathcal{S}(s)$, and the compiled method for $s$'s activation, $\mathcal{M}(s)$, are all co-resident with $s$:

$$s \rightarrow \mathcal{C}(s), s \rightarrow \mathcal{R}(s), s \rightarrow \mathcal{S}(s), s \rightarrow \mathcal{M}(s)$$

- For any compiled method $m$, $m$'s literals are all co-resident with $m$:

$$\forall l \in \text{literals}(m), m \rightarrow l$$

#### 4.2.1.2 Edge Marking

For edge marking, references to non-resident objects are represented as tagged immediate persistent object identifiers (PIDs), using the hitherto unassigned tag 0001 (see Figure 4.4).[4] Immediate values map to their class based on the tag, rather than by dereferencing the object pointer to obtain the class. Tagged PIDs map to the null class pointer. The full method lookup must first check if the receiver's class is null; if it is, then the cache is primed with an entry redirecting invocations of the given message on non-resident objects to the faulting primitive. Subsequent invocations of that message on non-resident objects will then hit that entry in the cache and proceed directly to the object faulting code without the overhead of an explicit residency check.

[4] Mneme PIDs are only 28 bits, leaving room for the 4-bit tag on a 32-bit machine.

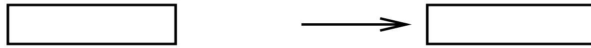


**Figure 4.5** Node marking: fault blocks. A non-resident object is referred to by a pointer to its tagged persistent identifier.

#### 4.2.1.3 Node Marking

A similar trick is used to obtain check-free message sends for node marking. Fault blocks (see Figure 4.5) contain a tagged immediate PID in place of the class pointer normally stored in the header of a resident object. Similarly, indirect blocks (Figure 4.6) contain a forwarding pointer with the two low-order tag bits set, so that they too are distinguishable from resident objects. When sending a message to a resident object, the VM probes the method cache using the class pointer stored in the object's header. If the message is sent to an object represented by a fault or indirect block, then the probe will instead use the PID or tagged forwarding pointer stored in the block. The full method lookup must first check if the initial class for lookup is tagged (either as a fault or indirect block); if it is, then the cache is primed with a faulting entry valid for that message and fault/indirect block combination. Thus, subsequent invocations of the given message on that particular fault or indirect block will hit the faulting entry and proceed directly to the object faulting code without any residency checks.

The implementation of the page protection variation for fault blocks achieves the same effect, but makes sure that the send bytecodes never see a reference to a non-resident receiver — loading the "class" of an access-protected fault or indirect block will cause a trap. The trap handler unprotects the pages containing fault and indirect blocks, overwrites the offending fault block with an indirect block, and arranges for the load instruction that caused the fault to be restarted with a direct pointer to the resident object. The handler reprotects the fault and indirect block pages before returning execution to the VM.

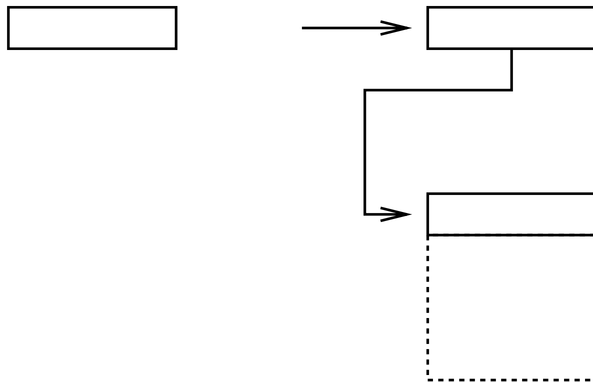


**Figure 4.6** Node marking: indirect blocks. An indirect reference to a resident persistent object is a pointer to a tagged forwarding pointer.

#### 4.2.1.4 Object Faulting Summary

By preloading objects critical to the forward progress of computation, bytecodes execute without residency checks, except on the receiver in message sends, and then only in the case of full method lookup when a method cache miss occurs. The persistent VM must still check the residency of objects it accesses directly whose residency is not guaranteed by the constraints. For example, full method lookup must perform residency checks as it ascends the class hierarchy, to ensure that the superclasses and their method dictionaries are resident. Similarly, primitive methods (implemented directly in the VM) must perform residency checks on any objects (excluding the receiver) they access directly.

### 4.2.2 Swizzling

The residency constraints described in Section 4.2.1.1 require that constrained references be swizzled to contain direct pointers to their resident targets at the time their source object first becomes resident as the result of a fault; if the targets are not yet resident then they also become resident as a result of the fault. In contrast, an unconstrained reference is typically represented in its unswizzled form (i.e., tagged PID or a pointer to a fault/indirect block),

regardless of the residency of its target. Nevertheless, when the target finally does become resident as the result of a subsequent object fault, there is no reason why the unconstrained reference should not be swizzled to point directly to it. Indeed, there may be a significant performance advantage to swizzling even unconstrained references if they are traversed frequently during execution.

One approach to swizzling unconstrained references is to avoid creating the indirection inherent in an unswizzled reference to a resident object in the first place, by *selectively* swizzling the fields of an object when it is first faulted: if a field contains a reference to an object that is already resident then it is swizzled to a direct pointer. Unfortunately, this can only ensure a direct pointer representation for fields containing references to objects that are already resident at the time of the fault, or become resident as a result of the fault. The remaining fields become eligible for swizzling only when their targets are made resident by subsequent object faults. They can be swizzled only through active elimination of indirect references.

While the garbage collector is one place where indirection can be eliminated, a more aggressive approach may be beneficial: eliminate all sources of unswizzled references when their target objects are first made resident. To eliminate indirect references in transient objects the system can scan all objects in the transient space (including active stack frames), swizzling any indirect references it finds. For persistent space a more focussed approach is needed, since persistent space is typically much larger than transient space, making scanning prohibitively expensive. The VM and run-time system maintain a *remembered set* [Ungar, 1984; Ungar, 1987] for each logical segment of persistent objects, recording all memory locations in persistent space that contain unswizzled references for objects in the segment. When an object is made resident, the locations recorded in its segment's remembered set are examined and swizzled. Thus, every indirect reference to an object is converted to a direct pointer when that object is first made resident, either through scanning of transient objects or using remembered sets.

Such aggressive swizzling would appear to be particularly important for the virtual memory page trap implementation of node marking, because it will preempt further unnecessary and expensive page traps on unswizzled references to objects that are already resident.

### 4.2.3 The Write Barrier: Detecting and Logging Updates

The lightweight mechanisms used for detecting updates in this study are inspired by solutions to the *write barrier* problem in garbage collection: the act of storing a pointer in an object is noted in order to minimize the number of pointer locations examined during garbage collection [Hosking et al., 1992]. Similarly, efficient checkpointing requires keeping track of all updates to objects, to minimize the number of locations unswizzled when generating the log (recall that a log record is generated only if there are differences between the new version of an object and the original in the client buffer pool).

This study examines several implementations of the write barrier, including three approaches previously used in garbage collection and here now applied for the first time to the problem of detecting and logging updates for resilience. Note that since the log consists of difference information obtained by comparing old and new versions of objects, all schemes end up generating exactly the same log information. The schemes vary mostly in the granularity of the update information they record, and hence in the amount of unswizzling and comparison required to generate the log.

#### 4.2.3.1 Object-Based Schemes

The first two schemes record updates at the logical level of objects. One approach is to mark updated objects by setting a bit in the header of the object when it is modified. To perform a checkpoint all cached objects must be scanned to find those objects that are marked as updated. A marked object must be unswizzled and compared to its original in the buffer pool to determine any differences that must be logged. The drawback of this

approach is the additional checkpoint overhead required to scan the cached objects to find those that are marked.

To avoid scanning, the second scheme uses a *remembered set* [Ungar, 1984] to record modified persistent objects. A checkpoint need only process the entries in the remembered set to locate the objects that must be unswizzled and possibly logged. The remembered set is implemented as a dynamic hash table.

So that the remembered set does not become too large, an inline filter is applied to record only updates to persistent objects, as opposed to newly-created transient objects — Smalltalk is a prodigious allocator, so the vast majority of updates are to transient objects. This requires a check to see that the updated object is located in the separately managed persistent area of the volatile heap, determined by taking the high bits of its address to index a table that contains such information. If the updated object is indeed persistent then a subroutine is invoked to hash the object's pointer into the remembered set.

Remembered sets have the advantage of being both concise and accurate, at the cost of filtering and hashing to keep the sets small — repeated updates to the same object result in just one entry in the remembered set, but incur repeated overhead to filter and hash.

#### 4.2.3.2 Card-Based Schemes

For small objects, the object-based schemes are ideal. However, updates to larger objects may suffer from poor locality with respect to the object size, resulting in unnecessary unswizzling and comparison upon checkpoint, bounded solely by the size of the object. An alternative is to record updates based on fixed-size units of the virtual memory space, by dividing the memory into aligned logical regions of size $2^k$ bytes — the address of the first byte in the region has $k$ low bits zero. These regions are called *cards* after Sobalvarro [1988]. Each card has a corresponding entry in a card table indicating whether the card contains updated locations. Mapping an address to an entry in this table is simple: shift the

address right by $k$ bits and use the result as an index into the table. Whenever an object is modified, the corresponding card is *dirtied*.

One of the most attractive features of card marking is the simplicity of the write barrier. Ignoring cache effects, the per-update overhead is constant. Keeping this overhead to a minimum is highly desirable. By implementing the card table as a byte array (rather than a bit map), and interpreting zero bytes as dirty entries and non-zero bytes as clean, a store is recorded using only a shift, index, and byte store of zero [Wilson and Moher, 1989a; Wilson and Moher, 1989b].[5]

The checkpoint operation scans only the dirty cards containing persistent objects, to perform unswizzling and obtain differences for logging. Unswizzling requires locating all pointers within the card. Moreover, the log records must be generated with respect to the modified objects in the card, recording the object identifier and contiguous ranges of modified bytes. Since the formats of the objects in the card are encoded in their object headers, the header of the first object within a given card must be located to start the scan. For this purpose, a table of card offsets parallel to the dirty card table records the location of the *last* (highest address) object header within each card. Thus, given a card for scanning, the header of the first object in the card can be found at the end of the last object in the previous card.

Dirty cards are marked clean after scanning. To reduce the overhead of scanning, contiguous dirty cards are scanned as a batch, running from the first to the last in one scan. Also, the implementation takes great pains to avoid unnecessary memory accesses when scanning the card table to locate a run of dirty cards, by loading an entire memory word of the table at a time.

The size of the cards is an important factor influencing checkpoint costs, since large cards mean fewer cards and smaller tables. However, larger cards imply unnecessary checkpoint

[5] Some modern RISC architectures either do not provide a byte store instruction, or implement it as a read-modify-write word instruction. On such machines, it may be cheaper to code the read-modify-write explicitly as a sequence of instructions, or even revert to a bit map implementation of the card table.

overhead to perform unswizzling and comparison of objects that are unmodified, but just happen to lie in a dirty card. Thus, an interesting question arises as to whether there exists an optimal card size which minimizes the sum of these competing overheads. The performance evaluation of Chapter 5 answers this question for the program behaviors considered there.

#### 4.2.3.3 Page-Protection Schemes

A variant of the card-based approach uses the hardware-supported page protection primitives of the operating system to detect stores into clean cards. A card in this scheme corresponds to a page of virtual memory. All clean pages are protected from writes. When a write occurs to a protected page, the trap handler dirties the corresponding entry in the card table and unprotects the page. Subsequent writes to the now dirty page incur no extra overhead.[6] Because of this, keeping the dirty page table as small as possible may be more important than the overhead to dirty an entry in the page table. Thus, one might prefer to implement the dirty page table as a bit table, reducing the table's size by a factor of eight. Here, we continue to use a byte table, for more direct comparison with the corresponding-sized card scheme.

[6] An operating system could more efficiently supply the information needed in the page protection scheme by offering appropriate calls to obtain the page dirty bits maintained by most memory management hardware [Shaw, 1987].

# CHAPTER 5

# PERFORMANCE EVALUATION

Having described an architecture for persistence, and several realizations of the read and write barrier mechanisms required to support that architecture, we now examine the performance of the prototype implementation, for each implementation alternative.

## 5.1 Benchmarks

The performance evaluation draws on the OO1 object operations benchmarks [Cattell and Skeen, 1992] to compare the alternative implementation approaches. These benchmarks are retrieval-oriented and operate on substantial data structures, although the benchmarks themselves are simple, and so easily understood. Their execution patterns include phases of intensive computation, so that memory residence is important, in particular contrast to the on-line transaction processing style of computation, which involves only simple accesses and updates. Moreover, though the OO1 benchmarks are relatively low-level, they are sufficient for an exhaustive exploration of the behavior of our minimal read and write barrier persistence mechanisms.

For higher-level benchmark operations, such as set-oriented queries or relational-style operations, the reader is referred to the Hypermodel [Anderson et al., 1989] and OO7 [Carey et al., 1993] benchmarks. Both of these use a richer database schema over which more complicated operations are defined, though they also retain simple operations very similar to those of OO1. There is little (if anything) to be gained by using these benchmarks in this study of low-level persistence mechanisms.

### 5.1.1 Benchmark Database

The OO1 benchmark database consists of a collection of 20 000 *part* objects, indexed by part numbers in the range 1 through 20 000, with exactly three *connections* from each part to other parts. The connections are randomly selected to produce some locality of reference: 90% of the connections are to the "closest" 1% of parts, with the remainder being made to any randomly chosen part. Closeness is defined as parts with the numerically closest part numbers.

The part database and the benchmarks are implemented entirely in Smalltalk, including the B-tree used to index the parts (source code is listed in Appendix A). Performance measurements are gathered using a specially instrumented version of the Smalltalk virtual machine. The instrumentation is kept constant across all implementation variants being considered, to enable direct comparisons.

The Mneme database file, including the base Smalltalk virtual image as well as the parts data, consumes 179 physical segments, for a total size of around 6MB. Each physical segment is at least 32KB in size, although some may be larger since any Smalltalk object larger than 32KB is allocated in its own private segment. There are on average 3.2 logical segments per physical segment, of up to 255 objects each. Newly created objects are clustered into segments only as they are encountered when unswizzling, using an essentially breadth-first traversal similar to that of copying garbage collectors [Cheney, 1970]. The part objects are 68 bytes in size (including the object header). The three outgoing connections are stored directly in the part objects. The string fields associated with each part and connection are represented by references to separate Smalltalk objects of 24 bytes each. Similarly, a part's incoming connections are represented as a separate Smalltalk Array object containing references to the parts that are the source of each incoming connection. The B-tree index for the 20 000 parts consumes around 165KB.

### 5.1.2 Benchmark Operations

The OO1 benchmarks comprise three separate operations and measure response time for execution of each operation. The first two operations are read-only, leaving the permanent database untouched. Being retrieval-oriented, and involving no modification to the database, they are good measures for comparing object faulting schemes. They operate as follows:

- **Lookup** fetches 1 000 randomly chosen parts from the database. A null procedure is invoked for each part, taking as its arguments the *x*, *y*, and *type* fields of the part.
- **Traversal** fetches all parts connected to a randomly chosen part, or to any part connected to it, up to seven hops (for a total of 3 280 parts, with possible duplicates). Similarly to the Lookup benchmark, a null procedure is invoked for each part, taking as its arguments the *x*, *y*, and *type* fields of the part.[1]

The third operation requires updating the permanent database, and so is more appropriate for comparing update detection schemes for resilience:

- **Insert** allocates 100 new parts in the database, each with three connections to randomly selected parts as described in Section 5.1.1 (i.e., applying the same rules for locality of reference). The index structure must be updated, and the entire set of changes committed to disk.

Although this operation is a reasonable measure of update overhead, it is hampered by a lack of control over the number and distribution of the locations modified, and its mixing of updates to parts and the index. A more controlled benchmark is:

- **Update** [White and DeWitt, 1992] operates in the same way as the Traversal measure, but instead of calling a null procedure it performs a simple update to each part object encountered, with some fixed probability. The update consists of incrementing the *x* and *y* scalar integer fields of the part. All changes must be committed to disk.

The probability of update can vary from one run to the next to change the frequency and density of updates.

---

[1] OO1 also specifies a *reverse* Traversal operation, swapping "from" and "to" directions. This reverse Traversal operation is of minimal practical use because the random nature of connections means that the number of "from" connections varies among the parts — while every part has three *outgoing* connections, the number of *incoming* connections varies randomly. Thus, different iterations of the reverse Traversal vary randomly in the number of objects they traverse, and so the amount of work they perform.

These benchmarks are intended to be representative of the data operations in many engineering applications. The Lookup benchmark emphasizes selective retrieval of objects based on their attributes, while the Traversal benchmark illuminates the cost of raw pointer traversal. The Update variant measures the costs of modifying objects and making those changes permanent. Additionally, the Insert benchmark measures both update overhead and the cost of creating new persistent objects.

OO1 calls for each benchmark measure to be *iterated* ten times, the first when the system is *cold*, with none of the database cached (apart from any schema or system information necessary to initialize the system). Thus, before each run of ten iterations we execute a "chill" program on the client to read a 32MB file from the server, scanning first forwards then backwards; this flushes the operating system kernel buffer cache on both client and server, so that the first iteration is truly cold. Each successive iteration accesses a *different* set of random parts. Still, there may be some overlap in the parts accessed by different iterations, in which case implementations that cache data from one iteration to the next will exhibit a warming trend, with improved performance for later *warm* iterations that access data cached by earlier iterations.

In addition to the ten cold through warm iterations, it is useful to measure the elapsed time for *hot* iterations of the Traversal and Update benchmarks, by beginning each hot iteration at the same initial part used in the last of the warm iterations. The hot runs are guaranteed to traverse only resident objects, and so are free of any overheads due to swizzling and retrieval of non-resident objects.

## 5.2 Experimental Setup

The client machine on which the experiments were run is a SPARCstation 2 (Cypress Semiconductor CY7C601 integer unit clocked at 40MHz) running SunOS 4.1.3.[2] The SPARCstation 2 has a 64KB unified cache (instruction and data) with a line size of 32 bytes.

[2] SPARCstation is a trademark of SPARC International, licensed exclusively to Sun Microsystems. SunOS is a trademark of Sun Microsystems.

Read misses cost 24-25 cycles. On a hit, writes update both cache and memory — a 16-byte write buffer reduces this overhead, though if the buffer is full the processor will stall for 4-5 cycles for one slow memory cycle to complete. Write misses invalidate, but do not allocate, the corresponding cache line.

The system has 32MB of main memory (DRAM), sufficient for the entire benchmark database to be cached in memory. Thus, buffer management policies can be ignored when interpreting the experimental results. The log file is written locally to an internal SUN0424 SCSI disk (414 360KB unformatted capacity, 2.5-3.0MB/s peak data rate, ~2.9MB/s sustained data rate,[3] 14ms average seek time). The log records are buffered by Mneme, and written as a batch using calls to `write`, followed immediately by a call to `fsync` to force the log data to the local disk before the checkpoint completes. Thus, checkpoints break down into two phases (*unswizzling* and *writing*), which are measured separately.

The Smalltalk virtual machine is compiled with the GNU C compiler (`gcc`) version 2.5.7 at optimization level 2 (option `-O2`), and all libraries are linked statically (`gcc` link option `-static`). The benchmarks were run with the client in single-user mode and disconnected from the network, to minimize interference from network traffic, virtual memory paging, and other operating system activity.

The database is stored locally (i.e., the client is its own Mneme database server), for the simple reason that the experimental apparatus was thus easier to obtain and control, and also because results for a remote database would differ only in the network latency for retrieval of Mneme physical segments from a remote server's disk. Local disk latencies turn out to be sufficiently high to demonstrate the caching effects inherent in the implementation. The database resides on an external SUN1.3G SCSI disk (1 336 200KB unformatted capacity, 3.25-4.5MB/s peak data rate, ~3.5MB/s sustained data rate, 11ms average seek time).

[3]The data rate varies because the disk has a constant linear density for all tracks, with outer tracks yielding a faster bit rate than inner tracks. Peak data rate is the data rate possible for a single sector. Sustained data rate is calculated as the number of 512-byte sectors per track multiplied by the angular velocity — this number is only approximate, since the number of sectors per track decreases from outer to inner tracks.

Elapsed time for the execution of the benchmark operations is measured using the SunOS system call `gettimeofday`, which directly accesses the system hardware clock. On the SPARCstation 2 the resolution provided by `gettimeofday` is $1\mu$s. This fine-grained accuracy permits the separate phases of execution (running, swizzling, disk retrieval, logging, etc.) to be measured separately with minimal disturbance of the results due to measurement overheads. All times are reported in seconds (unless stated otherwise), and exclude the time to initialize the Smalltalk system prior to beginning the benchmark.

A benchmark *run* consists of the ten cold through warm iterations, plus a single hot iteration. To guarantee repeatability, the permanent database is kept entirely static (checkpointed updates are written only to the log, never propagated to the permanent database) so that different runs are always presented with the same physical database. Moreover, every run begins with the same random number seed, so the *n*th iteration of any given benchmark run always accesses the same parts as the *n*th iteration within any other benchmark run. In other words, although a different set of random parts are accessed from one cold/warm iteration to the next *within* a run, corresponding iterations *across* runs always access the same set of parts, so corresponding iterations from different runs are directly comparable.

Nevertheless, there may be other uncontrollable variations in system behavior from one run to the next. For example, the disk state (track and block position of the disk read/write arm) may influence read performance from one run to the next. To get an idea of the significance of this variation in the comparison of different implementations we repeat the runs for each implementation and calculate the results as 90% confidence intervals for mean elapsed time. The confidence interval is a measure of the repeatability of the experiments — small confidence intervals indicate a high degree of repeatability and reveal the significance of differences in the experimental results among the alternative implementations.

As well as measuring elapsed time for the benchmarks, the runs were simulated using Shade (part of Sun's SPARC Performance Analysis Tools),[4] to obtain precise execution profiles for each run. Using modified versions of Shade's tools for cache simulation (`cachesim5`) and instruction profiling (`spixcounts`), this yielded precise counts of cache misses (for the SPARCstation 2 cache configuration described above) and execution frequency, per instruction address.

### 5.2.1 The Read Barrier: Object Faulting

The object faulting experiments use the Lookup and Traversal benchmarks to compare several versions of the virtual machine, varying different parameters of the implementation:

- Representation of references to non-resident objects: this dictates the specific code for residency checks:
  - tagged PIDs (ID)
  - pointers to tagged fault blocks (FB)
  - pointers to page-protected fault blocks (trap)
- Faulting and swizzling: the number of objects made resident (and swizzled) per fault:
  - single object: target of the fault (obj)
  - logical segment: target of the fault along with all the objects in its LSEG (LSEG)
  - physical segment: target of the fault along with all the objects in its PSEG (PSEG)
- Indirection elimination: the time when unconstrained references are converted to point directly to their resident target object:
  - never: this makes sense only for ID schemes (the other schemes obtain a pointer to a fault/indirect block when swizzling, which directly encodes whether the target object is resident, whereupon it can be swizzled to a direct pointer) (no-bypass)
  - when the reference is first swizzled: references are converted to direct pointers as they are encountered during swizzling and only if their target object is already resident (swizzle-bypass)
  - when the reference is first swizzled (if the target is already resident); otherwise, when the target is first made resident as the result of a fault (fault-bypass)

[4] Shade is freely available for research use under license from Sun. Contact address is: Bob Cmelik, Sun Microsystems Laboratories, Inc., 2550 Garcia Avenue, MS 29-225, Mountain View, CA 94043, (415) 336-1709, rfc@sun.com.

