Pointer Swizzling Techniques for Object-Oriented Database Systems

Seth John White

Technical Report #1242

September 1994

POINTER SWIZZLING TECHNIQUES
FOR
OBJECT-ORIENTED DATABASE SYSTEMS

by

SETH JOHN WHITE

A thesis submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN — MADISON

1994

© Copyright 1994

by

Seth John White

All Rights Reserved

ABSTRACT

In this thesis, we examine a technique that has come to be known as *pointer swizzling* and whose aim is to improve the performance of object-oriented database management systems (OODBMSs) while accessing persistent objects that have been cached in main memory. It is crucial that OODBMSs provide high-performance when accessing in-memory data, since the applications that OODBMSs must support, i.e. CAD, CASE, GIS, and multi-media, are computationally very intensive. Pointer swizzling improves performance by converting pointers from their disk format (an object identifier) to a more efficient in-memory format (a direct memory address) when objects are faulted into memory by the OODBMS. The thesis presents an in-depth analysis of the performance of several different approaches to pointer swizzling, including several schemes that perform software-based swizzling. The thesis also presents the design of a memory-mapped storage manager, QuickStore, which provides full support for pointer swizzling using standard virtual memory hardware. The performance results presented in the thesis give an accurate and comprehensive picture of the differences in performance between software and hardware-based swizzling techniques.

An issue that is closely related to pointer swizzling, and which appears several times in the thesis, is that of providing recovery services in an OODBMS. Implementing recovery in an OODBMS poses new challenges due to the tight integration between the application programming language used to access the database and the database system itself. In addition, the characteristics of the applications that OODBMSs must support, and the use of a client-server architecture, both add new performance issues. In the thesis we study several alternative ways of performing recovery in the context of an OODBMS that also provides full support for pointer swizzling.

The results concerning recovery show that using differencing to generate log records for updates at the client workstations is generally superior in terms of performance to logging whole pages. The differencing approach is better because it takes advantage of the aggregate CPU power available at the clients to lessen the overall burden placed on the server to support recovery. This provides much better scalability, as it prevents the server from becoming a performance bottleneck as quickly when the number of clients accessing the database increases.

ACKNOWLEDGMENTS

I have been very fortunate to have had Professor David DeWitt serve as my advisor while working on my Ph.D. David has been an inspiration to me, and he provided much encouragement and support when they were needed most during the past few years. I am extremely grateful to David for providing the excellent systems infrastructure and environment that I needed to carry out my research. The opportunities that he gave me for conducting research were outstanding—it has been an honor to have been his student. I am also very grateful to Professor Michael Carey. Mike provided excellent comments and suggestions to me, whenever I turned to him for help, and he always made time to talk with me—I can't thank him enough. I would also like to thank Professors Jeffrey Naughton, Marvin Solomon, and Miron Livny for all of the assistance that they provided during my graduate career.

I would like to offer special thanks to the members of the EXODUS and SHORE projects. In particular, I learned a lot from working with Mike Zwilling, who very patiently answered all of the many questions that I asked him as I struggled to understand EXODUS. Dan Schuh also deserves special thanks. I learned a lot from Dan about the implementation of E, and about pointer swizzling. I could not have conducted my research without the implementation work on E that Dan did. I would also like to thank the other hard working, industrious staff members of the EXODUS and SHORE projects including: Tan, Nancy Hall, and Bolo for all of their help. I also learned a lot from the other students in the database group at Wisconsin, including: Manish Mehta, Kurt Brown, Mike Franklin, Craig Freedman, Mark McAuliff, Odysseas Tsatalos, Janet Wiener, Shaul Dar, Paul Bober, Dan Lieuwen, Joe Hellerstein, Praveen Seshadri, Ambuj Shatdal, and Jignesh Patel, among others.

I would like to thank my family, especially my mom, for always believing in me and for helping me to believe that I could earn a Ph.D. Obtaining a Ph.D. is incredibly hard work (an obvious fact since it is 1:30 a.m. as I write this), and I could not have made it without their love. And finally, I would like to express my sincere gratitude and thanks to my good friend Kevin Looby, for the may weekends when he let me crash at his awesome pad in Lake Forest; when the stress of graduate school was too much and I needed to put some distance between myself and my work for a few days. Thanks, Kevin!

TABLE OF CONTENTS

ABSTRACT ii

ACKNOWLEDGMENTS iii

TABLE OF CONTENTS iv

Chapter 1: INTRODUCTION 1

Chapter 2: OVERVIEW OF POINTER SWIZZLING 4

2.1 Swizzling vs. No Swizzling 4

2.2 Hardware vs. Software-based Swizzling 4

2.3 In-place vs. Copy Swizzling 5

2.4 Uncaching vs. No Uncaching 6

2.5 Eager vs. Lazy Swizzling 6

2.6 Direct vs. Indirect Swizzling 7

2.7 Partial vs. Full Swizzling 7

2.8 Survey of Related Work 7

Chapter 3: POINTER SWIZZLING IN E 11

3.1 Introduction 11

3.2 EPVM 2.0 Design Concepts 12

3.2.1 Object Caching 12

3.2.2 Pointer Swizzling in EPVM 2.0 15

3.3 Performance Experiments 19

3.3.1 Software Versions 20

3.3.2 Benchmark Database 25

3.3.3 Hardware Used 25

3.4 Benchmark Results 26

3.4.1 Small Database Results 26

3.4.2 Large Database Results 34

3.5 EPVM 2.0 Performance in Detail 37

3.5.1 Hot Traversal Performance 37

3.5.2 Comparison of EPVM and ObjectStore 39

3.5.3 Cold Traversal Performance 40

3.6 Conclusions 42

Chapter 4: THE DESIGN OF QUICKSTORE 44

4.1 Overview of the Memory-Mapped Architecture 44

4.2 Implementation Details 47

4.3 In-Memory Data Structures 48

4.4 Pointer Swizzling in QuickStore 51

4.5 Buffer Pool Management 54

4.6 Recovery 55

4.7 Comparison With Other Systems 56

Chapter 5: THE PERFORMANCE OF QUICKSTORE 58

5.1 The OO7 Benchmark Database 58

5.2 The OO7 Benchmark Operations 61

5.2.1 Traversals 61

5.2.2 Queries 62

5.3 Hardware Used 62

5.4 Software Used 62

5.5 Database Systems Tested 63

5.5.1 E 63

5.5.2 QuickStore 64

5.6 Performance Results 65

5.6.1 Database Sizes 65

5.6.2 Small Cold Results 65

5.6.3 Small Hot Results 74

5.6.4 Medium Cold Results 76

5.6.5 Effect of Collisions 79

5.7 Conclusions 80

Chapter 6: RECOVERY IN QUICKSTORE 83

6.1 Background and Motivation 83

6.2 Recovery Strategies 84

6.2.1 Recovery in EXODUS 84

6.2.2 The Page Diffing Approach 85

6.2.2.1 Enabling Recovery for Page Diffing 85

6.2.2.2 Generating Log Records for Pages 87

6.2.3 The Sub-Page Diffing Approach 88

6.2.3.1 Enabling Recovery for Sub-Page Diffing 89

6.2.3.2 Generating Log Records for Sub-Pages 90

6.2.4 The Whole-Page Logging Approach 90

6.2.4.1 Actions Performed at Clients 91

6.2.4.2 Actions Performed at the Server 92

6.2.4.3 Recovering from a Crash 93

6.2.5 The Redo-at-Server Approach 94

6.2.6 Implementation Discussion 94

6.3 Performance Experiments 95

6.3.1 Benchmark Database 95

6.3.2 Benchmark Operations 96

6.3.3 Software Versions 96

6.4 Performance Results 97

6.4.1 Unconstrained Cache Results 97

6.4.2 Constrained Cache Results 101

6.4.3 Big Database Results 104

6.5 Related Work 106

6.6 Conclusions 109

Chapter 7: SUMMARY 110

References: 112

CHAPTER 1

INTRODUCTION

Beginning in the mid-1980s a great deal of interest arose in the database research community concerning object-based data management systems. This change in direction occurred because the object-based approach appeared to offer substantial advantages over the traditional relational approach to building database systems, especially in meeting the needs of emerging applications such as computer aided design and manufacturing (CAD/CAM), computer aided software engineering (CASE), geographic information systems (GIS), and office information systems. The relational approach is inadequate for these application domains primarily because it offers a restricted type system (flat relations, tuples, and attributes) [Codd70] that lacks the modeling power and expressiveness required to represent and manipulate the complex structures that they generally require. This creates an "impedance mismatch" between the type system of the DBMS and that of the programming language used to access the database system [Zdon90], that hurts performance and makes applications difficult to develop and maintain.

Since the 1980's object-oriented database technology has advanced significantly, and during the last five years, Object-Oriented Database Management Systems (OODBMSs), have gained a foothold in the commercial marketplace. For example, current commercial OODBMSs include ObjectStore [Lamb91], Objectivity [Objy92], Ontos [Ontos92], O_2 [Deux91], GemStone [Butter91], and Versant [Versan92]. Despite their greater expressive power, however, whether OODBMSs continue to gain in popularity will ultimately depend upon their ability to meet the stringent performance demands of their potential market. In particular, it is crucial that OODBMSs provide high-performance when accessing in-memory data, since the applications that OODBMSs must support are computationally very intensive. For example, CAD systems typically read an engineering design into memory, and then simulate the function of the design, or in the case of a VLSI CAD system, optimize the design using certain constraints.

This thesis examines a technique that is known as *pointer swizzling* whose aim is to improve the performance of OODBMSs while accessing persistent objects that have been cached in main memory. Pointer swizzling improves performance by converting pointers from their disk format (an object identifier) to a more efficient in-

memory format (a direct memory address) when objects are faulted into memory by the OODBMS. The thesis presents an in-depth analysis of the performance of several different approaches to pointer swizzling, including several schemes that perform software-based swizzling. The thesis also presents the design of a memory-mapped storage manager, QuickStore, which provides full support for pointer swizzling using standard virtual memory hardware. The performance results presented in the thesis give an accurate and comprehensive picture of the differences in performance between software and hardware-based swizzling techniques.

An issue that is closely related to pointer swizzling, and which appears several times in the thesis, is that of providing recovery services [Gray78] in an OODBMS. Implementing recovery in an OODBMS poses new challenges due to the tight integration between the application programming language used to access the database and the database system itself. In addition, the characteristics of the applications that OODBMSs must support, and the use of a client-server architecture, both add new performance issues. In the thesis we study several alternative ways of performing recovery in the context of an OODBMS that also provides full support for pointer swizzling.

The remainder of the thesis is organized as follows. Chapter 2 presents an overview of pointer swizzling. We list the major criteria that can be used to distinguish different pointer swizzling techniques and discuss the major advantages and disadvantages of different approaches to pointer swizzling.

Chapter 3 discusses the design and implementation of EPVM 2.0 (E Persistent Virtual Machine), an interpreter for the E programming language that incorporates a novel software swizzling technique. Chapter 3 also presents the results of a performance study comparing the performance of EPVM 2.0 with an alternative implementation of software-based pointer swizzling in E. The major distinction between the software-based schemes compared in the study is that EPVM 2.0 uses a *copy* swizzling technique, while the other swizzling scheme uses *in-place* swizzling (see Chapter 2). In addition, the performance of the software schemes is compared to that of ObjectStore, a commercial OODBMS.

Chapter 4 discusses the design of QuickStore, a memory-mapped storage system that supports hardware-based pointer swizzling. Unlike the implementation of pointer swizzling in EPVM 2.0, which uses software checks to detect references to non-resident objects, QuickStore uses standard virtual memory hardware to trigger the transfer of persistent data from secondary storage into main memory. The advantage of this approach is that access to in-memory persistent objects is just as efficient as access to transient objects, i.e. application programs access objects by dereferencing normal virtual memory pointers, with no overhead for software residency checks as in EPVM 2.0.

Chapter 5 presents the results of a performance study that compares the performance of QuickStore with that of a software-based swizzling technique implemented in the context of E. The results of the performance study present a clear and accurate picture of the differences between hardware and software-based pointer swizzling schemes.

Chapter 6 presents the design of several alternative ways of implementing recovery in the context of QuickStore. The two major approaches to implementing recovery that are considered are a scheme that uses *diffing* at clients to determine the modified portions of objects, and a technique know as whole-page logging which logs entire modified pages. Chapter 6 also presents a detailed performance study of the various recovery techniques.

Finally, Chapter 7 summarizes the contributions of the work presented in the thesis.

CHAPTER 2

OVERVIEW OF POINTER SWIZZLING

The number of different, possible pointer swizzling techniques is surprisingly large. The first attempt at presenting a thorough taxonomy of swizzling techniques appeared in [Moss92]. However, since that time the number of criteria that have been used to distinguish different swizzling schemes has grown considerably. This chapter discusses the salient features that distinguish alternative pointer swizzling techniques and presents a survey of related work on pointer swizzling. Figure 2.1 illustrates the different dimensions of pointer swizzling that are discussed.

2.1. Swizzling vs. No Swizzling

One obvious approach is to do no pointer swizzling at all. We note that O_2 [Deux91] which is a commercial system, performs no pointer swizzling. Under this approach the unique *object identifier* (OID) [Khosh86] contained in a pointer is used to *lookup* the actual memory location of the object referenced by the pointer whenever the pointer is dereferenced. The lookup process generally involves a relatively expensive search in an in-memory table. The goal of pointer swizzling is to avoid the lookup cost by converting (i.e. swizzling) a pointer from its OID form to a memory address. In the simple case, the swizzled pointer can then be used to directly access the object that it references.

2.2. Hardware vs. Software-based Swizzling

Pointer swizzling schemes have traditionally used software checks to determine if a pointer has been swizzled. More recently, hardware-based swizzling schemes [Wilso90, Lamb91] that use virtual memory access protection violations to detect accesses of non-resident objects have been proposed. The main advantage of the hardware-based approach is that accessing memory-resident persistent objects is just as efficient as accessing transient objects because the hardware approach avoids the overhead of residency checks incurred by software approaches.

A disadvantage of the hardware-based approach is that it makes providing many useful kinds of database functionality more difficult, such as fine-granularity locking, referential integrity, crash recovery, and flexible buffer management policies. In addition, the hardware approach limits the amount of data that can be accessed during a

transaction to the size of virtual memory. This limitation could conceivably be overcome by using some form of garbage collection to reclaim memory space, but this would add additional overhead and complexity to the system. The hardware approach has been used in several commercial and research systems, including ObjectStore [Lamb91], Texas [Singh92], Cricket [Shek90], Dali [Jagad94], and QuickStore (see Chapter 4).

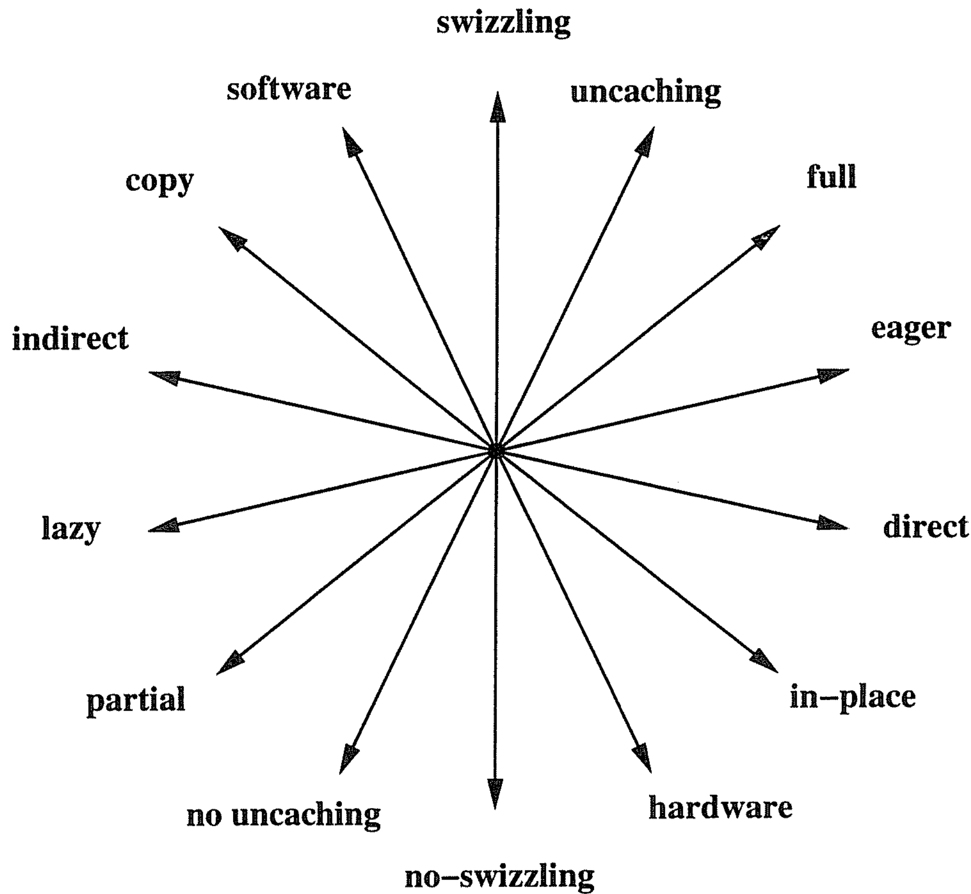

Figure 2.1. Dimensions of pointer swizzling.

2.3. In-place vs. Copy Swizzling

Copy and *in-place* strategies differ primarily in where they cache persistent objects in main memory. *In-place* refers to an approach that allows applications to access objects in the buffer pool of the underlying storage manager, while the copy approach copies objects from the buffer pool into a separate area of memory, typically called an *object cache*, and applications are only allowed to access objects in the object cache. These techniques can be used independently of whether swizzling is being done. While the copy approach incurs some cost for copying objects, it has the potential to make more efficient use of memory by only caching objects that are actually used by the

application. In addition, if pointer swizzling is being done, then the copy approach can save in terms of unswizzling work since, in the worst case, only the modified objects have to be unswizzled. Depending on the type of swizzling used, an in-place scheme may have to unswizzle an entire page of objects in the buffer pool whenever any object on the page is updated.

2.4. Uncaching vs. No Uncaching

The issue of uncaching vs. no uncaching separates systems that support the incremental uncaching of objects during a transaction from those that do not. This distinction was not made in [Moss92] since none of the techniques described there supported uncaching. The difficulty involved with allowing objects that are referenced by swizzled pointers to leave main memory during a transaction arises due to the fact that once an object is no longer in memory, the swizzled pointers that reference it become dangling references. Special care must be taken to deal with swizzled pointers in this case.

2.5. Eager vs. Lazy Swizzling

Eager swizzling schemes perform very aggressive pointer swizzling. In fact, pure eager swizzling, as defined in [Moss92], swizzles all of the pointers contained in a collection of objects (i.e. a set of objects that reference one another either directly or transitively) before an application begins accessing objects in the collection. The advantage of this approach is that no checks are needed to distinguish swizzled and unswizzled pointers.

Lazy swizzling schemes take a more incremental approach, with pointers being swizzled as a side effect of the actions taken by the application program at run-time. Lazy swizzling techniques have been categorized according to the granularity at which swizzling occurs. For example, swizzling can be performed a page-at-a-time (all pointers on a page are swizzled at once), an object-at-a-time (all pointers in an object are swizzled together), or a pointer-at-a-time (pointers are swizzled individually). Lazy swizzling techniques that swizzle individual pointers can be further categorized by the set of basic pointer operations (dereference, compare, fetch, and store) used to trigger swizzling [McAul94].

We note here that the term *eager swizzling* has a slightly different meaning in [Kemper93] and [McAul94] than in [Moss92]. [Moss92] defines an eager technique as one that eliminates the need for pointer checks by doing aggressive swizzling, while [Kemper93] and [McAul94] consider any swizzling technique that swizzles all of the pointers in an object (or page) together to be an eager technique.

2.6. Direct vs. Indirect Swizzling

Direct swizzling techniques place the in-memory address of the referenced persistent object directly in the swizzled pointer itself, while under indirect swizzling a swizzled pointer points to some intermediate data object (usually termed a *fault block*) which itself points to the target object when it is in memory. Indirect swizzling has the advantage of increased flexibility which can improve performance. For example, the indirect approach makes it easier to support the incremental uncaching of objects when software swizzling is used, since instead of having to unswizzle all of the pointers that reference an object when it is replaced, the pointer to the object contained in its corresponding fault block can simply be set to null. The disadvantages of indirect swizzling include the additional cost for the extra level of indirection and the cost of managing fault blocks.

2.7. Partial vs. Full Swizzling

Systems that use partial swizzling only swizzle a subset of all possible pointers, while systems that perform full swizzling may swizzle any pointer that references a persistent object. [Schuh90] uses partial swizzling to avoid the difficult problem (mentioned above) of unswizzling swizzled pointers when the objects that they reference leave main memory. The technique described in [Schuh90] only swizzles pointers that are local variables in functions. Variables of this type are maintained on a special *pointer stack* that is separate from the usual procedure activation stack. This significantly simplifies the task of locating swizzled pointers.

2.8. Survey of Related Work

This section discusses related work on pointer swizzling. In the discussion that follows, we first cover related work on software-based pointer swizzling techniques; following this, work that has been done on hardware-based swizzling schemes is discussed.

Some early work on software implementations of pointer swizzling was done as part of an implementation of PS-Algol [Atkin83, Cock84]. This approach used pointer dereferences to trigger the transfer of objects from secondary storage into main memory. [Moss92] presents a more recent study of software swizzling techniques that, like the study presented in Chapter 3, examines the issue of accessing persistent objects in the buffer pool of the underlying object manager versus copying them into an object cache. [Moss92] also examines the differences between eager and lazy swizzling. The lazy swizzling scheme described in [Moss92] takes an object-at-a-time approach to swizzling, in which objects that are in memory are classified as either swizzled or unswizzled. Under this approach, all pointers in an unswizzled object are swizzled immediately upon the first use of the object. This causes the

objects that the pointers reference to be faulted into memory and marked as unswizzled, after which the initial object is marked as swizzled.

One advantage of the object-at-a-time approach over the page-at-a-time approach of [Wilso90] (see below) is that it should generally perform less unnecessary swizzling and unswizzling work. A disadvantage, however, is that objects that are not accessed by a program can be faulted into memory by the swizzling mechanism, resulting in unnecessary I/O operations. In particular, unswizzled objects, while they are memory resident, have, by definition not been referenced. A restricted form of software swizzling is supported by EPVM 1.0 [Schuh90]. The major difference between the approach discussed in [Schuh90] and those presented in [Moss92], is that [Schuh90] allows objects to be replaced in memory during a transaction.

[Kemper93] examines the performance of several software swizzling schemes which, like the approach of [Schuh90], also allow objects to be replaced during a transaction. [Kemper93] explores the issues of both direct versus indirect swizzling and eager versus lazy swizzling. The results of the study in [Kemper93] were inconclusive, however, in finding a swizzling technique that was clearly superior to the others, so [Kemper93] advocates an approach termed *adaptable pointer swizzling* that attempts to combine several swizzling schemes into a single hybrid approach. [McAul94] contains a similar study to the one presented in [Kemper93], comparing swizzling performance using several different workloads. The results in [McAul94] differ from those found in [Kemper93], and indicate that indirect swizzling performs best. The study in [McAul94] differs from the work presented here in that it only examines the relative CPU cost of operations involving pointers, while we compare different swizzling schemes based on their overall system-level performance. [McAul94] also does not discuss hardware-based swizzling techniques.

A detailed proposal advocating the use of virtual memory techniques to trigger the transfer of persistent objects from disk to main memory first appeared in [Wilso90]. The basic approach described in [Wilso90] is termed "pointer swizzling at page fault time" since under this scheme all pointers on a page are converted from their disk format to normal virtual memory pointers (i.e. swizzled) by a page-fault handling routine before an application is given access to a newly resident page. In addition, pages of virtual memory are allocated to non-resident pages one step ahead of their actual use; they are access protected so that references to these pages will cause a page-fault to be signaled. The technique described in [Wilso90] allows programs to access persistent objects by dereferencing standard virtual memory pointers, eliminating the need for software residency checks.

The basic ideas presented in [Wilso90] were, at the same time, independently developed by the designers of ObjectStore [Objec90, Lamb91], a commercial OODBMS product from Object Design, Inc. However, the implementation of ObjectStore, outlined briefly in [Objec90], differs in some interesting ways from the scheme described in [Wilso90]. The most notable differences lie in the way that pointer swizzling is implemented and in how pointers are represented on disk.

Under the approach outlined in [Objec90], pointers between persistent objects are stored on disk as virtual memory pointers instead of being stored in a different disk format as in [Wilso90]. In other words, in ObjectStore pointer fields in objects simply contain the value that they last were assigned when the page was resident in main memory. When a page containing persistent objects is first referenced by an application program, ObjectStore attempts to assign the page to the same virtual address as when the page was last memory resident. If all of the pages accessed by an application can be assigned to their previous locations in memory, then the pointers contained on the pages can retain their previous values, and need not be "swizzled" (i.e., changed to reflect some new assignment of pages to memory locations) as part of the faulting process. If any page cannot be assigned to its previous address (because of a conflict with another page), then pointers that reference objects on the page will need to be altered (i.e swizzled) to reflect the new location of the page.

This scheme requires that the system maintain some additional information describing the previous assignment of disk pages to virtual memory addresses. The hope is that processing this information will be less expensive on average than swizzling the pointers on pages that are faulted into memory by the application program. The Texas [Singh92] and Cricket [Shek90] storage systems also use virtual memory techniques to implement persistence. Texas stores pointers on disk as 8-byte file offsets, and swizzles pointers to virtual addresses at fault time as described in [Wilso90]. Cricket, on the other hand, uses the Mach external pager facility to map persistent data into an application's address space (see [Shek90] for details).

[Hoski93] examines the performance of several object faulting schemes in the context of a persistent Smalltalk implementation. [Hoski93] includes one scheme that uses virtual memory techniques to detect accesses to non-resident objects. The approach described in [Hoski93] allocates fault-blocks, special objects that stand in for non-resident objects, in protected pages. When the application tries to access an object through its corresponding fault block, an access violation is signaled. The results presented in [Hoski93] show this scheme to have very poor performance. It is not clear, however, whether this is due to the overhead associated with using virtual memory or the extra work that must be performed during each object fault to locate and eliminate any outstanding pointers to the

fault block that caused the fault. (This work involves examining the pointer fields of all transient and persistent objects that contain pointers to the fault block.) Finally, we note that the effects of page replacement in the buffer pool and updates are also not considered in [Hoski93].

CHAPTER 3

POINTER SWIZZLING IN E

This chapter examines the performance of several software-based pointer swizzling schemes that were implemented in the context of the E programming language [Rich93]. In addition, the performance of the software-based schemes is compared to that of ObjectStore, a commercial OODBMS that incorporates hardware support for pointer swizzling. The issue of whether it is even beneficial to do pointer swizzling is also examined.

3.1. Introduction

E is a persistent programming language that was originally designed to ease the implementation of data-intensive software systems, such as database management systems, that require access to huge amounts of persistent data. The implementation of E uses an interpreter, the E Persistent Virtual Machine (EPVM), to coordinate access to persistent data [Schuh90] that is stored using the EXODUS Storage Manager [Carey89]. Under the approach taken by EPVM 1.0 (the original implementation of EPVM), memory resident persistent objects are accessed in-place in the buffer pool of the EXODUS Storage Manager (ESM). In addition, EPVM 1.0 provides support for a limited form of pointer swizzling.

This chapter presents an alternative implementation of EPVM (EPVM 2.0) that is targeted at CAD environments. One common example of a CAD application is a design tool that loads an engineering design into main memory, repeatedly traverses the design while performing some computation over it, and then saves the design again on secondary storage. An important property of design applications is that they perform a considerable amount of focused work on in-memory persistent objects. A major fraction of this work involves the manipulation of persistent objects via pointers.

The approach employed by EPVM 2.0 is to maintain an object cache in virtual memory containing the set of persistent objects that have been accessed by an E program. Objects are copied from the ESM buffer pool and inserted into the cache as they are accessed by a program. In addition, the cumulative effects of updates to a persistent object are propagated back to ESM via a single write operation when a transaction commits. The object cache supports full swizzling of pointers that reference small, sub-page objects, while pointers to large, multi-page

objects are not swizzled. Pointers to small objects are swizzled one-at-a-time as they are used by an E program. If the volume of data accessed by an individual transaction exceeds the size of real memory, objects are swapped to disk in their swizzled format by the virtual memory subsystem.