**Table 5.1** Schemes compared in object faulting experiments

| Scheme | Unswizzled references | Fault granularity | When to bypass |
|---|---|---|---|
| non-persistent | direct pointer | — | — |
| ID, no-bypass, obj | tagged PID | object | never |
| ID, no-bypass, LSEG | " | LSEG | " |
| ID, no-bypass, PSEG | " | PSEG | " |
| ID, swizzle-bypass, obj | " | object | swizzling |
| ID, swizzle-bypass, LSEG | " | LSEG | " |
| ID, swizzle-bypass, PSEG | " | PSEG | " |
| FB, swizzle-bypass, obj | pointer to tagged fault/indirect block | object | " |
| FB, swizzle-bypass, LSEG | " | LSEG | " |
| FB, swizzle-bypass, PSEG | " | PSEG | " |
| ID, fault-bypass, obj | tagged PID | object | swizzling/faulting |
| ID, fault-bypass, LSEG | " | LSEG | " |
| ID, fault-bypass, PSEG | " | PSEG | " |
| FB, fault-bypass, obj | pointer to tagged fault/indirect block | object | " |
| FB, fault-bypass, LSEG | " | LSEG | " |
| FB, fault-bypass, PSEG | " | PSEG | " |
| trap, fault-bypass, obj | pointer to page-protected fault/indirect block | object | " |
| trap, fault-bypass, LSEG | " | LSEG | " |
| trap, fault-bypass, PSEG | " | PSEG | " |

As a basis for comparison, the non-persistent virtual machine was run against the memory-resident database (non-persistent). Table 5.1 enumerates the variants. Note that there is no trap,swizzle-bypass variant — trap overhead is so high that indirect references must be bypassed when their target is made resident for performance to be anywhere near acceptable.

### 5.2.2 The Write Barrier: Detecting and Logging Updates

For the experiments comparing the alternative update detection schemes for resilience, the object fault handling scheme is kept fixed (FB,LSEG,fault-bypass), while the mechanism for tracking updates varies. The experiments use the Insert and Update benchmarks, with update probabilities of 0.00, 0.05, 0.10, 0.15, 0.20, 0.50, and 1.00, to measure the performance of object marking (objects), remembered sets (remsets), access-protected

**Table 5.2** Schemes compared in resilience experiments

| Scheme | On update | On checkpoint |
|---|---|---|
| scan | do nothing | scan all objects |
| objects | set bit in header of changed object | scan for changed objects |
| remsets | enter object reference into update set | iterate over update set |
| cards-*n* | dirty card table entry | scan dirty table for dirty cards (process objects and fragments in cards) |
| pages | dirty and unprotect page | scan dirty table for dirty pages (process objects and fragments in pages) |

All schemes write only differences, for minimal log volume.

pages (pages), and card marking (cards-*n*, card size $n \in \{16, 64, 256, 1024, 4096\}$ bytes). As a "worst-case" scenario, we also include a scheme (scan) which does not record updates at all, but simply scans the entire set of resident persistent objects, and compares each one with its unmodified copy in the Mneme buffers. The page size on the SPARCstation 2 is 4096 bytes. The update detection schemes are summarized in Table 5.2. In addition to the cold through warm and hot iterations, which perform one update traversal per checkpoint, we also measure the performance for longer transactions, varying the number of hot update traversals performed as a single transaction. This allows us to derive an estimate of the run-time overheads (excluding swizzling and disk accesses for retrieval) for the different update detection schemes. We measure the total elapsed time for 5, 10, 15, 20, 50, 100, and 500 iterations per checkpoint.

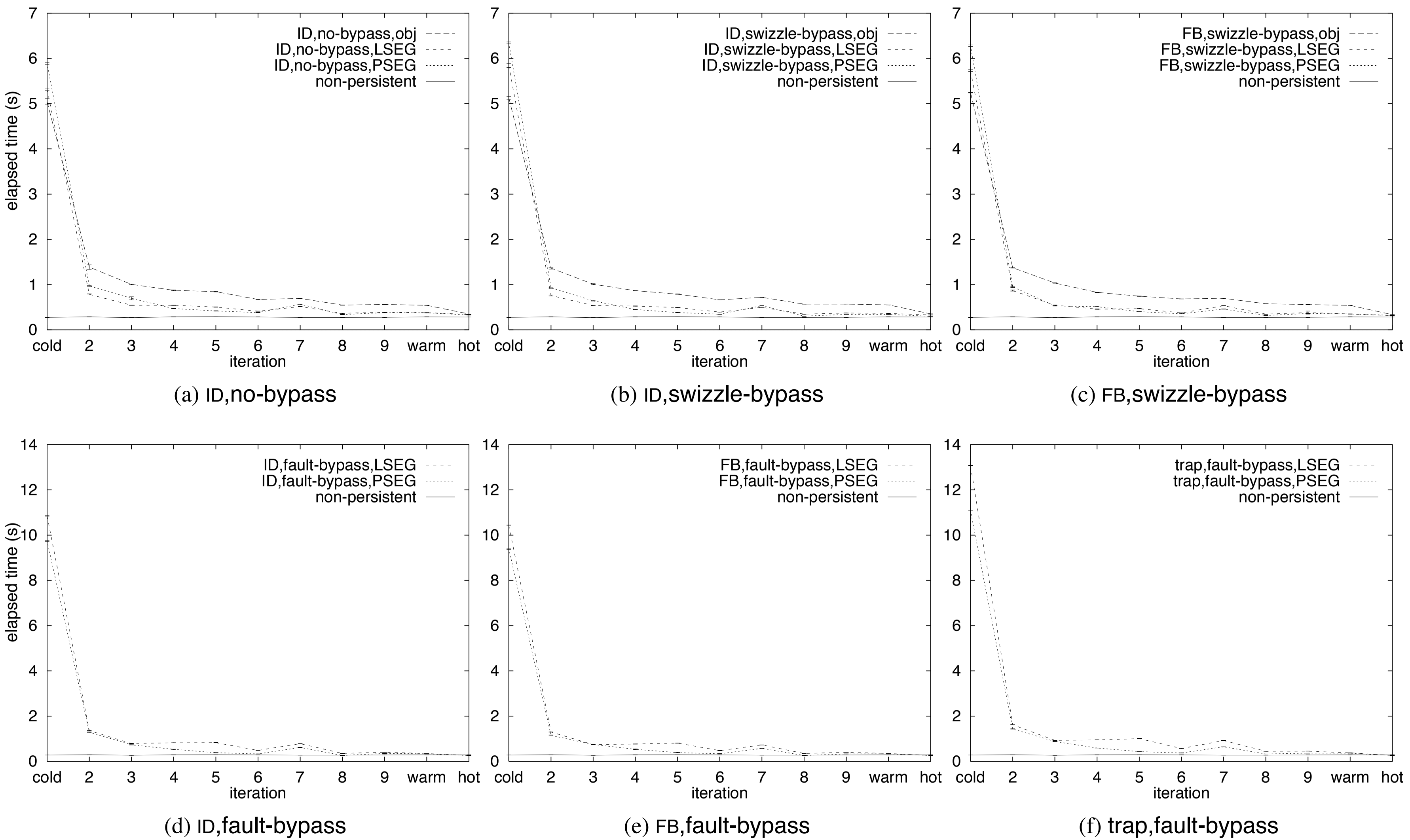
## 5.3 Results: Object Faulting

The first set of results are for the Lookup and Traversal benchmarks, which compare object faulting alternatives, including alternatives for the representation of references to non-resident objects, alternative faulting and swizzling policies, and whether indirections are eliminated. The graphs presented here give total elapsed time (in seconds) for execution of the benchmarks. Detailed breakdowns of the total elapsed time by phases of execution of each benchmark appear in Appendix B.

### 5.3.1 Lookup

The full set of results for the Lookup benchmark are plotted in Figure 5.1. Each graph shows total elapsed time for execution of the benchmark from cold through warm to hot iterations, varying just one of the implementation variables; i.e., there are plots for each of the fault granularities (obj/LSEG/PSEG) for each possible representation/swizzling combination (ID/FB/trap, no-/swizzle-/fault-bypass). The non-persistent results are also plotted as a baseline measure of the benchmark's pure running time (i.e., time spent only in the interpreter executing bytecodes). Note the difference in scale for the fault-bypass variants, which incur extra overhead at each fault to eliminate indirect references to resident objects. Also, results for the fault-bypass,obj variants are omitted from the plots because of their poor performance for cold through warm iterations — since they make just one object resident per fault, object faults are so frequent that the high overhead to eliminate indirections at every fault dominates execution; even for the warm iteration the obj schemes still experience object faults (cf. Figures B.1(d)-(f), p. 134).

All variations exhibit a clear warming trend as the system caches more and more objects from one iteration to the next, requiring successively fewer disk accesses. The warmup is almost immediate, for all schemes, since the first cold Lookup iteration touches enough of the database to bring most of the database's physical segments into Mneme's client buffers, whence objects can be swizzled very quickly. This behavior is intrinsic to Lookup since it randomly touches 1000 unrelated objects with each iteration. It also illustrates the value of clustering objects into segments for efficient transfer from disk. Taking the LOOM approach of transferring an object at a time from disk would require disk activity for every object swizzled, so the warming trend would be much less immediate.




(a) ID,no-bypass

(b) ID,swizzle-bypass

(c) FB,swizzle-bypass

(d) ID,fault-bypass

(e) FB,fault-bypass

(f) trap,fault-bypass

**Figure 5.1** Lookup, by object faulting scheme

Ignoring the issue of indirection elimination, for cold through warm iterations there is little difference among the alternative representations for references to non-resident objects (cf. Figures 5.1(a)-(c) and (d)-(f)), except that the trap,fault-bypass schemes have higher overhead than the ID/FB,fault-bypass schemes because of the high cost of fielding page protection violations and managing page protections. All approaches have performance close to non-persistent at the hot iteration (relative to the colder iterations), indicating that overheads for disk access and swizzling dominate cold through warm execution, while the residency constraints eliminate most of the overhead for explicit residency checks.

The key cold/warm/hot results (with 90% confidence intervals) are summarized in Table 5.3. We now examine each of these separately in more detail.

#### 5.3.1.1 Cold Lookup

The cold iteration is dominated by disk retrieval (elapsed time is an order of magnitude greater than for non-persistent), so the best overall cold performance is given by the ID,no-bypass,obj scheme, which retrieves the minimal set of objects required to complete the iteration and swizzles them simply by *copying* them into the application memory space without converting them in any way. Schemes that swizzle more eagerly require correspondingly higher overhead on retrieval: swizzle-bypass to bypass indirections while swizzling and fault-bypass to eliminate indirections at every fault. Similarly, schemes that fault larger granularities incur higher overhead because more objects must be copied. Nevertheless, whereas *smaller* granularities mean overall better performance for the no-bypass and swizzle-bypass schemes, faulting *larger* granularities yields better performance for the fault-bypass schemes because it results in fewer object faults; thus the high per-fault overhead of indirection elimination occurs less frequently. These results are highlighted by the boldface figures in the cold column of Table 5.3.

**Table 5.3** Lookup: key cold/warm/hot results

| Scheme | Cold | Warm | Hot | | | | |
|---|---|---|---|---|---|---|---|
| | elapsed time (s) | elapsed time (s) | elapsed time (s) | instruction references | instruction misses | data read misses | data write misses |
| non-persistent | $0.2756 \pm_{90\%} 0.0001$ | $0.2852 \pm_{90\%} 0.0001$ | $0.2820 \pm_{90\%} 0.0001$ | 5633352 | 25654 | 42945 | 14467 |
| ID, no-bypass, obj | $\mathbf{5.05} \pm_{90\%} 0.06$ | $0.542 \pm_{90\%} 0.002$ | $0.3501 \pm_{90\%} 0.0001$ | 7492288 | 27412 | 50398 | 1539 |
| ID, no-bypass, LSEG | $5.32 \pm_{90\%} 0.03$ | $0.379 \pm_{90\%} 0.001$ | $0.3407 \pm_{90\%} 0.0001$ | 7487326 | 10792 | 54687 | 1150 |
| ID, no-bypass, PSEG | $5.89 \pm_{90\%} 0.02$ | $\mathbf{0.376} \pm_{90\%} 0.002$ | $\mathbf{0.3284} \pm_{90\%} 0.0001$ | 7499011 | 11813 | 34277 | 202 |
| ID, swizzle-bypass, obj | $\mathbf{5.13} \pm_{90\%} 0.03$ | $0.5511 \pm_{90\%} 0.0007$ | $0.3522 \pm_{90\%} 0.0001$ | 7492288 | 32135 | 49204 | 1539 |
| ID, swizzle-bypass, LSEG | $5.85 \pm_{90\%} 0.05$ | $0.3635 \pm_{90\%} 0.0005$ | $0.3275 \pm_{90\%} 0.0001$ | 7052368 | 17385 | 50101 | 1150 |
| ID, swizzle-bypass, PSEG | $6.34 \pm_{90\%} 0.02$ | $\mathbf{0.3403} \pm_{90\%} 0.0008$ | $\mathbf{0.2998} \pm_{90\%} 0.0001$ | 6600871 | 17220 | 31014 | 200 |
| FB, swizzle-bypass, obj | $\mathbf{5.244} \pm_{90\%} 0.006$ | $0.5406 \pm_{90\%} 0.0008$ | $0.3340 \pm_{90\%} 0.0002$ | 6921155 | 14881 | 49706 | 8879 |
| FB, swizzle-bypass, LSEG | $5.73 \pm_{90\%} 0.02$ | $0.350 \pm_{90\%} 0.001$ | $\mathbf{0.3126} \pm_{90\%} 0.0001$ | 6588506 | 9392 | 46065 | 6178 |
| FB, swizzle-bypass, PSEG | $6.28 \pm_{90\%} 0.02$ | $\mathbf{0.350} \pm_{90\%} 0.002$ | $0.3196 \pm_{90\%} 0.0002$ | 6416510 | 6770 | 68109 | 4700 |
| ID, fault-bypass, obj | $106.42 \pm_{90\%} 0.04$ | $14.770 \pm_{90\%} 0.003$ | $0.3191 \pm_{90\%} 0.0001$ | 5637346 | 54258 | 66643 | 456 |
| ID, fault-bypass, LSEG | $10.86 \pm_{90\%} 0.02$ | $0.341 \pm_{90\%} 0.001$ | $0.2583 \pm_{90\%} 0.0001$ | 5632351 | 11452 | 22690 | 1077 |
| ID, fault-bypass, PSEG | $\mathbf{9.74} \pm_{90\%} 0.03$ | $\mathbf{0.332} \pm_{90\%} 0.002$ | $\mathbf{0.2567} \pm_{90\%} 0.0001$ | 5637346 | 8482 | 22397 | 94 |
| FB, fault-bypass, obj | $97.22 \pm_{90\%} 0.01$ | $13.772 \pm_{90\%} 0.001$ | $\mathbf{0.2542} \pm_{90\%} 0.0001$ | 5772880 | 2725 | 17457 | 452 |
| FB, fault-bypass, LSEG | $10.43 \pm_{90\%} 0.02$ | $0.345 \pm_{90\%} 0.002$ | $0.2592 \pm_{90\%} 0.0001$ | 5723937 | 6380 | 24690 | 1077 |
| FB, fault-bypass, PSEG | $\mathbf{9.40} \pm_{90\%} 0.02$ | $\mathbf{0.3320} \pm_{90\%} 0.0007$ | $0.2559 \pm_{90\%} 0.0002$ | 5642352 | 8259 | 22215 | 94 |
| trap, fault-bypass, obj | $138.22 \pm_{90\%} 0.03$ | $18.844 \pm_{90\%} 0.004$ | $\mathbf{0.2559} \pm_{90\%} 0.0001$ | N/A | N/A | N/A | N/A |
| trap, fault-bypass, LSEG | $13.072 \pm_{90\%} 0.008$ | $0.374 \pm_{90\%} 0.002$ | $0.2600 \pm_{90\%} 0.0001$ | N/A | N/A | N/A | N/A |
| trap, fault-bypass, PSEG | $\mathbf{11.09} \pm_{90\%} 0.02$ | $\mathbf{0.360} \pm_{90\%} 0.001$ | $0.2599 \pm_{90\%} 0.0001$ | N/A | N/A | N/A | N/A |

Boldface values are the best cold/warm/hot performance within each group of object faulting schemes.

#### 5.3.1.2 Warm Lookup

In contrast to the cold results, eagerness in faulting and swizzling in the cooler iterations pays off at warmer iterations, since fewer unswizzled references (each requiring an ID lookup, indirection bypass, or object fault) are encountered. Thus, the best warm Lookup performance is for the fault-bypass,PSEG schemes, except that the trap approach suffers from the high cost to field page protection violations to trigger object faults and to manage page protections (there are sufficient faults for these overheads to be apparent). The boldface figures in the warm column of Table 5.3 highlight this trend favoring eager faulting and swizzling for improved warm Lookup performance.

#### 5.3.1.3 Hot Lookup

The hot Lookup results are also summarized in Table 5.3. Similarly to the warm results, the schemes are essentially ranked by their eagerness to fault and swizzle, although the elapsed time results are slightly obscured by method lookup cache and underlying hardware cache effects — different fault granularities result in a different object layout in virtual memory, so cache footprints vary among the schemes. For this reason, the table also lists the number of instruction references, instruction cache misses, and data read and write misses, incurred by each of the schemes for the hot Lookup iteration.[5] The best hot Lookup performance for each faulting scheme is given in boldface.

The ID,no-bypass schemes all have comparable cost in terms of number of instructions executed since they do no swizzling of unconstrained references — differences are a result of variable method lookup cache behavior. Thus, most references encountered are in the form of tagged IDs, requiring expensive lookup to obtain a pointer to their targets, and fault

[5] Unfortunately, Shade results for the trap scheme are not available (N/A), since Shade cannot simulate our trap handler correctly: the handler relies on precise knowledge of the contents of particular registers at the time of the trap, in order to restart the trapping instruction with the address of the newly-resident object; Shade does not map these registers directly, so our trap handler does not work properly under Shade.

granularity has little intrinsic impact on performance — ID,no-bypass,PSEG is best (by chance) only because it incurs fewer hardware cache misses.

The swizzle-bypass schemes are ranked by faulting granularity, since faulting more objects at a time reduces the number of locations containing tagged IDs or pointers to fault blocks. Still, the FB,swizzle-bypass schemes require fewer instructions (if not less time because of variations in cache behavior) than the corresponding ID,swizzle-bypass schemes — for the remaining unswizzled references it is cheaper to dereference an indirect block than to look up an ID. Of the ID,swizzle-bypass schemes, ID,swizzle-bypass,PSEG is best since it leaves fewer references in their unswizzled tagged ID format. Of the FB,swizzle-bypass schemes, FB,swizzle-bypass,LSEG is better only because it has a better cache footprint than the PSEG variant, which executes fewer instructions because it encounters fewer indirect blocks.

As expected, overall best hot Lookup performance is achieved for the fault-bypass schemes since they convert all references to direct pointers — unswizzled references to resident objects are never encountered. The improvement in performance for the fault-bypass schemes over those which bypass indirections less eagerly is around 20% — significant enough for applications dealing with large amounts of hot data to pay the up-front cost of indirection elimination. In fact, the elapsed time executions for these schemes are mostly better than for non-persistent, once again because of improved cache behavior, and despite the fact that the software variants require additional overhead for the read barrier. This result illustrates the extreme importance of our residency constraints, since they go most of the way to eliminating software residency checks, so the overhead for the read barrier is negligible. Method lookup cache and hardware cache variations for different fault granularities account for the remaining differences in performance within each of the ID-, FB-, and trap- fault-bypass object faulting variants.

#### 5.3.1.4 Running Time Overheads

Lastly, to get some sense of the running time cost of the page traps and software-mediated object faults, we obtain linear regression fits of the running time (time spent in the interpreter actually executing bytecodes as opposed to retrieval, swizzling, or bypassing indirections) versus the number of faults occurring for each iteration of the benchmark. The variables and coefficients for the regression equation $y = a + bx$ are:

$y$ = running time (excluding swizzling and other fault handling overheads);
$a$ = base running time overhead per iteration;
$b$ = incremental running time overhead per fault;
$x$ = number of faults.

The fitted coefficients (including 90% confidence intervals) appear in Table 5.4, along with the sample correlation coefficient. The $b$ coefficient (overhead per fault) is a measure of the number of seconds required to get in and out of the object fault handler, either through software checks to detect faults or through a protection trap handler. Correlations are best for those schemes where objects faults occur frequently enough, or detecting a fault is sufficiently expensive, to have noticeable impact on running time performance. Where other effects have more impact, the correlations are not so good.

The results for the trap schemes are of particular interest, since they reveal the very high cost to field page protection traps in user-mode. In comparison, the overheads for software-mediated object faulting are at least an order of magnitude smaller. Note that the overhead to field page protection traps like this, interspersed throughout a program's normal execution, is likely to be significantly higher than that to field page protection traps in a tight loop, since the operating system trap handling code and data structures needed to service the trap may not be in the hardware caches. Indeed, in an earlier study Hosking and Moss [1993] show that fielding interspersed traps like this can cost an order of magnitude more than traps performed in a tight loop.

**Table 5.4** Lookup: fault detection overheads

| Scheme | | | Base running time (s) | Overhead per fault ($\mu$s) | sample correlation coefficient |
|---|---|---|---|---|---|
| ID, | no-bypass, | obj | $0.37 \pm_{90\%} 0.02$ | $50 \pm_{90\%} 10$ | 0.9469 |
| ID, | no-bypass, | LSEG | $0.344 \pm_{90\%} 0.005$ | $200 \pm_{90\%} 100$ | 0.7862 |
| ID, | no-bypass, | PSEG | $0.340 \pm_{90\%} 0.007$ | $90 \pm_{90\%} 200$ | 0.2310 |
| ID, | swizzle-bypass, | obj | $0.364 \pm_{90\%} 0.008$ | $48 \pm_{90\%} 4$ | 0.9904 |
| ID, | swizzle-bypass, | LSEG | $0.328 \pm_{90\%} 0.006$ | $200 \pm_{90\%} 100$ | 0.7791 |
| ID, | swizzle-bypass, | PSEG | $0.307 \pm_{90\%} 0.006$ | $100 \pm_{90\%} 200$ | 0.3564 |
| FB, | swizzle-bypass, | obj | $0.348 \pm_{90\%} 0.006$ | $57 \pm_{90\%} 3$ | 0.9952 |
| FB, | swizzle-bypass, | LSEG | $0.32 \pm_{90\%} 0.01$ | $300 \pm_{90\%} 200$ | 0.6607 |
| FB, | swizzle-bypass, | PSEG | $0.309 \pm_{90\%} 0.004$ | $80 \pm_{90\%} 100$ | 0.3431 |
| ID, | fault-bypass, | obj | $0.36 \pm_{90\%} 0.02$ | $110 \pm_{90\%} 10$ | 0.9884 |
| ID, | fault-bypass, | LSEG | $0.261 \pm_{90\%} 0.004$ | $280 \pm_{90\%} 80$ | 0.9144 |
| ID, | fault-bypass, | PSEG | $0.267 \pm_{90\%} 0.008$ | $100 \pm_{90\%} 300$ | 0.2632 |
| FB, | fault-bypass, | obj | $0.30 \pm_{90\%} 0.02$ | $124 \pm_{90\%} 9$ | 0.9926 |
| FB, | fault-bypass, | LSEG | $0.258 \pm_{90\%} 0.002$ | $260 \pm_{90\%} 40$ | 0.9708 |
| FB, | fault-bypass, | PSEG | $0.2570 \pm_{90\%} 0.0006$ | $170 \pm_{90\%} 20$ | 0.9804 |
| trap, | fault-bypass, | obj | $0.9 \pm_{90\%} 0.5$ | $8300 \pm_{90\%} 200$ | 0.9989 |
| trap, | fault-bypass, | LSEG | $0.27 \pm_{90\%} 0.01$ | $8800 \pm_{90\%} 200$ | 0.9992 |
| trap, | fault-bypass, | PSEG | $0.28 \pm_{90\%} 0.02$ | $9000 \pm_{90\%} 600$ | 0.9944 |

### 5.3.2 Traversal

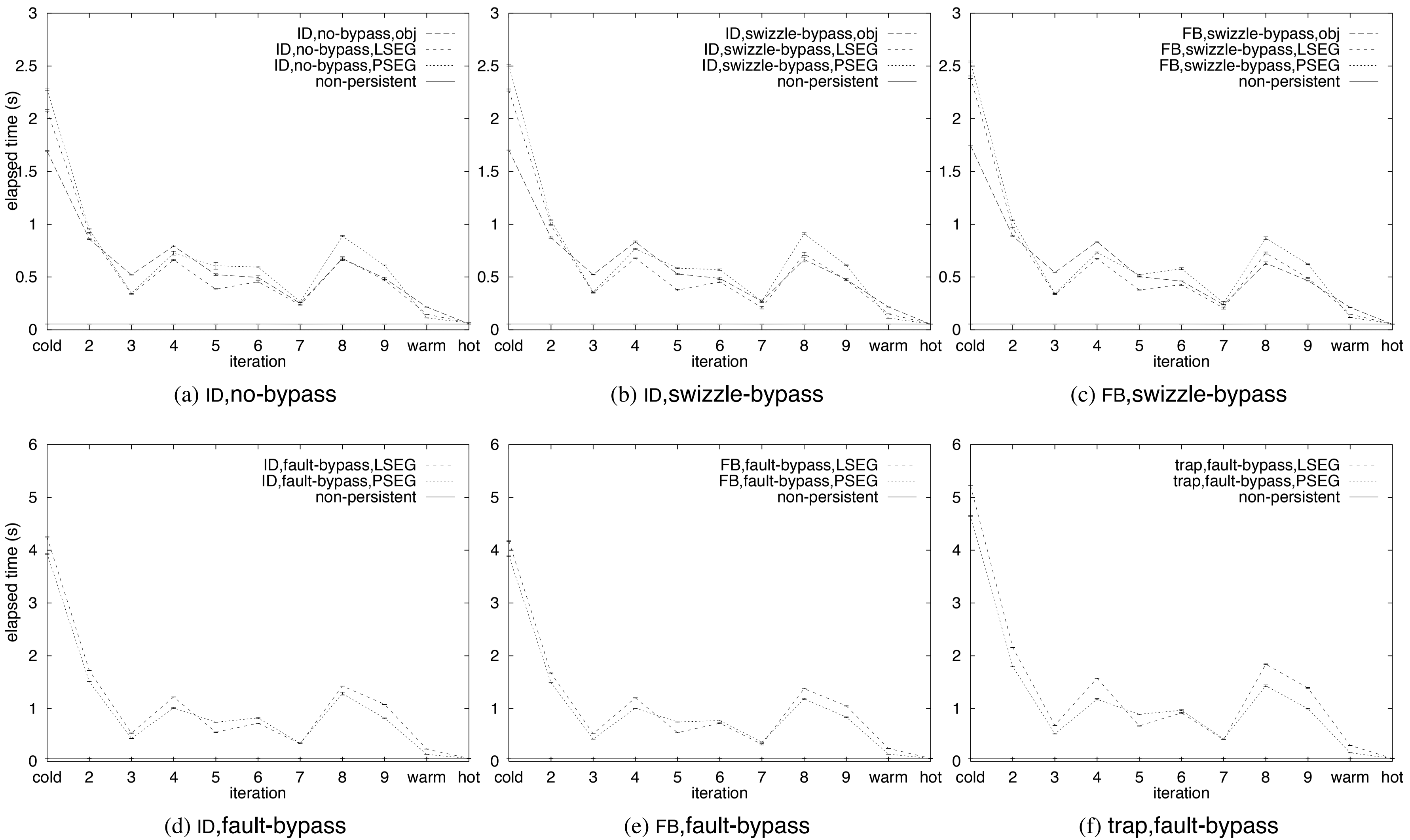
The full set of results for the Traversal benchmark are plotted in Figure 5.2, omitting those for fault-bypass,obj due to their poor performance (cf. Figures B.4(d)-(f), p. 137).

The warming trend is slower than for Lookup, despite the fact that each iteration accesses more parts. This is due to the locality of references encoded in the connections between parts, which has been replicated in the clustering used to group objects into physical segments. Thus, traversals mostly touch parts whose physical segments are already resident in Mneme's client buffers, with only a few connection traversals needing to be serviced by a disk access. When faults do occur, they signal the traversal entering a previously unvisited region of the database, requiring a burst of retrieval activity to fetch a cluster of "strongly-connected" parts. For this reason, the elapsed time for a Traversal does not necessarily decrease from one iteration to the next (nominally warmer) iteration, since subsequent iterations visit a different random set of parts, of which only a few may have been made resident in preceding iterations. This behavior makes the tradeoffs in swizzling and fault granularity more distinct.

Again, we summarize the key cold/warm/hot results (with 90% confidence intervals) in Table 5.5, and discuss each separately.

#### 5.3.2.1 Cold Traversal

The cold results are similar to those for Lookup. Eager swizzling and larger-granularity faulting incur higher overhead up front for retrieval, although the frequency of object faults (and hence indirection elimination) impact performance most for the fault-bypass schemes. Once again, best cold Traversal performance occurs for the ID,no-bypass,obj scheme.




(a) ID,no-bypass

(b) ID,swizzle-bypass

(c) FB,swizzle-bypass

(d) ID,fault-bypass

(e) FB,fault-bypass

(f) trap,fault-bypass

**Figure 5.2** Traversal, by object faulting scheme

#### 5.3.2.2 Warm Traversal

Unlike the Lookup benchmark, Traversal requires significant numbers of object faults even at the warm iteration. Thus, the overhead to eliminate indirections is still relevant to the warm Lookup performance of the fault-bypass schemes, which do not perform as well as ID,swizzle-bypass,PSEG which is best overall.

#### 5.3.2.3 Hot Traversal

Hot system performance is highlighted even more by the Traversal benchmark, since it is dominated by raw pointer traversal overheads much more than Lookup (which also incorporates the relatively fixed cost of the B-tree lookup). Once again, eliminating unswizzled references (fault-bypass) yields the best overall performance (around 15% improvement over other schemes), since all references are direct pointers, so the traversal can proceed at full speed. Interestingly, faulting one object at a time (obj) gives best performance across almost all object faulting variants (boldface), because of the improved locality of reference obtained by object-at-a-time placement: "strongly-connected" objects are placed close together in virtual memory, yielding improved hardware cache locality. The exception is for FB,swizzle-bypass, where the additional overhead due to cache misses is outweighed by the reduction in instruction cycles to bypass indirect blocks for LSEG over obj, so their elapsed times are much the same (52.5ms and 52.6ms respectively).

**Table 5.5** Traversal: key cold/warm/hot results

| Scheme | Cold | Warm | Hot | | | | |
|---|---|---|---|---|---|---|---|
| | elapsed time (s) | elapsed time (s) | elapsed time (s) | instruction references | instruction misses | data read misses | data write misses |
| non-persistent | $0.05563 \pm_{90\%} 0.00007$ | $0.05553 \pm_{90\%} 0.00007$ | $0.05465 \pm_{90\%} 0.00004$ | 1016219 | 6983 | 9109 | 50 |
| ID, no-bypass, obj | $\mathbf{1.693} \pm_{90\%} 0.005$ | $0.2157 \pm_{90\%} 0.0007$ | $\mathbf{0.0566} \pm_{90\%} 0.0002$ | 1303503 | 691 | 4263 | 99 |
| ID, no-bypass, LSEG | $2.08 \pm_{90\%} 0.01$ | $0.1479 \pm_{90\%} 0.0007$ | $0.0651 \pm_{90\%} 0.0003$ | 1303507 | 6955 | 9708 | 125 |
| ID, no-bypass, PSEG | $2.28 \pm_{90\%} 0.01$ | $\mathbf{0.1131} \pm_{90\%} 0.0009$ | $0.0655 \pm_{90\%} 0.0002$ | 1303593 | 7083 | 10584 | 323 |
| ID, swizzle-bypass, obj | $\mathbf{1.705} \pm_{90\%} 0.009$ | $0.2169 \pm_{90\%} 0.0009$ | $\mathbf{0.0532} \pm_{90\%} 0.0001$ | 1214100 | 493 | 3974 | 95 |
| ID, swizzle-bypass, LSEG | $2.27 \pm_{90\%} 0.01$ | $0.1508 \pm_{90\%} 0.0007$ | $0.0561 \pm_{90\%} 0.0001$ | 1059098 | 6907 | 9039 | 122 |
| ID, swizzle-bypass, PSEG | $2.508 \pm_{90\%} 0.009$ | $\mathbf{0.1110} \pm_{90\%} 0.0006$ | $0.0557 \pm_{90\%} 0.0002$ | 1039126 | 6823 | 9361 | 271 |
| FB, swizzle-bypass, obj | $\mathbf{1.747} \pm_{90\%} 0.004$ | $0.2136 \pm_{90\%} 0.002$ | $0.0526 \pm_{90\%} 0.0001$ | 1127224 | 497 | 6154 | 538 |
| FB, swizzle-bypass, LSEG | $2.39 \pm_{90\%} 0.01$ | $0.1491 \pm_{90\%} 0.0005$ | $\mathbf{0.0525} \pm_{90\%} 0.0002$ | 1036461 | 4717 | 7198 | 101 |
| FB, swizzle-bypass, PSEG | $2.537 \pm_{90\%} 0.009$ | $\mathbf{0.1180} \pm_{90\%} 0.0009$ | $0.05533 \pm_{90\%} 0.00003$ | 1027172 | 6866 | 9445 | 251 |
| ID, fault-bypass, obj | $17.521 \pm_{90\%} 0.006$ | $9.402 \pm_{90\%} 0.001$ | $\mathbf{0.04590} \pm_{90\%} 0.00008$ | 1016224 | 255 | 2445 | 83 |
| ID, fault-bypass, LSEG | $4.251 \pm_{90\%} 0.003$ | $0.2305 \pm_{90\%} 0.0006$ | $0.05441 \pm_{90\%} 0.00008$ | 1016219 | 6870 | 9033 | 76 |
| ID, fault-bypass, PSEG | $\mathbf{3.93} \pm_{90\%} 0.01$ | $\mathbf{0.1318} \pm_{90\%} 0.0006$ | $0.05452 \pm_{90\%} 0.00008$ | 1016224 | 6771 | 8969 | 230 |
| FB, fault-bypass, obj | $16.632 \pm_{90\%} 0.005$ | $8.9044 \pm_{90\%} 0.0007$ | $\mathbf{0.04631} \pm_{90\%} 0.00006$ | 1032629 | 274 | 2392 | 79 |
| FB, fault-bypass, LSEG | $4.18 \pm_{90\%} 0.01$ | $0.244 \pm_{90\%} 0.001$ | $0.0547 \pm_{90\%} 0.0003$ | 1016230 | 6857 | 9016 | 81 |
| FB, fault-bypass, PSEG | $\mathbf{3.90} \pm_{90\%} 0.01$ | $\mathbf{0.136} \pm_{90\%} 0.002$ | $0.05456 \pm_{90\%} 0.00005$ | 1016230 | 6785 | 8957 | 234 |
| trap, fault-bypass, obj | $21.42 \pm_{90\%} 0.01$ | $14.15 \pm_{90\%} 0.01$ | $\mathbf{0.0465} \pm_{90\%} 0.0002$ | N/A | N/A | N/A | N/A |
| trap, fault-bypass, LSEG | $5.224 \pm_{90\%} 0.006$ | $0.3010 \pm_{90\%} 0.0007$ | $0.05492 \pm_{90\%} 0.00003$ | N/A | N/A | N/A | N/A |
| trap, fault-bypass, PSEG | $\mathbf{4.651} \pm_{90\%} 0.009$ | $\mathbf{0.162} \pm_{90\%} 0.002$ | $0.0552 \pm_{90\%} 0.0002$ | N/A | N/A | N/A | N/A |

Boldface values are the best cold/warm/hot performance within each group of object faulting schemes.

**Table 5.6** Traversal: fault detection overheads

| Scheme | Base running time (s) | Overhead per fault ($\mu$s) | sample correlation coefficient |
|---|---|---|---|
| ID, no-bypass, obj | $0.058 \pm_{90\%} 0.002$ | $52 \pm_{90\%} 2$ | 0.9970 |
| ID, no-bypass, LSEG | $0.0664 \pm_{90\%} 0.0008$ | $160 \pm_{90\%} 30$ | 0.9610 |
| ID, no-bypass, PSEG | $0.0668 \pm_{90\%} 0.0006$ | $170 \pm_{90\%} 40$ | 0.9215 |
| ID, swizzle-bypass, obj | $0.054 \pm_{90\%} 0.002$ | $55 \pm_{90\%} 2$ | 0.9981 |
| ID, swizzle-bypass, LSEG | $0.0577 \pm_{90\%} 0.0009$ | $160 \pm_{90\%} 30$ | 0.9455 |
| ID, swizzle-bypass, PSEG | $0.0572 \pm_{90\%} 0.0008$ | $170 \pm_{90\%} 60$ | 0.8844 |
| FB, swizzle-bypass, obj | $0.055 \pm_{90\%} 0.002$ | $54 \pm_{90\%} 3$ | 0.9964 |
| FB, swizzle-bypass, LSEG | $0.056 \pm_{90\%} 0.002$ | $190 \pm_{90\%} 60$ | 0.8989 |
| FB, swizzle-bypass, PSEG | $0.0571 \pm_{90\%} 0.0007$ | $180 \pm_{90\%} 50$ | 0.9233 |
| ID, fault-bypass, obj | $0.0470 \pm_{90\%} 0.0008$ | $102 \pm_{90\%} 1$ | 0.9999 |
| ID, fault-bypass, LSEG | $0.0554 \pm_{90\%} 0.0004$ | $140 \pm_{90\%} 10$ | 0.9867 |
| ID, fault-bypass, PSEG | $0.0559 \pm_{90\%} 0.0005$ | $140 \pm_{90\%} 40$ | 0.9203 |
| FB, fault-bypass, obj | $0.049 \pm_{90\%} 0.002$ | $106 \pm_{90\%} 3$ | 0.9992 |
| FB, fault-bypass, LSEG | $0.0555 \pm_{90\%} 0.0004$ | $140 \pm_{90\%} 10$ | 0.9880 |
| FB, fault-bypass, PSEG | $0.0560 \pm_{90\%} 0.0005$ | $140 \pm_{90\%} 40$ | 0.9236 |
| trap, fault-bypass, obj | $0.7 \pm_{90\%} 1.6$ | $7000 \pm_{90\%} 2000$ | 0.8995 |
| trap, fault-bypass, LSEG | $0.07 \pm_{90\%} 0.01$ | $8300 \pm_{90\%} 500$ | 0.9956 |
| trap, fault-bypass, PSEG | $0.062 \pm_{90\%} 0.004$ | $8800 \pm_{90\%} 300$ | 0.9984 |

#### 5.3.2.4 Running Time Overheads

Because Traversal performance is so strongly correlated with the overhead of raw pointer traversal, the regression fits to measure object faulting overheads are all excellent (see Table 5.6). Once again, we see the very high overhead to trap page protection violations, with the software-mediated schemes having much lower overhead. Also, cache locality produced by object-at-a-time faulting is evident in the obj variants having much lower run-time object faulting overhead than for larger-granularity faulting — not only is data more likely to be resident in the cache, but the more frequent faulting incurred by the obj schemes means that the code to trigger faults is also more likely to be cache resident.

### 5.3.3 Conclusions

The results are conclusive in establishing that software object fault detection mechanisms can provide performance very close to optimal, even surpassing the performance of

comparable hardware-assisted schemes. This has been achieved through careful assumptions about residency. In particular, the object-oriented execution paradigm allows many residency checks to be elided, with residency checks being restricted mostly to method invocation. This approach can be applied to any language that includes dynamic binding of method calls, by arranging for fault blocks to respond to all methods by first faulting the target object and then forwarding the invocation to it. We have also shown that it can pay to be eager in object swizzling, by swizzling *related* objects in advance of the application's need for them. Naturally, this depends on the number of times a each object reference is used, just as eager swizzling incurs higher overhead when the system is cold. Thus, the break-even point for swizzling depends on the execution characteristics of the application.[6]

## 5.4 Results: Detecting and Logging Updates

The second set of results concerns the detection of updates to support resilience, using both the Insert and Update benchmarks. Both of these benchmarks include a checkpoint operation, thereby enabling the comparison of different approaches to detecting updates and generating the log.

### 5.4.1 Insert

Results for Insert are plotted in Figure 5.3, and summarized in Table 5.7. The baseline non-persistent implementation treats checkpoint as a null operation.

For cold transactions, the results for the pages scheme reveal the high cost of calls to the operating system to manage page protections. Only as the system warms up does performance for pages fall below the "worst-case" scan approach, which pays consistently high overhead to scan the entire set of resident objects on checkpoint.

[6] We leave a detailed study of the break-even points for swizzling to future work, since the current swizzling implementation is not particularly well-tuned. Its performance is also dependent on database clustering, which we have also not attempted to tune. As such, the current implementation is unlikely to give a complete picture of the tradeoff between swizzled and non-swizzled implementations.

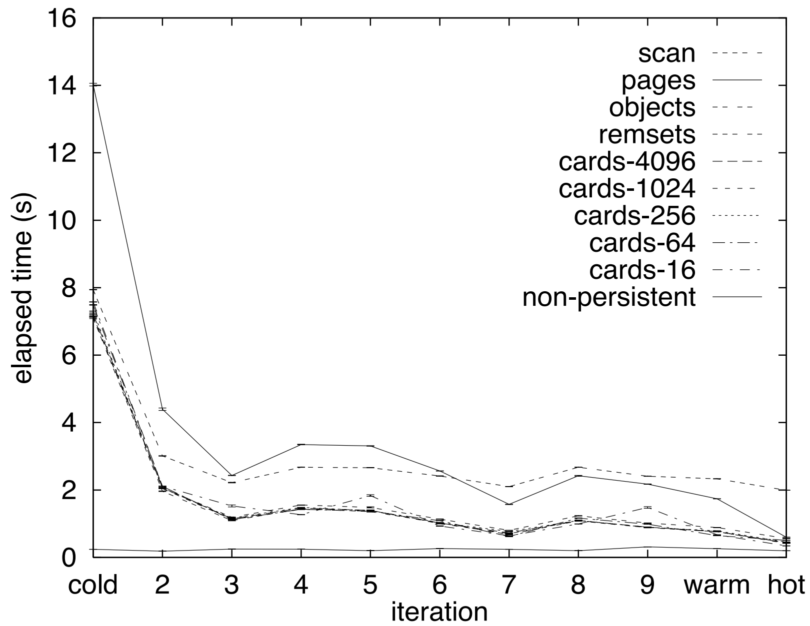



**Figure 5.3** Insert

**Table 5.7** Insert

| Scheme | Cold | Warm | Hot |
|---|---|---|---|
| non-persistent | $0.2426 \pm_{90\%} 0.0001$ | $0.26014 \pm_{90\%} 0.00009$ | $0.19914 \pm_{90\%} 0.00006$ |
| scan | $7.942 \pm_{90\%} 0.008$ | $2.333 \pm_{90\%} 0.005$ | $1.993 \pm_{90\%} 0.001$ |
| objects | $7.161 \pm_{90\%} 0.007$ | $0.887 \pm_{90\%} 0.003$ | $0.562 \pm_{90\%} 0.002$ |
| remsets | $7.10 \pm_{90\%} 0.02$ | $0.766 \pm_{90\%} 0.004$ | $0.436 \pm_{90\%} 0.002$ |
| cards-16 | $7.61 \pm_{90\%} 0.02$ | $0.673 \pm_{90\%} 0.003$ | $0.345 \pm_{90\%} 0.003$ |
| cards-64 | $7.55 \pm_{90\%} 0.01$ | $0.646 \pm_{90\%} 0.003$ | $0.511 \pm_{90\%} 0.003$ |
| cards-256 | $7.327 \pm_{90\%} 0.009$ | $0.794 \pm_{90\%} 0.003$ | $0.454 \pm_{90\%} 0.003$ |
| cards-1024 | $7.189 \pm_{90\%} 0.009$ | $0.766 \pm_{90\%} 0.003$ | $0.425 \pm_{90\%} 0.004$ |
| cards-4096 | $7.24 \pm_{90\%} 0.01$ | $0.764 \pm_{90\%} 0.004$ | $0.436 \pm_{90\%} 0.005$ |
| pages | $14.03 \pm_{90\%} 0.02$ | $1.722 \pm_{90\%} 0.002$ | $0.585 \pm_{90\%} 0.005$ |

The hot transaction makes insertions to the same set of objects as the last of the warm transactions. Thus, the hot transaction includes no object faults or swizzling, so the page protection scheme is no longer penalized for having to manipulate page protections during swizzling, and therefore achieves performance closer to that of the other schemes — it still incurs page traps when clean pages are modified, and must manipulate page protections at checkpoint time. For the other schemes, there is little to distinguish one from another in Figure 5.3 for the cold through warm transactions.