To evaluate the effectiveness of the design of EPVM 2.0, this chapter also presents the results of a number of performance experiments that were conducted using the OO1 benchmark [Catte91]. The experiments compare EPVM 2.0 with three alternative pointer swizzling architectures. The first architecture is represented by EPVM 1.0 which supports a more limited form of pointer swizzling than EPVM 2.0. The second architecture does not support pointer swizzling, and corresponds to using a conventional non-persistent programming language, i.e. C++, to call ESM directly. The final system is ObjectStore V1.2 [Objec90], a commercially available object-oriented DBMS. ObjectStore uses a memory-mapped approach to support pointer swizzling and fault objects into main memory.

The experimental results illustrate the tradeoffs between the different implementations of object faulting and pointer swizzling (including doing no swizzling) and examine the impact of the different schemes on the generation of recovery information. In the case of EPVM 2.0, alternative ways of managing the migration of persistent objects from the ESM buffer pool into the object cache are also examined. All of the systems included in the study are based on a client-server architecture and feature full support for transactions, concurrency control, and recovery. The client-server version of ESM [Frank92, Exodu93] was used to store persistent data for the experiments based on EPVM 2.0, EPVM 1.0, and C++.

The remainder of the chapter is organized as follows. Section 3.2 presents a detailed description of the implementation of EPVM 2.0. Section 3.3 describes the benchmark experiments. Section 3.4 presents the performance results. Section 3.5 takes a more detailed look at the CPU costs associated with the approach of EPVM 2.0. Section 3.6 summarizes the conclusions reached in this chapter.

3.2. EPVM 2.0 Design Concepts

3.2.1. Object Caching

As mentioned in Section 3.1, the ESM is used to provide disk storage for the persistent objects that are accessible to an E program. EPVM 2.0 copies objects from the ESM client buffer pool into the object cache as they are accessed. Separate schemes are used to cache objects that are smaller than a disk page, hereafter referred to as small objects, and large objects that can span any number of pages on disk. Small objects are copied from the ESM client buffer pool in their entirety and stored in the cache in individual contiguous regions of virtual memory. A

bitmap, which is appended to the beginning of each region, is used to record the locations of all swizzled pointers contained in the small object.

Large objects are cached a page-at-a-time in units of 8K bytes. Individual large object pages are cached on demand, so that only the pages that have been referenced are cached. Each cached page has appended to it a bitmap that keeps track of all swizzled pointers on the page. Different pages of a large object are not necessarily stored contiguously in virtual memory. This fact has important implications for pointer swizzling since it essentially means that pointers to large objects cannot be swizzled because accesses to large objects through such pointers can span page boundaries.

Objects that have been cached are organized using a hash table on each object's identifier (OID). Entries in this hash table are pointers to object descriptors (see Figure 3.1). In the case of a small object, the object descriptor contains a single pointer to the copy of the object in virtual memory. Paired with this pointer are a low and a high byte count that are used to keep track of the range of modified bytes in the object. For each update of a small object the range is expanded by decrementing and incrementing the low and high byte counts respectively, as needed. Note that this method of keeping track of the modified portion of an object works best when there is some locality of updates to objects. The range of modified bytes, together with the bitmap stored at the beginning of the object, determines the portion of the object that must be written to disk, and the subset of swizzled pointers in the object that must be unswizzled when a transaction completes.

The object descriptor of a large object contains an array of pointers to pages of the large object. Each large object page has associated with it a low and high byte count that are used to keep track of the modified portion of the page, in a manner analogous to that used for small objects.

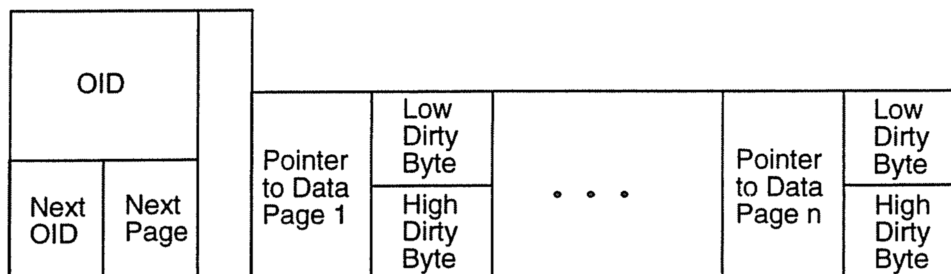

Figure 3.1. An object descriptor for a large object.

Figure 3.2 shows an example of small and large objects that have been cached. The small objects' descriptors each contain a single pointer to their respective objects, while the large object's descriptor contains pointers to two pages of the large object. Note that these pointers point to the beginning of the object/page and not to the corresponding bitmap. In Figure 3.2, the last page of the large object has not been referenced, so the object descriptor points only to the first two pages.

The object descriptors of small objects are organized in a second hash table according to the disk page on which their corresponding objects reside. All small objects that reside on the same disk page are in the same overflow chain. This allows the effects of updates to all objects on the same page to be propagated back to the ESM buffer pool at the same time when a transaction commits. Large objects are kept in a separate linked list that is traversed at the end of a transaction to write back dirty portions of large object pages.

Figure 3.2 depicts two small objects that reside on the same disk page. The objects are linked together by the next page pointers in their object descriptors. Of course, it is possible that objects from different disk pages may be found in the same overflow chain if their page numbers hash to the same value. In practice, however, the low cost

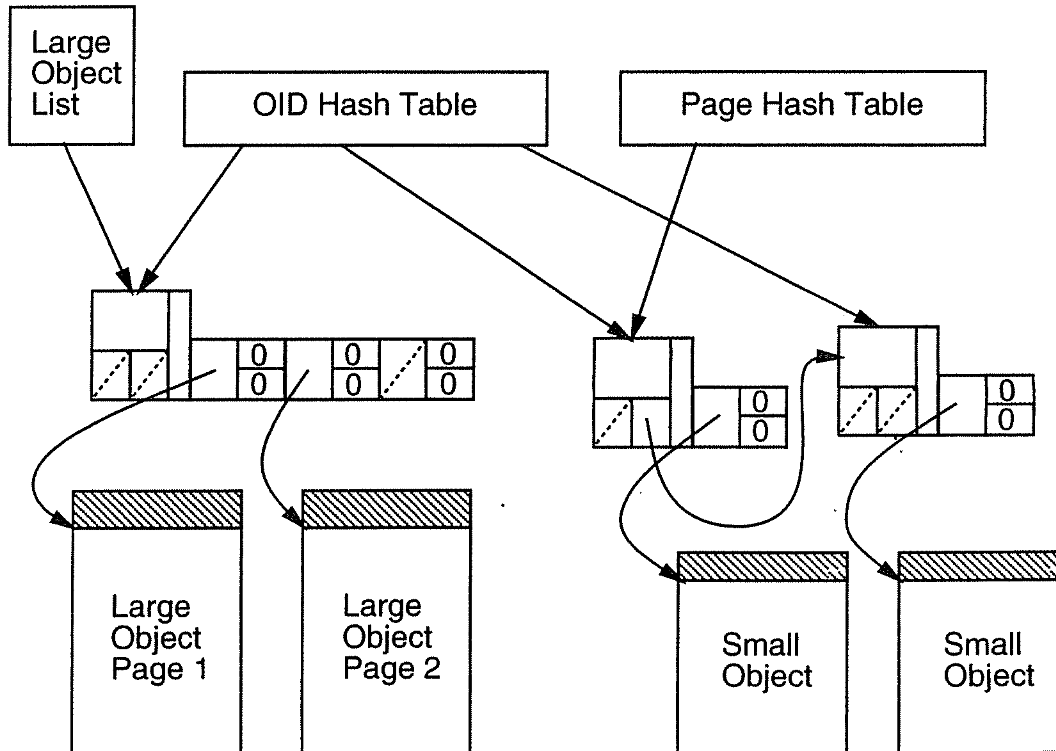

Figure 3.2. Object cache containing small and large objects.

of this strategy, plus the fact that such collisions are rare, allows it to perform well.

3.2.2. Pointer Swizzling in EPVM 2.0

Since all persistent objects are accessed through pointers in E, it is important to provide an efficient mapping from E pointers to persistent objects. When a pointer is dereferenced by an E program it may be in one of two states: **unswizzled**, in which case it contains the value of an object identifier (OID); or **swizzled**, meaning that it contains a direct memory pointer. Dereferencing an unswizzled pointer basically incurs the cost overhead of a lookup in the OID hash table in order to obtain a pointer to the referenced object. Dereferencing a swizzled pointer avoids this cost as a swizzled pointer contains a direct memory pointer to the object that it references.

While the difference in dereferencing cost may seem small, it is important to remember that tens or hundreds of thousands of pointer dereferences can occur during the execution of a program. Hence, the potential savings offered by pointer swizzling is indeed large. A key assumption of any pointer swizzling scheme is that pointers are used often enough (on average) to justify the costs of doing the pointer swizzling.

EPVM 2.0 supports a pointer swizzling scheme that converts pointers from their OID form to direct memory pointers incrementally during program execution. The goal is to quickly and cheaply convert pointers to their swizzled format so that a program "sees" only swizzled pointers during the majority of the time it is executing.

Software checks are used to distinguish swizzled and unswizzled pointers. This seems reasonable since the price of such checks should be a very small part of overall program execution time; this fact has been independently confirmed in [Moss92]. Furthermore, it is possible to do standard kinds of compiler optimizations to eliminate checks from a program (though the E compiler currently does not do this). The swizzling scheme used in EPVM 2.0 is further characterized by the fact that it swizzles pointers one-at-a-time, as opposed to the approach described in [Wilso90] which swizzles them a page-at-a-time, and [Moss92] which swizzles pointers at the granularity of objects. The type of swizzling scheme used by EPVM 2.0 is referred to as an 'edge marking' scheme in [Moss92].

Implementation Strategy

Since pointers are swizzled dynamically during program execution, the key decision that must be made is when during execution to actually do the swizzling. One possibility is to swizzle pointers when they are dereferenced. To see how this is done, consider in detail what happens when an unswizzled pointer is dereferenced during the execution of an E program. First, the memory address of the pointer is passed to an EPVM function that performs a

lookup in the OID hash table using the value of the OID contained in the pointer. An E pointer is composed of a 12 byte OID (volume id: 2 bytes, page id: 4 bytes, slot number: 2 bytes, unique field: 4 bytes) and a 4 byte offset field. If the referenced object is not found, then it must be obtained from the EXODUS Storage Manager (ESM), possibly causing some I/O to be done, copied into virtual memory, and inserted into the cache. The pointer may then be swizzled since the virtual memory addresses of both the pointer and the object it references are known. This type of swizzling will be referred to as **swizzling upon dereference**.

The advantage of this scheme is that pointers that are never dereferenced are never swizzled, so the amount of unnecessary swizzling work is minimized. Furthermore, since only pointers to referenced objects are swizzled, unnecessary I/O operations are avoided. Swizzling upon dereference does present a major problem, however. In particular, when a pointer is dereferenced, it has often already been copied into a temporary memory location, e.g. a local pointer variable somewhere on the activation stack. Swizzling upon dereference, therefore, fails to swizzle pointers between persistent objects and can, in effect, force programs that use these pointers to work with unswizzled pointers throughout their execution.

The approach used by EPVM 2.0 is to swizzle pointers within objects as they are "discovered", i.e. when the location of the pointer becomes known. We shall call this type of swizzling **swizzling upon discovery**. A pointer within an object may be discovered when it is involved in the basic pointer operations fetch or compare (see Section 2.5), i.e. when the value contained in the pointer is copied or compared to another pointer. In the context of EPVM, pointers are discovered as follows. First, EPVM is passed the persistent address of the pointer that is a candidate for swizzling. The contents of this persistent address are then used to locate the object containing the candidate pointer in the cache. (Note that this initial step may involve actually caching the object that contains the pointer to be swizzled.) Once the virtual memory address of the candidate pointer is known, its contents can be inspected, and if it is not already swizzled, used to perform a lookup in the OID hash table to find the object that it references. If the object denoted by the candidate pointer is found in the cache, then the candidate pointer is swizzled. Note that this swizzling scheme solves the major problem associated with swizzling upon dereference since pointers within persistent objects are swizzled.

Next, consider the case when the object referenced by the candidate pointer is not found in the cache. One alternative would be to go ahead and cache the object. This "eager" approach could result in unnecessary I/O operations, however, since the object referenced by the candidate pointer may in fact not be needed by the program. For example, consider a persistent collection object that is used to store pointers to objects of some other class. The

routine that implements deletion from the collection may need to compare the value of a pointer being deleted with an arbitrary number of pointers in the collection. Each of these comparisons discovers a pointer contained in the collection object, so the deletion operation could fault in a large number of objects if eager swizzling upon discovery were used. Since a swizzling scheme should avoid causing unnecessary I/O operations, EPVM 2.0 takes a lazy approach in which it does not swizzle the candidate pointer when the object that it references is not already cached. The reader should note that the distinction made here between eager and lazy swizzling upon discovery is somewhat different from the general distinction between eager and lazy pointer swizzling. Swizzling upon discovery of either kind falls into the general category of lazy swizzling techniques discussed in Chapter 2.

In summary, the swizzling scheme used by EPVM 2.0 uses only pointer dereferences to fault objects into the cache. Then, once an object is in the cache, pointers that reference the object are swizzled when their locations are discovered. Pointers to an object that are discovered before the object has been referenced are not immediately swizzled. In addition, there is a slight caveat that should be made at this point, which is that EPVM 2.0 also employs the limited form of swizzling upon dereference used by EPVM 1.0. This is done as an optimization to handle the case when an unswizzled local pointer variable may be dereferenced a number of times in succession. Swizzling the pointer when it is first dereferenced lessens the cost of the dereferences that follow and can improve performance [Schuh90]. It is important to note that the benefits of this type of swizzling are temporary, because when the function containing the local variable finishes its execution, the local variable that has been swizzled is lost. The swizzling upon dereference component of our swizzling scheme will not have a noticeable impact on the performance of EPVM 2.0 in the upcoming performance experiments, but we mention it here for the sake of completeness.

Lastly, note that our swizzling approach restricts swizzling activity to only those pointers that are actually used by a program, so programs that do not use many pointers do not have to pay a big price in terms of swizzling overhead. Also, only those objects actually needed by the program are cached, so no extra I/O activity results from swizzling.

The example in Figure 3.3 is designed to illustrate the differences between swizzling upon dereference and the eager and lazy variations of swizzling upon discovery that were described above. The function *TotalCost* traverses an assembly of persistent part objects (which is assumed to form a tree for simplicity) in depth first order and calculates the total cost of the assembly. Each part object contains a cost field and three pointers to subparts. We also assume that the collection of part objects is as shown in Figure 3.4a, i.e. there are eight part objects in the collection

whose OIDs are represented by the letters A to H, and the objects form a tree of height two. Figure 3.4a depicts the format of the part objects when they are stored on disk and the connections between parts are represented by OIDs.

Note that the only pointer that is actually dereferenced in the example is *root*; a transient, local pointer variable. If just swizzling upon dereference is used while executing *TotalCost*, then only *root* will be swizzled, and the *subPart* pointers contained within part objects will always remain in their unswizzled form. This implies that repeated traversals of the parts assembly will always encounter unswizzled pointers, i.e. the assembly will remain in the format shown in Figure 3.4a.

Pointers located within part objects are discovered by the *TotalCost* function when the expression *root->subPart[i]* is evaluated in line 8. Note that whenever a part object is visited, all three of the *subPart* pointers located in the object are discovered. Suppose that the collection of parts shown in Figure 3.4a is repeatedly traversed using the *TotalCost* function, beginning at object A, to a depth of 1. If the eager implementation of swizzling upon discovery is used, then all three subparts of each leaf node in the subtree visited by *TotalCost* are cached. Figure 3.4b shows the basic structure of the part assembly in memory after the first traversal of the parts using this method. In this example, a total of eight part objects are read from disk and cached, which is double the number of

```
1  dbstruct part {              // structure of a part
2     dbint pCost;
3     part *subPart[3];
4  };

5  int TotalCost(part *root, int depth) {
6     int totCost = 0;
7     for (int i = 0; i < 3; i++)
8        if (root->subPart[i] && depth>0 )
9           totCost += TotalCost(root->subPart[i], depth-1);
10    totCost += root->pCost;
11    return totCost;
12 }
```

Figure 3.3. Example E function.

objects actually needed.

Next, consider how the swizzling scheme used in EPVM 2.0 behaves when doing the same traversal. After the first traversal of the collection, the part objects that have been cached will appear as in Figure 3.4c. Note that all of the objects accessed by the program have been cached, but that the pointers among the objects are still in their unswizzled OID form. In this case, none of the *subPart* pointers have been swizzled since, when they are discovered on line 8 during the first traversal, the objects that they reference are not yet in the cache. The objects are faulted into the cache during the first traversal when the pointer *root* is dereferenced on line 8. After a second traversal, the structure of the collection is as in Figure 3.4d. Note that all of the pointers between objects that have been visited by the program are swizzled, and that further traversals of the collection will dereference only swizzled pointers.

3.3. Performance Experiments

The performance experiments were done using the traversal portion of the OO1 Benchmark [Catte91]. The traversal portion involves repeatedly traversing a collection of part objects, beginning at a randomly selected part, in

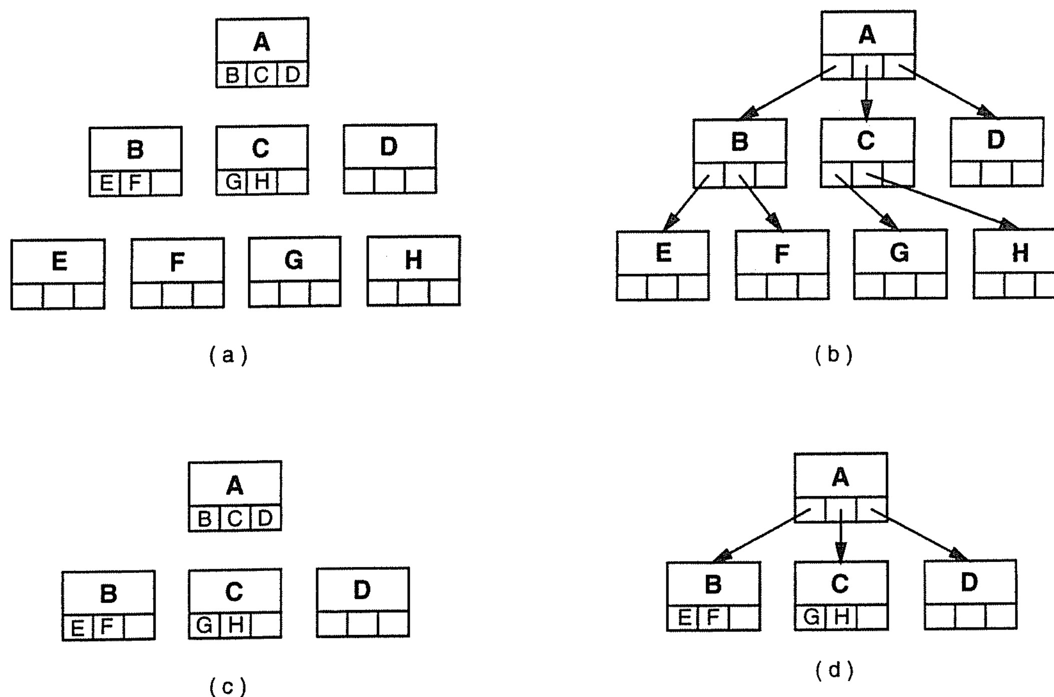

Figure 3.4. Different representations of a collection of objects.

a depth-first fashion to a depth of 7 levels. Each of the individual traversals is referred to as an iteration of the benchmark. As each part is visited during an iteration a simple function is called with four values in the part object as parameters. In addition, the ability to update part objects was added, so that each time a part object is visited, a simple update can be performed with some fixed probability. The update operation was defined as incrementing two 4-byte integer fields contained in the part object.

A total of eight different software versions were evaluated. These software versions can be classified into four basic architectures (see Section 3.3.1). The experiments compare the performance of the different architectures and investigate the relative performance of several versions of the approach used by EPVM 2.0. A number of experiments that vary the frequency with which updates are performed on objects were also conducted. This was done to assess the impact that the different swizzling approaches have on the generation of recovery information. All of the architectures that are examined offer equivalent transaction facilities, i.e page level locking, atomicity of transactions, and transaction rollback. Some architectures attempt to batch updates of objects together and generate recovery information for all of the updates made to an object at the end of the transaction, while other architectures take the traditional database approach of generating log records for each individual update. Both approaches have important implications for systems that do redo/undo logging. The experiments also compare the different software versions using a small database that fits into main memory and a large database that represents a working set size that is bigger than main memory [Catte91].

3.3.1. Software Versions

The first architecture, which is shown in Figure 3.5, results when a conventional non-persistent programming language, i.e. C++, is used to call ESM directly. This approach accesses objects in the client buffer pool of ESM using a procedural interface. The routines that make up the ESM interface are linked with the application at compile time and the client buffer pool is located in the application's private address space. In all of the experiments, the server process was located on a separate machine that was connected to the client over a network.

Accesses occur within a particular transaction, and take place during a visit to an object as follows. When an application first wants to read a value contained in an object, it calls an ESM interface function. The interface function requests the page containing the object from the server if necessary (possibly causing the server to perform some I/O on its behalf), and pins the object in the client buffer pool. Next, the interface function returns a data structure to the application, known as a user descriptor [Carey89], that contains a pointer to the object. The applica-

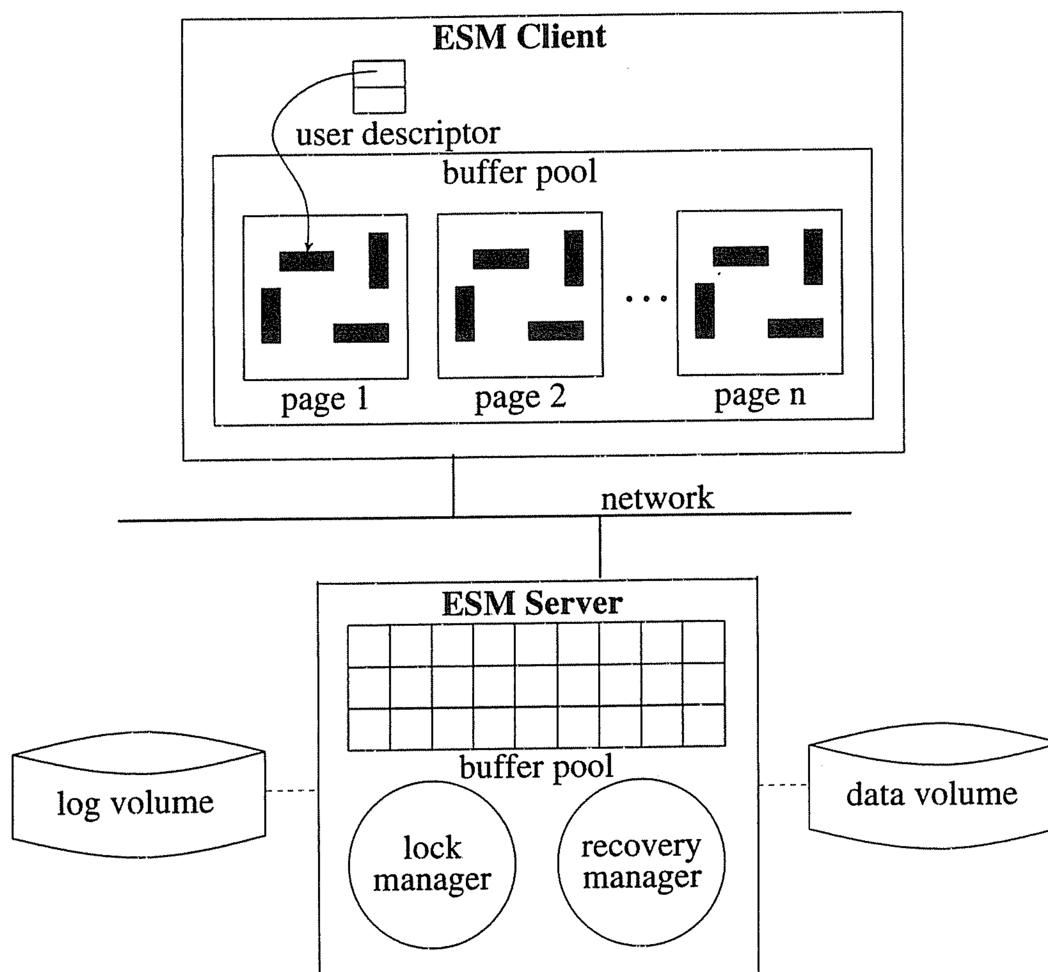

Figure 3.5. Architecture 1.

tion can then read values in the object any number of times by following the pointer contained in the user descriptor.

Each time the application wants to update a portion of an object it must call an interface function, passing in (among other things) the new value and a user descriptor pointing to the object as parameters. The update function then updates the specified portion of the object in the client buffer pool and generates a log record for the update using the old value contained in the object and the new value which was passed as a parameter. When an application is finished visiting an object it calls an ESM function to unpin the object in the client buffer pool. If all objects on the page are unpinned at this point, the page becomes a candidate for replacement by the client buffer manager.

Note that, in this architecture, a pin/unpin sequence of operations on an object generally takes place during a very short period of time relative to the life of a program, often during a single invocation of a function. This causes an object to be pinned and unpinned multiple times if it is visited more than once by a program. In addition, each

update operation causes a log record to be generated.

In the current release of ESM, data pages are cached in the client's buffer pool between transactions. However, the client must communicate with the server to reacquire locks for cached pages that are accessed in succeeding transactions. Transaction commit involves shipping dirty data pages and log pages back to the server, writing log pages to disk, and releasing locks [Frank92]. No pointer swizzling is done in this architecture. A single software version based on this architecture was used (referred to as CESM). The size of the ESM client and server buffer pools was 5 megabytes.

The second architecture represents the approach taken by EPVM 1.0 [Schuh90]. Figure 3.6 shows the client portion of this architecture (the server portion is identical to the server shown in Figure 3.5). EPVM 1.0 avoids calls to the storage manager by maintaining a cache of worthy objects in the ESM client buffer pool. Objects are accessed in the following way. The first time that an object is needed by an application, EPVM 1.0 calls an ESM interface function that pins the object in the client buffer pool, and returns a user descriptor through which the object can be referenced. This may involve communication between the client and the server and the server may in turn perform some I/O on behalf of the client. Next, EPVM 1.0 creates an entry for the object in a hash table based on the object's OID. The hash table maintains a mapping from OIDs to user descriptors that remains valid until either the client buffer pool becomes full or program execution completes.

Objects that are cached in the ESM buffer pool are accessed by doing a lookup in the OID hash table, or by following a swizzled pointer since EPVM 1.0 supports a limited form of pointer swizzling (see Section 2.7). Updates to objects, however, require EPVM 1.0 to invoke a storage manager interface function. The interface function updates the object in the buffer pool and generates a log record for the update. In addition to the usual operations performed by ESM to commit a transaction, transaction commit requires that EPVM 1.0 scan the OID hash table and unpin all objects.

In order to measure the effectiveness of the swizzling technique employed by EPVM 1.0, experiments were performed using two versions of this architecture. The first version had the limited form of pointer swizzling enabled, and the second had swizzling turned off. These versions will be referred to as EPVM1 and EPVM1-NO, respectively. Again, 5 megabyte client and server buffer pools were used.

The third architecture investigated corresponds to the approach taken by EPVM 2.0. Let us briefly review how objects are accessed with this architecture. When an object is first needed by an application program, EPVM 2.0 calls ESM on behalf of the application. ESM then pins the object in the client buffer pool, as shown in Figure 3.5.