Differences are better discerned by considering the raw elapsed times given in Table 5.7. For the cold Insert, the pages scheme is significantly more expensive (even than scan) because of the overhead to manage page protections as objects are made resident. Best is remsets closely followed by objects, since neither of these schemes incur any overhead as the resident set of objects grows, whereas the card schemes must grow their tables to cover the expanding set of resident objects. For warmer iterations, checkpoint cost is more important, so the smaller granularity schemes that require no scanning are to be preferred since they focus unswizzling. Notice how these two factors influence the results among the card schemes: moderately large cards (cards-1024) are better when the system is cold, and the resident set of objects grows frequently, since growing the card tables is cheaper; smaller cards are better for the warm (cards-64) and hot (cards-16) iterations where checkpoint overheads dominate.

For a better understanding of the behavior of the hot results, Figure 5.4 shows the breakdown of the elapsed time for each phase of execution of the benchmark:

- *running*: time spent in the interpreter executing the program, as opposed to unswizzling old and new objects to generate differences and writing those differences to the log (note that running includes the cost of noting modifications as they occur);
- *old*: time to unswizzle old modified objects and generate log entries for them;
- *new*: time to unswizzle new objects and generate log entries for them;
- *write*: time to flush the log entries to disk; and
- *other*: time for any remaining bookkeeping activities, such as modifying page protections, and scavenging free transient memory space.

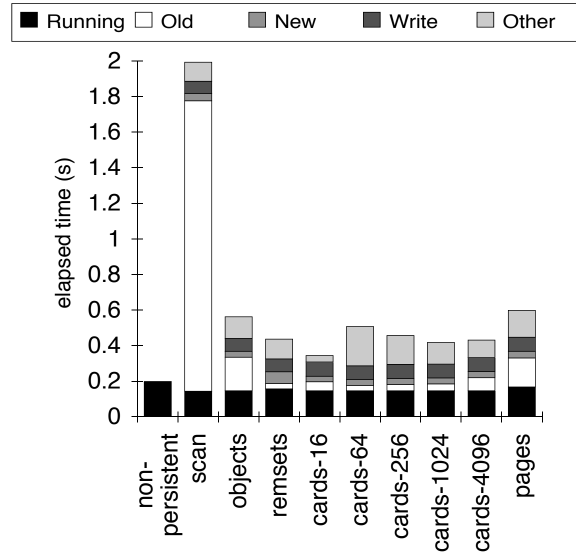



**Figure 5.4** Insert: Hot breakdown

The most interesting feature of Figure 5.4 is the *old* component, which reflects the amount of scanning required to determine the differences between a cached object and its original in the client buffer pool. For the card-based schemes there is an evident tradeoff between the size of the card table and the card size: small cards require more overhead to scan the card table but less overhead in scanning the cards themselves; larger cards have a smaller table, but more overhead to scan the larger cards. Variation among the schemes in the other components are due less to intrinsic costs of the schemes than to subtle underlying cache effects: that the *other* component exhibits such variation is a result of the scavenging of transient space having markedly different cache performance between schemes. Thus, we refrain from further discussion of the results for Insert, and move on to those for the Update benchmark, which offers more precise control of benchmark parameters.

### 5.4.2 Update

Comparison of the results for the implementation alternatives is easier if we consider the key cold, warm, and hot results separately.

#### 5.4.2.1 Cold Update

Figure 5.5 presents the elapsed time for the first (cold) iteration at each of the update probabilities; Figure 5.6 repeats the plot at an expanded scale omitting the results for pages and scan. There is little variation with update probability, since the cold times are dominated by I/O and swizzling costs. Nevertheless, pages is significantly more expensive (worse even than scan) due to the overheads of page protection management, both to trap updates to protected pages and to manipulate page protections during swizzling. Best overall is objects, closely followed by remsets — neither of these schemes incurs any overhead as the resident set of objects grows; in contrast, the card schemes must grow their tables to cover the expanding set of resident objects.

#### 5.4.2.2 Warm Update

At the tenth (warmest) iteration (Figures 5.7 and 5.8), checkpoint cost becomes more important. Worst overall is scan, since it must unswizzle all resident objects to generate the log. Next worst is pages, again because of the overhead to manage page protections. The card schemes are ranked by size, with smaller cards providing more precise information as to which objects are modified. The remsets scheme has performance very close to that of the smaller card schemes, because it concisely records just those objects that are modified.

#### 5.4.2.3 Hot Update

The hot transaction traverses exactly the same parts as the last of the ten cold through warm iterations, by beginning at the same part. Thus, the hot transaction includes no object faults or swizzling. The hot results (Figures 5.9 and 5.10) are similar to those for the warm transaction, except that with all objects needed by the traversal having already been cached, no fetching and swizzling of objects occurs. Thus, the page protection scheme is no longer penalized for having to manipulate page protections during swizzling, and therefore achieves performance closer to that of the page-sized card scheme. The

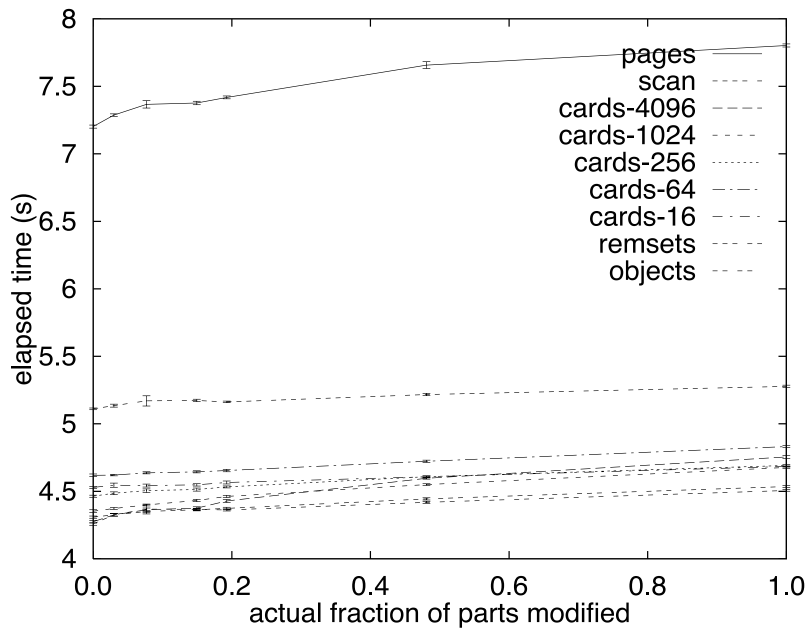



**Figure 5.5** Cold Update

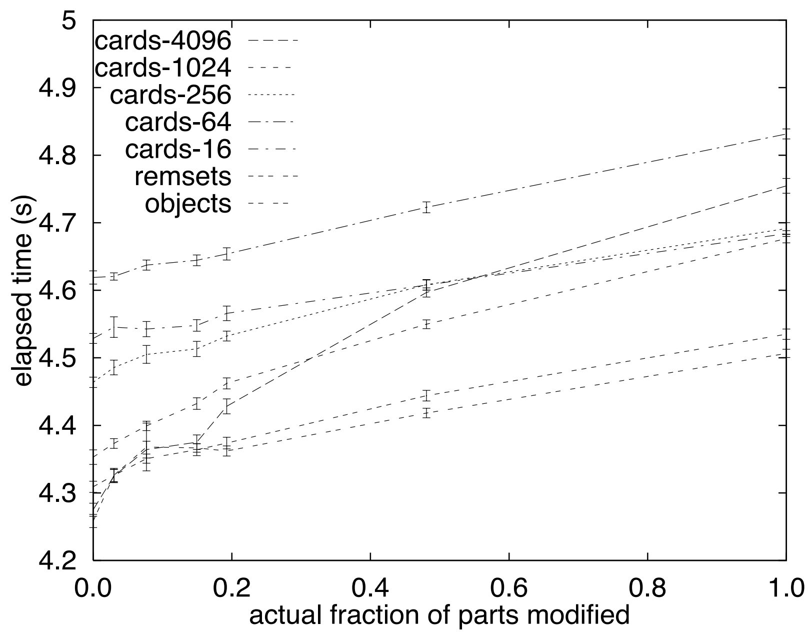



**Figure 5.6** Cold Update (expanded scale)

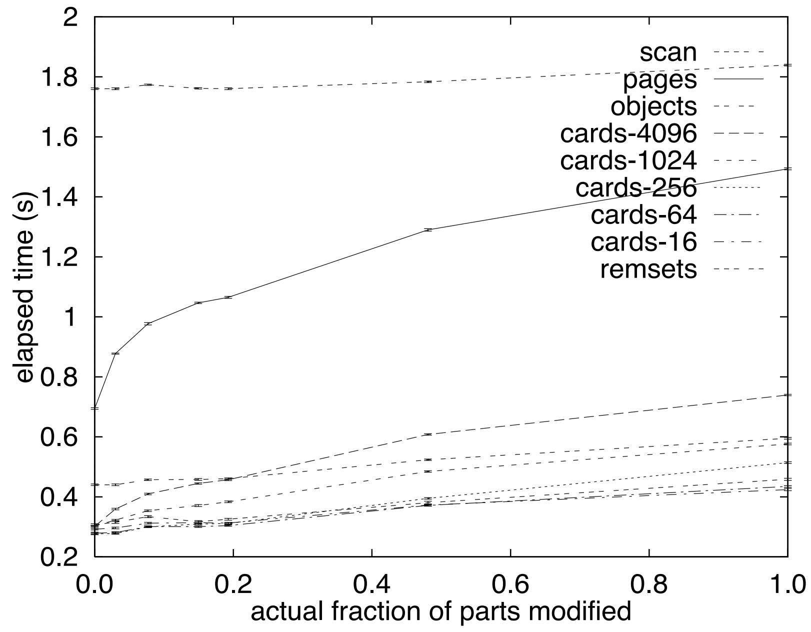



**Figure 5.7** Warm Update

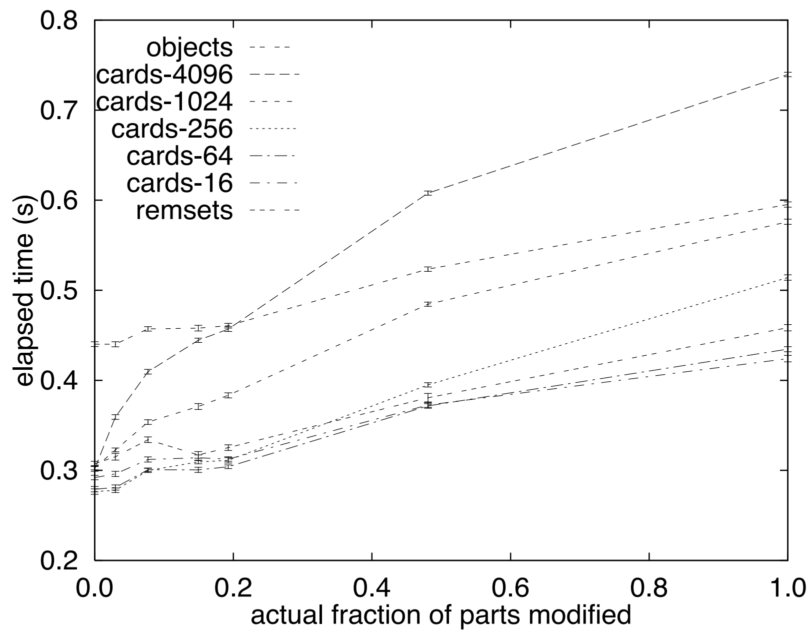



**Figure 5.8** Warm Update (expanded scale)

remaining difference between these two schemes is explained by the need to manipulate the protection of dirty pages at checkpoint time.

The breakdowns of the elapsed time for the hot Update at each update probability ($p$) are plotted in Figures 5.11(a)-(g) (note that the scale changes as update probability increases), omitting the results for scan and non-persistent. As we saw with Insert, checkpoint costs dominate performance, with the *old* component being the decisive difference between schemes, particularly at larger update probabilities. The tradeoff between card table size and card size is once again evident. At the smaller update probabilities the cost of scanning the card table has more influence; schemes with small cards but a larger card table fare worse than larger cards. At higher update probabilities there are more dirty cards to process, so unswizzling overheads dominate those of scanning the card table, with larger cards requiring more unswizzling to generate differences than smaller cards. The tradeoff is most pronounced for the 16-byte cards, which are substantially smaller than the average object size, so that unswizzling costs outweigh card table scanning costs only at the higher update probabilities. Overall, remembered sets offer the most concise record of updates, allowing modified objects to be unswizzled without scanning. The scanning overhead is clearly evident for the object marking scheme, especially at low update probabilities, where objects has the worst performance.

Apart from the pages scheme, variation among the schemes in the *running* component is not significant, indicating that checkpoint overheads are the dominating influence for this short transaction benchmark. The pages scheme does incur significant run-time overhead because of the high cost of the traps to note updates. Note that although no *part* objects are modified for $p = 0.0$, some updates do occur to other objects in the system, which cause page traps.

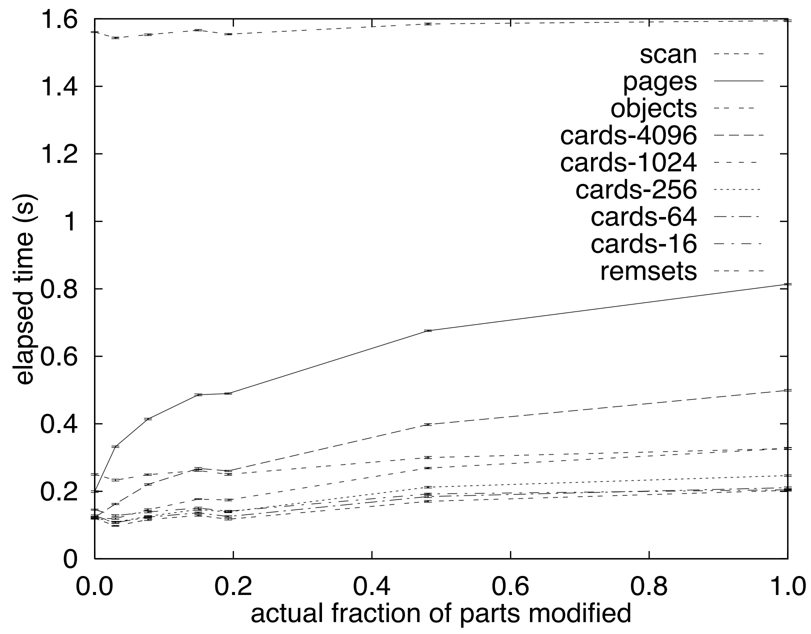



**Figure 5.9** Hot Update

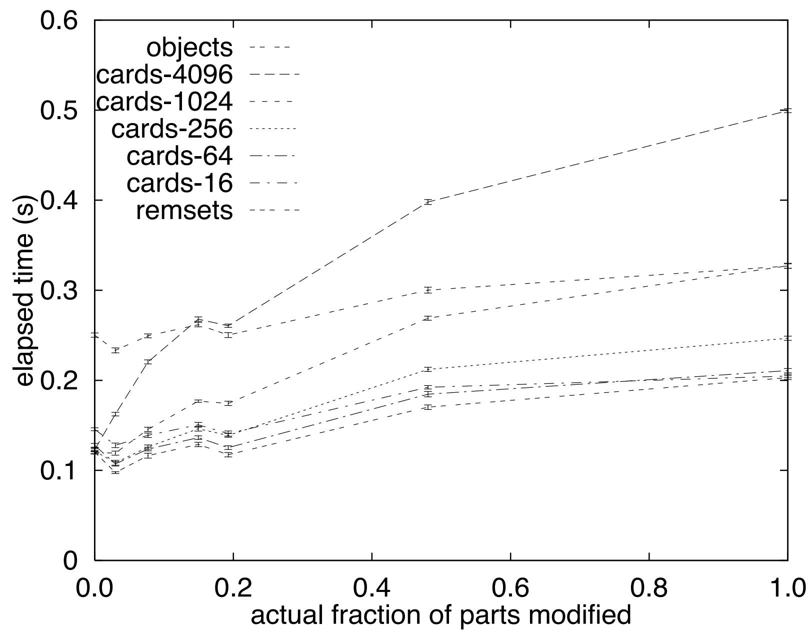



**Figure 5.10** Hot Update (expanded scale)

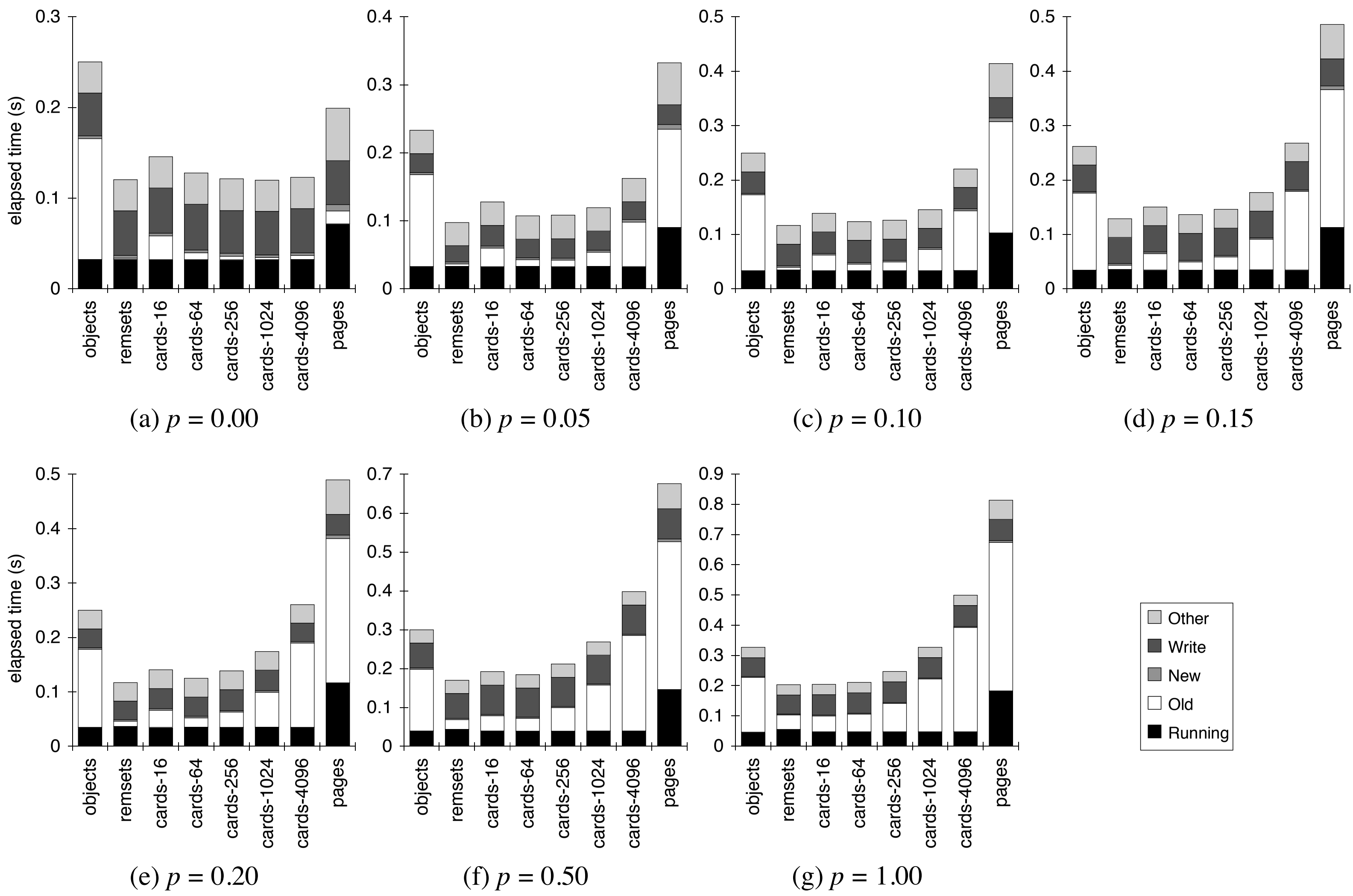



**Figure 5.11** Update: Hot breakdown

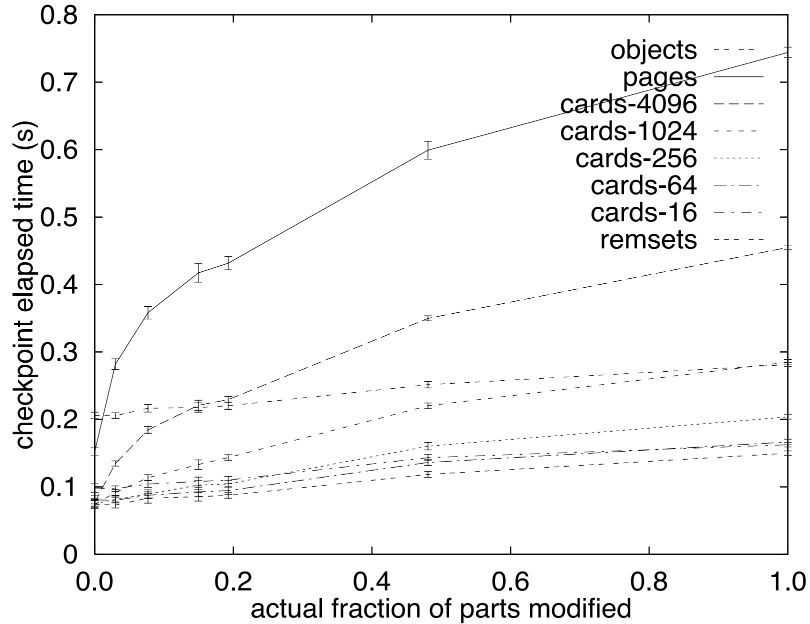



**Figure 5.12** Long-running Update: checkpoint overhead per transaction

5.4.2.4 Long Transactions

Run-time overheads come into play only when transactions are long enough for computation to dominate checkpoint overhead. The final set of results concerns the experiments in which multiple hot update traversals are performed as a single transaction. We generalize the results, by obtaining linear regression fits for each scheme, for the model: $y = a + bx$, where $y$ is the total elapsed time, and $x$ the number of update traversals per transaction. As expected, since a hot traversal will have constant cost no matter how many times it is performed, the fits are excellent. The $y$-axis intercept, $a$, approximates the checkpoint overhead per transaction, and has the familiar form we have seen for short-running transactions (see Figure 5.12).

The slope $b$ is a measure of the long-running per-traversal costs of each scheme, plotted in Figure 5.13(a). The remsets scheme has the highest overhead to note updates. The card schemes are clustered together in the mid-range of overhead, while objects has the least measured overhead apart from scan. Curiously, pages has the second worst measured overhead per update, even though it incurs no additional overhead for subsequent updates

to a page after it is first modified. The overhead to field these page protection traps is thus constant for each value of $x$ in the regression, so the trap overhead is extracted as a component of the $a$ coefficient.