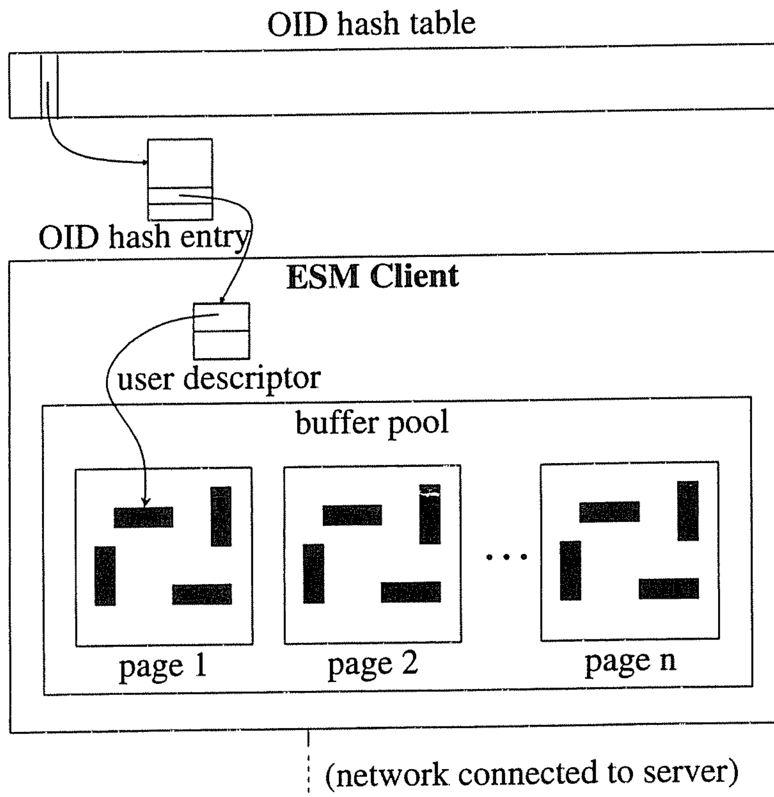

Figure 3.6. Architecture 2.

Next, EPVM 2.0 uses the user descriptor returned by ESM to copy the object into virtual memory, and EPVM 2.0 inserts the object into the object cache in the manner depicted in Figure 3.2. EPVM 2.0 then calls ESM to unpin the object in the client buffer pool. All subsequent reads or updates of the object during the current transaction occur in the cache and are handled exclusively by EPVM 2.0.

During transaction commit EPVM 2.0 scans the page hash table and for each small object that has been updated, EPVM 2.0 calls ESM to pin the object in the client buffer pool and update the object. Note that this may involve communication between the client and the server if the page containing the object is no longer present in the client buffer pool. When ESM updates the object in the client buffer pool, the new value of the modified portion of the object and the old value located in the client buffer pool are used to generate a log record for the update. Updates of large objects are handled in a similar manner, the only difference being that EPVM 2.0 invokes ESM once for each modified page of the large object.

The performance of two alternative ways of copying objects from the client buffer pool into the object cache were examined. The first copies objects one-at-a-time from the client buffer pool into the cache, while the other

copies all of the objects on a page when the first object on the page is accessed. These two schemes shall be referred to as *object caching* and *page caching* respectively. The tradeoff between the two approaches is that object caching generally requires more interaction with the storage manager, i.e. one interaction per object, while page caching requires only one interaction per page, but has the potential to perform unnecessary copying.

Four versions of this architecture were investigated. Two used object caching. In order to study the effect of buffer pool size on object caching, the size of the client buffer pool for one version was set at 5 megabytes and the client buffer pool for the other was set at 1 megabyte (while both used a 5 megabyte server buffer pool). These versions shall be referred to as OC5M and OC1M respectively. Both versions did pointer swizzling.

The third and fourth versions were designed to measure the benefit provided by the swizzling technique implemented in EPVM 2.0. Both versions do page caching and each was given a 1 megabyte client buffer pool to make the amount of memory that they used similar to the other versions. Again, a 5 megabyte server buffer pool was used for all of the experiments. The version referred to as PC1M does pointer swizzling, while the version labeled PC1M-NO does not.

The final architecture examined was that of ObjectStore V1.2 [Lamb91]. Like ESM, ObjectStore uses a client/server architecture in which both the client and server processes buffer recently accessed pages of objects. All interaction between the client and server in ObjectStore was configured to take place at the granularity of individual pages, just as in ESM. ObjectStore features basically the same transaction facilities as ESM, i.e. recovery for updates in the event of client or server failure, page level locking, and transaction rollback.

ObjectStore also supports inter-transaction caching of persistent data in the client's main memory [Lamb91]. Callback messages are sent by the server to clients in order to maintain the coherence of cached data. This allows the ObjectStore client to cache locks between transactions as well as data pages. To efficiently support callbacks, the ObjectStore client is divided into two processes [Orens92]: a callback process, and an application process. When only a single client is connected with the server, the two-process architecture does not have a noticeable effect on performance since the application process communicates directly with the server to obtain data pages and locks on those pages.

The most important difference between ObjectStore and the architectures already mentioned is that ObjectStore uses a memory-mapping scheme, similar to virtual memory, to implement pointer swizzling and fault objects from secondary storage into main memory (see Sections 2.2 and 2.8). Another important difference is that ObjectStore generates recovery information for updates of persistent data by logging entire dirty pages. Whole-page logging is

used to implement recovery largely due to the fact that ObjectStore applications are allowed to directly update objects by dereferencing normal virtual memory pointers. Because of this, ObjectStore is not able to keep track of the modified portions of pages (or objects) as is done in EPVM 2.0. The amount of real memory available to the client for caching pages of objects during a single transaction is fixed. We used 5 megabyte client and server buffer pools for all of the experiments. This architecture will be referred to as OS.

3.3.2. Benchmark Database

For ESM, the small benchmark database [Catte91] consumed a total of 489 8 K-byte disk pages (3.8 Mb) and consisted of a collection of 20,000 part objects (each object is an average of 176 bytes in size). Additionally, the Sun benchmark requires that objects be indexed, so the parts were indexed using an array of 20,000 object pointers (OIDs). An array of pointers was used instead of a B-tree index in order to keep performance differences due to differing B-tree implementations in ESM and ObjectStore from influencing the results. The total size of the index for ESM was 320,000 bytes.

The small database, including the part index, required 422 8 K-byte pages (3.3 Mb) using ObjectStore. Each part object contains connections to three other part objects in the database. These connections were implemented using pointers in both systems. The database required more disk space when using ESM largely because of differences in the way that pointers to persistent data are stored by the two systems.

The large benchmark database is identical to the small database except that 125,000 part objects were used. The large database occupied 3,057 disk pages (23.8 Mb) and the index size was 1.9 megabytes when using ESM. For ObjectStore, the large database, including the index, required 2,559 pages (19.9 Mb). 125,000 objects were used for the large database instead of 200,000 as specified in [Catte91] due to limitations in the amount of available swap space. Using 125,000 objects eliminated this problem while still providing a database that would not fit into the real memory of the workstations that were used.

3.3.3. Hardware Used

All experiments were performed using two identically configured SUN SPARCstation ELCs (approximately 20 mips). One ELC was used as the client machine and the other was used as the server. The two machines were connected via a private Ethernet. Both machines had 24 megabytes of main memory. Data was stored at the server using 70 megabyte raw disk partitions located on separate SUN0207 disk drives. One partition was used for the transaction log and the other was used to store normal data. The virtual memory swap area on the client machine

was also located on a SUN0207 and was 32 megabytes in size.

3.4. Benchmark Results

3.4.1. Small Database Results

This section contains the results of running several variations of the traversal portion of the OO1 benchmark using the small benchmark database of 20,000 objects. All of the experiments were repeated 3 times and then averaged to obtain the results that are shown. All times are listed in seconds.

Tables 3.1, 3.2, and 3.3 present the individual cold, warm, and hot iteration times when no updates are performed and the entire benchmark run is executed as a single transaction. The cold time is the execution time for the first iteration of the benchmark when no persistent data is cached in memory on either the client or server machines. The warm time is the execution time for the tenth iteration of the benchmark. The hot times were obtained by repeating the traversal done during the warm iteration, so that all of the objects were in memory and all swizzling was done prior to the beginning of the hot iteration. In Tables 3.1 and 3.2, the column labeled *I/O* gives the number of pages sent from the server to the client during each iteration. The times in Tables 3.1, 3.2, and 3.3 do not include the overhead for transaction begin and commit.

Table 3.1 compares one version from each of the four software architectures discussed in Section 3.3.1. The versions selected are generally comparable in the sense that each uses a similar amount of memory. CESM has the best time in the cold iteration, but EPVM1 does almost as well. Since CESM and EPVM1 are both in-place techniques, the small difference between CESM and EPVM1 is likely due to the overhead of inserting objects into the OID hash table for EPVM1. PC1M is 7% slower than EPVM1 due to the cost of copying full pages of objects into the object cache. OS does the worst during the cold iteration despite the fact that it performs the fewest I/O operations. Based on the the performance results obtained using QuickStore (see Chapter 5), we believe this is due to the overhead for using virtual memory and also to slower I/O performance for ObjectStore relative to ESM[1].

The ordering of times for the warm iteration in Table 3.1 is just the reverse of that for the cold iteration. OS is much faster than the other versions in the warm iteration since it incurs essentially no overhead for accessing in-memory objects in this case. PC1M is next in terms of performance. PC1M is 33% faster than EPVM1 because

[1] It is also possible that the poor performance of ObjectStore in this case is due, at least in part, to a network-related performance bug that was known to cause problems in some pre-version-2.0 ObjectStore installations on Sun systems [OO7].

Traversal without updates				
Version	Cold	I/Os	Warm	I/Os
CESM	10.586	325	0.285	2
EPVM1	10.655	327	0.180	2
PC1M	11.386	327	0.120	2
OS	12.530	217	0.066	1

Table 3.1. Single transaction without updates (times are in seconds).

Traversal without updates				
Version	Cold	I/Os	Warm	I/Os
OC5M	10.750	327	0.227	2
OC1M	11.799	434	1.979	171
PC1M	11.386	327	0.120	2
PC1M-NO	11.384	327	0.136	2

Table 3.2. Single transaction without updates (times are in seconds).

Traversal without updates		
Version	Hot	Hot w/o random
C	0.039	0.005
OS	0.039	0.005
PC1M	0.058	0.024
PC1M-NO	0.078	0.044
EPVM1	0.074	0.040
EPVM1-NO	0.082	0.048
CESM	0.230	0.196

Table 3.3. Single transaction without updates (times are in seconds).

EPVM1 incurs the overhead of inserting a large number of objects into the OID hash table while PC1M caches only 2 pages (80 objects). This is because PC1M is more aggressive at caching than EPVM1, so it already cached the additional objects during previous iterations. CESM has the worst performance in the warm iteration due to the overhead of calling ESM for each object that is visited during the iteration.

Table 3.2 presents the results for each of the versions based on EPVM 2.0. OC1M has the worst performance during the cold iteration because its small client buffer pool size forces it to reread pages from the server. It may seem surprising that OC1M is only 10% slower than OC5M given that it performs 33% more I/O operations. This is due to the fact that the server buffer pool is large enough to hold all of the pages read by the client in this case, and shipping pages from the server is much faster than reading them from disk. 0C5M is 6% faster than PC1M due to the overhead that PC1M incurs for copying full pages into the cache. The similarity of PC1M and PC1M-NO shows

that there is essentially no advantage or disadvantage to doing swizzling during the cold iteration.

In the warm iteration, Table 3.2 shows that PC1M has the best performance. PC1M and PC1M-NO do better than the object caching versions during the warm iteration because many more objects are being cached then than are pages. More precisely, 2 pages (80 objects) are cached during the warm iteration by the page caching versions, while 1097 objects are cached by the object caching versions. As above, PC1M caches fewer objects in the warm iteration because it has already cached the additional objects during previous iterations. This accounts for the seemingly strange fact that PC1M-NO (which does page caching and no swizzling) is 40% faster than OC5M (which does full swizzling). OC1M continues to reread pages from the server in the warm iteration and consequently has the worst performance. Turning to swizzling in the warm case, Table 3.2 shows that swizzling provides a 12% reduction in execution time for page caching. The times for EPVM1-NO are not shown in Tables 3.1 and 3.2. There was essentially no difference between EPVM1 and EPVM1-NO in the cold iteration for this experiment. In the warm iteration, swizzling made EPVM1 8% faster than EPVM1-NO.

The hot times in Table 3.3 represent the asymptotic behavior of each of the systems when no further conversion or copying of in-memory objects is taking place. An additional version, labeled C, has been added to Table 3.3. C represents an implementation of the benchmark coded in non-persistent C++ using transient in-memory objects. C represents the best performance that a persistent system could hope to achieve in the hot case.

We first examine architectural differences. OS does the best during the hot iteration. The fact that the performance of OS is identical to C shows that the memory-mapped architecture of OS imposes no additional overhead in the hot case. OS is 33% faster than PC1M because of the overhead that PC1M incurs for swizzle checks and EPVM 2.0 function calls. In addition, the fact that pointers to persistent objects in E are 16 bytes long as opposed to 4 bytes in OS further slows the performance of PC1M.

EPVM1 is third in terms of performance and is 21% slower than PC1M. This is because EPVM1 does not swizzle pointers between persistent objects and also because of the extra level of indirection imposed upon it by user descriptors. CESM has the worst performance in the hot iteration. Its hot time is approximately 3 times that of EPVM1 and nearly 6 times that of OS. This is due to the fact that CESM calls ESM to pin and unpin each object that is visited during the iteration. OC5M and OC1M were identical to PC1M in the hot iteration, so they are not shown. Comparing PC1M with PC1M-NO, we see that swizzling has improved performance by 26% in the hot case for page caching while swizzling makes a difference of just 10% for EPVM 1.0.

It may seem surprising that in the hot column of Table 3.3, OS is only 33% faster than PC1M. Upon closer inspection of the benchmark implementation, it was noticed that the Unix function *random* was being called during each visit of a part object as part of the overhead for determining whether or not to perform an update. The last column of Table 3.3 shows the results for the hot traversal when the overhead for calling *random* is removed. Note that OS now has approximately 5 times the performance of PC1M. This is closer to what one would expect given the differences between these two architectures. Similarly, the difference between PC1M and EPVM1 increases to 40%. Both sets of hot results have been included since we believe that they illustrate how quickly the difference in performance between the architectures diminishes when a small amount of computation is performed on each object access.

Table 3.4 contains the cold and warm iteration times for traversal without updates over the small database when each iteration is executed as a separate transaction. In Table 3.4 the relative performance of the different architectures is the same as in Table 3.1 during the cold iteration. Comparing the cold iteration times of Table 3.4 with Table 3.1 also shows that the overhead of transaction commit is relatively minor for all of the versions when no updates are done. The warm iteration results in Table 3.4 highlight the effects of inter-transaction caching. OS has the best performance in large part because it caches both data pages and locks between transactions. The OS client, therefore, only has to communicate with the server process once, to read one page that was not accessed during the previous nine iterations. The versions using ESM, on the other hand, must communicate with the server to read uncached data pages and to reacquire locks on all cached pages that are reaccessed. PC1M caches the fewest pages between transactions because it only has a 1 megabyte client buffer pool. This causes it to have the worst warm performance. We also ran this experiment without inter-transaction caching for ESM. Inter-transaction caching improved performance by 40% for CESM and EPVM1 and by just 7% for PC1M during the warm iteration.

Traversal without updates				
Version	**Cold**	**I/Os**	**Warm**	**I/Os**
CESM	10.669	325	1.962	133
EPVM1	10.712	327	2.075	135
PC1M	11.430	327	3.669	283
OS	12.734	217	0.404	1

Table 3.4. Multiple transactions w/o updates (times are in seconds).

The cold and warm times for the four versions based on EPVM 2.0 are not shown for the multiple transactions experiment. The cold iteration times were all within 1% of those shown in Table 3.2. The warm iteration times were, of course, slower than the warm times in Table 3.2 since locks on pages had to be reacquired. OC5M had the best performance in the warm iteration. It was 42% faster than PC1M and 50% faster than OC1M. Pointer swizzling made essentially no difference for either PC1M or EPVM1 in this experiment. In addition, the hot times were within 2% of the warm times for PC1M and OC1M. The hot time for OS was 0.377 seconds which is 7% faster than the warm time for OS (Table 3.4). The hot times for CESM, EPVM1, and OC5M were approximately 50% faster than their corresponding warm times.

We next consider the effect of adding updates to the traversal. Figure 3.7 presents the total execution time for a single transaction consisting of 1 cold, 9 warm, and 10 hot iterations when the update probability ranges between 0 and 1. The non-swizzling versions EPVM1-NO and PC1M-NO were each within 1% of EPVM1 and PC1M, respectively, and so are not shown. In addition, the performance of CESM was roughly 7% faster than EPVM1 throughout.

OS has the fastest time when no updates are done, however, the relative performance of OS degrades as updates are added due to the high cost of transaction commit. Based on our experience with QuickStore (see Chapter 6), we believe that transaction commit is more expensive for OS because whole-page logging is used. The performance of OS levels off once the frequency of updates is high enough so that all pages are being updated. PC1M is always faster than OS when updates are performed. The performance of EPVM1 continually degrades as the update probability is increased because it generates a log record for every update. It is a little surprising that EPVM1 is better than PC1M and OS in many cases. This is due in large part to the fact that the log records generated by EPVM1 can be processed asynchronously by the server while the transaction is running. PC1M is faster than EPVM1 when the update probability is greater than about .3. The commit time for PC1M is constant once all of the objects visited during the transaction have been updated.

OC1M has the worst performance overall in Figure 3.7 since it must reread pages from the server while the transaction is running and also during the commit phase. The performance of OC5M shows that object caching can perform quite well when its client buffer pool is large enough to avoid having to reread data pages. The difference between PC1M and OC5M is because PC1M must reread pages during transaction commit in order to generate recovery information. If PC1M is given a bigger client buffer pool then its performance is nearly identical to the

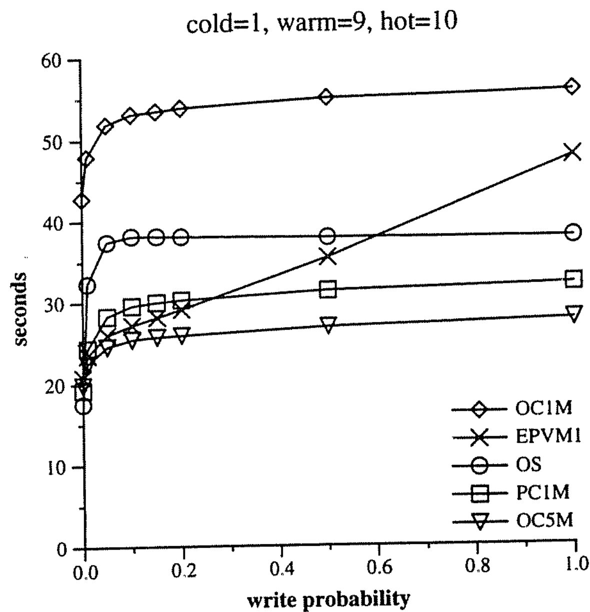

Figure 3.7. Benchmark run as a single transaction.

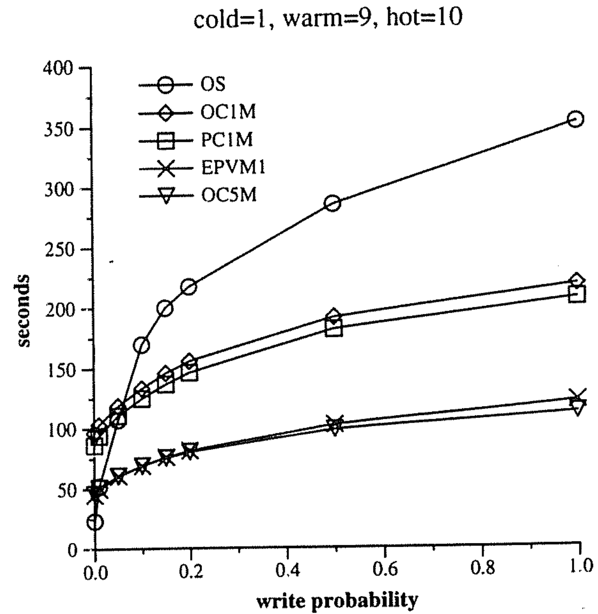

Figure 3.8. Benchmark run as multiple transactions.

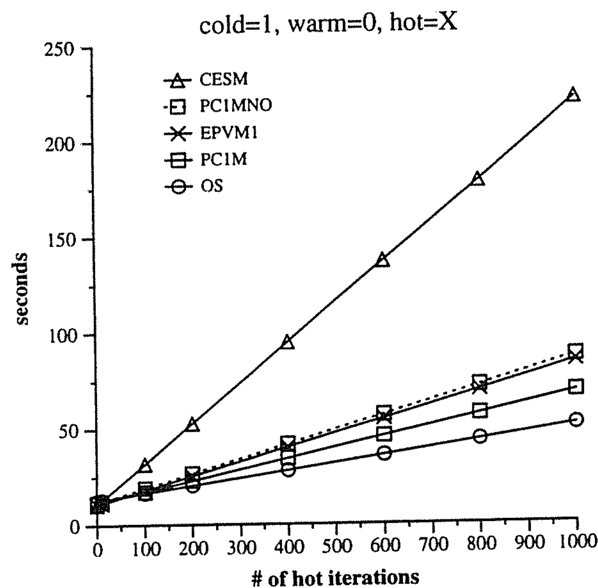

Figure 3.9. Single read-only transaction.

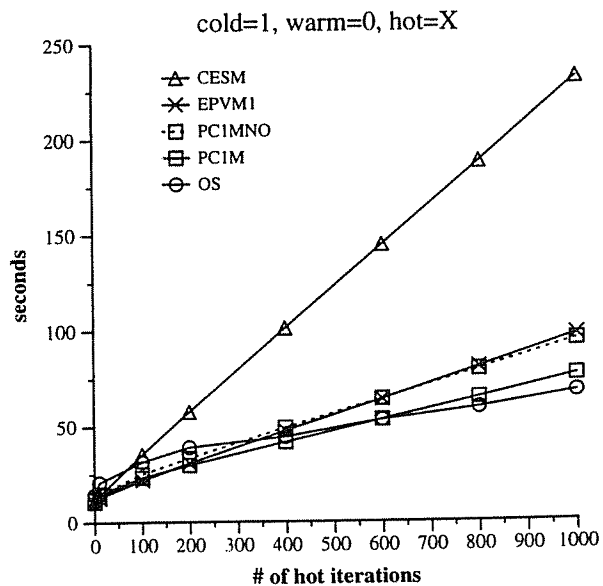

Figure 3.10. Single transaction (update prob. = .01).

performance of OC5M.

Figure 3.8 presents the overall execution time for the traversal with a varying write probability when each iteration of the benchmark constitutes a separate transaction. The curves for PC1M-NO, EPVM1-NO, and CESM are again omitted due to their similarity to the curves for PC1M, and EPVM1. OS has the best performance when the update probability is low. As the update probability is increased, however, the relative performance of OS

degrades due to the high cost of transaction commit. Comparing the relative performance of OS in Figure 3.8 with Figure 3.7, we see that OS performs worse when transactions are short. This is because the overhead for transaction commit is higher for OS than the other systems, since OS uses whole-page logging to support recovery.

OC1M is a little slower than PC1M in Figure 3.8 because OC1M must reread pages from the server while a transaction is executing. This overhead is greater than the cost of the extra copying done by PC1M. The performance of OC1M relative to PC1M improves in Figure 3.8 relative to Figure 3.7 because PC1M incurs higher caching costs when transactions are short and because the overhead for reacquiring locks for both systems during each transaction diminishes the differences in their performance.

The curve for OC5M in Figure 3.8 shows that if enough memory is available, then object caching performs the best in most cases. EPVM1 does quite well in Figure 3.8 because it avoids the extra copying overhead of PC1M and the object caching versions and also does not have to reread data pages from the server in the context of any single transaction. In addition, EPVM1 performs better in Figure 3.8 than in Figure 3.7 relative to the other systems because the costs of recovery are a smaller fraction of the total traversal cost. Thus, the performance of EPVM1 does not degrade relative to the performance of the other systems in Figure 3.8 to the degree that it does in Figure 3.7. Finally, we note that inter-transaction caching improved performance for OC5M and EPVM1 from 43% (read-only) to 29% (write prob. = 1). The improvement was smaller when updates were done because of the fixed overhead for sending dirty data pages back to the server at the end of each transaction. PC1M and OC1M posted a 5% gain in performance when caching was added.

Figures 3.9 and 3.10 fix the number of cold and warm iterations at 1 and 0, respectively, and vary the number of hot iterations between 0 and 1000. In both figures each benchmark run was a single transaction. In Figure 3.9 the update probability was 0 and in Figure 3.10 it was .01. Both figures illustrate the large difference in CPU requirements between CESM and the other versions when a large number of hot traversals are performed. Although it is not easy to see in Figure 3.9, EPVM1 has the best performance when the number of hot iterations is between 1 and 60 and OS does the best when the number of hot iterations is greater than 60. PC1M is better than EPVM 1.0 after approximately 60 hot iterations have been performed as well.

When 1000 hot iterations are done, OS is 25% faster than PC1M, 39% faster than EPVM1 and posts a 76% improvement over CESM. PC1M always does better than PC1M-NO and shows an improvement of 21% when 1000 iterations are done. The results for EPVM1-NO are not shown; however, EPVM1 was 7% faster than EPVM1-NO after 1000 iterations. The times for OC5M and OC1M were within 1% of PC1M in Figure 3.9, so their

times have been omitted as well.

In Figure 3.10, EPVM1 has the best performance when the number of hot traversals is between 1 and 160. PC1M is the best when the number of hot traversals is between 160 and 600. After 600 hot iterations OS is always the fastest. It is surprising that 600 hot traversals must be performed in order for OS to perform the best, but this is due to the relatively high cost of transaction commit for OS. Turning to swizzling, after 1000 iterations PC1M-NO was 24% slower than PC1M, and EPVM1-NO (not shown) was 7% slower than EPVM1. OC5M and OC1M are also not shown in Figure 3.10. OC1M was within 1% of PC1M in all cases. The performance of OC5M was initially 15% faster than PC1M and 6% faster than PC1M after 1000 iterations.

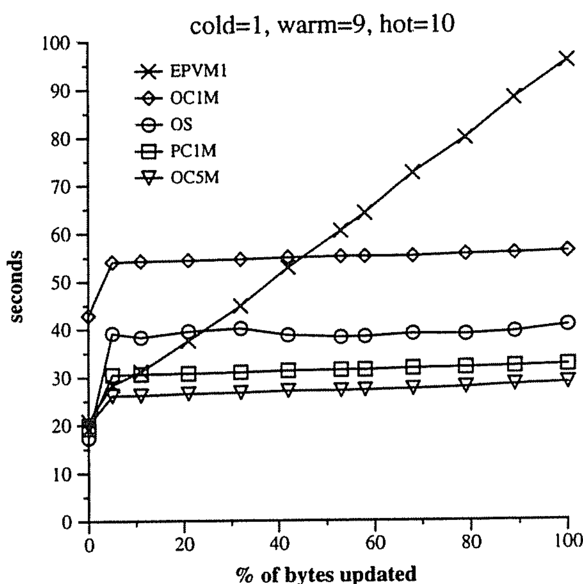

Figure 3.11. Single transaction (update prob. = .3).

Figure 3.11 demonstrates what happens when one varies the fraction of each part object that is updated while the update probability remains fixed. In this experiment, part objects were defined to contain an array of 19 integers (76 bytes) instead of the usual non-pointer data specified by the OO1 benchmark. The x-axis shows the percentage of this array that was updated. Not surprisingly, OS has relatively flat performance once any updates are done. This is because it does whole-page logging. The versions based on EPVM 2.0 also show little change in performance once updates are added. This is because while the number of log pages that were generated increased from 51 to 160 as the update fraction was increased, ESM only required about two seconds to process the extra log pages. The performance of EPVM1 degrades quickly as a larger portion is updated because it generates a log record for each

individual integer update. The number of log pages generated by EPVM1 varied from 171 (update 1 integer) to 3,325 (update whole array).

3.4.2. Large Database Results

In the large database experiments, the number of page faults that occurred was important for some systems. Page faults are listed in parentheses next to the number of normal I/O operations done by the client for the versions that experienced page faults. The number of page faults was obtained by using the Unix *getrusage* system call.

Tables 3.5 and 3.6 present the cold and warm times observed when the benchmark was executed as a single transaction and, as before, the time for transaction begin and commit is not included. CESM has the best performance in the cold iteration of Table 3.5. PC1M does fewer I/O operations, but is slower than CESM due to copying costs. EPVM1 is slower than CESM primarily because it does a less effective job of buffer management. OS has the worst performance in the cold iteration. Based on our experience with QuickStore, we believe this is due to the cost of managing paging in the client buffer pool for ObjectStore, and also to poorer I/O performance for ObjectStore relative to ESM [Orens92].

PC1M performs the best in the warm iteration, but comparing PC1M to the other architectures is not strictly fair in this case since it is allowed to use all of the available memory, as shown by the number of page faults that it experiences. CESM and EPVM1 are close in terms of performance, but EPVM1 is a little slower due to the fact that it performs more I/O and must insert objects into the OID hash table. OS is surprisingly 12% slower than EPVM1 in the warm iteration. The reasons for the poor performance of ObjectStore in this case are the same as those listed above for the cold iteration.