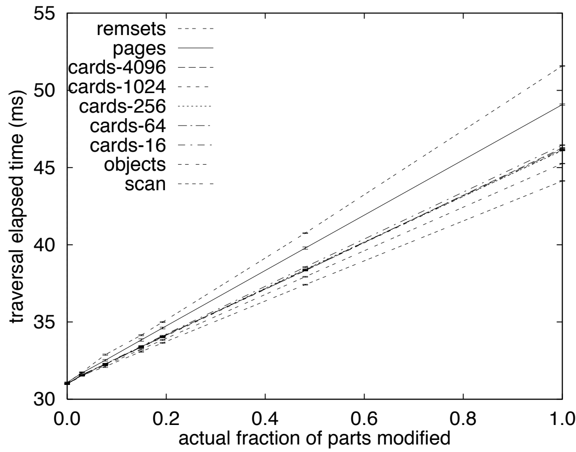
As it turns out, the pages scheme is the victim of anomalous hardware cache behavior. Indeed, taking per-traversal instruction references as our measure (plotted in Figure 5.13(b)), pages has overhead equivalent to that of scan. The anomaly is revealed in Figures 5.13(c) and (d), where we see that there is contention between data and instructions for cache lines in the unified instruction and data cache of the SPARCstation 2. Subsequent inspection of the cache simulation results indicates that the contention is restricted to a single instruction and data location, both of which are accessed for each part modified in the Update traversal — the slopes of the lines for pages in Figures 5.13(c) and (d) are approximately 1 miss per modified part. There is a similar anomaly for remsets, but only at update probabilities 0.15 and 0.2.

Taking linear regressions for the per-traversal elapsed time and instruction references, versus the actual number of parts modified (i.e., the slopes of the lines in Figures 5.13(a) and (b)), we obtain a measure of the per-update run-time overhead for each scheme, given in Table 5.8. Taking the non-persistent/scan results as the base level of overhead required to perform an unrecorded update, we calculate the intrinsic update overhead that can be attributed to each of the schemes as:

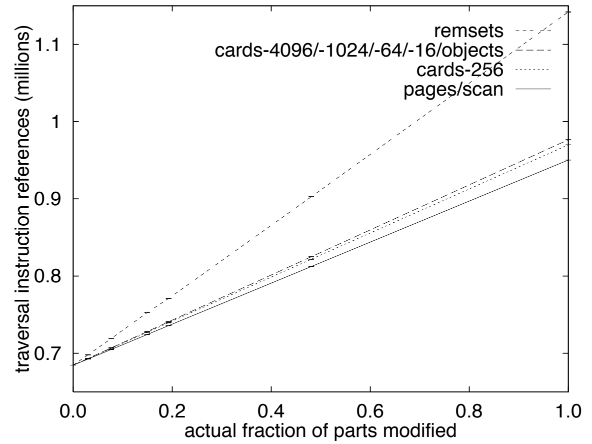
$$\text{intrinsic overhead per update} = \frac{\text{overhead per part} - \text{scan overhead per part}}{2}$$

The division by 2 reflects the fact that modifying a part actually consists of two updates: one to increment the $x$ attribute, and one to increment the $y$ attribute. These attributable per-update overheads are given in Table 5.9. The adjusted pages result is obtained by subtracting 25 cycles to correct for the anomalous 2 read misses per modified part incurred by the pages scheme ($\frac{2\times 25}{2} = 25$ cycles).

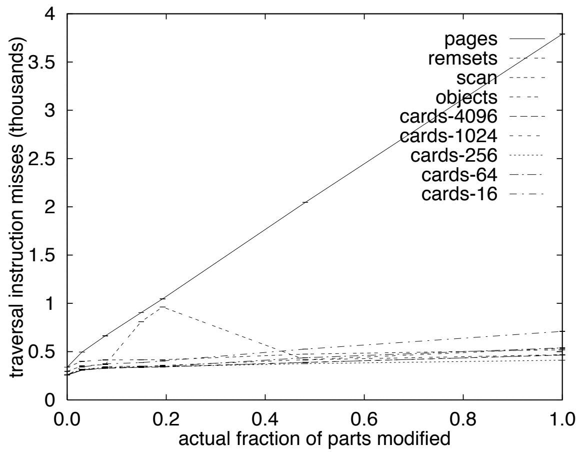
As validation of these results, the observed instruction overheads for the different schemes correspond to the actual instruction overheads revealed upon inspection of the code



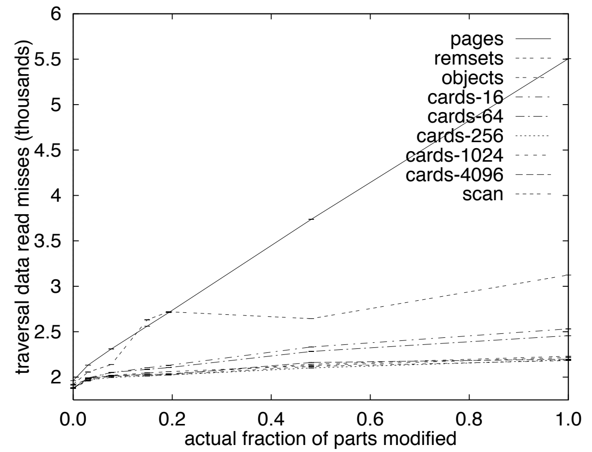
(a) Elapsed time



(b) Instruction references



(c) Instruction misses



(d) Data read misses

**Figure 5.13** Traversal overheads

generated by the compiler. Indeed, the cards-256 scheme requires only three instructions instead of the four used for the other card sizes because of shared code with the write barrier for the generation scavenging garbage collector, which also uses 256-byte cards.

These results precisely measure the run-time overheads of each of the schemes. The page protection scheme (pages) offers the least overhead per update of all the schemes that record updates (i.e., apart from scan and non-persistent), since each transaction entails many repeated updates to the same locations, so that only the first update to a location causes a page trap. Remaining updates proceed with no additional overhead. Note that the remaining 5 cycles overhead for pages is likely anomalous, unless it indicates lingering cache disturbance due to page protection traps. Meanwhile, the software-mediated card schemes show only marginally higher overhead, while the remsets scheme incurs the high cost of a call to hash the updated location into the remembered set on every update.

Finally, we also determine the break-even point between pages and cards-4096 for the hot Update benchmark, as the number of Update traversals, $t$, required for the up-front cost of pages (incurred in managing page protections at checkpoint time as well as the per-transaction run-time cost of the page traps) to equal the per-update run-time costs of cards-4096. This is calculated by taking their difference in checkpoint overhead per transaction (as plotted in Figure 5.12), say $p - c$, and dividing by the cards-4096 cost per update (from Table 5.9), $\frac{13 \text{ cycles}}{40\text{MHz}}$, times the number of updates per traversal, $2m$ where $m$ is the number of parts modified per traversal:

$$t = \frac{p - c}{\frac{13}{40000000} \times m \times 2}$$

The calculations are summarized in Table 5.10. Note how the frequency/density of update affects the tradeoff between the high per-transaction cost of pages versus the run-time costs of cards-4096: amortization of the up-front per-transaction overheads occurs with fewer Update traversals for higher update probabilities. The high break-even points show that pages is preferable only in extreme cases, when transactions are particularly long or updates extremely frequent and dense.

**Table 5.8** Long-running Update: traversal overheads (per part modified)

| Scheme | Time (cycles) | Instructions |
|---|---|---|
| non-persistent | 159$\pm_{90\%}$0 | 81$\pm_{90\%}$0 |
| scan | 159$\pm_{90\%}$1 | 81$\pm_{90\%}$0 |
| objects | 173$\pm_{90\%}$1 | 89$\pm_{90\%}$0 |
| remsets | 249$\pm_{90\%}$4 | 139$\pm_{90\%}$1 |
| cards-16 | 188$\pm_{90\%}$2 | 89$\pm_{90\%}$0 |
| cards-64 | 185$\pm_{90\%}$1 | 89$\pm_{90\%}$0 |
| cards-256 | 183$\pm_{90\%}$2 | 87$\pm_{90\%}$0 |
| cards-1024 | 184$\pm_{90\%}$1 | 89$\pm_{90\%}$0 |
| cards-4096 | 185$\pm_{90\%}$2 | 89$\pm_{90\%}$0 |
| pages | 219$\pm_{90\%}$1 | 81$\pm_{90\%}$0 |

**Table 5.9** Long-running Update: intrinsic run-time overheads (per update)

| Scheme | Time (cycles) | Instructions |
|---|---|---|
| non-persistent | 0$\pm_{90\%}$0 | 0$\pm_{90\%}$0 |
| scan | 0$\pm_{90\%}$1 | 0$\pm_{90\%}$0 |
| objects | 7$\pm_{90\%}$1 | 4$\pm_{90\%}$0 |
| remsets | 45$\pm_{90\%}$2 | 29$\pm_{90\%}$1 |
| cards-16 | 15$\pm_{90\%}$1 | 4$\pm_{90\%}$0 |
| cards-64 | 13$\pm_{90\%}$1 | 4$\pm_{90\%}$0 |
| cards-256 | 12$\pm_{90\%}$1 | 3$\pm_{90\%}$0 |
| cards-1024 | 13$\pm_{90\%}$1 | 4$\pm_{90\%}$0 |
| cards-4096 | 13$\pm_{90\%}$1 | 4$\pm_{90\%}$0 |
| pages (raw) | 30$\pm_{90\%}$1 | 0$\pm_{90\%}$0 |
| pages (adjusted) | 5$\pm_{90\%}$1 | 0$\pm_{90\%}$0 |

$$\text{intrinsic overhead per update} = \frac{\text{overhead per part} - \textsf{scan}\ \text{overhead per part}}{2}$$

**Table 5.10** Long-running Update: break-even points for cards-4096 versus pages

| update probability | checkpoint overhead (seconds) pages ($p$) | cards-4096 ($c$) | parts modified per traversal ($m$) | break-even point (traversals, $t$) |
|---|---|---|---|---|
| 0.00 | 0.152$\pm_{90\%}$0.006 | 0.077$\pm_{90\%}$0.006 | 0 | $\infty$ |
| 0.05 | 0.282$\pm_{90\%}$0.008 | 0.135$\pm_{90\%}$0.004 | 98 | 2308$\pm_{90\%}$ 366 |
| 0.10 | 0.358$\pm_{90\%}$0.009 | 0.184$\pm_{90\%}$0.005 | 252 | 1062$\pm_{90\%}$ 167 |
| 0.15 | 0.42 $\pm_{90\%}$0.01 | 0.221$\pm_{90\%}$0.008 | 490 | 625$\pm_{90\%}$ 105 |
| 0.20 | 0.43 $\pm_{90\%}$0.01 | 0.229$\pm_{90\%}$0.005 | 632 | 489$\pm_{90\%}$ 74 |
| 0.50 | 0.60 $\pm_{90\%}$0.01 | 0.350$\pm_{90\%}$0.004 | 1577 | 244$\pm_{90\%}$ 32 |
| 1.00 | 0.744$\pm_{90\%}$0.008 | 0.455$\pm_{90\%}$0.003 | 3280 | 136$\pm_{90\%}$ 16 |

$$t = \frac{p - c}{\frac{13}{40000000} \times m \times 2}$$

### 5.4.3 Conclusions

The results for the detection of updates and checkpointing show a clear ranking among the alternative schemes, with approaches that record updates at smaller granularities having a significant advantage when the transactions are short and the update locality is poor, since they greatly reduce the overheads of unswizzling and generation of differences for the log. For short transactions, the remembered set scheme is best over all update probabilities, since it provides a very concise summary of just those objects that have been modified. Still, small granularity cards also offer robust performance across a range of update probabilities, and have the advantage of lower and precisely bounded run-time overhead.

For longer intervals between checkpoints, the run-time costs of update detection come into play. Thus, the remembered set scheme loses its appeal due to the relatively high expense of managing the remembered set. The page protection scheme has the advantage that detection overhead is paid for up front in the page protection violation trap on the first write to a clean page, and subsequent updates proceed without cost. Meanwhile, the overheads of the card and object marking schemes change very little as update probability varies, with any difference being due to hardware cache effects. Even so, the differences in run-time overheads of the schemes are slight when compared to those of checkpointing.

The length of the interval between checkpoints is an important factor because of this tension between the run-time and checkpointing overheads of the various schemes. Long intervals between checkpoints are likely to result in correspondingly more updates, increasing the checkpoint latency. Only when the volume of modified data is small with respect to the length of time between checkpoints should the run-time costs of the schemes be permitted to guide the choice of update detection mechanism. The overwhelming influence of unswizzling and generation of log records indicates that the general bias should be towards the more accurate smaller granularities than to schemes with low run-time overheads.

With respect to the hardware approach embodied in the page protection scheme we have seen that it can involve substantial extra overhead for "typical" operations as represented

by the benchmarks. In the abstract, the hardware approach is an attractive one. However, current realizations which must use expensive calls to the operating system seem to be limited in their effectiveness. Moreover, the large granularity of page size remains the most serious deficiency of this scheme, even if improved operating system support can succeed in lowering the costs of managing the update information through access to page dirty bits (Hosking and Moss [1993] explore the ramifications of such support in more detail).

We offer three guidelines for the generation of recovery information in persistent programming languages:

- Avoid large granules of update detection, to minimize checkpoint overheads.
- Choose a checkpoint frequency corresponding to the rate of generation of new update information, so that checkpoint delays are tolerable. Long-running transactions that perform few updates need infrequent checkpoints.
- Use page protection mechanisms only where update locality is good and checkpoints are infrequent.

## 5.5 Related Performance Studies

This study is loosely related to that of White and DeWitt [1992] (mentioned earlier in Chapter 2), which compares the overall performance of various object faulting and pointer swizzling schemes for C++, as supported by several different persistent object stores, including ObjectStore and the EXODUS Storage Manager. While our basic architecture is similar to the object caching scheme of White and DeWitt, the thrust of our study is significantly different. Instead of comparing several different architectures for persistence we keep the architecture fixed, while varying the mechanisms to fault objects and generate log information. The representations we use for references to non-resident objects are much more lightweight than those of White and DeWitt, as are our mechanisms to support generation of the log.

Nevertheless, White and DeWitt's results do suggest that the method used to generate recovery information can have a significant impact on the performance of the system, with

fine-grained update information being most beneficial when transactions are short and there is poor update locality. Similarly, they note the high overhead of faulting objects using a page protection scheme like ObjectStore or Texas. We have explored these issues directly here, addressing the specific question of which mechanisms are best, and what factors determine a method's effectiveness, within the fixed framework of a single persistent programming language.

In a subsequent study, White and DeWitt [1994] extend ESM to support a memory-mapped style of access like that of ObjectStore and Texas. They compare their system, called QuickStore, with the software-mediated faulting scheme of E (using the OO7 benchmarks). Once again, although their results do reveal the high cost of managing page mappings, the comparison is between two different systems, which perform faulting and swizzling quite differently, while our experimental framework operates within the same basic persistent system, where everything but the primitive faulting and logging mechanisms are fixed. For future work, White and DeWitt [1994] speculate on the advantages to be gained in augmenting the language implementation to generate log information instead of relying on QuickStore's memory mapping mechanisms to generate per-page difference records. This is precisely what we have done here with our studies of different mechanisms for noting updates for logging.

# CHAPTER 6

# CONCLUSIONS

We have explored the lightweight implementation of two key mechanisms for fine-grained persistence: the detection and handling of references to non-resident objects, and the efficient generation of recovery information. Our experiments used recognized benchmarks to compare the performance of alternative realizations of these mechanisms, within a prototype persistent programming language. Most importantly, we have demonstrated that software-mediated techniques can be a competitive alternative to hardware-assisted techniques, and that adding persistence to a programming language does not necessarily degrade its performance. Moreover, the results show that persistence can be implemented on general-purpose machines without imposing significant overhead above and beyond the fundamental costs of data transfer to and from secondary storage. In fact, our results indicate that performance differences between non-persistent and persistent languages are reducible to mere differences in hardware cache performance between the persistent and non-persistent implementation.

In addition, the performance evaluation of Chapter 5 represents the first comprehensive exploration of implementation alternatives for persistence within a single prototype system, where intrinsic differences between the alternatives are more clearly discerned. The experimental methodology is (to our knowledge) unique, and allows precise measurement of the low-level costs of the mechanisms being studied.

## 6.1 Implications

These results have implications beyond the realm of persistence. At a general level, the results are an indictment of the performance of operating system virtual memory primitives.

Both the overhead to field page protection violations within user-level signal handlers, and the high cost of calls to the operating system to modify page protections mean that applications relying on these primitives pay unnecessarily high overheads. Much of the problem lies with the large granularity (relative to fine-grained objects) of virtual memory pages in modern operating systems and architectures, which can significantly affect overall performance. As processor speeds improve, and physical memories grow, page sizes are likely to become larger, further degrading the performance of virtual memory solutions in applications that have a naturally smaller granularity.

It is worthwhile noting that similar results have been obtained in other very different application areas, such as efficient implementation of data breakpoints for debuggers [Wahbe, 1992] and generational garbage collection [Hosking et al., 1992; Hosking and Moss, 1993]. We acknowledge that sub-page protection and dirty bits, along with appropriate operating system interfaces, might somewhat overcome the performance disadvantages we observed (cf. Thekkath and Levy [1994]). However, it is clear that while user level virtual memory primitives offer transparent solutions to various memory management problems, requiring no modification of the programming language implementation (except in the run-time to handle protection violations) they do not *necessarily* offer the best performance.

## 6.2 Future Work

There are several promising directions for extension of this work: examination of the issues in a compiled setting; alternative architectures for persistence; language and implementation support for coarse-grained persistent data and large collections of data; and support for distribution and concurrency.

### 6.2.1 Compilation Versus Interpretation

The fact that the results have been obtained for an interpreted language cannot be taken lightly, since run-time overheads for interpretation are several times higher than those of

compiled programs. Nevertheless, we see no reason why the results will not carry over to a compiled setting. We acknowledge that compilation will shrink the running-time portion of total execution, so that the overheads of object faulting and update noting will become more pronounced *relative* to total execution time. However, the various mechanisms should retain their rankings with respect to one another, since their absolute costs will remain the same (modulo shifting and possibly spurious hardware cache effects).

Some languages (e.g., Modula-3 [Nelson, 1991; Harbison, 1992], C++ [Stroustrup, 1986]) do not enforce the pure object-oriented style of execution that enables residency checks to be piggy-backed upon method invocation. Operations on an object can be performed without necessarily invoking a method on it. This means that explicit residency checks must be compiled into the code in advance of such operations, to ensure that the object is resident. Compiler optimizations [Richardson and Carey, 1990; Richardson, 1990; Hosking and Moss, 1990; Hosking and Moss, 1991; Moss and Hosking, 1994] may allow these explicit residency checks to be merged or eliminated. For example, control-flow information may reveal that multiple traversals of a particular object reference along a given execution path require only one residency check, rather than a check per traversal. Similar optimizations may merge or eliminate the noting of updates at certain store sites. These optimizations will have the effect of reducing the importance of run-time overheads of the software-based schemes, so that swizzling, disk retrieval, and checkpoint overheads become the dominant factor influencing the choice of object faulting and update detection scheme. The effect of such optimizations in a compiled setting is an interesting avenue of further research.

### 6.2.2 Other Architectures

We acknowledge that our architectural framework is one of several that might be considered. For example, we have chosen a copy swizzling approach, whereas it may be possible for applications to manipulate objects directly in the client buffer pool. Consider-

ation of such alternatives would further our understanding of the relative merits of different architectures.

We also recognize that our hardware-mediated object faulting approach, which allocates fault blocks in protected pages, does not compare directly with the memory-mapped file approaches of ObjectStore and Texas. In particular, ObjectStore makes some effort to allow objects to be mapped directly into the same memory locations in which they were originally allocated, thus reducing or eliminating the need to swizzle pointers upon object fault. While our results stand as a relative comparison of object faulting techniques within a fixed architectural framework, and it is reasonable to assume that the relative standing of our fault detection mechanisms will remain the same even if the underlying persistent object storage architecture changes, side-by-side comparison with direct-mapped approaches would still be interesting.

To some extent, this is the focus of the efforts of White and DeWitt [1992] and White and DeWitt [1994]. However, devising an experimental framework in which irrelevant differences in the alternative architectures are abstracted away, so as to highlight their intrinsic characteristics, poses a challenge that has yet to be fully addressed.

### 6.2.3 Higher-Level Functionality

Finally, there are several features of database systems that have not been explored in this work. Here, we have focused on the navigational style of data access supported by programming languages. Database systems also typically support query-oriented associative access over collections of data — providing efficient support for this alternative access style, and integrating it within our framework for persistence is a significant open problem. Similarly, new applications demand support for accessing coarse-grained data significantly different from the fine-grained objects we consider here. For example, multimedia databases incorporate text, images, and audio/video streams, all of which are relatively coarse-grained.

Supporting the management of such data within a persistent programming language seems a ripe research problem.

Other aspects of database systems that we have not considered here are support for concurrency and distribution. Allowing simultaneous manipulation of the database by concurrently executing persistent programs requires concurrency control mechanisms to ensure that the database remains in a consistent state. We have argued that several approaches to concurrency control can certainly be incorporated into our architecture, but concurrency control also requires that we incorporate some form of transaction semantics in the persistent programming language — but there remain many interesting questions as to what those transaction semantics should be, how they should be presented to the programmer, and what their performance impact will be. Distribution is also a logical extension of this work, since there are synergies to be exploited in the way we represent references to non-resident objects and the way distributed systems must address objects distributed over many machines in a network.

## 6.3 Closing Remarks

Persistence promises to revolutionize the way in which programmers think about long-term data storage solutions. This dissertation opens the way to persistence becoming ubiquitous in programming languages and systems, by demonstrating that persistence is a feasible, maturing technology, with competitive performance, deserving of inclusion in the canon of desirable language features.

# APPENDIX A

# SMALLTALK SOURCE LISTING OF OO1 BENCHMARKS

The following source code listing gives the Smalltalk code for the OO1 benchmark database and operations used in the experiments of Chapter 5.