In Table 3.6, OC1M has the worst performance in the cold iteration because it performs more I/O operations. PC1M is a little slower than OC5M due to the overhead of copying full pages. Swizzling makes no difference for

Traversal without updates				
Version	**Cold**	**I/Os**	**Warm**	**I/Os**
CESM	38.643	1093	30.973	909
EPVM1	39.582	1149	32.716	928
PC1M	38.894	1014	24.180	33 (699)
OS	48.614	839	36.567	615

Table 3.5. Single transaction without updates (times are in seconds).

PC1M in the cold iteration. The relative times in the warm iteration are similar to the cold iteration due to the large size of the database. However, the performance of PC1M-NO is actually a little better than PC1M since swizzling dirties pages in virtual memory causing them to be written to disk more often. This fact doesn't show up in the number of page faults shown in Table 3.6 since these numbers only give the number of pages read (not written) from the swap area by the process. The times for EPVM1-NO were essentially identical to EPVM1 in both the cold and warm iterations and so are not shown.

When each iteration was executed as a separate transaction, the cold times were all within 2% of the times shown for the versions in Tables 3.5 and 3.6. In the warm iteration, the times for the versions included in Table 3.5 were also all within 2%—except for PC1M, whose performance was slower by 23%. The decrease in performance for PC1M was due to the fact that it was not able to cache as much data in virtual memory and that it also performed a lot of unnecessary copying. OC1M was 12% slower than PC1M during the warm iteration and OC5M was just 2% faster than PC1M. PC1M-NO and EPVM1-NO were each within 1% of PC1M and EPVM1, respectively, in the multiple transactions experiment. Repeating the experiment without inter-transaction caching showed that caching had much less impact on performance when using the large database. Caching improved the performance of EPVM1 by 4% and PC1M by just 2% during the warm iteration.

Figure 3.12 presents the total execution time for the traversal of the large database when 1 cold, 9 warm, and 0 hot iterations are run together as a single transaction. OS has the worst performance in most cases. It may be surprising, given the results presented in Table 3.5, that OS is better than PC1M in the read only case. PC1M is slower in this case because when it scans the page hash table during transaction commit to determine which objects have been updated, it causes a significant amount of virtual memory swapping activity. This poor performance during the commit phase makes PC1M slower than the other versions as well.

Traversal without updates				
Version	**Cold**	**I/Os**	**Warm**	**I/Os**
OC5M	38.098	1082	23.401	675
OC1M	43.107	1516	27.035	988
PC1M	38.894	1014	24.180	33 (699)
PC1M-NO	38.838	1014	22.540	33 (705)

Table 3.6. Single transaction without updates (times are in seconds).

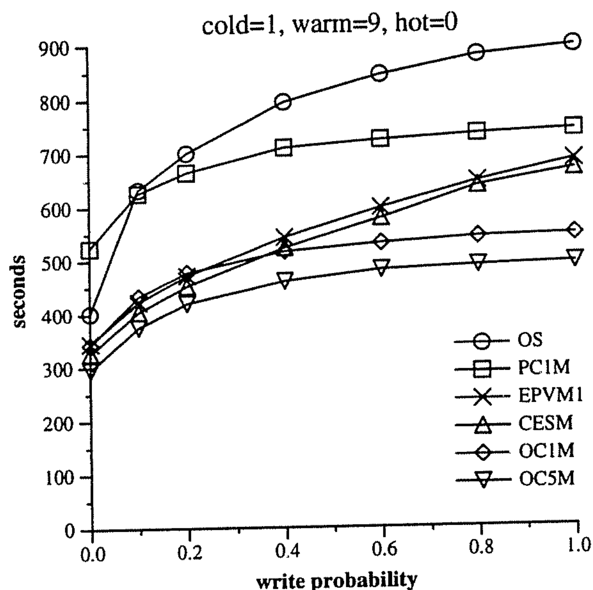

Figure 3.12. Benchmark run as a single transaction.

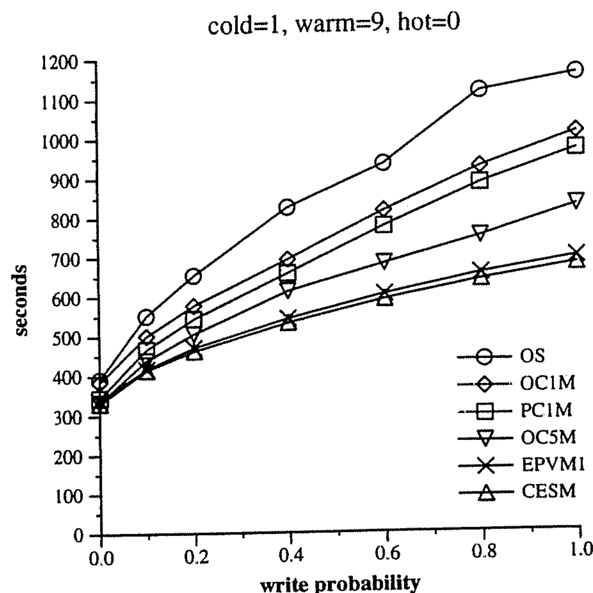

Figure 3.13. Benchmark run as multiple transactions.

It should be noted that in the large database case it is not strictly fair to compare OS, EPVM1, and CESM to the page caching and object caching versions of EPVM 2.0 since the EPVM 2.0 versions are allowed to use more memory. The comparison between EPVM1, CESM, and OS is fair, however, since these versions were given equal amounts of memory. The times for EPVM1-NO (not shown) were all within 1% of EPVM1. PC1M-NO, which is also not shown in Figure 3.12, was 4% faster than PC1M in the read only case because swizzling dirtied pages in virtual memory for PC1M caused an increase in paging activity. The difference between PC1M and PC1M-NO gradually diminished as more updates were performed, and PC1M-NO was well within 1% of PC1M when the update probability was 1.

Figure 3.13 presents the total execution time for the traversal when 1 cold, 9 warm, and 0 hot iterations are each executed as separate transactions. OS has the worst performance in Figure 3.13, even when no updates are performed. This differs from Figure 3.12 where OS was faster than PC1M in the read-only case. We believe that OS performs worse when transactions are short because of the overhead for reestablishing the memory-map at the beginning of each transaction. CESM has the best performance in Figure 3.13, but is only slightly faster than EPVM1. The relative performance of CESM and EPVM1 is better in Figure 3.13 compared to Figure 3.12. This is due to the fact that when transactions are short the caching versions generate log records more frequently. This causes the performance of all of the systems to degrade at approximately the same rate in Figure 3.13 as the update probability is increased.

Turning to object and page caching, the performance of page caching is intermediate between OC1M and OC5M in Figure 3.13. This again illustrates the tradeoff made by object caching which must reread pages from the server and page caching which caches more objects and copies more data into virtual memory. The performance of PC1M improves relative to OC1M in Figure 3.13 compared to Figure 3.12, because PC1M caches fewer objects per transaction when transactions are short. Thus, PC1M does not experience any virtual memory paging (which hurts performance) in Figure 3.13. EPVM1-NO (not shown) and PC1M-NO (not shown) were always within 1% of EPVM1 and PC1M, respectively, in Figure 3.13.

3.5. EPVM 2.0 Performance in Detail

This section presents a detailed analysis of the CPU costs associated with the approach of EPVM 2.0. In performing the analysis, PC1M—page caching using a 1 megabyte client buffer pool—was selected to serve as representative of the EPVM 2.0 approach. The results presented in this section were obtained by using the Unix pixie profiling tool [Ultrix90] and by analyzing assembly language listings of the relevant functions.

3.5.1. Hot Traversal Performance

We begin by breaking down the hot time of 58 milliseconds shown for PC1M in Table 3.3 of Section 3.4. The analysis also holds for OC5M and OC1M, the object caching versions, which perform identically in this case.

The hot iteration required a total of 776,524 CPU cycles, of which 40.97% were spent executing EPVM functions. The cost of invoking these functions was an additional 5.48% of all cycles, making the total fraction of time devoted to EPVM functions 46.45%. In addition, the E compiler also generates in-line code to handle some work associated with accessing persistent data. This work includes the swizzle checks that are performed before each pointer dereference, dereferences of swizzled pointers, and resetting the pointer stack when functions reach the end of their activation. Table 3.7 presents the combined costs of both in-line operations and EPVM functions, which together accounted for 63.75% of the hot traversal time.

Maintaining the pointer stack accounted for 8% of the total time. In Table 3.7, the time spent maintaining the stack is broken down into the time for calling and executing the EPVM function that updates the stack and the time for resetting the stack (handled by in-line code). Since updating the stack is a fairly simple operation, it could in principle also be handled by in-line code. We estimate that if this were done, the amount of time devoted to maintaining the pointer stack would decrease from 8.01% to 4.81%.

description	cost components	cycles	total cycles	%
pointer stack maintenance	call setup	3	9,840	1.26
	update pointer stack	14	45,920	5.91
	restore pointer stack	2	6,560	0.84
subtotal		19	62,320	8.01
in-line dereferences	in-line dereference check	6	59,040	7.60
	in-line swizzled dereference	7	68,880	8.87
subtotal		13	127,920	16.47
pointer fetches	call setup	5	16,400	2.11
	embedded dereference check	2	6,560	0.84
	embedded swizzle check	4	13,120	1.69
	other costs	48	157,440	20.27
subtotal		54	193,520	24.91
dummy function cost	call setup	5	16,400	2.11
	function body	29	95,120	12.25
subtotal		34	111,520	14.36
total			495,280	63.75

Table 3.7. EPVM 2.0 hot traversal cost summary.

We next consider the cost of dereferencing swizzled pointers. In EPVM, the cost of dereferencing a pointer that refers to a value contained in a persistent object depends a great deal on the value's type. For example, dereferences of pointers that refer to scalar values, such as integers, are handled by in-line code, while a dereference of a pointer that refers to a pointer, also called a pointer fetch, is always handled by an EPVM function, even when the pointer being dereferenced is swizzled.

Table 3.7 shows that in-line dereferences accounted for 16.47% of the total traversal time. An in-line dereference consists of two parts: a check to see if the pointer is swizzled (7.60%) and the actual work required to dereference the pointer (8.87%). Dereferencing pointers that referred to pointers, i.e. fetching pointers, required roughly 25% of the total time. The cost of dereference checks incurred by pointer fetches was .84%. (The dereference check is used to distinguish whether a swizzled or unswizzled pointer is being dereferenced.) Individual embedded dereference checks were cheaper than the in-line dereference checks (2 cycles vs. 6 cycles) for two reasons. The first is that the in-line code begins by loading the address of the pointer being dereferenced from memory into a register which requires 2 additional cycles (1 for the load instruction and 1 for a delay). These cycles are avoided by the embedded check because the address of the pointer being dereferenced is already in a register. (It is passed as an argument to the function.) The in-line code also inserts 2 additional delay instructions that are avoided by the

optimizer for the case of the embedded code.

Under our swizzling scheme, the location of a pointer is "discovered" when a pointer to it is dereferenced, so the referenced pointer becomes a candidate for swizzling. The checks to determine whether or not the candidate pointer could be swizzled accounted for just 1.69% of the total cost of the traversal, compared to 8.44% for dereference checks. The other work cost component of the pointer fetch category includes the cost for determining whether the pointer being fetched lies in a large object (in which case it could be split across two different large object pages), the cost for copying the pointer to a temporary location, the cost for checking whether the temporary is allowed to be swizzled, plus some additional implementation costs including the cost for returning from the EPVM function. Adding the costs for fetching pointers and in-line dereferences yields the total cost for dereferencing pointers, which is 41.38% for EPVM 2.0. Finally, the *dummy function cost* of 14.36%, shown in Table 3.7, represents the overhead for preparing an argument to be passed to the "dummy" function [Catte91] that is called once during each object access.

3.5.2. Comparison of EPVM and ObjectStore

Table 3.7 highlights the contribution of pointer dereferencing costs for EPVM 2.0, which are an important component (41.38%) of the total traversal cost. The cost of a single in-line dereference for EPVM 2.0 was 13 cycles, while dereferences that were handled by EPVM functions required 54 cycles each. By way of contrast, pointer dereferences require just 1 cycle when using OS (ObjectStore) or C (non-persistent C++) and, as Table 3.8 shows, only accounted for 4.8% of the hot traversal time for OS. It is interesting that there is a factor of 25 difference in absolute pointer dereferencing cost between the two systems, given that their overall performance differs by only 33%. The relatively small difference in overall performance is caused by the overheads for generating random numbers, function entries and exits, etc., which are incurred by both systems and which lessen the impact of pointer dereferencing costs on overall performance.

description	cost components	cycles	total cycles	%
pointer dereferences	return a scalar value	1	9840	3.6%
	return a pointer value	1	3280	1.2%
total				4.8%

Table 3.8. Costs of pointer dereferences for OS.

3.5.3. Cold Traversal Performance

This section analyzes the CPU costs associated with the cold traversal time of 11.386 seconds shown for PC1M in Table 3.1 of Section 3.4. The dominant cost factor during the cold traversal is I/O cost; however, looking at CPU cost provides some insight into the magnitude of the overhead for pointer swizzling and object faulting. Note that when faulting and swizzling costs are removed, the remaining traversal cost is basically identical to the cost of doing the hot traversal (776,524 cycles).

We first consider the overhead for caching objects, which requires the majority of the CPU time (85.93%) in the cold case. Recall that objects are faulted into the cache as the result of dereferencing an unswizzled pointer that refers to a non-resident object. The faulting cost shown for EPVM in Table 3.9 includes all additional costs needed to cache objects beyond that required for an in-line dereference of a swizzled pointer. Table 3.9 divides the caching cost into 3 categories: the time spent calling and executing ESM functions (6.44%), the time spent by EPVM copy-

description	cost components	cycles	total cycles	%
caching objects	ESM	33	426,940	6.44
	EPVM copy cost	149	1,954,010	29.46
	other EPVM costs	253	3,319,271	50.04
subtotal		434	5,700,221	85.93
swizzling upon discovery	hash & test	8	16216	0.24
	hash comparison	11	18,766	0.28
	additional checks	11	18,766	0.28
	call setup	4	6,824	0.10
	swizzle pointer	29	49,474	0.75
	set bitmap	16	27,296	0.41
	miscellaneous	3	5,118	0.08
subtotal		82	142,460	2.15
swizzling upon dereference	swizzle checks	19	6,099	0.09
	call setup	4	1,284	0.02
	swizzle pointer	21	6,741	0.10
subtotal		44	14,124	0.21
traversal work	in-line dereference checks	6	59,040	0.89
	embedded dereference checks	2	6,560	0.10
	swizzle checks	4	13,120	0.20
	other work	. . .	697,802	10.52
subtotal			776,522	11.71
total			6,633,327	100.00

Table 3.9. EPVM 2.0 cold traversal cost summary.

ing objects from the ESM buffer pool into the object cache (29.46%), and other caching work done by EPVM (50.04%). The latter ("other EPVM costs") cost includes overheads such as inserting objects into hash tables, initializing object descriptors, initializing bitmaps, and allocating space for objects in the cache.

Because page caching is being used, 13,124 part objects are cached during the cold iteration. These objects occupied 321 pages on disk and, of the objects that are cached, only 1,465 were actually referenced during the iteration. The large number of objects that are cached explains why the caching cost is relatively high, even though the per object cost is low (434 cycles).

Pointer swizzling required a relatively small amount of CPU time (2.15%) during the cold iteration. Table 3.9 divides the swizzling cost into the cost for swizzling upon discovery (2.15%) and the cost for swizzling upon dereference (0.21%). Swizzling upon discovery attempted to swizzle 2,026 pointers during the cold traversal. Of these 2,026 attempts, 1,706 (84%) actually resulted in a pointer being swizzled. The failed attempts were caused by the fact that in 321 cases the object referred to by the candidate pointer was non-resident.

The cost of swizzling an individual pointer for swizzling upon discovery was 82 cycles. Table 3.9 gives a breakdown of this cost. When swizzling upon discovery is used, a hash table lookup must first be done to see if the referenced object is cached. The cost for this lookup and the comparison of the pointer with the OID contained in the hash table entry totaled 19 cycles. Once the referenced object is found, some additional checks are needed by our implementation to ensure that the candidate pointer is indeed swizzleable (basically to distinguish large and small objects). These checks cost an additional 11 cycles. Finally, call setup and execution of the function to swizzle the pointer accounted for 4 and 29 cycles, respectively, and the work necessary to set the bit in the object's bitmap marking the location of the swizzled pointer accounted for 16 cycles.

A failed swizzle attempt for swizzling upon discovery, due to the referenced object not being present in the cache, cost 8 cycles. This cost is for the hash table lookup to see if the object is resident. The 8 cycle figure is a best case estimate that assumes no hash table collisions. This is a reasonable assumption since our results showed that collisions were rare during the experiment. The total cost due to failed swizzle attempts was only .04% of the CPU time. Finally, swizzling upon dereference swizzled 321 pointers, with a cost of 44 cycles for each pointer swizzled. Note that swizzling upon dereference swizzled one pointer for the first object that was referenced on each page that was faulted in. Because page caching was used, the remaining pointers were swizzled upon discovery.

The cost for dereference checks and swizzle checks was relatively low in the cold iteration, although in absolute terms the cost is the same as during the hot iteration. The total cost of all checks was 1.19%. The low cost of

swizzling pointers and pointer checks correlates well with the nearly identical times shown for PC1M and PC1M-NO in Figure 3.2 of Section 3.4.

3.6. Conclusions

This chapter presented a detailed discussion of the implementation of pointer swizzling and object caching in EPVM 2.0. The relative performance of several versions of EPVM 2.0 was then analyzed using the OO1 benchmark. EPVM 2.0 was also compared to some alternative methods of supporting pointer swizzling, including the memory-mapped approach of ObjectStore.

We begin by summarizing the performance results for the systems based on EPVM 2.0. The page caching approach (PC1M) delivered better overall performance than object caching (OC1M, OC5M) when the small database was used. PC1M (page caching using a small buffer pool) had better overall performance than OC1M (object caching using a small buffer pool) in this case because PC1M avoided the need to reread pages from the server during normal transaction execution as was done by OC1M. Page caching also performs well because the cost of copying full pages is relatively small compared to the cost of copying individual objects when the small database is used.

When the large database was used, the performance results for object caching and page caching were more mixed, although object caching appears to have an advantage. For example, OC1M performed much better than PC1M when executing long transactions because PC1M performed a lot of unnecessary copying work and experienced paging of virtual memory, both of which lowered its performance. When shorter transactions were used, however, and no virtual memory paging occurred, PC1M had slightly faster performance than OC1M.

We now discuss the EPVM 2.0 swizzling results. The swizzling scheme used by EPVM 2.0 never noticeably hurt performance when the small database was used, and it improved performance by as much as 45% in some cases (see Table 3.3 column 4). When the large database was used, swizzling did not improve performance and resulted in a 4% performance decrease in a few cases due to the fact that it caused an increase in the amount of virtual memory paging activity. The increase in paging was due to the fact that pointer swizzling dirtied virtual memory pages, causing them to be written to the swap area by the virtual memory sub-system. Lastly, we note that the swizzling scheme used by EPVM1 improved performance by 16% in some cases when using a small database and had no effect when using the large database.

The detailed analysis of the performance of EPVM 2.0 during the hot traversal showed that the software checks needed to support swizzling were a relatively small portion (10%) of the total hot traversal cost. However, the

percentage of time spent dereferencing pointers to persistent data (41%) was significant for EPVM 2.0. It was interesting to note that although ObjectStore spends a much smaller portion of the total time dereferencing pointers (4.8%), the overall performance of EPVM 2.0 and ObjectStore only differed by 33% during the hot iteration. This is because the differences in pointer dereferencing costs between the two systems are outweighed by common costs necessary to perform the traversal itself. The detailed cold traversal results showed that swizzling costs are extremely low compared to other costs (such as data caching).

We now turn to a comparison of ObjectStore and EPVM 2.0. The OO1 cold iteration times for ObjectStore were slower than the cold times for the architectures based on ESM when using both a small and a large database. ObjectStore had the fastest warm iteration time when using the small database, but when the large database was used, ObjectStore had the worst warm performance. Given the results obtained using QuickStore (see Chapter 5), these results suggest that the I/O performance of ObjectStore is worse than that of ESM. The poor performance of ObjectStore is also likely due to the overhead for maintaining the memory-map that maps data into the process's address space. The hot iteration results (obtained using the small database) showed that the memory-mapped scheme used by ObjectStore can achieve five times faster performance than the software approach of EPVM 2.0 when operating on in-memory data. However, it was observed that the difference in performance was only 33% once a small amount of additional computation was added.

It was also shown that PC1M generally performed better than ObjectStore when updates were performed. The main reason for this appears to be that ObjectStore does whole-page logging in order to support crash recovery. EPVM1 and CESM performed better than ObjectStore and PC1M when the frequency of updates was low and when multiple transactions were used. When a large database was used, the memory-mapped approach of ObjectStore always had slower performance than EPVM1 and CESM. Although it is interesting to speculate about the differences in performance between ObjectStore and the other swizzling schemes, it is virtually impossible to learn the true reasons for the differences since ObjectStore is a proprietary system. Because of this, the next chapter introduces QuickStore, a memory-mapped OODBMS which, like the software-based swizzling schemes presented in this chapter, is built on top of ESM. QuickStore was built so that an accurate and detailed comparison of hardware and software swizzling techniques could be made (see Chapter 5).

CHAPTER 4

THE DESIGN OF QUICKSTORE

This chapter describes the design of QuickStore, a memory-mapped storage system for persistent C++ that was built using the EXODUS Storage Manager (ESM) [Carey89]. QuickStore uses standard virtual memory hardware to trigger the transfer of persistent data from secondary storage into main memory [Wilso90]. The advantage of this approach is that access to in-memory persistent objects is just as efficient as access to transient objects, i.e. application programs access objects by dereferencing normal virtual memory pointers, with no overhead for software residency checks in contrast to [Moss92, Schuh90, White92].

QuickStore is implemented as a C++ class library that can be linked with an application, requiring no special compiler support. Instead, QuickStore uses a modified version of gdb to obtain information describing the physical layout of persistent objects (see Section 4.4 for details). The memory-mapped architecture of QuickStore supports "persistence orthogonal to type", so that both transient and persistent objects can be manipulated using the same compiled code. Because QuickStore uses ESM to store persistent data on disk, it features a client-server architecture with full support for transactions (concurrency control and recovery), indices, and large objects. QuickStore places no additional limits on the size of a database, and the amount of data that can be accessed in the context of any single transaction is limited only by the size of virtual memory.

The remainder of the chapter is organized as follows. Sections 4.1 and 4.2 describe the memory-mapping scheme used by QuickStore to give application programs access to persistent data. The in-memory data structures used by QuickStore are described in Section 4.3. Section 4.4 discusses pointer swizzling, and Sections 4.5 and 4.6 cover buffer pool management and recovery, respectively. Finally, Section 4.7 compares the design of QuickStore with the design of other memory-mapped persistent stores.

4.1. Overview of the Memory-Mapped Architecture

As mentioned above, QuickStore uses ESM to store persistent objects on disk. ESM features a page-shipping [DeWitt90] architecture, in which objects are transferred from the server to the client a page-at-a-time. Once a page of objects has been read into the buffer pool of the ESM client, applications that use QuickStore access objects on

the page directly in the ESM client buffer pool, by dereferencing normal virtual memory pointers. We note that objects are always accessed in the context of a transaction in QuickStore.

This section describes the memory-mapping scheme used by QuickStore to give application programs access to persistent objects. To understand the approach, it is useful to view the virtual address space of the application process as being divided into a contiguous sequence of *frames* of equal length. In our case, these frames are 8 K-bytes in size, the same size as pages on disk. The ESM client buffer pool can also be viewed as a (much smaller) sequence of 8 K-byte frames. To coordinate access to persistent objects, QuickStore maintains a physical mapping from virtual memory frames to frames in the buffer pool. This physical mapping is **dynamic**, since paging in the buffer pool requires that the same frame of virtual memory be mapped to different frames in the buffer pool at different points in time. The mapping can also be viewed as a logical mapping from virtual memory frames to disk pages. When viewed this way, the mapping is **static** since the same virtual frame is always associated with the same disk page during the course of a transaction.

Figure 4.1 illustrates this mapping scheme in more detail. The buffer pool shown in Figure 4.1 contains 7 frames (labeled from 1 to 7). Virtual memory frames are denoted using upper-case letters, while disk pages are specified in lower-case. In the discussion that follows, we sometimes refer to the virtual memory frame beginning at address A as frame A.

Before a page is read from disk by QuickStore, the virtual memory frame corresponding to the page is selected and access protected. When the application first attempts to access an object on the page by dereferencing a pointer into its frame, a page-fault is signaled and a fault handling routine that is part of the QuickStore runtime system is invoked. This fault handling routine is responsible for reading the page from disk, updating various data structures, and enabling access permission on the virtual frame that caused the fault so that execution of the program can resume. For example, in Figure 4.1a page a has been read from disk into frame *4* of the buffer pool. Page a is "mapped" to virtual address A. Read access has been enabled on frame A, so that the application can read the objects contained on page a. We note that once the mapping from virtual address A to page a has been established, the application can access objects on page a by dereferencing pointers to frame A at any time. Thus, the mapping from A to a must remain valid until the end of the current transaction (or longer, if requested) in order to preserve the semantics of any pointers that the application may have to objects on page a. (This is not the case for the mapping of virtual frame A to buffer frame *4*, however, as we will see in a moment.)

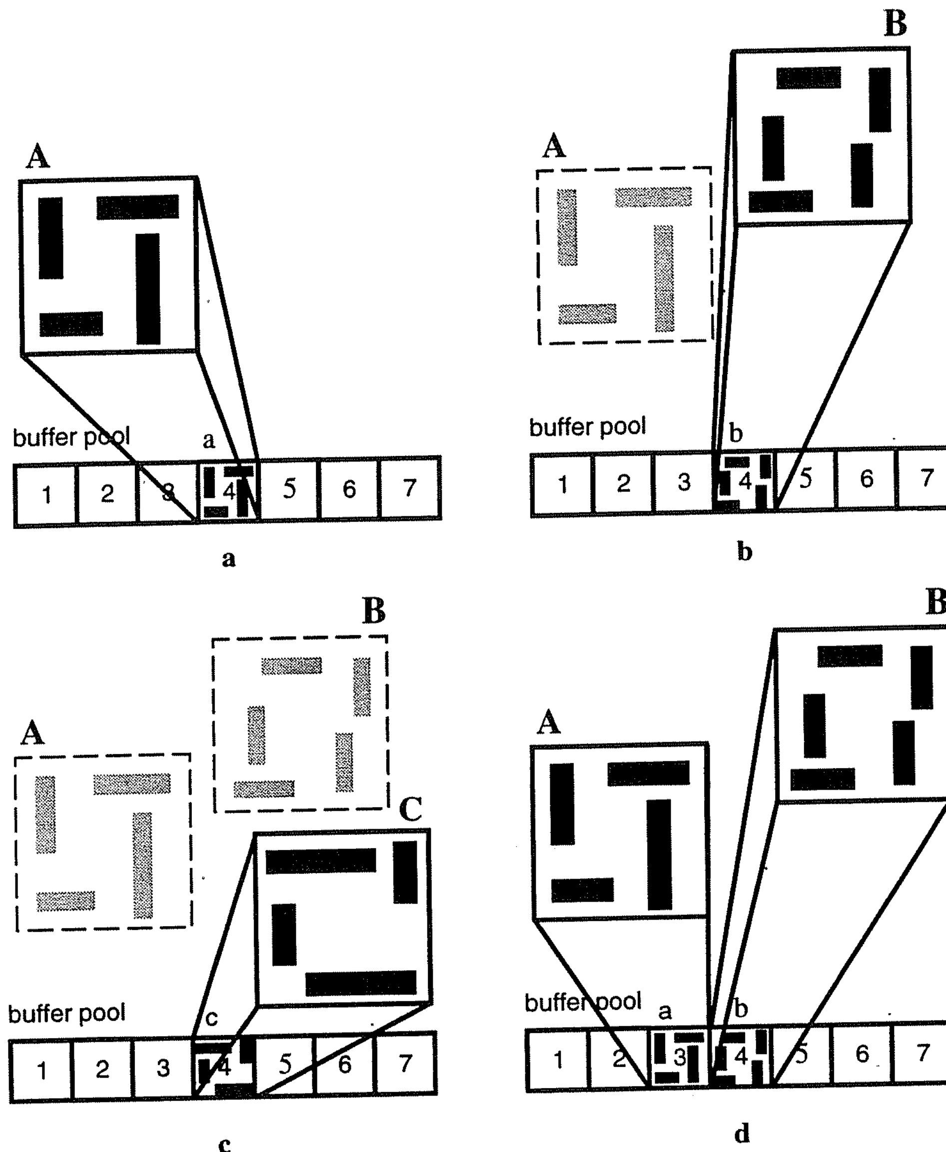

Figure 4.1. Mapping virtual frames into the buffer pool.

If the objects on page *a* contain pointers to objects on other non-resident pages, then virtual frames are assigned to these pages when page *a* is faulted into memory (if they haven't been already). The mechanism for assigning virtual frames to disk pages is covered in detail in Section 4.4, which discusses pointer swizzling. A frame for a non-resident page remains access protected until the program attempts to deference a pointer into the frame, which then causes a page fault that results in the page being read into the buffer pool. Figure 4.1 doesn't explicitly show any of the frames being referenced by pointers on page *a* since they are not important to the current discussion.

When the buffer pool becomes full, paging will occur and page *a* may be selected for replacement by the buffer manager. (See Section 4.5 which discusses buffer pool management for details.) This is what has happened in Figure 4.1b. Here, page *b* has been read from disk into frame *4* of the buffer pool, replacing page *a*. Page *b* has been mapped to virtual address *B* and read access on *B* has been enabled. Note that since we assume that the buffer pool is full in Figure 4.1b, additional virtual frames (not shown) will also have been mapped to the remaining 6 frames in the buffer pool other than frame *4*. If the application continues to access additional pages of objects in the database, then the situation shown in Figure 4.1c may result. In Figure 4.1c, page *c* has been read into memory and replaced page *b* in frame *4* of the buffer pool. Page *c* has been mapped to virtual frame *C* and read access on *C* has been enabled. This illustrates that, in general, any number of virtual frames may be associated with a particular frame of the buffer pool over the course of a transaction.

At this point, the reader may be wondering, what would happen in Figure 4.1b if the application attempted to dereference pointers into virtual frame *A* after page *a* has been replaced in the buffer pool by page *b*. Won't these pointers refer to data on page *b*? This problem is avoided by disabling read access on frame *A* when page *a* is not in memory. If the application again dereferences pointers into frame *A*, a page-fault will be signaled and the fault handling routine will be invoked. The fault handling routine will call ESM to reread page *a*, map virtual frame *A* to the frame in the buffer pool that now contains *a*, and enable read permission on frame *A* once again.

To illustrate this, Figure 4.1d shows what might result if page *a* were immediately referenced after being replaced in Figure 4.1b. In this case, *a* has been reread by ESM into frame *3* in the buffer pool and frame *3* has been mapped to virtual memory address *A*. This further illustrates the dynamic nature of the mapping from virtual memory frames to frames in the buffer pool as virtual frame *A* is mapped to buffer frame *4* in Figure 4.1a and then remapped to buffer frame *3* in Figure 4.1d. Note, however, that the mapping between virtual frames and disk pages is static—virtual frame *A* is always mapped to disk page *a*.

4.2. Implementation Details

QuickStore uses the UNIX *mmap* system call to implement the physical mapping from virtual memory frames to frames in the ESM client buffer pool and to control virtual frames' access protections. It was necessary to modify the ESM client software slightly in order to accommodate the use of *mmap* since *mmap* really just associates virtual memory addresses with offsets in a file, while ESM normally calls the UNIX function *malloc* to allocate space in memory for its client buffer pool. To make ESM and *mmap* work together, the buffer pool allocation code was

changed so that it would first open a file (and resize it if necessary) equal in size to the size of the client buffer pool. The buffer allocation code then calls *mmap* to associate a range of virtual memory with the entire file. The rest of the ESM client software uses this range of memory to access the buffer pool just as though the memory had been allocated using *malloc*.

The important thing to note is that the file serves as backing store for the buffer pool. Swap space and actual physical memory are never allocated for the virtual frames that are mapped into the file by *mmap*, so mapping a huge amount of virtual memory into the buffer pool doesn't affect the size of the process, although it may increase the size of page tables maintained by the operating system. One should also note that the contiguous range of addresses used by the ESM client to access the buffer pool is different from the 8 K-byte ranges of addresses that the application program uses to access pages in the buffer pool. The former is simply used to integrate an already existing storage manager (ESM) with the memory mapped approach and would not, in general, be required by a memory mapped implementation.

We should point out that using *mmap* in the particular way that we did, caused some minor performance problems in the implementation. Because the workstation used as the client machine in the benchmark experiments (a Sun ELC) had a virtually mapped CPU cache, accessing the same page of physical memory in the buffer pool via different virtual address ranges caused the CPU cache to be flushed whenever the process switched between the address ranges. This increased the number of *min faults*, which are virtual memory page faults that do not require I/O (in Unix terminology), experienced by the application. We note the effects of this phenomena when discussing the performance results in Section 5.6.

4.3. In-Memory Data Structures

QuickStore maintains an in-memory table that keeps track of the current mapping from virtual memory frames to disk pages. At a given point in time, the table contains an entry for every page that has been faulted into memory, plus entries for any additional pages that are referenced by pointers on these pages. We refer to a page for which there is a table entry as being "in the current mapping". In essence, any page containing persistent data that the application program can deference a pointer to must be in the current mapping. Entries in the table are called *page descriptors* and are 60 bytes long. Figure 4.2 shows the format of a page descriptor. We note that disk pages themselves come in two types: pages that contain sets of objects that are smaller than a disk page which are called *small object pages*, and pages that contain individual pages of multi-page objects, which are called *large object pages*.

Table entries for small object pages and large object pages differ in some respects, so they are discussed separately.

A page descriptor for a small object page contains the range of virtual addresses associated with the page, the physical address of the page on disk, and a pointer to the page when it is pinned in the buffer pool. The physical address of the page, in our implementation, is the OID of a special meta-object (24 bytes) located on each small object page. Page descriptors also contain other fields such as flags that indicate what types of access are currently allowed on the frame associated with the page (read, write, and none), whether an exclusive page lock has been obtained, and whether or not the page has previously been read into memory during the current transaction. This last flag is useful since it is not necessary to do any swizzling work for a page when it is reread during a transaction; the pointers on such pages are guaranteed to be valid. The page descriptor also contains a heap pointer that is used for recovery purposes.

The scheme used for large object pages is somewhat more complicated than the scheme for small object pages. The virtual memory frames associated with a multi-page object must be contiguous, so they are reserved all at once. To avoid maintaining individual table entries for every page of a multi-page object, multi-page objects that have not been accessed—but which are in the mapping—are represented by a single entry in the mapping table. The range of virtual addresses in this entry is the entire range of contiguous addresses associated with the object, and the physical address field contains the OID of the object. When the first page of a multi-page object is accessed by the application program, the table entry is split so that there is one entry in the table for the page that has been accessed, and an entry for each contiguous sub-sequence of unaccessed pages. Table entries for sub-sequences of unaccessed pages of multi-page objects are split in turn when one of the pages contained in the sub-sequence is accessed.

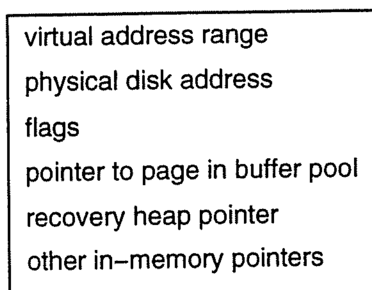

Figure 4.2. Format of a page descriptor.

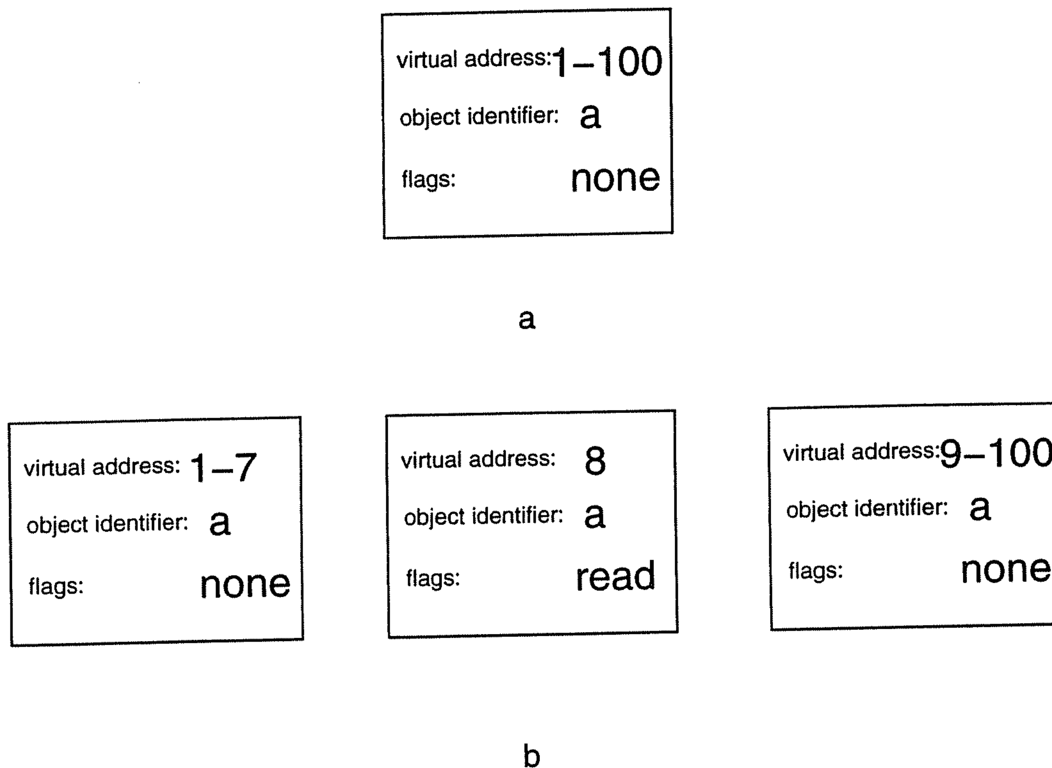

Figure 4.3. Splitting a large object descriptor.

Figure 4.3 illustrates the splitting process for a large object descriptor. Figure 4.3a shows the descriptor for a large object that has not yet been accessed. The object is 100 pages long and has been mapped to virtual memory frames 1 through 100. The OID of the object is denoted by the letter *a*. Figure 4.3b shows what happens when the eighth page of the object is accessed. In Figure 4.3b a page descriptor has been allocated for the recently accessed page. The virtual address range in the descriptor records the fact that the descriptor is now only associated with virtual memory frame *8* and that read access on frame *8* has been enabled. The descriptor also contains a pointer (not shown) to the copy of the page that is cached in the buffer pool. Figure 4.3b shows that two additional page descriptors are used to represent the remaining pages of object *a*. One descriptor represents the pages in object *a* that precede page *8*, while the other represents the pages that follow page *8*. We note that the offset (also not shown) of the first page of the object that is represented by a descriptor is stored in the page descriptor as well, so that the descriptor can be associated with the correct pages of the object.

The table organizes page descriptors according to the range of virtual memory addresses that they contain using a height balanced binary tree. One reason for using a binary tree is that it makes the splitting operation associ-

ated with large objects efficient. It is also helpful to keep the ranges of addresses currently allocated to persistent data ordered. For example, our current scheme for allocating virtual frames to disk pages uses a global counter (stored on disk) that is incremented by the frame size each time that a frame is allocated to a disk page. This counter needs to be persistent so that successive runs of a program don't reuse the same virtual memory addresses unnecessarily when allocating new objects. If the database becomes bigger than the size of virtual memory then this counter will wrap around and it may become necessary to scan the in-memory binary tree in order to find a virtual frame that is currently not in use.

Page descriptors are also hashed based on their physical address (OID) and inserted in a hash table. (For large objects only the entry containing the first page of the object is inserted in the hash table.) The hash table implements a reverse mapping from physical disk address to virtual memory address. The hash table is used by the fault handling routine as part of the pointer swizzling process (see the next section for details).

4.4. Pointer Swizzling in QuickStore

QuickStore stores pointers on disk as virtual memory addresses in exactly the same format that they have when they are in memory. Figure 4.4 shows the format of a pointer in QuickStore. The high order bits contained in a pointer can be viewed as identifying a virtual memory frame, while the low order bits identify an offset into the frame where the actual object referenced by the pointer is located. (Objects are not allowed to move within pages, so the offset identifies a unique object or portion of an object.) Since virtual memory pointers are only meaningful in the context of an individual process, the system maintains some additional meta-data that associates pointers on disk with the objects that they reference. We first describe how this meta-data is stored in QuickStore, and then briefly touch on some possible alternative implementation strategies.

QuickStore associates meta-data with individual disk pages. In the case of small object pages, each page contains a direct pointer (OID) to a *mapping object* containing the meta-data for the page. (Actually, the pointer is contained in the meta-object located on the page.) The term *mapping object* is used since the object records the

Figure 4.4. Format of a pointer in QuickStore.

mapping between virtual frames referenced by pointers on the page and disk pages that was in effect when the page was last memory resident. Mapping objects are essentially just arrays of <virtual address range, disk address> pairs.

Mapping information is stored separately instead of on the disk pages containing objects themselves because the space required to store the mapping information for a page can vary over time. For example, if the pointers on a page are updated, the number of frames referenced by pointers on the page may change, thus changing the number of entries in the mapping object. Multi-page objects are implemented similarly to small object pages, except that there is an array of meta-objects appended to the end of the large object containing one meta-object for each page of the large object. Finally, we note that each meta-object also contains a pointer (OID) to a bitmap object that records the locations of pointers on the page so that they can be swizzled.

QuickStore uses a modified version of *gdb* to get the type information for objects that is used to maintain the bitmaps associated with pages. The modified version of *gdb* outputs a description of the layout of a type that is stored in the schema for an application on disk. When an object is created the information in the schema is used to update the bitmap for the page containing the object.

Figure 4.5 illustrates the way that the structures described above are used to ensure that all of the pointers seen by an application program are valid swizzled pointers. In Figure 4.5a, disk page *a* is mapped to virtual frame *A*; however, page *a* has not been accessed by the application program, so read access on frame *A* has not been enabled. We now consider the actions that are taken by the QuickStore fault-handling routine when page *a* is first accessed. These are illustrated in Figure 4.5b.

In Figure 4.5b, the fault-handler has read both page *a* and the page containing the mapping object (as well as other mapping objects) associated with page *a* into main memory. The fault-handler then examines each entry in the mapping object, and uses the disk address contained in the entry to lookup the page descriptor associated with that disk address in the in-memory table. If no entry is found in the table, then one is created using the information contained in the mapping object entry. In the example in Figure 4.5b, the mapping object indicates that page *a* contains pointers that reference objects on three distinct disk pages. These include page *a* itself and two other pages, page *b* and page *c*. When the fault handler looks up pages *b* and *c*, it discovers that there is not a page descriptor entry for either page in the mapping table, so two entries are created.

When a table entry is created, the disk page (or pages, in the case of a large object) is assigned to its previous virtual frame (obtained from the mapping object entry) if the virtual frame is unused; otherwise a new frame is selected. If an entry for a disk page is found in the table, the system checks to see if the page is currently mapped to

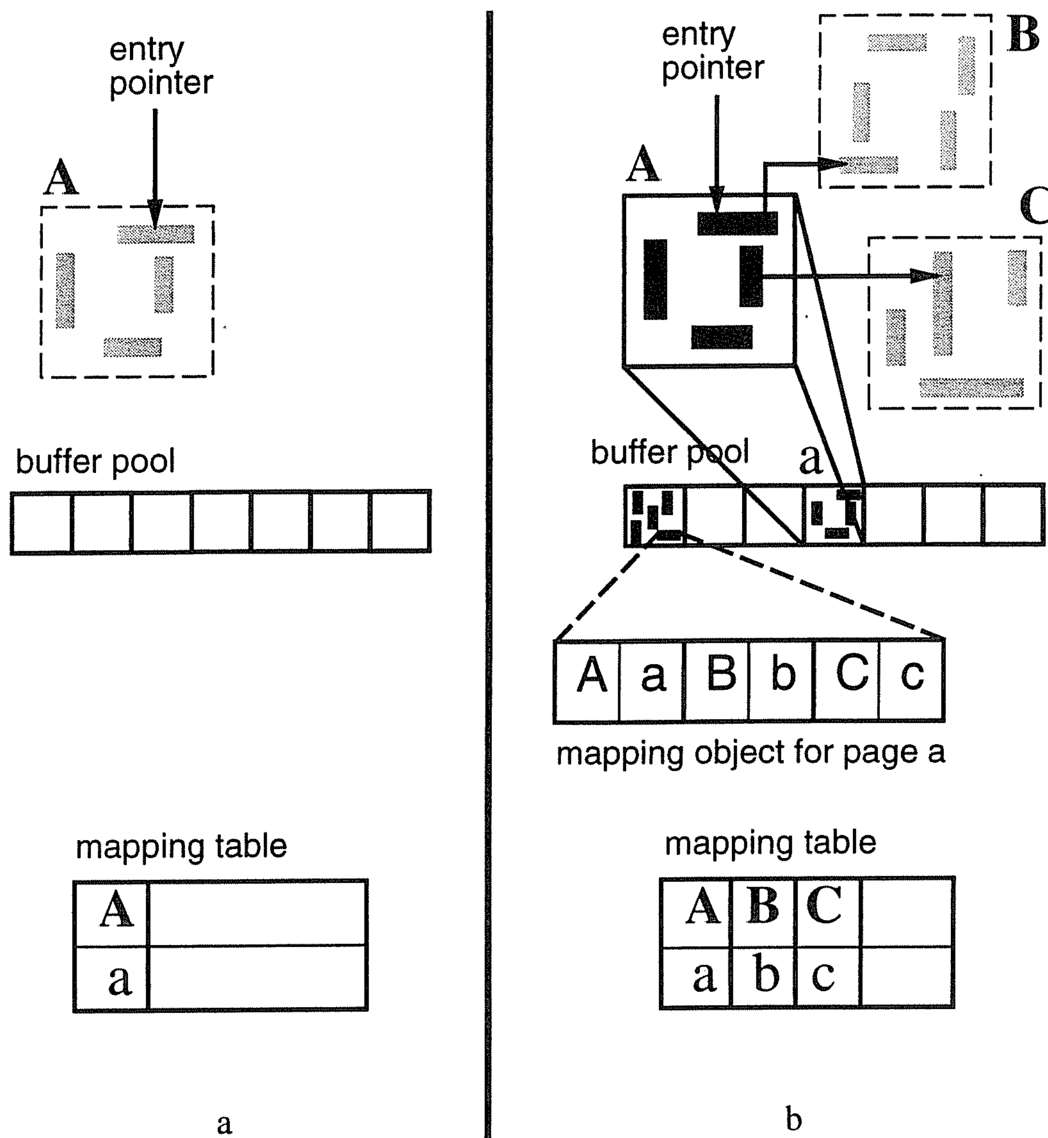

Figure 4.5. Pointer swizzling with no collisions.

the same virtual frame as the one contained in its mapping object entry. If each disk page can be mapped to the virtual frame contained in its corresponding entry, then the swizzling process terminates. In Figure 4.5b, page *a* is already mapped to the frame *A*. In addition, pages *b* and *c* can be assigned to frames *B* and *C*, since frames *B* and *C* are currently not being used. Since all of the pages referenced by pointers contained on page *a* are assigned to the same memory locations that they occupied the last time page *a* was in memory, it isn't necessary to update (i.e. swizzle) any of pointers on page *a*, and the swizzling process terminates.

If some disk pages had been mapped to new locations in Figure 4.5b, then additional swizzling work would have been required. For example, suppose that page *c* couldn't have been assigned to frame *C* because another page had already been assigned to that frame. In that case, the pointers on page *a* that reference objects on page *c* would have to have been updated so that they referenced the new frame associated with page *c*. When pointers need to be swizzled, the bitmap object associated with the page is read from disk and used to find and update any pointers on the page that need to be changed. Although only a subset of the pointers contained on a page may need to be changed, all of the pointers on the page must be examined since it is not known in advance which pointers actually need to be updated. Note that even though bitmap objects are fixed in size, they are stored separately from their corresponding data page since they hopefully will not have to be used in most cases.

We now comment on some possible alternative ways of storing the mapping information that is used to support pointer swizzling. Instead of storing information concerning the mapping of virtual memory frames to disk pages on a per page basis, one could also store it for groups of pages. For example, some object-oriented systems group pages into units called segments or clusters. One could even store the information at the level of the entire database or possibly a file. We chose not to follow this approach because it makes keeping the mapping information up to date considerably more complex. For example, suppose that a database is composed of a large number of clusters, each containing 10 pages, and that mapping tables are maintained for individual clusters. If the database is bigger than virtual memory, or is distributed (the same virtual frame may have been associated with different pages in the database in this case), then pages may need to be relocated when they are brought into memory. Page relocations are a problem if only a few pages of a cluster are accessed by an application, since to keep the mapping information for the cluster up to date one must either read the rest of the pages in the cluster and update them so that they are consistent with the new mapping, or keep multiple versions of the mapping information—one version for the pages whose mapping has changed, and one for the pages that are still using the old mapping. If many versions of the mapping information are stored then this scheme could be much worse than storing mapping information for individual pages.

4.5. Buffer Pool Management

Buffer managers in traditional database systems typically use an LRU algorithm or a clock style algorithm that approximates an LRU page replacement policy. We also felt that a clock algorithm was the best choice for use in QuickStore. However, implementing this type of scheme turned out to be more difficult in the context of a memory-mapped system, where objects in the buffer pool are accessed by dereferencing virtual memory pointers.

The reason for this is that there is less information available to the buffer manager indicating which pages have been accessed recently.

Recall that in a traditional implementation of clock, a bit is usually kept for each frame in the buffer pool to indicate whether or not the frame has been accessed since the clock hand last swept over it. This bit is set by the database system each time the page is accessed and is reset by the clock algorithm. There is no way to set such a flag, however, when dereferencing a pointer in QuickStore.

One solution to this problem is to have the clock algorithm access-protect the virtual frame corresponding to a buffer pool frame when the clock hand reaches it. If the frame is subsequently reaccessed, a page-fault will occur and the fault handling routine can re-enable access to the page. This scheme replaces the usual setting and unsetting of bits in a traditional clock algorithm with the enabling and disabling of access permissions on virtual memory frames. We experimented with this solution, but in our experience the extra overhead of manipulating the page protections and handling additional page-faults made this approach prohibitively expensive in terms of performance.

To avoid the problem described above, QuickStore uses a simplified clock algorithm. Under this scheme the clock hand begins its sweep from wherever it stopped during the previous invocation of the algorithm. As soon as the clock hand reaches a page for which access is not enabled, the algorithm selects that page for replacement. If the clock hand reaches the end of the buffer pool without finding a candidate for replacement, however, then the **entire** virtual address space of the process being used for persistent data is reprotected with a single call to *mmap* and the algorithm is restarted, i.e. the clock hand begins scanning starting from the first frame in the buffer pool. This scheme performed much better than the original scheme outlined above in our experiments, and compared favorably with the more traditional clock replacement algorithm used by E (see Section 5.6).

4.6. Recovery

Implementing recovery for updates poses some special problems in the context of a memory-mapped scheme as well. For example, since application programs are able to update objects by dereferencing virtual memory pointers, it is difficult to know what portions of objects have been modified and require logging. Furthermore, it is desirable to batch the effects of updates together and log them all at once, if possible, since some applications may update the same object many times during a transaction. Due to the considerations mentioned above, QuickStore uses a page differencing (or diffing) scheme to generate log records for objects that have been updated. Chapter 6 discusses the implementation of recovery in QuickStore in detail, but we highlight the important aspects here.

Virtual memory frames that are mapped to pages in the database that have not been updated do not have write access enabled, so the first attempt by the application program to update an object on such a page causes a page-fault. When the fault-handler detects that an access violation is due to a write attempt, it copies the original values contained in the objects on the page into an in-memory heap data structure. The fault-handler also obtains an exclusive lock on the page from ESM, if needed, and enables write access on the virtual frame that caused the fault before returning control to the application. The application program can then update the objects on the page directly in the buffer pool.

At transaction commit time, when paging in the buffer pool occurs, or when the recovery heap becomes full, the old values of objects contained in the heap and the corresponding updated values of objects in the buffer pool are compared (diffed) to determine if log records need to be generated. The diffing algorithm used by QuickStore is more sophisticated than the simple approach of generating a log record for each modified region in an object, sometimes combining individual modified regions and logging them together (see Chapter 6 for details).

Care must also be taken when processing updates to update the mapping tables associated with each modified page. Recall that the mapping object for a page keeps track of the set of pages that are referred to by pointers on the page. Updates to objects on a page can change the pages that are members of this set, making it necessary to update the mapping object as well. Updating the mapping object for a page requires that each pointer contained on the page be examined and the in-memory table consulted to determine which page in the database the pointer references. The bitmap for the page is used to locate the pointers that it contains and from these pointers a new set of referenced pages is constructed. This new set is then compared element by element with the old set to see if the set has changed. If it has, then the mapping object for the page is updated to reflect the new set of referenced pages.

4.7. Comparison With Other Systems

This section discusses previous work that has been done on memory-mapped stores and compares the design of QuickStore to other memory-mapped systems. [Lamb91] describes the design of ObjectStore, a commercially available OODBMS that is based on a memory-mapped architecture. The design of QuickStore was heavily influenced by the design of ObjectStore, so QuickStore and ObjectStore are similar in several respects. For example, both systems are client-server database systems that allow application programs to access pages of objects directly in the client buffer pool. QuickStore and ObjectStore also both store pointers on disk as virtual memory pointers, and maintain additional meta-data describing the pointers' semantics. Thus, the pointer swizzling schemes

used in the two systems are nearly identical. Finally, both QuickStore and ObjectStore are based on persistent C++, and both limit the amount of data that can be accessed during a transaction to the size of virtual memory.

The Texas [Singh92] and Cricket [Shek90] storage systems have also used virtual memory techniques to support object faulting and swizzling. Texas stores pointers on disk as 8-byte file offsets, and swizzles pointers to virtual addresses as described in [Wilso90] at fault time. This means that whenever a page is faulted into memory, Texas must swizzle all of the pointers on the page. Currently, all data is stored in a single file (implemented on a raw *Unix* disk partition) in Texas [Singh92]. Both QuickStore and Texas are implemented as C++ libraries that add persistence to C++ programs without the need for compiler support. Both systems also support the notion of "persistence orthogonal to type" (as does ObjectStore). This allows the same compiled code to manipulate both transient and persistent objects. Both systems also allow the database size to be bigger than the size of virtual memory.

Texas is currently a single user, single processor system; QuickStore, since it is built on top of client-server EXODUS, features a client-server architecture with full transaction support—including concurrency control, recovery, and support for distributed transactions. QuickStore is also different in that it manipulates objects directly in the ESM client buffer pool, while Texas copies pages of objects into a separate heap area allocated in virtual memory. This limits the amount of data that can currently be accessed during a single transaction by Texas to the size of the disk swap area backing the application process. QuickStore also manages paging in the ESM client buffer pool explicitly, while Texas simply allows pages to be swapped to disk by the virtual memory subsystem when the process size exceeds the size of physical memory. Cricket, on the other hand, uses the Mach external pager facility to map persistent data into an application's address space (see [Shek90] for details).

A very recent related system is Dali [Jagad94] which is designed to be a main memory storage manager. Dali uses memory mapping techniques, but since it is specifically designed to handle main memory databases, it differs substantially from QuickStore. For example, Dali itself performs no pointer swizzling. In Dali, the database is partitioned into units called *database files*, each of which is mapped contiguously into a process's address space. Database pointers in Dali contain a database file identifier and an offset. Dereferencing a *database pointer* usually involves indexing into a fixed size array of open *database files* and adding the starting address of the *database file* (contained in the array) to the offset contained in the pointer. If a large number of database files are being used (which is not expected to happen often) then a search in an in-memory tree structure is required to determine the starting address of the database file.

CHAPTER 5

THE PERFORMANCE OF QUICKSTORE

This chapter presents an in-depth analysis of the differences in performance between the memory-mapped pointer swizzling approach of QuickStore and the software-based pointer swizzling scheme used in E. QuickStore and E exemplify two of the basic approaches (hardware vs. software) that have been used to implement persistence in OODBMSs. Moreover, both QuickStore and E are based on the same underlying storage manager (ESM) and compiler. This allows us to make a truly accurate comparison of the hardware and software swizzling schemes, something which has not been done previously.