# BTree

| | |
|---|---|
| class name | **BTree** |
| superclass | **SequenceableCollection** |
| instance variable names | **tally children** |
| class variable names | *none* |
| pool dictionaries | *none* |
| category | **Benchmarks-OO1** |

*I am a minimal (insert-only) implementation of a B⁺-tree index. My instances are nodes in the tree.*

*Instance variables:*

| | |
|---|---|
| *tally* | *<SmallInteger> The number of keyed objects stored in this node.* |
| *children* | *<Array \| Nil> The children of this node.* |

*Protocol for accessing*

**add: anObject**

```
"Insert anObject into the BTree rooted at myself"

| new newChildren |
self isFull
    ifTrue:
        ["Split the root"
        new ← self species new: self minDegree.
        1 to: tally do: [:i | new basicAt: i put: (self basicAt: i)].
        new tally: tally.
        new children: children.
        tally ← 0.
        self setChildren at: 1 put: new.
        self splitChild: 1].
↑self insert: anObject
```

**at: anInteger**

```
"Access the nth element of the BTree. This allows the BTree to be
enumerated like any other SequenceableCollection"

| index tree i child delta |
index ← anInteger.
tree ← self.
i ← 0.
```

```
    [tree children notNil & (i <= tree tally)]
        whileTrue:
            [i ← i + 1.
            child ← tree children at: i.
            delta ← index − child size − 1.
            delta < 0
                ifTrue:
                    [i ← 0.
                    tree ← child].
            delta = 0
                ifTrue:
                    [index ← i.
                    i ← tree tally + 1].
            delta > 0 ifTrue: [index ← delta]].
    index > tree tally ifTrue: [self errorSubscriptBounds: anInteger].
    ↑tree basicAt: index
```

**at: anInteger put: anObject**

```
    "Storing into a BTree with at:put: is not allowed."

    self error: 'to add to a BTree, you must use add:'
```

**findKeyOrNil: key**

```
    | index element |
    index ← self indexForInserting: key.
    index > 0
        ifTrue:
            [element ← self basicAt: index.
            key = element key ifTrue: [↑element]].
    children isNil ifTrue: [↑nil].
    ↑(children at: index + 1)
        findKeyOrNil: key
```

**size**

```
    | size |
    size ← tally.
    children notNil ifTrue: [1 to: tally + 1 do: [:i | size ← size + (children at: i) size]].
    ↑size
```

*Protocol for copying*

**shallowCopy**

```
    "Answer a copy of the receiver which shares the same elements; in
    this case keyed objects"
```

```
| copy copyChildren |
(copy ← self species new: self minDegree) tally: tally.
1 to: tally do: [:i | copy basicAt: i put: (self basicAt: i)].
children notNil
    ifTrue:
        [copyChildren ← copy setChildren.
        1 to: children size do:
            [:i | copyChildren at: i put: (children at: i) shallowCopy]].
↑copy
```

*Protocol for enumerating*

**do: aBlock**

```
"Evaluate aBlock with each of the elements of the BTree rooted at
myself"

| index |
index ← 0.
children notNil
    ifTrue:
        [[(index ← index + 1) <= tally]
            whileTrue:
                [(children at: index)
                    do: aBlock.
                aBlock value: (self basicAt: index)].
        (children at: index)
            do: aBlock]
    ifFalse: [[(index ← index + 1) <= tally]
            whileTrue: [aBlock value: (self basicAt: index)]]
```

*Protocol for printing*

**printOn: aStream**

```
aStream nextPutAll: self class name, ' '.
self printOn: aStream limit: aStream position + self maxPrint
```

**printOn: aStream limit: limit**

```
| index |
aStream nextPut: $(.
index ← 0.
children notNil
    ifTrue:
        [[(index ← index + 1) <= tally] whileTrue:
            [(children at: index) printOn: aStream limit: limit.
            aStream space.
```

```
                aStream position > limit ifTrue: [aStream nextPutAll: '...etc...)'. ↑self].
                (self basicAt: index) key printOn: aStream.
                aStream space].
            (children at: index) printOn: aStream limit: limit.
            aStream space]
        ifFalse:
            [[(index ← index + 1) <= tally] whileTrue:
                [aStream position > limit ifTrue: [aStream nextPutAll: '...etc...)'. ↑self].
                (self basicAt: index) key printOn: aStream.
                aStream space]].
    aStream nextPut: $)
```

*Protocol for private*

**children**

```
    ↑children
```

**children: anArray**

```
    ↑children ← anArray
```

**indexForInserting: key**

```
    "Find the index of the child in which to insert key"

    | low high index |
    low ← 1.
    high ← tally.

    [index ← high + low bitShift: −1.
    low > high]
        whileFalse: [(self basicAt: index) key <= key
                ifTrue: [low ← index + 1]
                ifFalse: [high ← index − 1]].
    ↑high
```

**insert: anObject**

```
    "Insert anObject into the BTree rooted at myself"

    | index |
    index ← self indexForInserting: anObject key.
    index > 0 ifTrue:
        [(self basicAt: index) key = anObject key ifTrue:
            [self error: 'Attempting to insert object with duplicate key']].
    index ← index + 1.
    children isNil
        ifTrue:
```

```
            [tally
                to: index
                by: −1
                do: [:i | self basicAt: i + 1 put: (self basicAt: i)].
            tally ← tally + 1.
            ↑self basicAt: index put: anObject].
    (children at: index) isFull
        ifTrue:
            [self splitChild: index.
            anObject key > (self basicAt: index) key ifTrue: [index ← index + 1]].
    ↑(children at: index)
        insert: anObject
```

**isFull**

```
    ↑tally = self basicSize
```

**maxDegree**

```
    ↑self basicSize + 1
```

**minDegree**

```
    ↑self maxDegree // 2
```

**setChildren**

```
    ↑children ← Array new: self maxDegree
```

**setTally**

```
    tally ← 0
```

**splitChild: index**

*"Split the child at index"*

```
    | minDegree oldChild newChild oldChildren newChildren |
    minDegree ← self minDegree.
    (oldChild ← children at: index) tally: minDegree − 1.
    (newChild ← self species new: minDegree) tally: minDegree − 1.
    1 to: minDegree − 1 do:
        [:i | newChild basicAt: i put: (oldChild basicAt: minDegree + i)].
    (oldChildren ← oldChild children) notNil
        ifTrue:
            [newChildren ← newChild setChildren.
            1 to: minDegree do:
                [:i | newChildren at: i put: (oldChildren at: minDegree + i)]].
    tally + 1
        to: index + 1
        by: −1
        do: [:i | children at: i + 1 put: (children at: i)].
```

```
    children at: index + 1 put: newChild.
    tally
        to: index
        by: −1
        do: [:i | self basicAt: i + 1 put: (self basicAt: i)].
    self basicAt: index put: (oldChild basicAt: minDegree).
    tally ← tally + 1
```

**tally**

```
    ↑tally
```

**tally: anInteger**

```
    ↑tally ← anInteger
```

# BTree class

| | |
|---|---|
| class name | **BTree class** |
| superclass | **SequenceableCollection class** |
| instance variable names | *none* |
| class variable names | *none* |
| pool dictionaries | *none* |
| category | **Benchmarks-OO1** |

*Protocol for instance creation*

**new**

↑self new: 2

**new: minDegree**

↑(super new: 2 * (minDegree max: 2) − 1) setTally

# OO1

| | |
|---|---|
| class name | **OO1** |
| superclass | **Object** |
| instance variable names | **random parts lastId traverseIds insertConnections** |
| class variable names | *none* |
| pool dictionaries | *none* |
| category | **Benchmarks-OO1** |

*I am an OO1 database.*

*Instance variables:*

| | |
|---|---|
| *random* | *<Random> Random number generator used to generate the benchmark database and run the benchmarks.* |
| *parts* | *<BTree> BTree index for the parts database, indexed by part identifier.* |
| *lastId* | *<SmallInteger> The highest (most recently) assigned part identifier.* |
| *traverseIds* | *<Array> The randomly-selected parts to be used as the starting point for each iteration (cold through warm) of the traversal benchmarks.* |
| *insertConnections* | *<Array> The randomly-selected parts for connection to the newly-added parts in the insertion benchmark.* |

Protocol for initialize-release

**build: numParts**

```
    "Build a new OO1 database, containing numParts parts"

    | firstId |
    random ← Random new.

    "Parts index is a BTree with minimum degree 500; this value was chosen to give
approximately one BTree node per Mneme physical segment (32K PSEGS / 68 byte parts
= 482)"
    parts ← BTree new: 500.

    "Insert numParts parts"
    lastId ← 0.
    numParts timesRepeat: [parts add: (Part new id: (lastId ← lastId + 1))].
```

```
"Make clustered random connections from each part to 3 other parts"
parts do:
    [:part |
    part
        connectTo: (parts findKeyOrNil: (self randomConnection: part id))
        to: (parts findKeyOrNil: (self randomConnection: part id))
        to: (parts findKeyOrNil: (self randomConnection: part id)).
    "Initialize build field of part with a random integer in the range [0, 99].
    This determines which parts are modified by the update benchmark."
    part build: (self randomIntRange: 0 to: 99)].

"Precompute the 10 random parts to be used as starting points in the
cold through warm traversals"
traverseIds ← Array new: 10.
1 to: 10 do: [:i | traverseIds at: i put: (self randomIntRange: 1 to: lastId)].

"Precompute the 3 clustered random connections for each of the 100
parts added to the database in the 10 cold through warm insertions"
insertConnections ← ReadWriteStream on: (Array new: 1000).
firstId ← lastId.
1000
    timesRepeat:
        [lastId ← lastId + 1.
        insertConnections nextPut: (Array
                with: (self randomConnection: lastId)
                with: (self randomConnection: lastId)
                with: (self randomConnection: lastId))].
insertConnections reset.
lastId ← firstId
```

*Protocol for benchmarks*

**btraverse**

```
"Run the 10 cold through warm backward traversals, followed by
several hot traversals varying the number of traversals per
measurement"

| id |
1 to: 10 do:
    [:i |
    id ← traverseIds at: i.
    self time: [self btraverse: id]].
#(0 1 5 10 15 20 50 100 500 ) do:
    [:i | self time: [i timesRepeat: [self btraverse: id]]]
```

**btraverse: fromId**

*"Do a backward traversal starting from the part with identifier fromId"*

```
(parts findKeyOrNil: fromId)
    btraverse: 7
```

**ftraverse**

*"Run the 10 cold through warm forward traversals, followed by several hot traversals varying the number of traversals per measurement"*

```
| id |
1 to: 10 do:
    [:i |
    id ← traverseIds at: i.
    self time: [self ftraverse: id]].
#(0 1 5 10 15 20 50 100 500 ) do: [:i | self time: [i timesRepeat: [self ftraverse: id]]]
```

**ftraverse: fromId**

*"Do a forward traversal starting from the part with identifier fromId"*

```
(parts findKeyOrNil: fromId)
    ftraverse: 7
```

**insert**

*"Run the 10 cold through warm insertions, followed by a hot insertion where the targets of the new connections are already resident"*

```
Smalltalk commit.
10 timesRepeat: [self
        time:
            [self insert100.
            Smalltalk commit]].
insertConnections position: 900.
self
    time:
        [self insert100.
        Smalltalk commit]
```

**insert100**

*"Insert 100 new parts, with precomputed clustered random connections "*

```
| toIds |
100
    timesRepeat:
```

```
        [toIds ← insertConnections next.
        (parts add: (Part new id: (lastId ← lastId + 1)))
            connectTo: (parts findKeyOrNil: (toIds at: 1))
            to: (parts findKeyOrNil: (toIds at: 2))
            to: (parts findKeyOrNil: (toIds at: 3))]
```

**lookup**

```
    "Run the 10 cold through warm lookups, followed by
    a hot lookup where the targets of the lookup are guaranteed all to
    be resident"

    | ids |
    10
        timesRepeat:
            [ids ← ReadWriteStream on: (Array new: 1000).
            1000 timesRepeat: [ids nextPut: (self randomIntRange: 1 to: lastId)].
            ids reset.
            self time: [self lookup: ids]].
    ids reset.
    self time: [self lookup: ids]
```

**lookup: ids**

```
    "Lookup the parts in ids"

    ids do: [:id | (parts findKeyOrNil: id) lookup]
```

**update: p**

```
    "Run 10 cold through warm update traversals, followed by
    several hot update traversals varying the number of traversals per
    measurement. The probability of any given part encountered during
    the traversal being updated is p%"

    | id |
    Smalltalk commit.
    1 to: 10 do:
        [:i |
        id ← traverseIds at: i.
        self
            time:
                [self update: id ifBuildLessThan: p.
                Smalltalk commit]].
    #(0 1 5 10 15 20 50 100 500 ) do: [:i | self
            time:
                [i timesRepeat: [self update: id ifBuildLessThan: p].
                Smalltalk commit]]
```

**update: fromId ifBuildLessThan: int0to100**

```
"Do an update traversal, starting at the part with the identifier
fromId, updating each part encountered whose randomly initialized
build field contains an integer less than int0to100"

(parts findKeyOrNil: fromId)
    update: 7 ifBuildLessThan: int0to100
```

*Protocol for private*

**beginBenchmark**

```
"Indicate to the VM that the benchmark has begun"

<primitive: 120>
↑self
```

**endBenchmark**

```
"Indicate to the VM that the benchmark has ended"

<primitive: 121>
↑self
```

**randomConnection: fromId**

```
"Randomly (modulo clustering) select a part as a target for a
connection from the part with identifier fromId"

| toId |
(self randomIntRange: 1 to: 10)
    > 1
    ifTrue:
        ["90% of time create connection to the closest1% of parts"
        toId ← fromId + (self randomIntRange: 1 to: lastId // 100)
            − 1 − (lastId // 200).
        " 'double up' at ends so stay in part-id range"
        toId < (lastId // 200) ifTrue: [toId ← toId + (lastId // 200)].
        toId > (lastId − (lastId // 200)) ifTrue: [toId ← toId − (lastId // 200)]]
    ifFalse: ["10 % of time create connection to any part 1...N"
        toId ← self randomIntRange: 1 to: lastId].
↑toId
```

**randomIntRange: low to: high**

```
"Generate a random integer in the range [low, high]"

↑(random next * (high − low)) rounded + low
```

**time: timedBlock**

*“Tell the VM to measure the elapsed time for execution of timedBlock”*

```
self beginBenchmark.
timedBlock value.
self endBenchmark
```

# Part

| | |
|---|---|
| class name | **Part** |
| superclass | **Object** |
| instance variable names | **id type x y build conn1To conn1Type conn1Len conn2To conn2Type conn2Len conn3To conn3Type conn3Len from** |
| class variable names | *none* |
| pool dictionaries | *none* |
| category | **Benchmarks-OO1** |

*Instance variables:*

| | |
|---|---|
| *id* | *<SmallInteger> unique part identifier* |
| *type* | *<String> 10-character part type* |
| *x y* | *<SmallInteger>* |
| *build* | *<SmallInteger>* |
| *conn1To conn2To conn3To* | *<Part> outgoing connections* |
| *conn1Type conn2Type conn3Type* | *<String> 10-character connection type* |
| *conn1Len conn2Len conn3Len* | *<SmallInteger>* |
| *from* | *<Array \| Nil> incoming connections* |

*Protocol for initialize-release*

**id: newId**

```
id ← newId.
type ← String new: 10.
x ← 0.
y ← 0.
build ← 0
```

*Protocol for accessing*

**build**

```
↑build
```

**build: value**

```
↑build ← value
```

**id**

```
↑id
```

**key**

```
↑id
```

*Protocol for connections*

**addFrom: aPart**

*"Add a connection to myself from aPart"*

```
from isNil
    ifTrue:
        [from ← Array with: aPart.
        ↑self].
from ← from copyWith: aPart
```

**connectTo: part1 to: part2 to: part3**

*"Connect myself to part1, part2, and part3"*

```
conn1To ← part1 addFrom: self.
conn1Type ← String new: 10.
conn1Len ← 0.
conn2To ← part2 addFrom: self.
conn2Type ← String new: 10.
conn2Len ← 0.
conn3To ← part3 addFrom: self.
conn3Type ← String new: 10.
conn3Len ← 0
```

*Protocol for operations*

**btraverse: level**

```
| count |
count ← 1.
self
    x: x
    y: y
    type: type.
level = 0 ifTrue: [↑count].
from notNil ifTrue:
    [from do: [:part | count ← (part btraverse: level − 1) + count]].
↑count
```

**ftraverse: level**

```
self
```

```
        x: x
        y: y
        type: type.
    level = 0 ifTrue: [↑self].
    conn1To ftraverse: level − 1.
    conn2To ftraverse: level − 1.
    conn3To ftraverse: level − 1
```

**lookup**

```
    self
        x: x
        y: y
        type: type
```

**update: level ifBuildLessThan: int0to100**

*"Perform update traversal, updating me if my build field is less than int0to100"*

```
    build < int0to100
        ifTrue:
            [x ← x + 1.
            y ← y + 1].
    level = 0 ifTrue: [↑self].
    conn1To update: level − 1 ifBuildLessThan: int0to100.
    conn2To update: level − 1 ifBuildLessThan: int0to100.
    conn3To update: level − 1 ifBuildLessThan: int0to100
```

**x: arg1 y: arg2 type: arg3**

*"Null procedure to be invoked on parts encountered during benchmarks"*

```
    ↑self
```

# Random

| | |
|---|---|
| class name | **Random** |
| superclass | **Stream** |
| instance variable names | **seed** |
| class variable names | **A M MF Q R** |
| pool dictionaries | *none* |
| category | **Numeric-Numbers** |

*A good, minimal standard, random number generator.*
*This is "Integer Version 1" of the random number generator presented in:*
*"Random Number Generators: Good Ones are Hard to Find",*
*Stephen K. Park and Keith W. Miller,*
*CACM 31 (10), Oct. 1988, pp. 1192−1201.*

*Protocol for accessing*

**contents**

```
↑self shouldNotImplement
```

**next**

```
"Answer with the next random number."

| hi lo test |
seed ← A * seed \\ M.
↑seed / MF
```

**nextPut: anObject**

```
↑self shouldNotImplement
```

*Protocol for testing*

**atEnd**

```
↑false
```

*Protocol for private*

**setSeed**

```
seed ← Time millisecondClockValue bitAnd: 16r1FFFFFFF
"Time millisecondClockValue gives a large Integer; I want a
SmallInteger. "
```

# Random class

| | |
|---|---|
| class name | **Random class** |
| superclass | **Stream class** |
| instance variable names | *none* |
| class variable names | **A M MF Q R** |
| pool dictionaries | *none* |
| category | **Numeric-Numbers** |

*Protocol for instance creation*

**new**

*“Answer a new random number generator.”*

```
↑self basicNew setSeed
```

*Protocol for class initialization*

**initialize**

```
A ← 16807.
M ← 16r7FFFFFFF.
MF ← M asFloat.
Q ← M // A.
R ← M \\ A
```

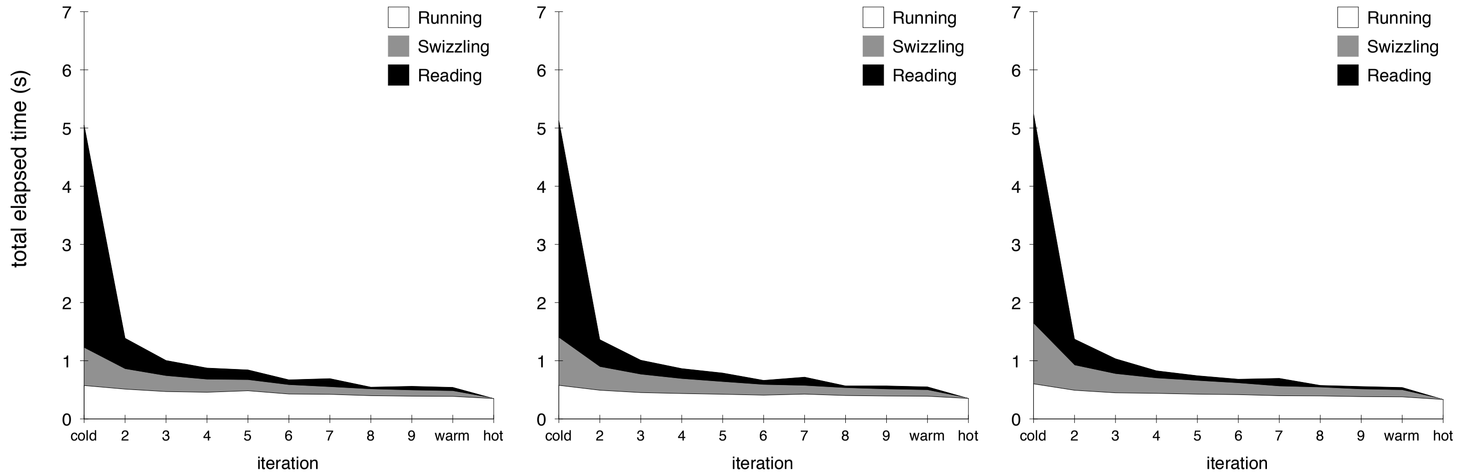
# APPENDIX B

# SUPPLEMENTAL RESULTS: OBJECT FAULTING

The following graphs plot the breakdown of elapsed time for each phase of execution of the object faulting benchmarks:[1]

- *running*: time spent in the interpreter executing the program; includes the overhead to trigger object faults, but not the time for disk reads, swizzling, or indirection elimination;
- *swizzling*: time spent copying and swizzling objects from Mneme's buffer pool;
- *reading*: time spent reading Mneme PSEGs from disk into the buffer pool; and
- *bypassing*: time spent bypassing indirect references to objects made resident at time of fault.

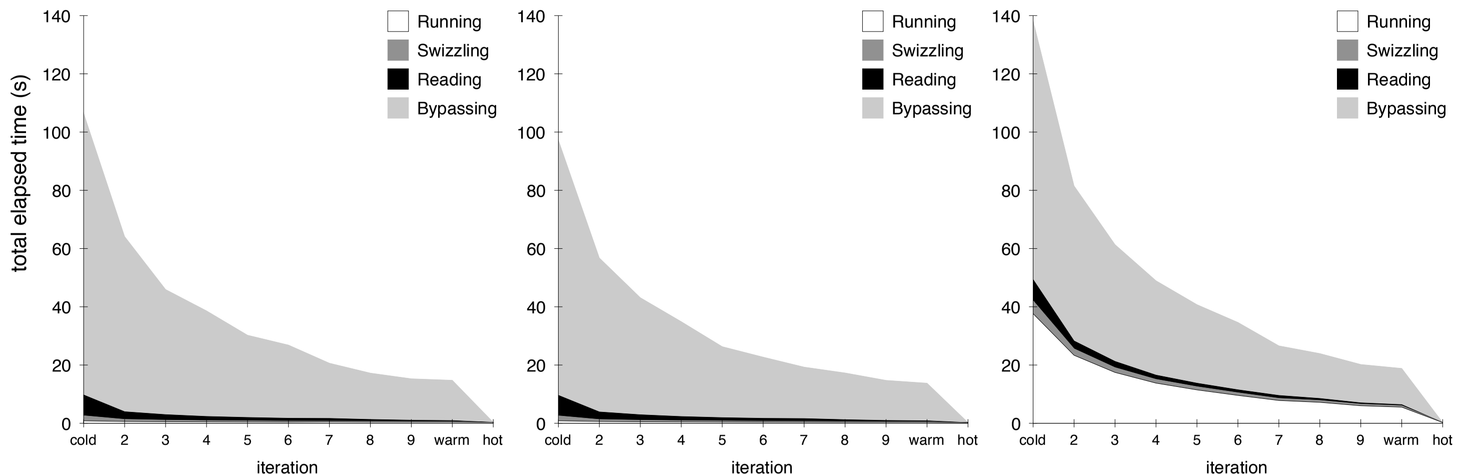
[1]Raw data for all results presented in this dissertation may be obtained by contacting the author via email: `hosking@cs.purdue.edu`. Check also WWW URLs `http://www.cs.umass.edu/~hosking` or `http://www.cs.purdue.edu/people/hosking`.