The rest of Chapter 5 is organized as follows. Section 5.1 and 5.2 describe the OO7 benchmark database and operations. Sections 5.3 and 5.4 describe the hardware and software used in the experiments. Section 5.5 offers further details on the database systems (QuickStore and E) studied. Section 5.6 presents the performance results. Finally, Section 5.7 presents the conclusions reached in this chapter.

5.1. The OO7 Benchmark Database

This section describes the structure of the OO7 benchmark database. The OO7 database [Carey93] is intended to be suggestive of many different CAD/CAM/CASE applications. There are two sizes of the OO7 database: small and medium. Table 5.1 summarizes the parameters of the database.

Parameter	Small	Medium
NumAtomicPerComp	20	200
NumConnPerAtomic	3	3
DocumentSize (bytes)	2000	20000
Manual Size (bytes)	100K	1M
NumCompPerModule	500	500
NumAssmPerAssm	3	3
NumAssmLevels	7	7
NumCompPerAssm	3	3
NumModules	1	1

Table 5.1. OO7 Benchmark database parameters.

A key component of the database is a set of *composite parts*. Each composite part is intended to suggest a design primitive such as a register cell in a VLSI CAD application. The number of composite parts per module is controlled by the parameter *NumCompPerModule* which was set to 500. Each composite part has a number of attributes, including the integer attributes **id** and **buildDate**. Associated with each composite part is a *document* object that models a small amount of documentation associated with the composite part. Each document has an integer attribute **id**, a small character attribute **title** and a character string attribute **text**. The length of the string attribute is controlled by the parameter *DocumentSize*.

Each composite part also has an associated graph of *atomic parts*. Intuitively, the atomic parts within a composite part are the units out of which the composite part is constructed. One atomic part in each composite part's graph is designated as the "root part". In the small database, each composite part's graph contains 20 atomic parts, while in the medium database, each composite part's graph contains 200 atomic parts.

Each atomic part has the integer attributes **id**, **buildDate**, **x**, **y**, and **docId**. The **buildDate** values in atomic parts are randomly chosen in the range *MinAtomicDate* to *MaxAtomicDate*, which is 1000 to 1999. Each atomic part is connected via a bi-directional association to three other atomic parts according to the parameter *NumConnPerAtomic*. The connections between atomic parts are implemented by interposing an information-bearing connection object between each pair of connected atomic parts. A connection object contains the integer field **length** and the short character array **type**. Figure 5.1 depicts a composite part, its associated document object, and its associated graph of atomic parts.

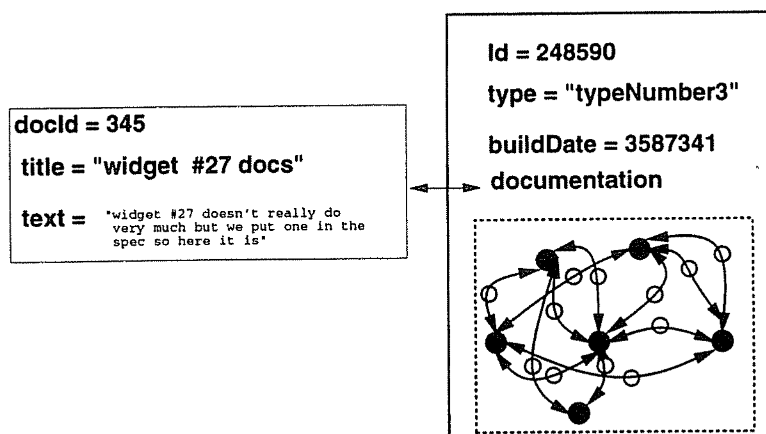

Figure 5.1. A composite part and its associated document object.

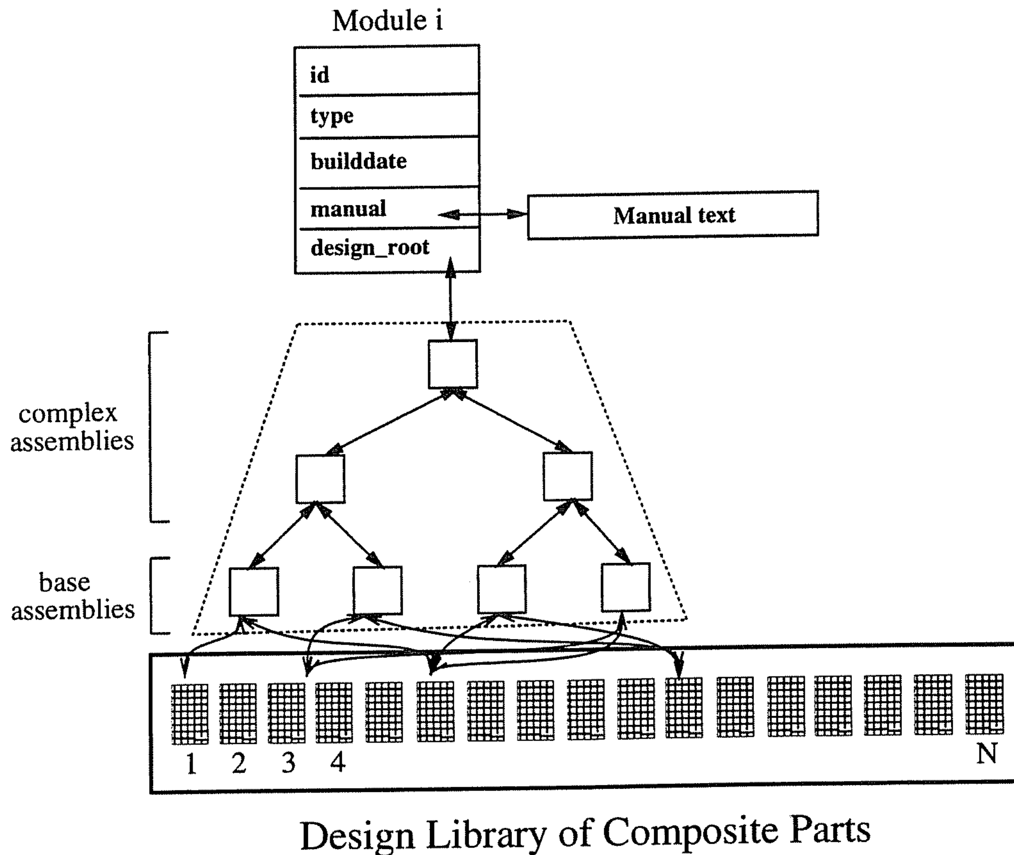

Figure 5.2. Structure of a module.

Additional structure is imposed on the set of composite parts by a structure called the "assembly hierarchy". Each assembly is either made up of composite parts (in which case it is a *base assembly*) or it is made up of other assembly objects (in which case it is a *complex assembly*). The first level of the assembly hierarchy consists of *base assembly* objects. Base assembly objects have the integer attributes **id** and **buildDate**. Each base assembly has a bi-directional association with three composite parts which are chosen at random from the set of all composite parts. Higher levels in the assembly hierarchy are made up of *complex assemblies*. Each complex assembly has a bi-directional association with three subassemblies, which can either be base assemblies (if the complex assembly is at level two in the assembly hierarchy) or other complex assemblies (if the complex assembly is higher in the hierarchy). There are seven levels in the assembly hierarchy.

Each assembly hierarchy is called a *module*. Modules are intended to model the largest subunits of the database application. Modules have several scalar attributes. Each module also has an associated *Manual* object, which is a larger version of a document. Manuals are included for use in testing the handling of very large (but simple)

objects. Figure 5.2 roughly depicts the full structure of the single user OO7 Benchmark Database. The picture is somewhat misleading in terms of both shape and scale; the actual assembly fanout used is 3, and there are only $(3^7-1)/2=1093$ assemblies in the small and medium databases, compared to 10,000 atomic parts in the small database and 100,000 atomic parts in the medium database.

5.2. The OO7 Benchmark Operations

This section describes the OO7 benchmark operations used in the study. Some of the OO7 operations were omitted because they didn't highlight any additional differences among the systems being compared. The operations, which are referred to by type and number, consist of a set of 10 tests termed traversals, and another set of 8 query tests. The queries were hand coded in C++ since neither QuickStore or E provide a declarative query language. We comment on the implementation used to execute the queries, although this is not part of the OO7 specification.

5.2.1. Traversals

The T1 traversal performs a depth-first traversal of the assembly hierarchy. As each base assembly is visited, each of its composite parts is visited and a depth-first search on the graph of atomic parts is performed. The traversal returns a count of the number of atomic parts visited, but otherwise no additional work is done during the traversal. The T6 traversal is similar to T1, except that instead of visiting the entire graph of atomic parts of each composite part, T6 just visits the root atomic part and returns.

T2 and T3 are also similar to T1, but they include updates. Each T2 traversal increments the (x,y) attributes contained in atomic parts as follows[1]:

T2A Update the root atomic part of each composite part.

T2B Update all atomic parts of each composite part.

T2C Update all atomic parts of each composite part four times.

The T3 traversals are similar to T2 except that the **buildDate** field of atomic parts in incremented. This field is indexed, so T3 highlights the cost of updates of indexed fields. We also used three traversals that are not based on T1. T7 picks a random atomic part and traverses up to the root of the design hierarchy. T8 scans the manual object

[1][Carey93] specifies that the (x, y) attributes should be swapped. We increment them instead so that multiple updates of the same object change the object's value. This guarantees that the diffing scheme used for recovery by QuickStore will always generate a log record.

associated with the module and counts the occurrences of a specified character, and T9 compares the first and last characters of the manual to see if they are the same.

5.2.2. Queries

Query Q1 randomly retrieves 10 atomic parts. This is done by using an index based on part id. Q2 selects the most recent 1% of atomic parts based on buildDate, while Q3 looks up the most recent 10%. Both queries do an index scan to find the parts. Query Q4 looks up 10 document objects at random using the index on their title field. It then visits all base assemblies that use the composite part corresponding to each document. This is done by traversing pointers from documents to composite parts and then processing a collection of pointers to base assemblies that is maintained as part of each composite part object. Q5 is referred to as a single level make operation. Q5 finds all base assemblies that use a composite part with a build date later than the build date of the base assembly. Q5 is implemented as a nested loops pointer join between base assemblies and composite parts, i.e it iterates over the collection of base assemblies maintained for the module and as each base assembly object is visited, the references to composite parts that it contains are traversed.

5.3. Hardware Used

As a test vehicle we used a pair of Sun workstations on an isolated Ethernet. A Sun IPX workstation configured with 48 megabytes of memory, two 424 megabyte disk drives (model Sun0424) and one 1.3 gigabyte disk drive (model Sun1.3G) was used as the server. One of the Sun 0424s was used to hold system software and swap space. The Sun 1.3G drive was used by ESM to hold the database, and the second Sun 0424 drive was used to hold the ESM transaction log. The data and recovery disks were configured as raw disks. For the client we used a Sun Sparc ELC workstation (about 20MIPS) configured with 24 megabytes of memory and one 207 megabyte disk drive (model Sun0207). This disk drive was used to hold system software and as a swap device.

5.4. Software Used

The systems examined in the study use ESM V3.0 to provide disk storage for persistent objects. The important features of ESM were discussed in Chapter 3, but we briefly summarize them here. ESM provides files of untyped objects of arbitrary size and B-tree indices. ESM uses a page-server architecture where client processes request pages that they need from the server via TCP/IP. If the server cannot satisfy the request from its buffer pool, a disk I/O is initiated by invoking a disk process to perform the actual I/O operation. After the disk process has read in the

page, the server process returns it to the requesting client process and keeps a copy in its own buffer pool. ESM also provides concurrency control and recovery services. Locking is provided at the page and file levels with a special non-2PL protocol being used for index pages. Recovery is based on logging the changed portions of objects.

ESM used a disk page size of 8 K-bytes (which is also the unit of transfer between a client and the server). The client and server buffer pools were set to 1,536 pages (12 Mb) and 4,608 pages (36 Mb) respectively. Release 4.1.3 of SunOS was run on both of the workstations used in the experiments. QuickStore was compiled using the GNU g++ compiler V2.3.1. The E compiler is a modified version of the GNU compiler.

5.5. Database Systems Tested

5.5.1. E

This section briefly describes the implementation of the E language that was used in the study. The version of E used in this chapter is based on EPVM 3.0, and is different from the versions of E studied in Chapter 3. EPVM 3.0 was selected over EPVM 2.0 because it uses an in-place swizzling scheme, like QuickStore, and because we wanted to study the performance of the systems using large data sets that were bigger than the swap space of the client machine. (Recall that EPVM 2.0 limits the amount of data that can be accessed during a transaction to the size of the swap area.)

The pointer swizzling scheme used in EPVM 3.0 is the same as the scheme used in EPVM 1.0, except that swizzled pointers point directly to objects in the buffer pool. In other words, EPVM 3.0 only swizzles pointers that are local variables in C++ functions. Pointers within persistent objects are not swizzled because this makes page replacement in the buffer pool difficult [White92]. The only truly significant difference between EPVM 3.0 and EPVM 1.0 is the way in which EPVM 3.0 implements recovery.

The recovery scheme used in EPVM 3.0 copies the original values of objects into a side buffer in main memory, allowing updates of objects in place in the buffer pool. Original values of objects and updated values are used to generate log records at transaction commit (or sooner if the side buffer or the buffer pool become full). However, no diffing is performed as in QuickStore. EPVM 3.0 employs a scheme that breaks medium to large objects into 1K chunks for logging purposes, while objects that are smaller than 1K are logged in their entirety.

5.5.2. QuickStore

A detailed description of QuickStore was given in Chapter 4. Although QuickStore and E offer nearly the same functionality, it is important to point out one fundamental way in which the two systems differ. This has to do with the degree to which the two systems support the notion of object identity. Chapter 4 described the scheme used by QuickStore to implement a mapping between virtual memory frames and disk pages. This mapping is maintained for pages when they are in memory as well as when they reside on disk. Because of this mapping, pointers to persistent objects can be viewed as a <virtual frame, offset> pair, where the high order bits of the pointer identify the virtual memory frame referenced by the pointer and the low order bits specify an offset into the frame. Virtual memory frames are mapped to disk pages, so pointers really just specify offsets or locations on pages.

To see why this scheme doesn't support object identity, consider what happens when an object, for which there are outstanding references, is deleted. The page that contained the object can be faulted into memory by subsequent program runs and mapped to some virtual memory frame. If the program then dereferences dangling pointers to the deleted object, no error will be explicitly flagged. If a new object occupies (or overlaps) the space on the page previously used by the deleted object then the dangling pointers will reference this object.

QuickStore doesn't fully support object identity or "checked references" to objects because the overhead would be prohibitive. For example, the meta-data that would be required to associate every unique pointer on a page with its corresponding OID would likely be an order of magnitude greater than the current scheme used by QuickStore. Furthermore, we are aware of no commercial or research system (including ObjectStore) that supports these types of references for normal pointers in the context of a memory-mapped scheme. E, on the other hand, supports object identity fully, including "checked references". E implements this by storing pointers as full 16 byte OIDs within objects. This is a reasonable approach, but it does incur certain costs. For example, since objects are larger in E than in QuickStore, the database as a whole is larger, and E generally performs more I/O as a result. Also, dereferencing big pointers is more expensive even in terms of CPU requirements than dereferencing regular virtual memory pointers.

Because of these differences, we included a third system in the performance study. This system is identical to QuickStore, except that the size of each object has been padded so that it is the same as the corresponding object in E. Comparing the performance of this system to the performance of E in the experiments where faults take place gives insight into the overhead of faulting for the memory-mapped approach, while comparing it with QuickStore indicates the advantage gained by QuickStore due to its smaller object size. In addition, one can think of this system

as approximating the performance of a hybrid memory-mapped scheme that allows large pointers to be embedded within objects, thus supporting both checked and unchecked references.

5.6. Performance Results

5.6.1. Database Sizes

The size of the OO7 database is important in understanding the performance results. Table 5.2 shows the database sizes for E, QuickStore (QS), and QuickStore with big objects (QS-B)[2]. The QS database is roughly 60% as big as the E database for both the small and medium cases. This is because of the different schemes used by the systems to store pointers. The QS-B database is slightly bigger than the E database due to the overhead for storing bitmaps that indicate the locations of pointers on pages and the mapping tables. Mapping objects accounted for approximately 3% of the database size for QS and QS-B, while bitmaps generally accounted for an additional 3%.

	Small	Medium
QS	6.6	54.2
E	10.5	94.1
QS-B	11.5	98.5

Table 5.2. Database Sizes (in megabytes)

5.6.2. Small Cold Results

This section presents the the cold results for the small database experiments. The cold results were obtained by running the OO7 benchmark operations when no data was cached in memory at either the client or server machines. The times presented represent the average of 10 runs of the benchmark operations, except where noted otherwise. The times were computed by calling the Unix function *getimeofday* which had a granularity of several microseconds on the client machine.

Read-Only Results

Figure 5.3 and Table 5.3 list the cold response times and the number of client I/O requests, respectively, for the read-only traversals. As Figure 5.3 shows, QS is 37% faster than E during T1[3]. This difference in performance is

[2]Objects in QS-B are padded to the same size as the corresponding objects in the E implementation.

[3]T1: DFS of assembly hierarchy visiting all atomic parts.

largely due to the smaller database size for QS which causes it to read 53% fewer pages from disk than E (Table 5.3). Almost all of the I/O activity during T1 resulted from reading clusters[4] of composite parts. Each composite part cluster occupied a little less than one page for QS, while two pages were required for E. This accounts for the roughly 2 to 1 ratio in the number of disk reads between the two systems. Comparing E with QS-B, we see that QS-B is 15% slower than E during T1. QS-B always issues slightly more I/O requests than E since QS-B must read mapping objects to support the memory-mapped scheme.

The performance of QS is only 4% better than E during T6[5]. Again, this difference is largely due to the number of disk reads that each system performs. Table 5.3 shows that the amount of I/O for QS is almost the same

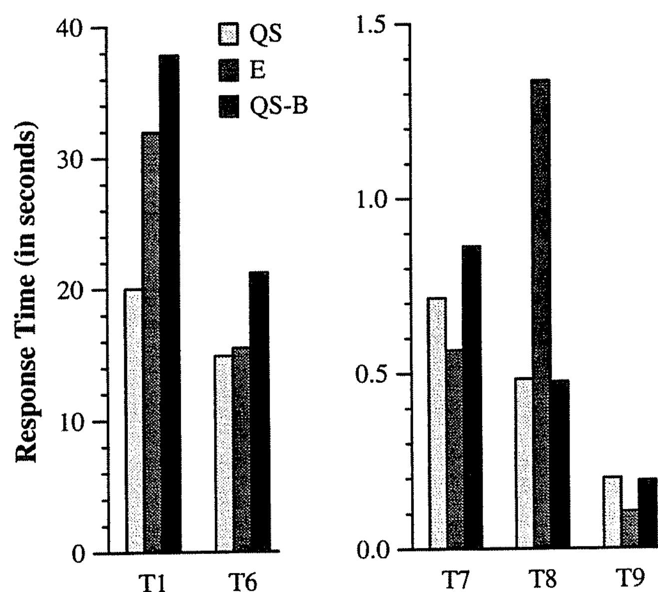

Figure 5.3. OO7 traversal cold times, small database.

	T1	T6	T7	T8	T9
QS	474	467	26	19	9
E	1018	600	25	18	7
QS-B	1047	639	31	19	9

Table 5.3. Client I/O requests, traversals, small database.

[4]The atomic part objects and connection objects associated with a composite part were clustered together on disk together with the composite part object itself in our implementation of OO7.

[5]T6: DFS of assembly hierarchy visiting only the root atomic part.

during T1 and T6, while the number of disk reads for E decreases by 41% during T6. This is due to the size difference of composite part clusters between the two systems. E does noticeably fewer I/O operations during T6 because, unlike QS, it generally doesn't read entire clusters. The performance of QS-B is 27% slower than E during T6. As the detailed faulting times shown below will illustrate, this difference in performance is close to the actual percentage difference in individual page fault costs for the systems, as the CPU cost of performing the traversal itself has less of an overall impact on performance during T6 than during T1.

E has the best performance during T7[6]. QS is 26% slower than E because of increased faulting costs relative to T1 and T6. One reason faults are more expensive for QS during T7 is that a large fraction of faults (86%) are due to reading pages of *base assembly* objects. These pages have larger mapping tables because pointers from *base assembly* objects to *composite part* objects are uniformly distributed among all *composite part* objects in the database. This increases the average I/O cost for reading the mapping tables and the number of table entries (139 on average for T7 vs 20 on average for T1) that must be examined per fault. QS-B is 34% slower overall during T7 relative to E.

Turning to T8[7], Figure 5.3 shows that E is roughly 3 times slower than QS. This is because the E interpreter is invoked once for each character of the manual that is examined by T8, while QS simply has to dereference a virtual memory pointer to access the manual. By contrast, Figure 5.3 shows that E is nearly twice as fast as QS on T9[8]. This difference is due almost entirely to faulting costs since very little work is done on the objects faulted in during T9. It is not surprising that faulting costs for QS are relatively high in this case. T9, like T7, touches only a few pages in the database and the pages that are accessed are not clustered. QS and QS-B have similar performance during T8 and T9 since character data is the same size for both systems.

The cold response time and client I/O requests for the queries are shown in Figure 5.4 and Table 5.4, respectively. E is 24% faster than QS and and 26% faster than QS-B during Q1[9]. The memory mapped scheme is slower in this case because the relatively small number of atomic part objects that are accessed during Q1 are accessed randomly. This causes one fault to be performed by all of the systems for each atomic part object that is accessed. In addition, Table 5.4 shows that QS performs several additional I/O operations (16%) to read mapping objects. The

[6]T7: Traverse up the assembly hierarchy starting from a randomly selected atomic part.

[7]T8: Scan the manual object counting occurrences of a specified character.

[8]T9: Compare first and last characters of the manual to see if they are equal.

[9]Q1: randomly retrieve 10 atomic parts.

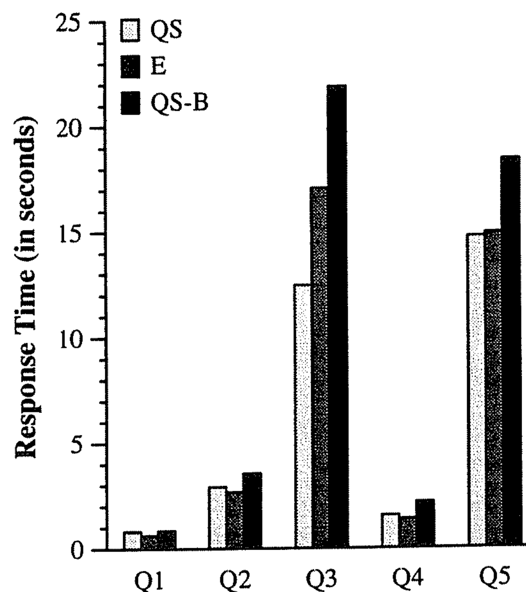

Figure 5.4. OO7 query cold times, small database.

	Q1	Q2	Q3	Q4	Q5
QS	31	109	413	62	467
E	26	104	641	59	558
QS-B	33	121	663	74	583

Table 5.4. Client I/O requests, queries, small database.

number of additional I/Os required to read mapping objects is relatively high during Q1 because mapping objects are clustered according to the disk pages to which they correspond, so when accesses to pages in the database are unclustered, more pages containing mapping objects tend to be read.

E also has the best performance during Q2[10]. As Table 5.4 shows, QS and QS-B again do more I/O than E during Q2 which causes them to have worse performance. The reasons for this are similar to those given for Q1. Since only a small fraction of atomic part objects (1%) are accessed during Q2, the accesses are unclustered to the point that roughly one fault is required per object accessed. When a larger percentage of objects (10%) are accessed as in Q3[11], QS has the best performance. Figure 5.4 shows that QS is 27% faster than E during Q3. QS faults in fewer pages than E in this case because object accesses are more highly clustered. E is 22% faster than QS-B during Q3 again illustrating the effect on performance of differences in faulting costs.

[10]Q2: retrieve most recent 1% of atomic parts.

[11]Q3: retrieve most recent 10% of atomic parts.

QS is 11% slower than E during Q4[12]. This is partly due to unclustered accesses to document and composite part objects which require roughly one fault per object accessed in both systems. A second factor is that a large fraction of the objects accessed by Q4 are base assembly objects, producing higher faulting costs due to the relatively large amount of mapping information associated with them. QS and E have basically the same overall performance during Q5[13] (Figure 5.4). QS performs 16% fewer I/O operations than E does, but the higher per fault cost for QS prevents it from performing faster than E. Finally, we note that E is 36% and 19% faster than QS-B during Q4 and Q5, respectively.

The results presented in Figures 5.3 and 5.4 for QS and E show that the memory-mapped scheme of QS offers better performance when objects are accessed in a clustered fashion. When objects are accessed in an unclustered fashion, however, E has better performance. In this case, the smaller object size of QS doesn't provide a noticeable savings in I/O, and QS actually performs slightly more total I/O operations than E because of the need to read mapping objects. The results presented in Figures 5.3 and 5.4 also demonstrated that the per page faulting cost for the memory mapped approach is noticeably higher than for the software approach. For example, except for T8 where other differences in CPU cost determined relative performance, QS-B always has slower performance than E in Figures 5.3 and 5.4.

Detailed Faulting Results

We next examine the differences in faulting costs between the systems in more detail. Table 5.5 shows the average cost per fault in milliseconds for each of the systems during T1 and T6. These times were calculated by subtracting the time required to execute a hot traversal from the time required for a cold traversal, and then dividing the result by the number of page faults to get the average per fault cost. To make sure that the results obtained were accurate, the numbers used to perform the calculation represented the average of 100 runs of each traversal experiment.

According to Table 5.5, the faulting cost for QS-B is slightly higher than for QS. We speculated that this was due to a larger number of pages that were being referenced on average by outbound pointers for QS-B, since there are more pages in the QS-B database. (An outbound pointer is a pointer that refers to an object on a page other than

[12] Q4: lookup 10 document objects and find the base assemblies that use their associated composite part.

[13] Q5: pointer join between base assemblies and composite parts.

the page containing the pointer.) This would increase the size of the mapping tables for QS-B. However, this turned out not to be the case since the average number of outbound pointers per page in the small database was 16 for QS and only 12 for QS-B. The comparison between QS and E in Table 5.5 is more interesting. It shows that individual page faults are roughly 20% more expensive for QS during T1 and T6. The corresponding figure for QS-B and E averages 26%, which correlates closely with the difference in response time between QS-B and E during T6.

To better understand the additional faulting overhead of the memory mapped scheme, Table 5.6 shows a detailed breakdown of the average faulting time for QS. As a check we present detailed numbers for both T1 and T6. One would except most of the costs for T1 and T6 to be similar since they fault in many of the same pages. The *min fault* entry in Table 5.6 is present due to the way our implementation interacts with the virtually mapped CPU cache of the client machine (see Chapter 4). This effect increased the average fault time by 6% and 5%, respectively for T1 and T6. The entry labeled *page fault* in Table 5.6 is the time that was required to detect the illegal page access and invoke the fault handler. Page faults comprised 3% of average faulting time for T1 and 2% for T6. We note that it was not possible to measure the times given for the *min fault* and *page fault* entries directly by running the benchmark. These times were obtained instead by measuring a test application that performed the operations several thousand times in a tight loop.

system	time(ms.)	
	T1	T6
QS	29.4	33.1
E	23.7	26.5
QS-B	31.6	34.5

Table 5.5. Average Faulting Cost.

description	time(ms.)	
	T1	T6
min faults	1.8	1.6
page fault	.8	.7
misc. cpu overhead	.5	.2
data I/O	24.8	28.5
map I/O	1.1	1.1
swizzling	.4	.4
mmap	.8	.8
total	30.2	33.3

Table 5.6. Detailed QS Faulting Times.

The remaining table entries break down the time spent in the fault handling routine. The entry for *misc. cpu overhead* includes time for looking up the address that caused the fault in the in-memory table, various residency and status checks to determine the appropriate action to take in handling the fault, and other miscellaneous work. *Data I/O* is the time needed to read the page of objects from disk and update the buffer manager's data structures. This accounted for largest fraction of faulting time, 82% for T1 and 85% for T6. The portion of time spent reading

mapping tables (*map I/O*) was 3.5% for T1 and 3.2% for T6. The *swizzling* entry gives the time needed to process the mapping table entries. Swizzling costs were quite low, accounting for 1% to 2% of the faulting cost on average. Since all of the pages read were mapped to the locations in memory that they occupied previously, the swizzling time doesn't include any overhead for updating pointers on pages that are inconsistent with the current mapping. The final entry, labeled *mmap*, gives the average time taken by the mmap system call to change the access protections. This accounted for a modest 3% of the faulting time. Finally, we note that the sums of the detailed times given in Table 5.6 correlate closely with the total per fault times given in Table 5.5.

Update Results

We next consider traversals T2 and T3 which include updates. Figure 5.5 shows the total response time for these traversals run as a single transaction. The read requests for T2 were nearly identical to T1 (Table 5.3), while the T3 traversals performed a few additional I/Os to read index pages. During T2A, which updates the root atomic part of each composite part, QS is 4% faster than E (Figure 5.5). This may seem surprising given that QS was 37% faster than E during T1 which does the same traversal as T2A, but without updates. The difference in performance between QS and E diminishes during T2A because the page-at-a-time scheme for handling updates of QS is more expensive than the object-at-a-time approach of E when sparse updates are done.

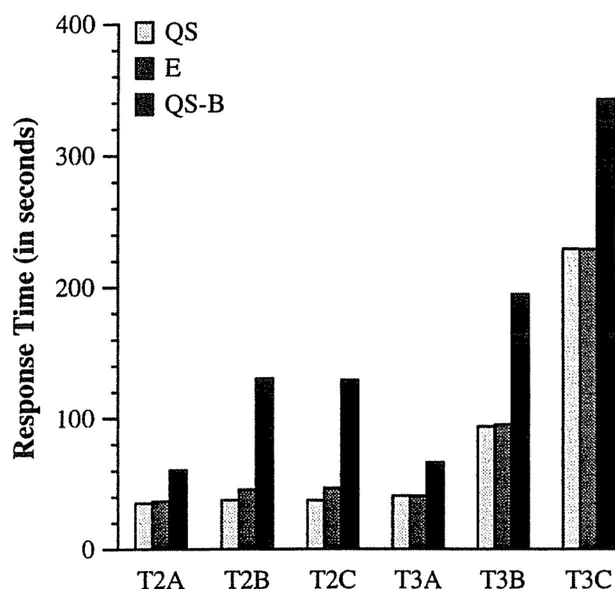

Figure 5.5. T2 and T3 Response Times.

Part of the increase in response time for QS is due to the fact that the number of page access violations increases from 454 during T1 to 878 during T2A, nearly doubling. The additional access violations occur when the first attempt is made to update an object on a page during the transaction. When this happens, a fault-handling routine is invoked to handle the access violation. As mentioned in Section 4.6, this routine performs several functions. First, it copies the objects contained on the page into the recovery buffer so that the original values contained in the objects may be used to generate logging records for updates (by diffing) at a later time. Next, it calls ESM, if necessary, to obtain an update (exclusive) lock on the page, and finally, it changes the virtual memory protections on the page so that the instruction that caused the exception can be restarted.

Our measurements showed that during T2A, a total of 5.198 seconds were needed to carry out this work for QS, which amounted to roughly 12.3 ms for each of the 423 pages updated. Of this, 7.3 ms was spent copying the objects on the page, 2.8 ms was used to upgrade the lock on the page, and .9 ms on average was spent calling *mmap* to change the page's protection to allow write access.

The response time of QS also increases relative to E during T2A because transaction commit is more expensive for QS (Figure 5.6). The commit time for QS can be broken down into the time required to perform three basic activities, plus a small amount of additional time to perform minor functions like reinitializing data structures, etc. The first of the basic operations involves *diffing* objects on pages that have been updated and calling ESM to

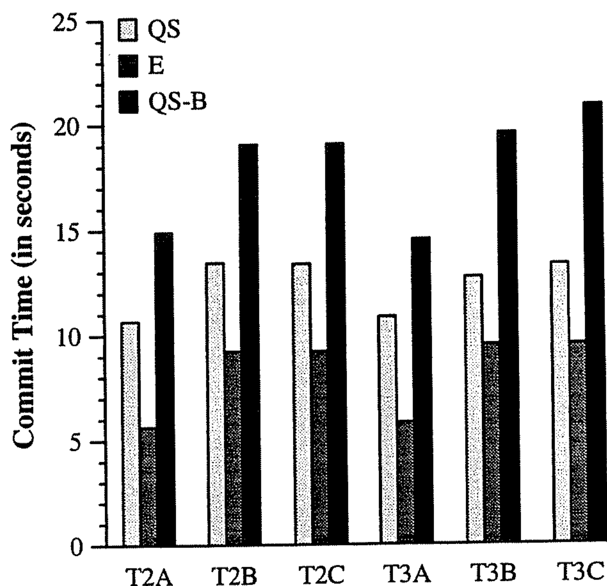

Figure 5.6. T2 and T3 Commit Times.

generate log records when it is determined that updates did occur. The diffing phase required a total of 3.035 seconds during T2A, of which .182 seconds was spent calling ESM to generate the 491 log records needed. Thus, the time needed on average to *diff* each of the 423 modified pages (not counting time to generate log records) was 6.7 milliseconds.

The second major task performed during transaction commit is to update the mapping object associated with each modified page. Our measurements showed that 3.084 seconds (7.2 ms per page) were required for this phase of commit processing. The final step in committing a transaction is performed by ESM. This involves writing all log records to disk at the server and flushing all dirty pages back to the server from the client. This phase of commit processing required 3.501 seconds during T2A.

Turning to T2B and T2C, we see that QS is 17% and 20% faster than E, respectively. As one would expect, QS does better relative to E during T2B and T2C when updates are more dense since QS copies and *diffs* fewer objects unnecessarily. In fact the absolute performance of QS degrades only slightly during T2B relative to T2A. This is due almost entirely to increased time during commit for *diffing* objects and generating log records. More precisely, 5.804 seconds was required do the *diffing* during T2B (.280 seconds of this was for generating log records). The average *diffing* cost per page was 12.9 ms during T2B (not counting logging). We also note that the performance of QS was basically the same during T2B and T2C, while the performance of E was 5% slower. This is because repeatedly updating an object is very inexpensive in QS, as objects are accessed via normal virtual memory pointers while updating an object under the approach used by E requires a function call per update.

The performance difference between QS and E narrows further during T3 relative to T2 and T1. QS has better performance than E in all cases, but nearly similar overheads for index maintenance make this difference less noticeable. In contrast to the relatively stable performance of the systems during T2, the response times of the systems steadily increase when going from T3A to T3B to T3C. This is because each update of an indexed attribute results in the immediate update and logging of the update to the corresponding index. Although the schemes employed by QS and E in using the ESM B-tree indices differ slightly, their performance is basically the same. Neither of the systems supports automatic index maintenance, so the index updates are coded as C++ method invocations on a class variable of type *index*. QS-B is always much slower than the other systems during T2 and T3, especially during the B and C traversals. This is because the 4Mb area used to hold recovery data wasn't big enough to hold all of the objects from modified pages during these traversals for QS-B.

5.6.3. Small Hot Results

The hot results were obtained by re-running the OO7 benchmark operations after all of the data needed by each operation had been cached in the client's main memory by the cold traversal. Figure 5.7 shows hot times for the traversals and Figure 5.8 shows hot times for the queries run on the small database. The times for QS-B are omitted since they were identical to those shown for QS.

As one would expect, the performance of QS is generally better than E. It is somewhat surprising, however, that E is just 23% slower than QS during T1. To determine the reasons for this relatively small difference, we used *qpt* [Ball92] to profile the benchmark application. Table 5.7 presents the profiling results for T1. The T1 hot traversal time has been broken down in Table 5.7 based on the percentage of CPU time spent in several groups of functions. Table 5.7 shows that E spent 33% of the time executing EPVM 3.0 interpreter functions. Most of this time was spent dereferencing unswizzled pointers. Both QS and E spent a considerable amount of time allocating and deallocating space in the transient heap (see the entry for *malloc*). This is because an "iterator" object is allocated in the heap for every node (assembly object, composite part, and atomic part) in the object graph that is visited during the traversal. The "iterator" object establishes a cursor over the collection of pointers to sub-objects so that the sub-objects can be traversed.

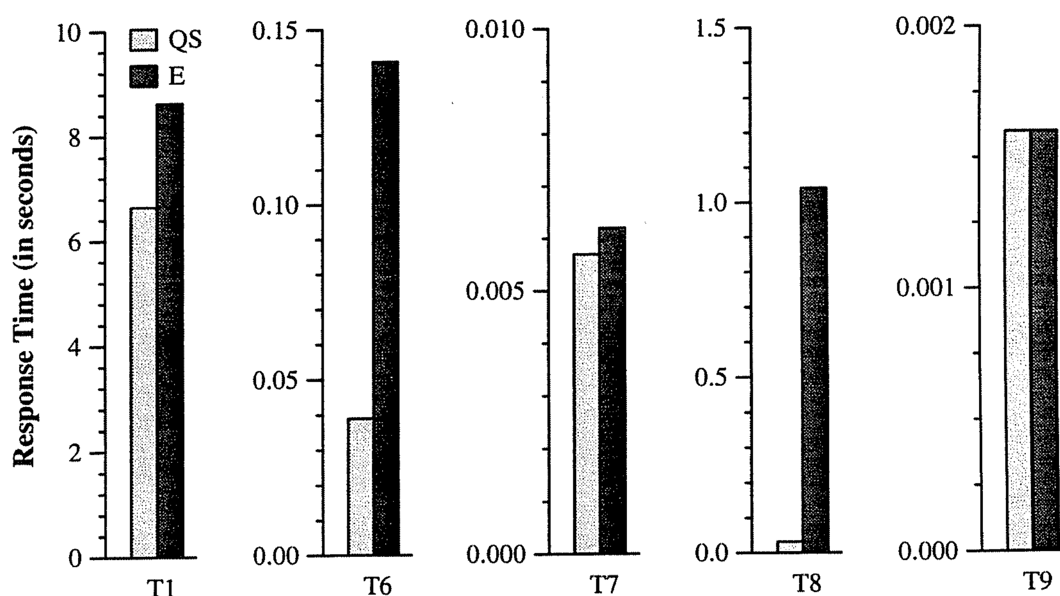

Figure 5.7. Traversal Hot Times.

The entry labeled *part set* in Table 5.7 gives the time spent executing functions that maintain the set of atomic part ids that have been visited in each composite part's subgraph of atomic parts. This set is needed so that the same atomic part is not visited more than once. The amount of time spent in other functions that implement the traversal, such as functions that iterate over collections of pointers to sub-objects and that implement the recursive traversal, was 8% for QS and 17% for E. The higher percentage for E reflects the additional cost of dereferencing large pointers in E. When each node in the object graph is visited, a simple function is called that examines a field in the object to make sure that the object is faulted into memory. The time spent in these functions was .7% for both systems. The detailed numbers in Table 5.7 are surprising because the small amount of additional work involving transient data structures that were needed to implement T1 accounts for a such a large percentage of the overall cost. The results show how quickly a small amount of additional computation can mask differences in the cost of accessing persistent data between the systems.

E is 3.6 times slower than QS during T6. QS performs better relative to E during T6 (the sparse traversal) than during T1 (the dense traversal) since there is less overhead for maintaining transient data structures during T6. For example, the sets of part ids are not maintained since only the root part is visited during T6. The performance of the systems is very close during T7. T7 visits very few objects in the database (10 to be precise) since it simply follows pointers from a single atomic part up to the root of the module. Thus, differences in traversal cost are easily diminished by other costs, such as the overhead to look the atomic part up in the index, etc. Figure 5.7 shows that E is a factor of 32 slower than QS during T8. T8 scans the manual object, a large object spanning several pages on disk. In the case of E, an EPVM 3.0 function call is performed for each character of the manual that is scanned, while QS accesses each character of the manual via a virtual memory pointer. E spent 91% of its time executing EPVM functions during T8. During T9 the systems have identical performance. T9 does very little work on persistent data that

description	**% of time**	
	QS	**E**
EPVM 3.0	-	33.31
malloc	56.13	24.99
part set	35.18	24.57
traverse	8.03	17.12
do nothing	0.64	0.70
misc.	0.02	0.01
total	100.00	100.00

Table 5.7. T1 hot traversal detail.

involves pointers, so the time shown in Figure 5.7 largely reflects the similarity in index lookup costs between the systems.

We now discuss the query hot times shown in Figure 5.8. QS and E have nearly the same performance for all of the queries except Q5, where E was 3.6 times slower than QS. QS is faster than E during Q5 because Q5 does a lot of pointer dereferences as part of the pointer join between base assemblies and composite parts. The times for Q1 and Q2 reflect the cost of index lookups, which are the same between the systems. QS is actually a little slower then E during Q3 because the collection scan code for QS is slightly less efficient than for E. This was due to coding differences and not to any fundamental difference between the systems. Q4 also performs several index lookups which make the performance of QS and E similar.

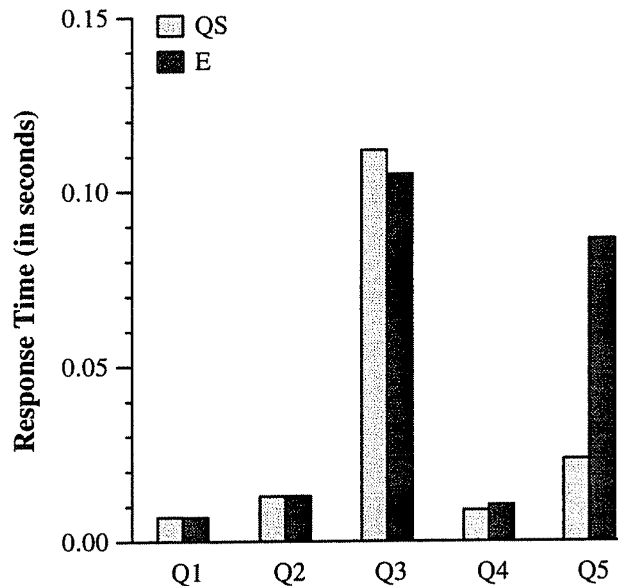

Figure 5.8. Query Hot Times.

5.6.4. Medium Cold Results

This section presents the cold times for the OO7 benchmark operations run on the medium database. The results presented represent the average of 5 runs of the benchmark experiments. Figure 5.9 presents the cold response times for the traversal operations and Table 5.8 gives the number of client I/O requests. We see in Figure 5.9 that, as in the case of the small database, QS has the best performance during T1. QS is 41% faster than E during T1 while it performs 63% fewer I/Os. E, on the other hand, is 36% faster than QS-B during T1. E has better performance than QS during T6 and T7. QS is slower during T6 because one page fault is required to read each

composite part in all of the systems and QS has higher per fault costs. The times shown for T7 and T8 are similar to the small database case. QS is slower due to higher per fault costs during T7. E is slower than QS during T8 due to the overhead of calling EPVM to scan each character of the manual. The results for T9 (not shown) were identical to the small case.

Figure 5.10 and Table 5.9 show the cold response times and client I/O requests for the queries run on the medium database. E always has the best performance during the queries. During Q1 and Q2 E benefits because accesses of atomic parts are very unclustered. The difference in performance between QS and E narrows during Q3 because accesses are more clustered in this case. It is interesting that QS is slower than E during Q3 since it performs significantly fewer I/O operations. QS is slower because of the overhead for managing paging in the client buffer pool. In Q4 and Q5 object accesses are again unclustered resulting in a high number of page-faults per object access, which causes QS to have slower performance than E.

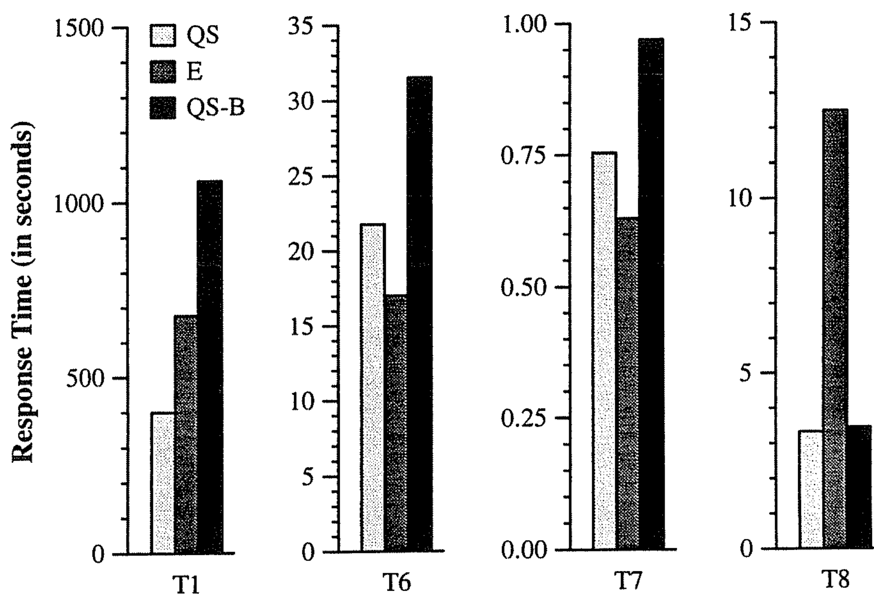

Figure 5.9. Medium Database, Traversal Cold Times.

	T1	T6	T7	T8
QS	13216	610	27	130
E	35622	558	25	129
QS-B	36963	802	32	130

Table 5.8. Traversal Cold I/Os.

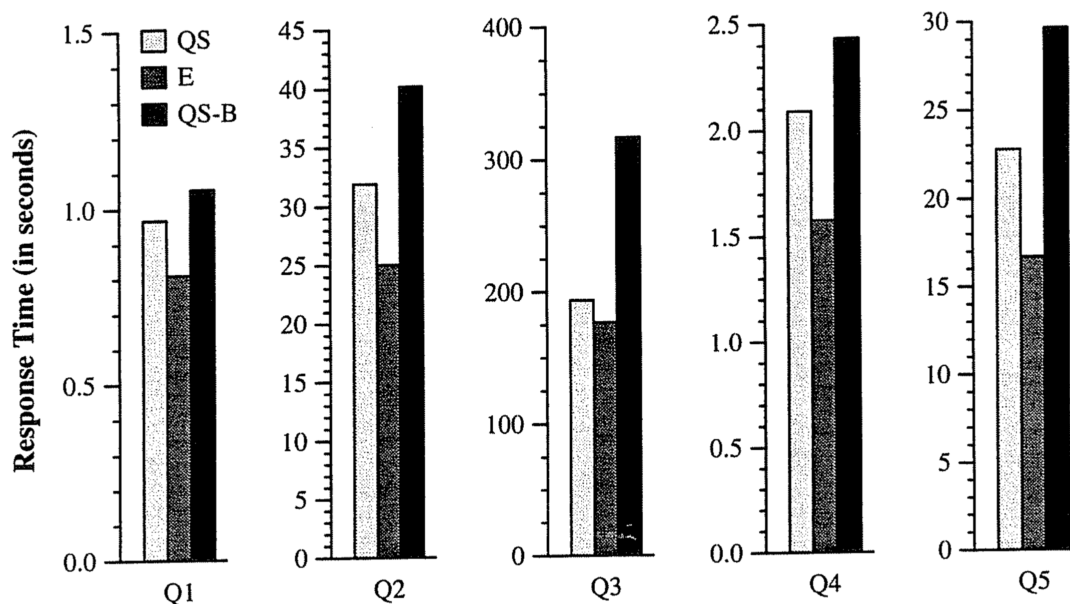

Figure 5.10. Medium Database, Query Cold Times.

	Q1	Q2	Q3	Q4	Q5
QS	34	901	5997	68	595
E	26	919	8045	58	558
QS-B	35	1095	10951	81	751

Table 5.9. Query Cold I/Os.

Turning now to traversals T2 and T3 (Figure 5.11) which perform updates, we see that QS outperforms E during the T2A and T3A traversals which only update the root atomic part of each composite part. This is understandable when one considers that both QS and E have to do basically the same amount of work to process the updates that they did in the small database case. This makes the cost difference of doing the traversal itself the main factor effecting their relative performance. The relative performance of QS worsens during T2B and T2C causing QS and E to have similar performance. Recovery is more expensive for QS during T2B and T2C since the buffer used for recovery is much smaller than the fraction of the database that is updated. QS-B has much worse performance than both QS and E in Figure 5.11. This is caused by the fact that in addition to higher traversal costs, QS-B has higher costs for recovery as well.

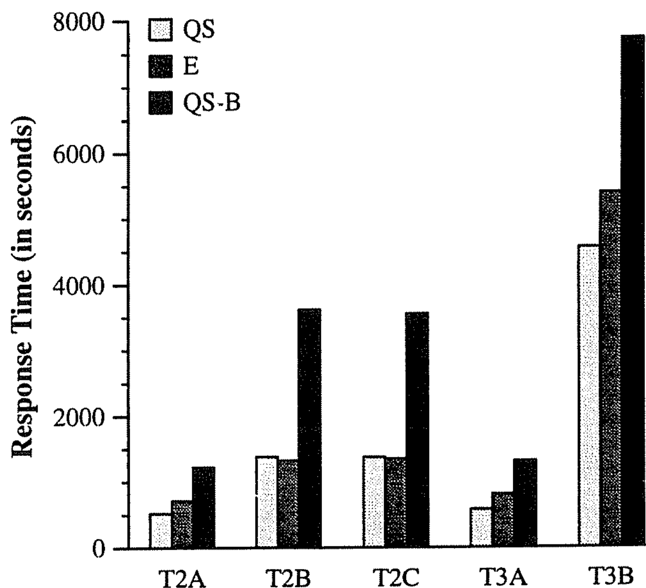

Figure 5.11. Medium Database, Traversal Cold Times.

5.6.5. Effect of Collisions

QuickStore always tries to assign a disk page to the virtual memory location that it last occupied when in memory. This section considers the effect on performance of relocating pages at different memory addresses when they are faulted into memory. This increases faulting costs because pointers between persistent objects must be updated to reflect the new assignment of disk pages to virtual memory addresses. We consider two approaches to dealing with page relocations. The first approach updates or swizzles pointers that need to be modified when pages are faulted into memory, but these changes are not written back to the database. This implies that the changes will have to be made again if the same data is accessed in subsequent transactions. We refer to this system as QS-CR (QuickStore with continual relocation). The second approach commits the changed mapping to the database. This approach is more costly initially, but may be able to avoid further relocations in the future. This approach also has the disadvantage that it can turn a read-only transaction into an update transaction. We refer to this approach as QS-OR (QuickStore with one-time relocation).

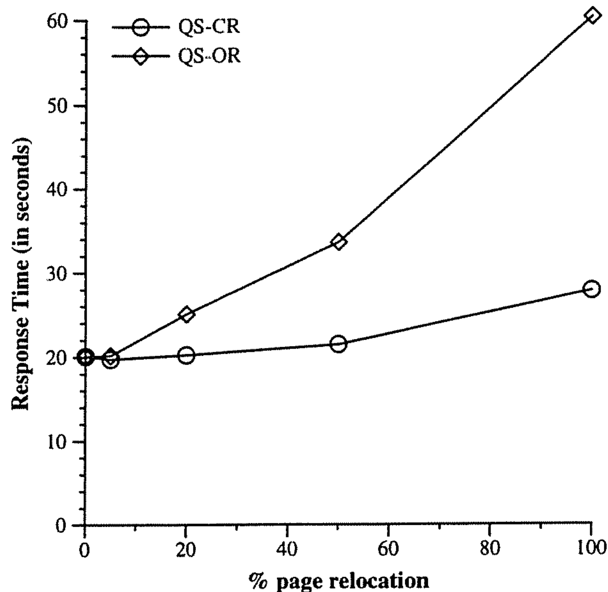

Figure 5.12. T1, vary % of relocations.

Figure 5.12 presents the results for T1 run on the small database when the percentage of pages that are relocated in memory is varied from 0 to 100%. The pages that were relocated in the experiment were picked at random. Figure 5.12 shows that when the number of relocations is small (5%), the performance of the systems is not significantly affected. However, when the relocation percentage is 20%, QS-OR is 25% slower than when no relocations occur. The difference in performance between QS-OR and QS-CR at this point is also about 25%. The performance of QS-CR slows by 7% and 38% when the percentage of relocated pages is 50% and 100%, respectively, while the corresponding decrease in performance for QS-OR is 67% and 116%, respectively. QS-OR is much slower than QS-CR when all pages are relocated since it must commit updates for all of the pages in the database.

5.7. Conclusions

This chapter compared the performance of QuickStore, which uses memory-mapping techniques to implement persistence, to the performance of E, a persistent version of C++ that uses a software interpreter. The OO7 benchmark was used as a basis for comparing the performance of the two systems. The results of the study give a clear and accurate picture of the tradeoffs between the two approaches, which we summarize below.

The results of the cold traversal experiments showed that when object accesses are clustered (T1), QuickStore has the best performance. This is because object sizes in QuickStore are smaller than in E, due to the different schemes used by the systems to represent pointers on disk. QuickStore's smaller object size allows it to perform

significantly fewer disk I/O operations to read the same number of objects when accesses were clustered. When object accesses were unclustered (i.e. in T6, Q1, Q2, etc.), the performance of QuickStore was comparable or worse than the performance of E. The reason for this change, relative to the clustered case, was that there was less difference between the systems in the number of pages faulted into memory per object access during the unclustered experiments. This exposed the fact that QuickStore has higher per faults costs than E. In addition, there were some cases (T7 and T9) when QuickStore always had lower performance due to higher faulting costs. The lower performance for QuickStore during T7 was influenced by the cost of reading a relatively large amount of mapping information to support the memory-mapping scheme that it uses.

The higher faulting costs for the memory-mapping scheme were also highlighted by the performance of QS-B (QuickStore with big objects). Except during the experiments that scanned large objects (T8), QS-B's performance was lower than E's during the read-only cold experiments. The memory-mapped schemes had better performance than E when large objects were accessed because large object accesses require significantly more CPU work with the software approach. This additional cost caused E to be slower even in the cold case.

For the traversals where faulting costs were examined in detail, it was shown that the average cost per fault for QuickStore was roughly 20% higher than for E. The largest component of the additional faulting cost for the memory mapping scheme was the time required to read mapping information from disk. This comprised 4% of the average cost per fault. The detailed cost analysis also showed that the overhead for handling page protection faults and manipulating page access protections were each 3%. The smallest component of the faulting cost for QuickStore was the CPU cost for swizzling pointers. This was just 1% of the average cost per fault.

The performance of QuickStore was generally better than E when updates were performed. The results of the update experiments showed, however, that the page-based diffing scheme used by QuickStore to generate log records was more expensive when updates were sparse and when the update activity was heavy enough to cause log records to be generated before transaction commit. This effect is examined in more detail in Chapter 6. QuickStore performed better relative to E when a higher percentage of objects were updated on each page, as QuickStore copied and diffed fewer objects unnecessarily in this case. The detailed times for the update experiments showed that the cost of diffing was ranged from 7 to 12 milliseconds per page for the OO7 update operations.

The hot results helped to quantify the performance advantage of the memory-mapped scheme when working on in-memory objects. In some cases (T1) the difference in performance between QuickStore and E was only 23%, while in others (T6) QuickStore was over 3 times faster than E. This showed how quickly the performance of the

systems converged when a small amount of additional work was performed, and correlates well with the hot results from Chapter 3. The results also showed that E was significantly slower than QuickStore when doing in-memory work on large objects since this required all accesses to be handled by the E interpreter.

Finally, we also examined the performance of QuickStore when pages of objects must be relocated in memory, which increases the amount of swizzling work performed by QuickStore. When the percentage of pages relocated was small, the performance of the systems did not noticeably worsen. However, a high percentage of relocations did have a noticeable effect on overall performance, particularly when the new mapping tables were written back to the database. The best approach overall appeared to be to avoid writing the changed mapping tables to disk and to continually relocate pages in memory since the negative impact on performance of this approach was small.

CHAPTER 6

RECOVERY IN QUICKSTORE

This chapter examines the performance of several alternative approaches to implementing recovery in QuickStore. Providing recovery services in QuickStore raises several challenging implementation issues, not only because it is a memory-mapped storage system, but also because of the kinds of applications that OODBMSs strive to support. In addition, QuickStore, since it is implemented on top of the EXODUS Storage Manager (ESM), is a client-server, page-shipping system [DeWitt90]. This raises additional performance concerns that are not present in database systems based on more traditional designs, i.e. centralized DBMSs or systems based on a query-shipping architecture.

The remainder of this chapter is organized as follows. In order to motivate the design of the recovery algorithms studied in this chapter, Section 6.1 presents a detailed discussion of the factors that make recovery in QuickStore an interesting problem. Section 6.2 describes several alternative techniques for implementing recovery in QuickStore. Next, Section 6.3 describes the performance study that was carried out to compare the performance of the different schemes. The study was conducted using the OO7 database benchmark. Section 6.4 presents the results of the performance study. Section 6.5 discusses related work. Finally, Section 6.6 presents our conclusions.