(a) ID,no-bypass,obj

(b) ID,swizzle-bypass,obj

(c) FB,swizzle-bypass,obj

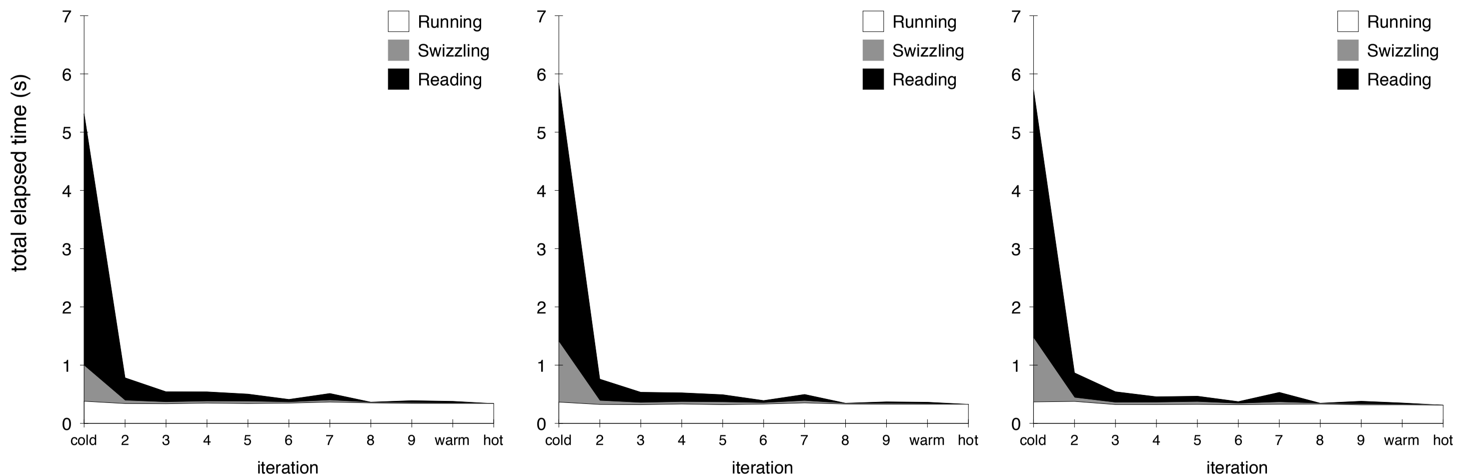


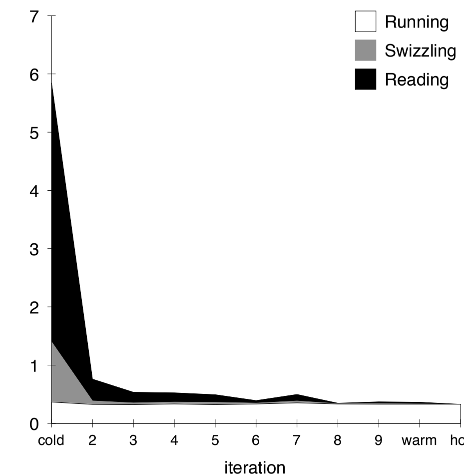

(d) ID,fault-bypass,obj

(e) FB,fault-bypass,obj

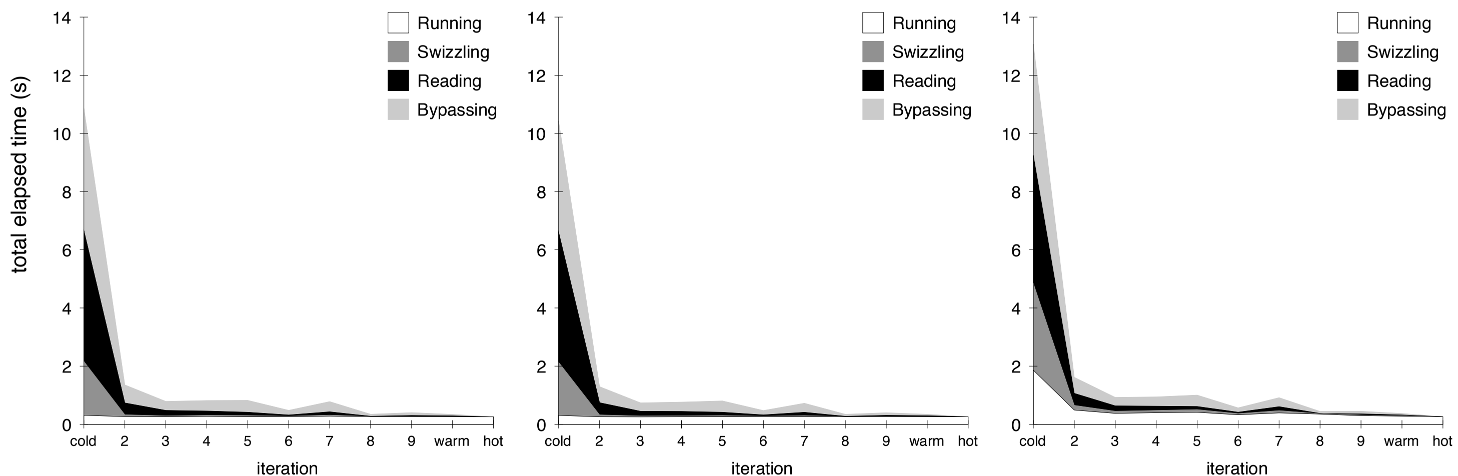
(f) trap,fault-bypass,obj

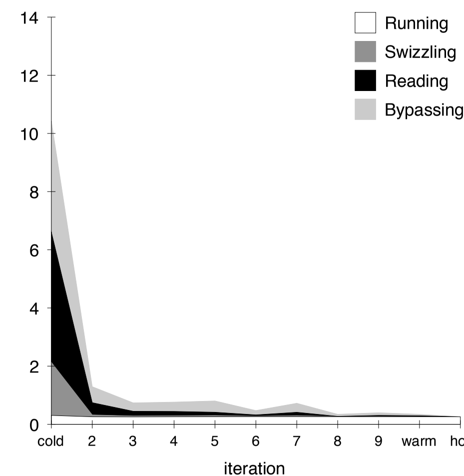
**Figure B.1** Lookup, swizzling one object per fault





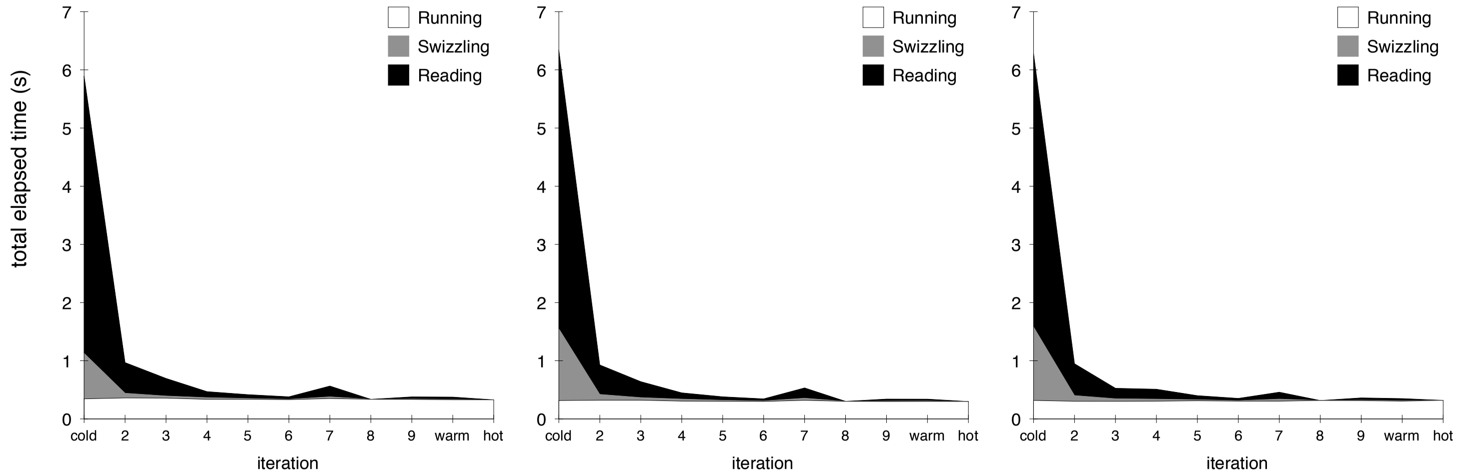

(a) ID,no-bypass,LSEG (b) ID,swizzle-bypass,LSEG (c) FB,swizzle-bypass,LSEG






(d) ID,fault-bypass,LSEG (e) FB,fault-bypass,LSEG (f) trap,fault-bypass,LSEG

**Figure B.2** Lookup, swizzling one LSEG per fault



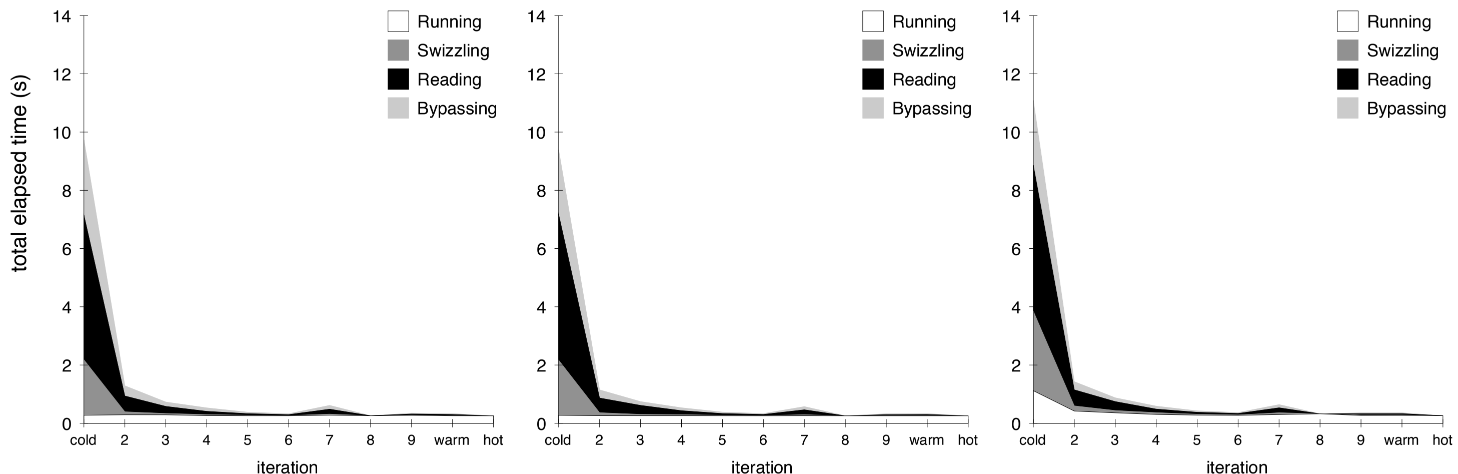
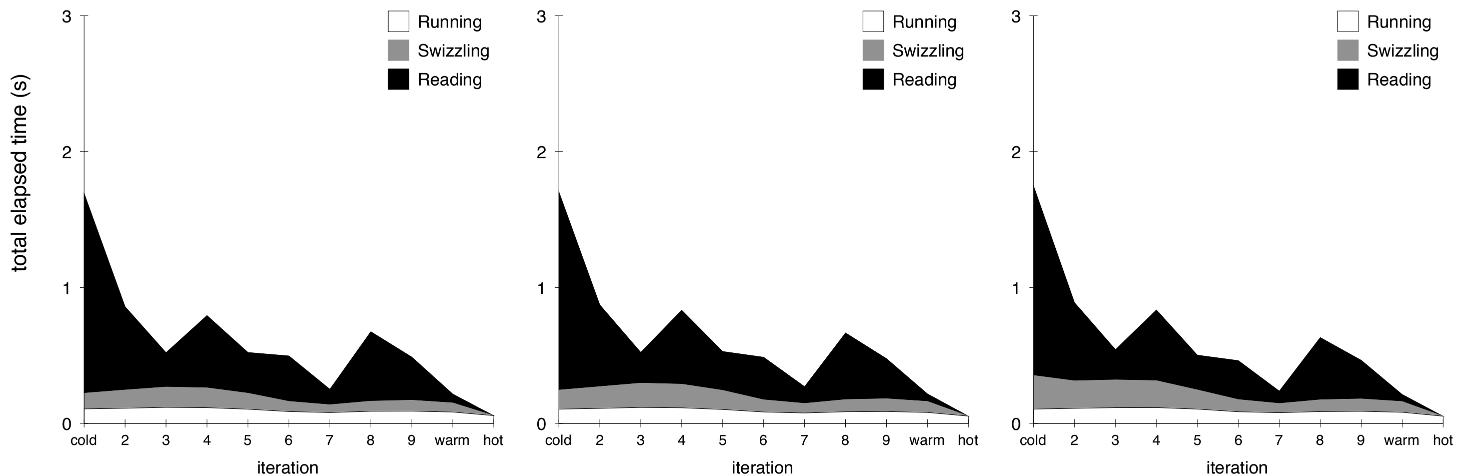
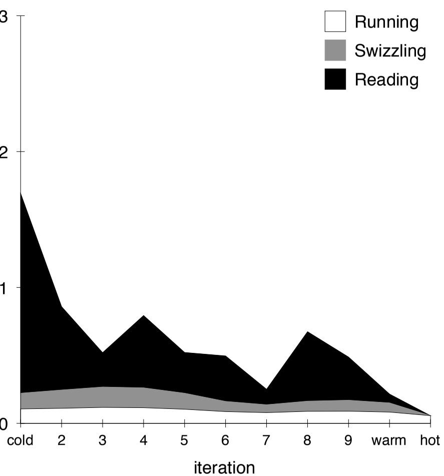
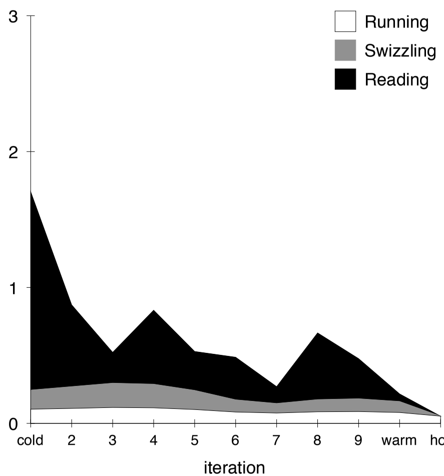



**Figure B.3** Lookup, swizzling one PSEG per fault

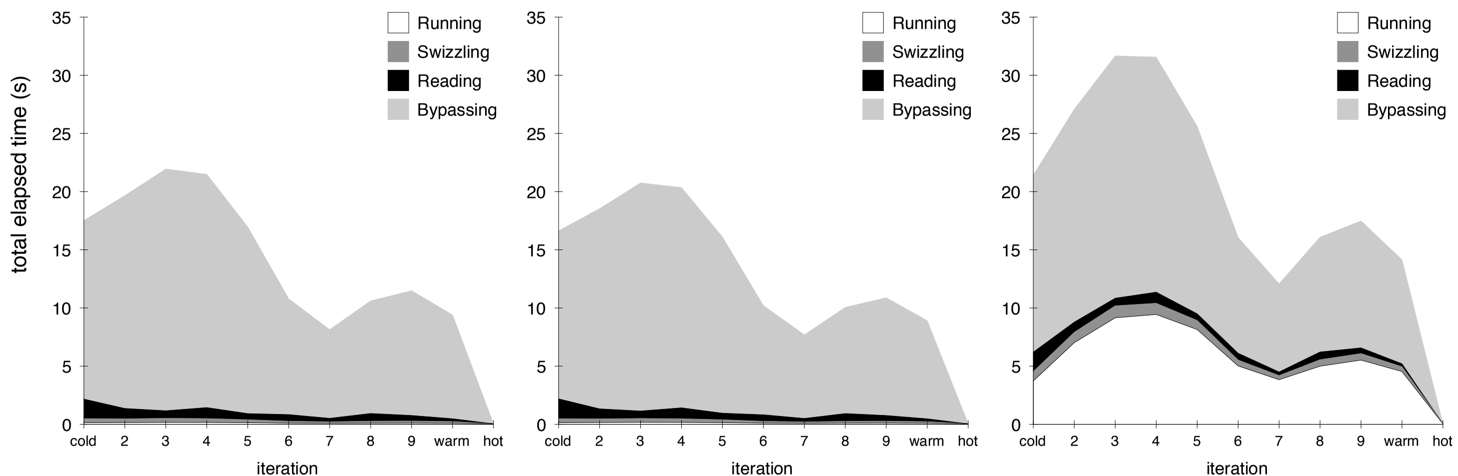
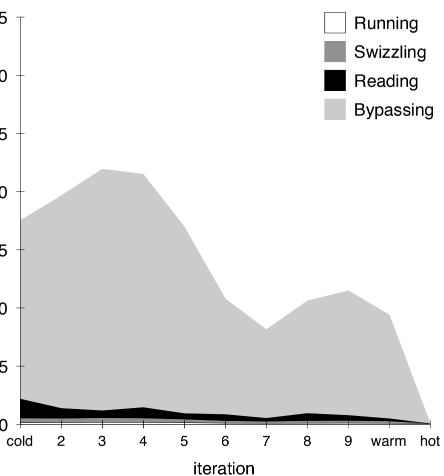




(a) ID,no-bypass,obj

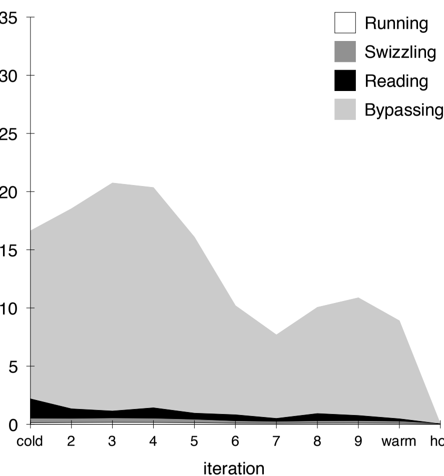



(b) ID,swizzle-bypass,obj

(c) FB,swizzle-bypass,obj

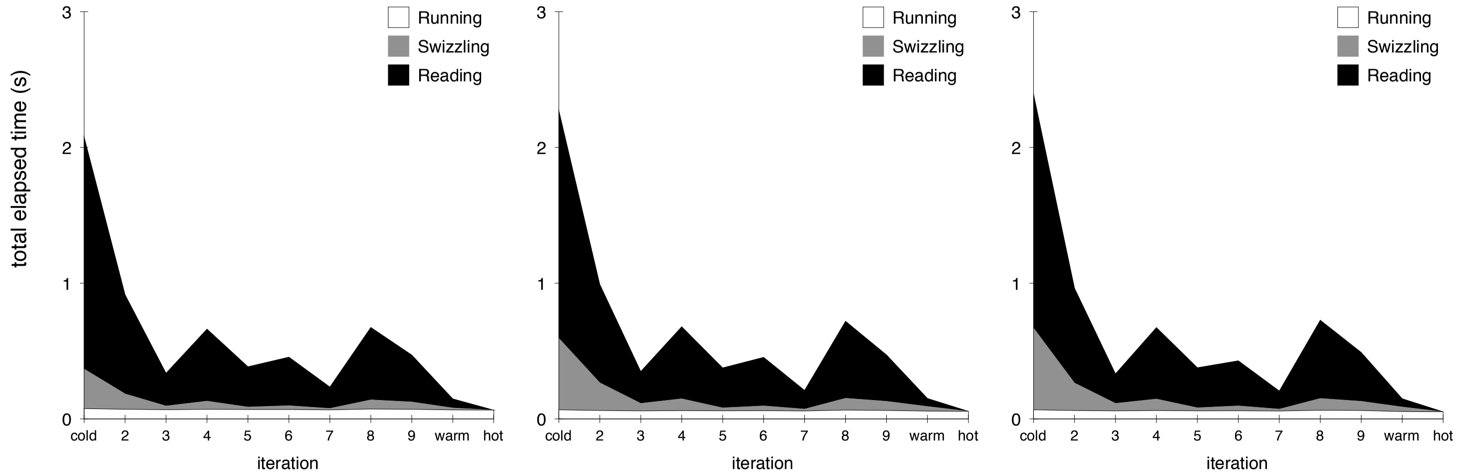

(d) ID,fault-bypass,obj



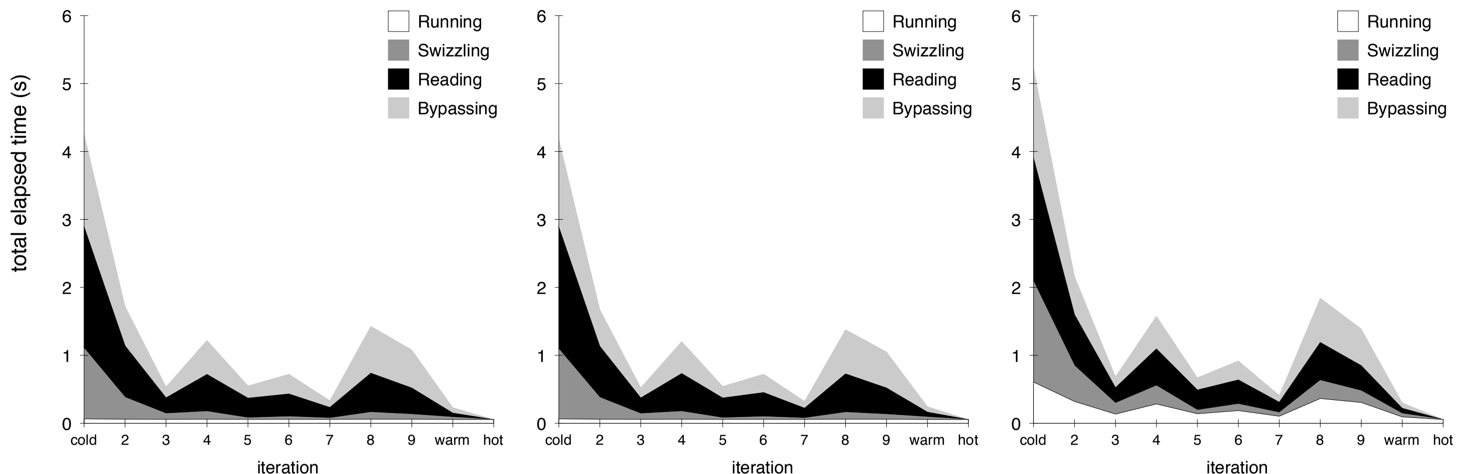

(e) FB,fault-bypass,obj

(f) trap,fault-bypass,obj

**Figure B.4** Traversal, swizzling one object per fault

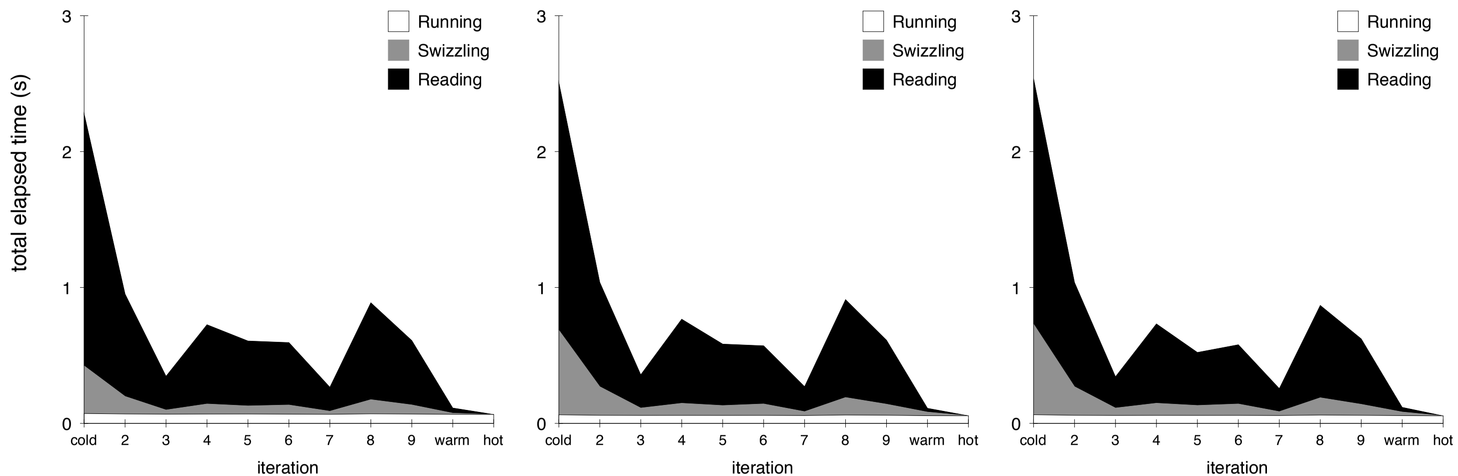
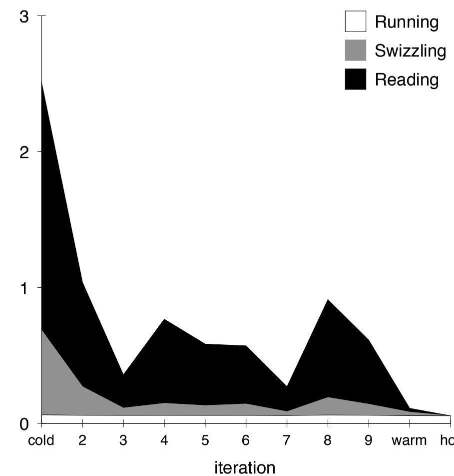
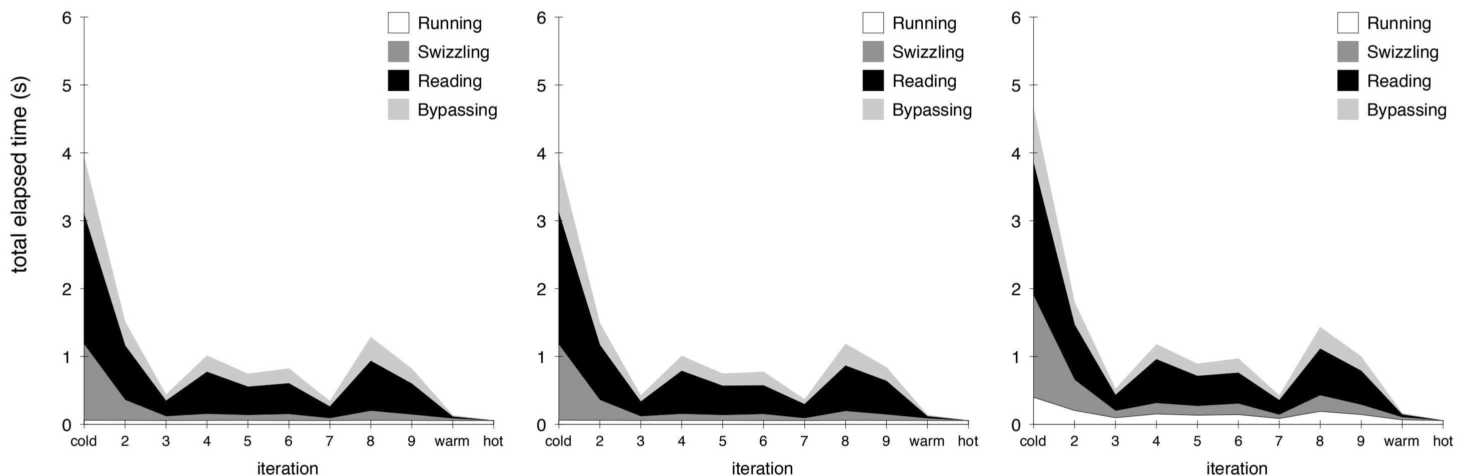
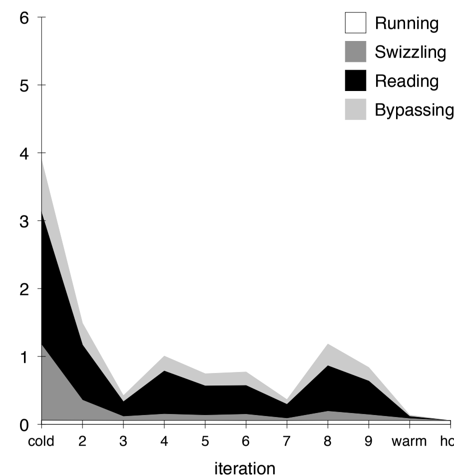
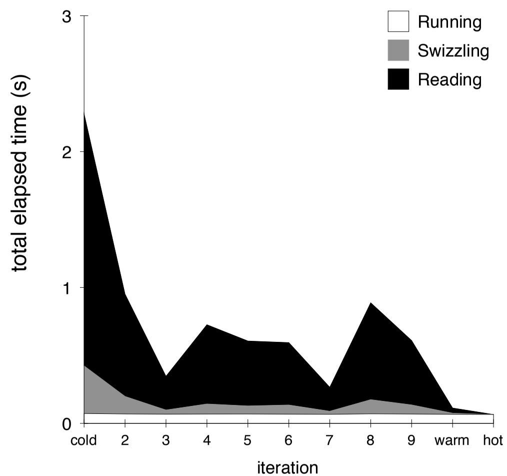



(a) ID,no-bypass,LSEG (b) ID,swizzle-bypass,LSEG (c) FB,swizzle-bypass,LSEG




(d) ID,fault-bypass,LSEG (e) FB,fault-bypass,LSEG (f) trap,fault-bypass,LSEG

**Figure B.5** Traversal, swizzling one LSEG per fault


(a) ID,no-bypass,PSEG

(b) ID,swizzle-bypass,PSEG

(c) FB,swizzle-bypass,PSEG

(d) ID,fault-bypass,PSEG

(e) FB,fault-bypass,PSEG

(f) trap,fault-bypass,PSEG

**Figure B.6** Traversal, swizzling one PSEG per fault

# BIBLIOGRAPHY

Agrawal, R. and Gehani, N. H. (1989a). ODE (Object Database and Environment): The language and the data model. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 36–45, Portland, Oregon. *ACM SIGMOD Record 18*, 2 (May 1989).

Agrawal, R. and Gehani, N. H. (1989b). Rationale for the design of persistence and query processing facilities in the database language O++. In Hull et al. [1989], pages 25–40.

Albano, A., Cardelli, L., and Orsini, R. (1985). Galileo: A strongly-typed, interactive conceptual language. *ACM Transactions on Database Systems*, 10(2):230–260.

Albano, A., Ghelli, G., and Orsini, R. (1989). Types for databases: The Galileo experience. In Hull et al. [1989], pages 196–206.

Anderson, T. L., Berre, A. J., Mallison, M., Porter, H., and Schneider, B. (1989). The Tektronix HyperModel benchmark specification. Technical Report 89-05, Computer Research Laboratory, Tektronix Laboratories.

Andrews, T. and Harris, C. (1987). Combining language and database advances in an object-oriented development environment. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 430–440, Orlando, Florida. *ACM SIGPLAN Notices 22*, 11 (November 1987).

Atkinson, M., Chisolm, K., and Cockshott, P. (1982). PS-Algol: an Algol with a persistent heap. *ACM SIGPLAN Notices*, 17(7):24–31.

Atkinson, M. P., Bailey, P. J., Chisholm, K. J., Cockshott, P. W., and Morrison, R. (1983a). An approach to persistent programming. *The Computer Journal*, 26(4):360–365.

Atkinson, M. P. and Buneman, O. P. (1987). Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2):105–190.

Atkinson, M. P., Chisholm, K. J., Cockshott, W. P., and Marshall, R. M. (1983b). Algorithms for a persistent heap. *Software: Practice and Experience*, 13(7):259–271.

Atkinson, M. P. and Morrison, R. (1985). Procedures as persistent data objects. *ACM Transactions on Programming Languages and Systems*, 7(4):539–559.

Bancilhon, F., Barbedette, G., Benzaken, V., Delobel, C., Gamerman, S., Lécluse, C., Pfeffer, P., Richard, P., and Velez, F. (1988). The design and implementation of $O_2$, an object-oriented database system. In Dittrich [1988], pages 1–22.

Banerjee, J., Chou, H., Garza, J. F., Kim, W., Woelk, D., Ballou, N., and Kim, H. (1987). Data model issues for object-oriented applications. *ACM Transactions on Office Information Systems*, 5(1):3–26.

Bell, J. R. (1973). Threaded code. *Communications of the ACM*, 16(6):370–372.

Bretl, R., Maier, D., Otis, A., Penney, J., Schuchardt, B., Stein, J., Williams, E. H., and Williams, M. (1989). The GemStone data management system. In Kim and Lochovsky [1989], chapter 12, pages 283–308.

Carey, M. J., DeWitt, D. J., and Naughton, J. F. (1993). The OO7 benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 12–21, Washington, DC. *ACM SIGMOD Record 22*, 2 (June 1993).

Carey, M. J., DeWitt, D. J., Richardson, J. E., and Shekita, E. J. (1986). Object and file management in the EXODUS extensible database system. In *Proceedings of the Twelfth International Conference on Very Large Data Bases*, pages 91–100, Kyoto, Japan. Morgan Kaufmann.

Carey, M. J., DeWitt, D. J., Richardson, J. E., and Shekita, E. J. (1989). Storage management for objects in EXODUS. In Kim and Lochovsky [1989], chapter 14, pages 341–369.

Cattell, R. G. G. and Skeen, J. (1992). Object operations benchmark. *ACM Transactions on Database Systems*, 17(1):1–31.

Cheney, C. J. (1970). A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678.

Copeland, G. and Maier, D. (1984). Making Smalltalk a database system. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 316–325, Boston, Massachusetts. *ACM SIGMOD Record 14*, 2 (1984).

Dearle, A., Conner, R., Brown, F., and Morrison, R. (1989). Napier88—A database programming language? In Hull et al. [1989], pages 179–195.

Dearle, A., Shaw, G. M., and Zdonik, S. B., editors (1990). *Proceedings of the Fourth International Workshop on Persistent Object Systems*, Martha's Vineyard, Massachusetts. Published as *Implementing Persistent Object Bases: Principles and Practice*, Morgan Kaufmann, 1990.

Detlefs, D. D., Herlihy, M. P., and Wing, J. M. (1988). Inheritance of synchronization and recovery in Avalon/C++. *IEEE Computer*, 21(12):57–69.

Deux, O. et al. (1990). The story of $O_2$. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108.

Dittrich, K. and Dayal, U., editors (1986). *Proceedings of the International Workshop on Object-Oriented Database Systems*, Pacific Grove, California. IEEE Computer Society Press, Washington, D.C.

Dittrich, K. R., editor (1988). *Proceedings of the Second International Workshop on Object-Oriented Database Systems*, volume 334 of *Lecture Notes in Computer Science*, Bad Münster am Stein-Ebernburg, Federal Republic of Germany. *Advances in Object-Oriented Database Systems*, Springer-Verlag, 1988.

Elhardt, K. and Bayer, R. (1984). A database cache for high performance and fast restart in database systems. *ACM Transactions on Database Systems*, 9(4):503–525.

Ellis, M. A. and Stroustrup, B. (1990). *The Annotated C++ Reference Manual*. Addison-Wesley.

Fegaras, L., Sheard, T., and Stemple, D. (1989). The ADABTPL type system. In Hull et al. [1989], pages 207–218.

Goldberg, A. and Robson, D. (1983). *Smalltalk-80: The Language and its Implementation*. Addison-Wesley.

Gray, J. N. (1978). Notes on database operating systems. In Bayer, R. et al., editors, *Operating Systems: An Advanced Course*, Lecture Notes in Computer Science. Springer-Verlag.

Haerder, T. and Reuter, A. (1983). Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15:287–318.

Harbison, S. P. (1992). *Modula-3*. Prentice Hall, New Jersey.

Hornick, M. F. and Zdonik, S. B. (1987). A shared, segmented memory system for an object-oriented database. *ACM Transactions on Office Information Systems*, 5(1):70–95.

Hosking, A. L. (1991). Main memory management for persistence. Position paper presented at the OOPSLA '91 Workshop on Garbage Collection.

Hosking, A. L. and Moss, J. E. B. (1990). Towards compile-time optimisations for persistence. In Dearle et al. [1990], pages 17–27.

Hosking, A. L. and Moss, J. E. B. (1991). Compiler support for persistent programming. COINS Technical Report 91-25, University of Massachusetts, Amherst, MA 01003.

Hosking, A. L. and Moss, J. E. B. (1993). Protection traps and alternatives for memory management of an object-oriented language. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 106–119, Asheville, North Carolina. *ACM Operating Systems Review 27*, 5 (December 1993).

Hosking, A. L., Moss, J. E. B., and Stefanović, D. (1992). A comparative performance evaluation of write barrier implementations. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 92–109, Vancouver, Canada. *ACM SIGPLAN Notices 27*, 10 (October 1992).

Hull, R., Morrison, R., and Stemple, D., editors (1989). *Proceedings of the Second International Workshop on Database Programming Languages*, Gleneden Beach, Oregon. Morgan Kaufmann.

Kaehler, T. (1986). Virtual memory on a narrow machine for an object-oriented language. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 87–106, Portland, Oregon. *ACM SIGPLAN Notices 21*, 11 (November 1986).

Kaehler, T. and Krasner, G. (1983). LOOM—large object-oriented memory for Smalltalk-80 systems. In Krasner [1983], chapter 14, pages 251–270.

Kim, W., Ballou, N., Banerjee, J., Chou, H., Garza, J. F., and Woelk, D. (1988). Integrating an object-oriented programming system with a database system. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 142–152, San Diego, California. *ACM SIGPLAN Notices 23*, 11 (November 1988).

Kim, W., Ballou, N., Chou, H., Garza, J. F., and Woelk, D. (1989). Features of the ORION object-oriented database system. In Kim and Lochovsky [1989], chapter 11, pages 251–282.

Kim, W., Garza, J. F., Ballou, N., and Woelk, D. (1990). Architecture of the ORION next-generation database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):109–124.

Kim, W. and Lochovsky, F. H., editors (1989). *Object-Oriented Concepts, Databases, and Applications*. ACM Press/Addison-Wesley, New York, New York.

Krasner, G., editor (1983). *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley.

Lamb, C., Landis, G., Orenstein, J., and Weinreb, D. (1991). The ObjectStore database system. *Communications of the ACM*, 34(10):50–63.

Lécluse, C. and Richard, P. (1989). The $O_2$ database programming language. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 411–422, Amsterdam, The Netherlands. Morgan Kaufmann.

Lécluse, C., Richard, P., and Velez, F. (1988). $O_2$, an object-oriented data model. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 424–433, Chicago, Illinois. *ACM SIGMOD Record 17*, 3 (September 1988).

Maier, D. and Stein, J. (1986). Indexing in an object-oriented DBMS. In Dittrich and Dayal [1986], pages 171–182.

Matthes, F. and Schmidt, J. W. (1989). The type system of DBPL. In Hull et al. [1989], pages 219–225.

Morrison, R., Brown, A., Carrick, R., Connor, R., Dearle, A., and Atkinson, M. P. (1989). The Napier type system. In Rosenberg, J. and Koch, D., editors, *Proceedings of the Third International Workshop on Persistent Object Systems*, pages 3–18, Newcastle, Australia. Springer-Verlag, 1990.

Moss, J. E. B. (1987). Managing stack frames in Smalltalk. In *Proceedings of the ACM SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pages 229–240, St. Paul, Minnesota. *ACM SIGPLAN Notices 22*, 7 (July 1987).

Moss, J. E. B. (1990). Working with persistent objects: To swizzle or not to swizzle. COINS Technical Report 90-38, University of Massachusetts, Amherst, MA 01003. Submitted for publication.

Moss, J. E. B. and Hosking, A. L. (1994). Expressing object residency optimizations using pointer type annotations. In *Proceedings of the Sixth International Workshop on Persistent Object Systems*, Tarasçon, France. Springer-Verlag.

Moss, J. E. B., Leban, B., and Chrysanthis, P. K. (1987). Finer grained concurrency control for the database cache. In *Proceedings of the Third International Conference on Data Engineering*, pages 96–103, Los Angeles, CA. IEEE.

Mylopoulos, J., Bernstein, P. A., and Wong, H. K. T. (1980). A language facility for designing database-intensive applications. *ACM Transactions on Database Systems*, 5(2):185–207.

Nelson, G., editor (1991). *Systems Programming with Modula-3*. Prentice Hall, New Jersey.

Nixon, B., Chung, L., Lauzon, D., Borgida, A., Mylopoulos, J., and Stanley, M. (1987). Implementation of a compiler for a semantic data model: Experiences with Taxis. In SIGMOD [1987], pages 118–131.

ObjectStore (1990). *ObjectStore User Guide*. Object Design, Inc. Release 1.0.

Purdy, A., Schuchardt, B., and Maier, D. (1987). Integrating an object server with other worlds. *ACM Transactions on Office Information Systems*, 5(1):27–47.

Richardson, J. E. (1990). Compiled item faulting: A new technique for managing I/O in a persistent language. In Dearle et al. [1990], pages 3–16.

Richardson, J. E. and Carey, M. J. (1987). Programming constructs for database implementations in EXODUS. In SIGMOD [1987], pages 208–219.

Richardson, J. E. and Carey, M. J. (1990). Persistence in the E language: Issues and implementation. *Software: Practice and Experience*, 19(12):1115–1150.

Rowe, L. A. and Shoens, K. (1979). Data abstraction, views and updates in RIGEL. In SIGMOD [1979], pages 71–81.

Schmidt, J. W. (1977). Some high level language constructs for data of type relation. *ACM Transactions on Database Systems*, 2(3):247–261.

Schmidt, J. W. and Mall, M. (1980). Pascal/R report. Report 66, Fachbereich Automatik, University of Hamburg, Hamburg, Germany.

Schuh, D., Carey, M., and DeWitt, D. (1990). Persistence in E revisited—implementation experiences. In Dearle et al. [1990], pages 345–359.

Shaw, R. A. (1987). Improving garbage collector performance in virtual memory. Technical Report CSL-TR-87-323, Stanford University.

Sheard, T. and Stemple, D. (1989). Automatic verification of database transaction safety. *ACM Transactions on Database Systems*, 14(3):322–368.

Shipman, D. W. (1981). The functional data model and the data language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173.

Shopiro, J. E. (1979). Theseus—A programming language for relational databases. *ACM Transactions on Database Systems*, 4(4):493–517.

SIGMOD (1979). *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, Boston, Massachusetts.

SIGMOD (1987). *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, San Francisco, California. *ACM SIGMOD Record 16*, 3 (December 1987).

Singhal, V., Kakkad, S. V., and Wilson, P. R. (1992). Texas, an efficient, portable persistent store. In *Proceedings of the Fifth International Workshop on Persistent Object Systems*, pages 11–33, San Miniato, Italy.

Skarra, A., Zdonik, S. B., and Reiss, S. P. (1986). An object server for an object oriented database system. In Dittrich and Dayal [1986], pages 196–204.

Smith, J. M., Fox, S., and Landers, T. (1983). *ADAPLEX: Rationale and Reference Manual*. Computer Corporation of America, Cambridge, Mass., second edition.

Sobalvarro, P. G. (1988). A lifetime-based garbage collector for LISP systems on general-purpose computers. B.S. Thesis, Dept. of EECS, Massachusetts Institute of Technology, Cambridge.

Stemple, D., Socorro, A., and Sheard, T. (1988). Formalizing objects for databases using ADABTPL. In Dittrich [1988], pages 110–128.

Straw, A., Mellender, F., and Riegel, S. (1989). Object management in a persistent Smalltalk system. *Software: Practice and Experience*, 19(8):719–737.

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

Taylor, J. R. (1982). *An Introduction to Error Analysis*. University Science Books, Mill Valley, California.

Thekkath, C. A. and Levy, H. M. (1994). Hardware and software support for efficient exception handling. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 110–119, San Jose, California. *ACM SIGPLAN Notices 29*, 11 (November 1994).

Ungar, D. (1984). Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, Pittsburgh, Pennsylvania. *ACM SIGPLAN Notices 19*, 5 (May 1984).

Ungar, D. M. (1987). *The Design and Evaluation of a High Performance Smalltalk System*. ACM Distinguished Dissertations. The MIT Press, Cambridge, MA. Ph.D. Dissertation, University of California at Berkeley, February 1986.

Verhofstad, J. S. M. (1978). Recovery techniques for database systems. *ACM Computing Surveys*, 10(2):167–195.

Wahbe, R. (1992). Efficient data breakpoints. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 200–212, Boston, Massachusetts. *ACM SIGPLAN Notices 27*, 9 (September 1991).

Walsh, N. (1994). *Making TEX Work*. O'Reilly and Associates, Inc.

Wasserman, A. L. (1979). The data management facilities of PLAIN. In SIGMOD [1979], pages 60–70.

White, S. J. and DeWitt, D. J. (1992). A performance study of alternative object faulting and pointer swizzling strategies. In *Proceedings of the Eighteenth International Conference on Very Large Data Bases*, pages 419–431, Vancouver, Canada. Morgan Kaufmann.

White, S. J. and DeWitt, D. J. (1994). QuickStore: A high performance mapped object store. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 395–406, Minneapolis, Minnesota. *ACM SIGMOD Record 23*, 2 (June 1994).

Wilson, P. R. and Kakkad, S. V. (1992). Pointer swizzling at page fault time: Efficiently and compatibly supporting huge address spaces on standard hardware. In *Proceedings of the 1992 International Workshop on Object Orientation in Operating Systems*, pages 364–377, Paris, France. IEEE Press.

Wilson, P. R. and Moher, T. G. (1989a). A "card-marking" scheme for controlling intergenerational references in generation-based garbage collection on stock hardware. *ACM SIGPLAN Notices*, 24(5):87–92.

Wilson, P. R. and Moher, T. G. (1989b). Design of the opportunistic garbage collector. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 23–35, New Orleans, Louisiana. *ACM SIGPLAN Notices 24*, 10 (October 1989).