6.1. Background and Motivation

One challenge that QuickStore recovery faces is the way that updates occur. As was mentioned in Chapter 4, QuickStore allows application programs to update objects directly in the ESM client buffer pool by dereferencing normal virtual memory pointers. This strategy allows applications to perform updates at memory speeds with essentially no overhead, but it also makes detecting the portions of objects that have been updated much more difficult than in systems that use more traditional implementation techniques. For example, in E, an interpreter function is called to perform each update. This provides a hook that the system can use to record the fact that the update has occurred. We note that the approach used in E requires special compiler support to insert function calls for updates into the application code at compile time. Commercial OODBMSs that support automatic index maintenance, such as ObjectStore [Lamb91], typically include a similar mechanism to trap updates so that index maintenance can be

performed whenever a relevant update occurs. However, like QuickStore, ObjectStore allows unindexed attributes to be updated by dereferencing normal virtual memory pointers.

A second factor affecting the design of a recovery scheme is that QuickStore, like most OODBMSs, is designed to handle non-traditional database applications, e.g. CAx, GIS, and office information systems. Applications of this type typically read objects into memory and then work on them intensively, repeatedly traversing relationships between objects and updating the objects as well. This behavior differs dramatically from the behavior exhibited by relational database systems, which usually update an individual tuple just once during a particular update operation. For example, giving all of the employees in a company an annual raise requires only a single update to each employee tuple. Since relational database systems typically update each tuple only once, they generate a log record for recovery purposes for each individual update. However, such a strategy is not practical in an OODBMS where an object may be updated many times during a single method invocation. Here it is necessary to batch the effects of updates together in order to achieve good performance by attempting to generate a single log record that records the effects of several updates to a particular object.

A final consideration arises from the fact that QuickStore is based on a client-server architecture in which updates are performed at client workstations. This raises the issue of cache consistency between clients and the server [Frank93] since both the client and the server buffer pools can contain cached copies of a page that has been updated. In addition, the stable copy of the transaction log is maintained by the server, so clients are typically required to ship log records describing updates over the network to the server before a transaction can commit. This differs from the traditional approach used in centralized database systems, where updates are performed at the server and log records are generated locally at the server as well.

6.2. Recovery Strategies

This section describes the four recovery schemes that we implemented and evaluated: page diffing, sub-page diffing, whole-page logging, and redo at server. We begin by describing the implementation of recovery in ESM since several of the techniques are built on top of or involve modifications to the ESM recovery scheme.

6.2.1. Recovery in EXODUS

The EXODUS Storage Manager is a client-server, page-shipping system [DeWitt90] in which both clients and servers manage their own local buffer pools. When a client needs to access an object on a page that is not currently cached in its local buffer pool, it sends a request to the appropriate server asking for the page. If necessary, the

server reads the page from secondary storage into main memory, sends a copy of the page to the client, and also retains a copy of the page in its own buffer pool as well.

Objects are updated at the clients and clients also generate log records that describe the updates for recovery purposes. Log records for updates contain both redo and undo information for the associated operation. For example, if a range of bytes within an object is updated, then the log record will contain the old and new values of that portion of the object. Log records are collected and sent back to the server a page-at-a-time. For simplicity, ESM enforces the rule that log records generated for a page are always sent back to the server before the page itself is sent. Thus, the server never has a page cached in its buffer pool for which it doesn't also have the log records describing the updates present on the page.

The server manages a circular, append-only log on secondary storage and uses a STEAL/NO-FORCE buffer management policy [Haerd83]. Clients can cache pages in their local buffer pools across transaction boundaries. However, inter-transaction caching of locks at clients is not supported. Also, all dirty pages are sent back to the server at commit time in order to maintain cache consistency between the clients and the server and simplify recovery. The log records generated on behalf of a transaction must be written to the log by the server before the transaction commits, but the dirty pages themselves are not forced to disk. See [Frank92] for a more detailed description of the ESM recovery scheme.

6.2.2. The Page Diffing Approach

The ESM recovery mechanism handles the generation of log records for updates. However, if recovery is done in a straightforward way by using the services provided by ESM, then each time an update is performed, a log record would be generated. This is a situation that we would like to avoid, if possible. Furthermore, there is no obvious way for a QuickStore application to detect the fact that an update has occurred since application programs are allowed to update objects by simply dereferencing standard virtual memory pointers.

6.2.2.1. Enabling Recovery for Page Diffing

The problems mentioned above can be addressed by employing a page diffing (PD) scheme to generate log records. This approach works as follows. In QuickStore, write access is not automatically enabled on virtual memory frames that are mapped to pages in the client buffer pool. In particular, write access is never enabled on frames that are mapped to pages for which the actions necessary to *enable* recovery have not been taken. Thus, any attempt by an application program to update an object on a page for which recovery is not enabled will result in a

page-fault, thus causing QuickStore's fault-handling routine to be invoked.

The fault-handling routine starts by looking up the page descriptor corresponding to the virtual memory address that caused the fault. Upon inspecting the status flags in the relevant page descriptor entry, the fault-handler will detect that the access violation is due to a write attempt. If recovery is not already enabled on the page, the fault-handler then copies the page into an area in memory termed the *recovery buffer* and sets the page descriptor entry to point to the copy. The fault-handler also obtains an exclusive lock on the page from ESM, if needed, and enables write access on the virtual frame that caused the fault. At this point, all of the work needed to enable recovery on the faulted-on page is complete, so control is returned to the application program which then can proceed to update objects on the page directly in the buffer pool.

Figure 6.1 shows the effect of the actions described above on the data structures maintained by QuickStore. In Figure 6.1 page *a*, which is cached in the buffer pool, is shaded to show that it has been updated. A copy containing the value of page *a* before recovery was enabled has been placed in the recovery buffer, and the page descriptor for page *a* has been set to point to the copy. As Figure 6.1 shows, the recovery buffer contains room for M pages, where M is fixed and $1 \leq M \leq N$ (where N is the number of pages in the ESM client buffer pool). Since M is fixed, the recovery buffer can become full, so it may be necessary to free up space periodically when an additional page is updated, by generating log records for a page that has already been copied into the recovery buffer. Space in the recovery buffer is managed using a simple FIFO replacement policy.

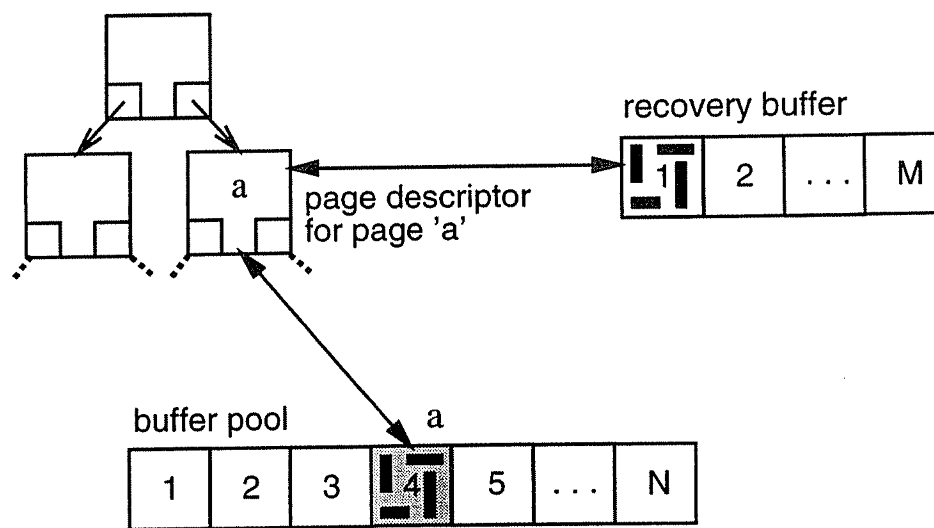

Figure 6.1. The page diffing approach.

6.2.2.2. Generating Log Records for Pages

At transaction commit time, or when paging in the buffer pool occurs or the recovery buffer becomes full, the old values of objects contained in the recovery buffer and their corresponding updated values in the buffer pool are compared (diffed) to determine if log records need to be generated. The actual algorithm used for generating log records is slightly more sophisticated than the simple approach of generating a single log record for each modified region of an object. This simple approach was rejected since it has the potential to generate a great deal of unnecessary log traffic. For example, consider an object in which the first and third words (1 word = 4 bytes in QuickStore) have been updated. The simple approach would generate two log records for the object. Since each ESM log record contains a header of approximately 50 bytes, the total space used in the log would be 116 bytes (50 bytes for each log header plus 4 bytes for each before and after image). On the other hand, if just one log record were generated, only 74 bytes would have been used (50 bytes, plus 12 bytes for each before and after image), providing a 36% savings in the amount of log space used.

The actual algorithm for generating log records uses diffing to identify consecutive modified regions in each object on a page; if only one region exists then a single log record is generated for the object. For example, Figure 6.2a shows an object that contains three modified regions, labeled *R1*, *R2*, and *R3*. The diffing algorithm starts from the beginning of the object, so initially it would identify the two modified regions *R1* and *R2*. It is easy to show,

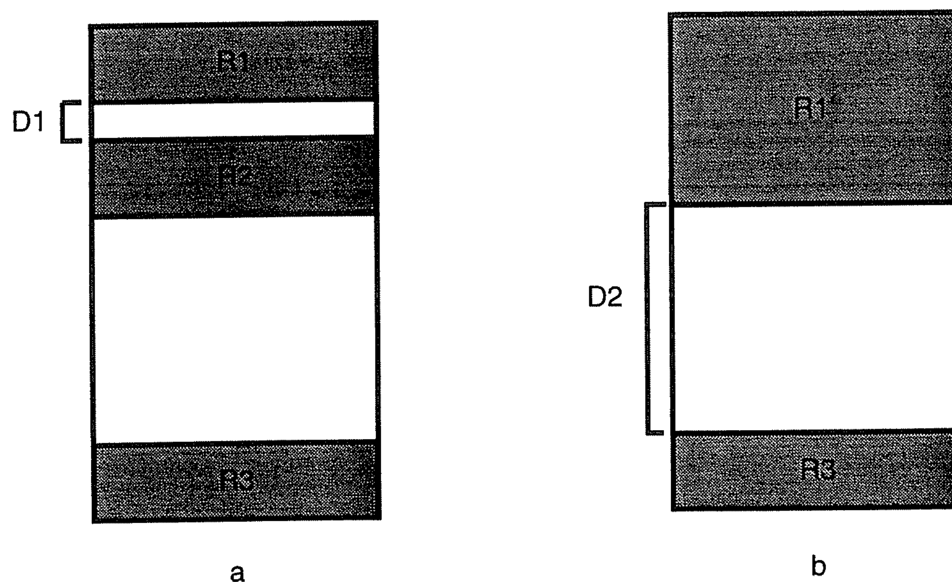

Figure 6.2. Combining modified regions in an object.

given the before/after-image format of log records, that if the distance $D1$ between $R1$ and $R2$ satisfies the equation $2 * \text{size}(D1) > H$, where H is the size of a log record header, then generating separate log records for each region will generate the least amount of log traffic. If this were the case in the example, then the algorithm would generate a log record for $R1$ before proceeding to consider additional regions. However, a log record for $R2$ is not immediately generated, since it may be better to combine $R2$ with some region that has not yet been discovered. If, on the other hand, $2 * \text{size}(D1) <= H$, i.e. the distance between $R1$ and $R2$ is small, then the algorithm combines the two regions into a single region. Figure 6.2b illustrates the second case. Here, $R1$ and $R2$ have been combined into a single region $R1'$. Again, no log record is generated at this step, as the algorithm may decide to combine $R1'$ with additional regions.

In either case mentioned above, the algorithm continues by identifying the next modified region in the object (if there is one) and repeats the previous check using the newly discovered region and either the combined region from the previous iteration or the region from the previous iteration for which no log record has yet been generated. In the example shown in Figure 6.2b, the algorithm would next examine regions $R1'$ and $R3$ to see if they should be logged separately or combined. Since the distance between $R1'$ and $R3$ is large, the algorithm will generate separate log records for $R1'$ and $R3$. Finally, we note that the decision concerning whether or not to combine consecutive modified regions depends only on the distance between them and not on their size, so the order in which the regions are examined does not matter. Thus, the algorithm is guaranteed to generate the minimum amount of log traffic.

6.2.3. The Sub-Page Diffing Approach

The page diffing recovery scheme described in the previous section has some potential disadvantages. Its most obvious disadvantage is that the CPU overhead for copying and diffing a whole page may be fairly high, especially when very few updates have actually been performed on the page. In addition, page diffing has the potential to waste space in the recovery buffer by copying a whole page when only a few objects on the page have been updated. This can increase the number of log records generated during a transaction, if the recovery buffer becomes full. However, the page-wise granularity of the page diffing scheme is necessary if applications are allowed to update objects via normal virtual memory pointers as virtual memory is page-based.

An alternative approach would be to interpret update operations in software. One way that this can be accomplished, in general, is by compiling persistent applications using a special compiler that inserts additional code to handle update operations. This code can simply be a function call that replaces the usual pointer dereference at the

points in the application program where updates occur. In our case, the function that is invoked is part of the QuickStore runtime system. This approach yields a system in which objects may be read at memory speed using standard virtual memory pointers, but in which update operations are more heavy-weight, requiring a function call and other software overhead. The hope when using such an approach is that the extra cost incurred on each update will be repaid through reduced recovery costs.

6.2.3.1. Enabling Recovery for Sub-Page Diffing

Figure 6.3 illustrates the in-memory data structures used by QuickStore to implement the sub-page diffing (SD) approach. Under the SD approach, each page is divided into a contiguous sequence of regions called blocks. Blocks are uniform in size (and we experimented with block sizes ranging from 8 to 64 bytes.). As Figure 6.3 shows, the page descriptor for a page that has been updated holds a pointer to an array containing pointers to copies of blocks that have actually been modified. There is one entry in the array for each block on the page, and array entries for unmodified blocks are null. Blocks were used as the sub-page unit of copying and diffing instead of objects for two reasons. First, it is cheaper in terms of CPU cost to identify the block on a page that is being updated than it is the object, when an update occurs (see below). And second, objects within a page may be rather big (i.e. up to 8K-bytes in size). If this is the case, then the advantages of the sub-page diffing approach will be lost when updates are sparse.

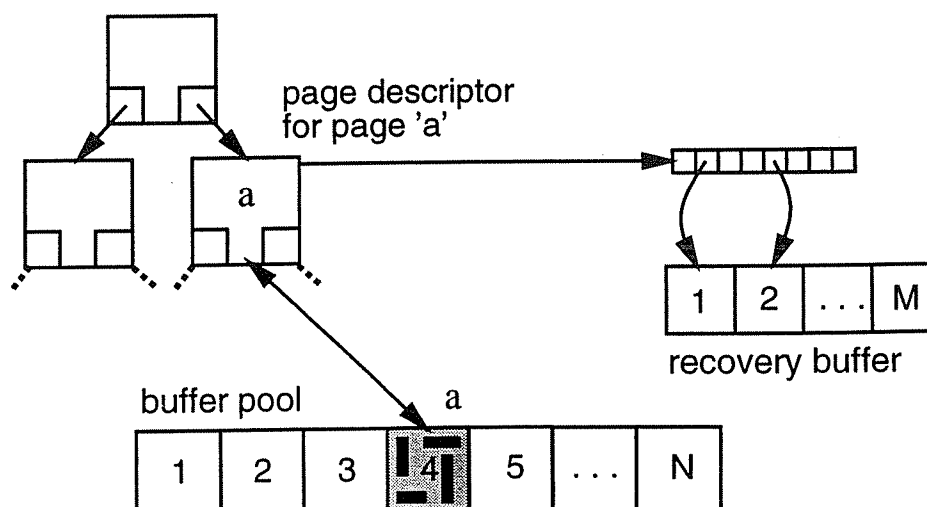

Figure 6.3. The Sub-page diffing approach.

Each time the QuickStore update function is called, it looks up the page descriptor of the appropriate page using the address of the memory location that is being updated (which is passed as a parameter). Once the page descriptor is found, a check is made to see if the block that is about to be updated has been copied yet. This check is relatively inexpensive since the address of the location in memory being updated can be used to index the array of block pointers contained in the page descriptor (after applying some simple logical operations to it). If a copy of the block has not yet been made, then a copy is placed in the recovery buffer. In addition, the status flags in the page descriptor are examined to see if an exclusive lock has been acquired for the page, and write access is enabled on the virtual frame mapped to the page (if it is not already). Finally, the update itself is performed. We note that write access could be enabled automatically on virtual frames when using the sub-page approach. We chose not to do this because not enabling write access allows the runtime system to catch erroneous writes to virtual frames that have not been updated, and because the extra cost of the approach we used is relatively low.

6.2.3.2. Generating Log Records for Sub-Pages

Like the page diffing approach, the SD approach generates log records at transaction commit time, when a modified page is paged out by the buffer manager, or when the recovery buffer becomes full. Log records can be generated by diffing the original values of the blocks contained in the recovery buffer with the modified versions of the same blocks located in the buffer pool, using the diffing scheme described in Section 6.2.2. In addition, one could avoid the expense of diffing altogether by simply logging entire blocks. We experimented with both techniques, and refer to the sub-page approach without diffing as sub-page logging (SL) in the upcoming discussion on performance.

6.2.4. The Whole-Page Logging Approach

This section describes the third recovery algorithm included in the study. This algorithm is termed whole-page logging (WPL) since entire modified pages are written to the log instead of log records for updated regions of objects. As was mentioned in Chapter 3, this is the basic approach used in ObjectStore [Lamb91]. The advantages of WPL are that it avoids the client CPU cost that is incurred by the two diffing schemes for copying and diffing. WPL also avoids the memory overhead at clients for storing the original values of pages/blocks. This can potentially improve performance by allowing WPL to allocate more memory to the client buffer pool. Finally, WPL allows applications to update objects at memory speeds by dereferencing normal virtual memory pointers, so the cost of actually performing updates is low.

The disadvantage of whole-page logging is that it logs entire after-images of dirty pages at the server. This means that all of the pages dirtied by a transaction must be forced to the log at the server before the transaction commits. We note, however, that the cost of shipping dirty pages back to the server does not add any additional costs in ESM since its recovery algorithm always sends updated pages back to the server when a transaction commits (see Section 6.2). The WPL scheme does not rely on the support provided by ESM for recovery as the diffing schemes do. Thus, WPL requires modifications in the actions taken at both clients and the server to support recovery.

6.2.4.1. Actions Performed at Clients

The whole-page logging algorithm works as follows at the clients. When an application first attempts to update a page at a client, a page-fault is signaled, as usual. The QuickStore page-fault handling routine marks the copy of the page cached in the buffer pool as dirty, in addition to requesting an exclusive lock if necessary, and enables write access on the virtual memory frame that is mapped to the page. Control is then returned to the application program. Dirty pages are shipped back to the server when the transaction commits, or possibly sooner if paging in the client buffer pool occurs. Note that no log records are generated for updates at the clients under this approach; only dirty pages are shipped back to the server. Figure 6.4 shows the basic client data structures used by QuickStore to support whole-page logging. Figure 6.4 also illustrates the fact that no additional memory is used for recovery at the client under WPL.

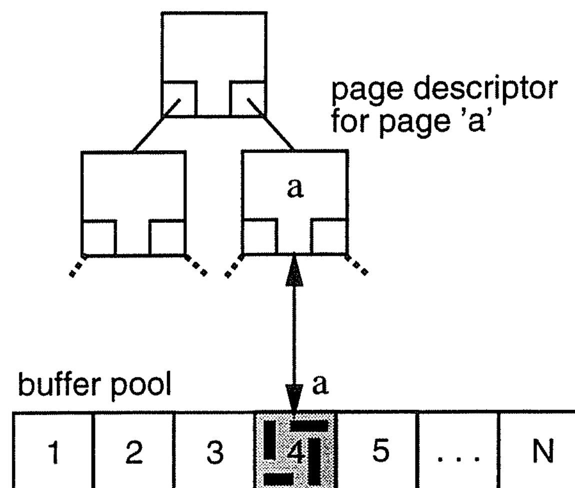

Figure 6.4. The whole-page logging approach.

6.2.4.2. Actions Performed at the Server

When the server receives a dirty page, it appends the page to the log and caches a copy of the page in its own buffer pool. The server does not allow the original copy of the page on disk to be overwritten with the new copy of the page until after the transaction that updated the page commits. If paging in the server's buffer pool causes a dirty page to be replaced during a transaction, then the page is read from the log if it is needed subsequently by a client. When a transaction reaches its commit point, the original values of any updated pages are still located on disk in their permanent locations, and the updated values of the pages have been flushed to the log together with a commit log record for the transaction. This makes it possible to abort a transaction at any time before the commit point is reached by simply ignoring, from then on, any of its updated values of pages located in the log or cached in memory, no undo processing for updates is required.

A page updated by a committed transaction, must be maintained in the log until one of two things happens. The first is that the page is read from the log and used to overwrite its permanent location on disk. The log space for the page can then be reused since the copy of the page contained in the log is no longer needed for recovery. The reason for this is fairly obvious. For example, suppose that transaction *T* updates page *P* and then commits, i.e. *P* is forced to the log. If a crash resulting in the loss of the server's volatile memory occurs any time after *T* commits, but before *P* overwrites its permanent location on disk, then the value of *P* stored in the log must be available in order for the system to correctly restart. If *P* has safely overwritten its permanent location on disk, however, then that copy of *P* can be used following a restart.

Space for a page in the log can also be reused if a subsequent transaction updates the page and commits, thereby forcing a new copy of the page *C2* to the log, before the initial copy of the page *C1* in the log overwrites the permanent location of the page. In effect, *C1* is not needed at this point, since following a crash *C2* will be used.

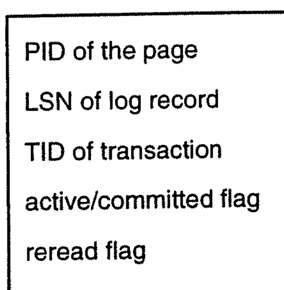

Figure 6.5. Format of a WPL table entry.

Note, however, that both *C1* and *C2* must be maintained in the log until the transaction that wrote *C2* commits.

The server maintains an in-memory table, called the WPL table, to keep track of pages contained in the log that are needed for recovery purposes. Figure 6.5 shows the format of a table entry. Each table entry contains the page id (PID) of the page, which identifies its permanent location on disk. Entries also contain the log sequence number (LSN) of the log record generated for the page. The LSN identifies the physical location of the page in the log. Additional fields stored in each entry include the transaction id (TID) of the transaction that last dirtied the page and some additional status information. When a page is initially written to the log, the status information records the fact that the transaction that dirtied the page has not yet committed. Finally, table entries contain a pointer that refers to the entry for a previously logged copy (if any) of the same page if that copy is still needed for recovery.

The server also maintains a list, for each active transaction, of the pages that have been logged for that particular transaction. When a transaction commits the WPL table entry for each page on this list is updated to show that the transaction that modified the page has committed. In order to reclaim log space there is a background thread that asynchronously reads pages modified by committed transactions from the log. (As an optimization, pages modified by a transaction that are still cached in the server buffer pool at commit time are simply marked as having been read.) Once a page has been read from the log, the server is free to flush the page to disk at any time. Once this happens, the entry for the page in the WPL table is removed.

6.2.4.3. Recovering from a Crash

In order to be able to recover from an unexpected crash, the server periodically takes checkpoints. At checkpoint time, the WPL table is written to the log. In order to recover from an unexpected failure, the server must be able to reconstruct the WPL table so that it contains entries for all of the pages that have been updated by committed transactions, but not yet written to their permanent disk locations. Once this is done, the server can resume normal operation.

Restart after a crash requires a single pass through the log that begins at the end of the log and proceeds backward to the most recent checkpoint record. At the start of the pass, a list that records committed transactions, termed the committed transactions list (CTL), is initialized to empty. During the pass a transaction is added to the CTL when a commit record for the transaction is encountered during the backward scan. When a log record for a modified page is encountered, an entry for the page is inserted into the WPL table if the transaction that updated the page is in the committed transaction list. Otherwise, the modify record is ignored since the associated transaction

did not commit before the crash.

When the checkpoint record is reached, the committed transaction list contains an entry for each transaction that committed after the checkpoint was taken. The contents of the checkpoint record itself are then examined. Entries in the checkpoint record that pertain to members of the CTL, or which are marked as pertaining to transactions that committed before the checkpoint was taken, are added to the newly constructed WPL table. At this point the WPL table has been fully reconstructed and normal processing can resume.

6.2.5. The Redo-at-Server Approach

The final recovery algorithm that we examined is termed redo-at-server (REDO). This algorithm is a modification of the ARIES-based recovery algorithm used by ESM in which clients send log records, but not dirty pages, back to the server. Under REDO, when a log record is received at the server the redo information in the log record is used to update the server's copy of the page. The disadvantage of REDO is that the server may have to read the page from secondary storage in order to apply the log record.

In addition to the obvious advantage of not having to ship dirty pages from clients to the server, REDO is also appealing from an implementation standpoint. It simplifies the implementation of a storage manager by providing cache consistency between clients and the server [Frank93]. REDO is currently being used in the initial version of SHORE [Carey94], a persistent object system being developed at Wisconsin. Since REDO only involves changes at the storage manager level, it can be used in combination with any of the recovery schemes mentioned previously that make use of the recovery services provided by EXODUS. Here, we will study its use in conjunction with the PD scheme.

6.2.6. Implementation Discussion

Each of the four recovery schemes described in this section has been implemented in the context of QuickStore/ESM. The two diffing schemes did not require any changes to the base recovery services provided by ESM. Instead of modifying the gnu C++ compiler to insert function calls for updates to support the sub-page diffing approach, the necessary function calls were inserted by hand at the application level to save time. In implementing whole-page logging we made use of the existing ESM recovery code whenever possible. For example, ESM already supported whole-page logging for newly created pages. The changes made to the ESM client to support WPL were therefore fairly minor. The changes made to the server were more substantial, and involved tasks such as maintenance of the WPL table (Section 6.2.4) and rereading pages from the log. Implementing redo-at-server

was again relatively easy. The ESM server already supported redo as part of the ARIES-based recovery scheme that it uses, and it wasn't hard to get the server to apply each log record to the appropriate page as log records were received by inserting a function call in the appropriate spot in the server code.

6.3. Performance Experiments

This section describes the performance study that was conducted to compare the performance of the recovery algorithms described in the previous section.

6.3.1. Benchmark Database

The OO7 object-oriented database benchmark was used as a basis for carrying out the study. The structure of the OO7 database was discussed in detail in Section 5.1. We included two different database sizes in the study, termed small and big. Table 6.1 shows the parameters used to construct the two databases.

We note that the parameters used here do not correspond exactly to the "standard" OO7 database specification of [Carey93]. As indicated in Table 6.1, a module in the small database here is the same size as a module in the small database of [Carey93]; however, modules in the big database differ from the small database in that they contain 2,000 composite parts instead of 500, and there are 8 levels in the assembly hierarchy in the big database versus 7 in the small database. Both the small and big database here contain five modules, as the number of clients that access the database will be varied from one to five. During a given experiment, each module will be accessed by a single client, so a module represents private data to the client that uses it. We decided not let the clients share data in order to avoid locking conflicts and deadlocks both of which can have a major effect on performance. Removing these effects simplified the experiments and allowed us to concentrate on the differences in performance that were

Parameter	**Small**	**Big**
NumAtomicPerComp	20	20
NumConnPerAtomic	3	3
DocumentSize (bytes)	2000	2000
Manual Size (bytes)	100K	100K
NumCompPerModule	500	2000
NumAssmPerAssm	3	3
NumAssmLevels	7	8
NumCompPerAssm	3	3
NumModules	5	5

Table 6.1. OO7 Benchmark database parameters.

due to the recovery mechanisms being studied.

Table 6.2 lists the total size of the databases and the size of a module within each database. The size of a module in the small database is 6.6 Mb which is small enough that an entire module can be cached in main memory at a client (12 Mb). In addition, the total size of the small database (33 Mb) is small enough that it fits into main memory of the server (36 Mb). Thus, the experiments performed using the small database test the performance of the recovery algorithms when the entire database can be cached in main memory. The size of a module in the big database, however, is larger than the main memory available at any single client. In addition, when more than a single client is used the amount of data accessed is also bigger than the memory available at the server. Experiments performed using the big database test the relative performance of the algorithms when a significant amount of paging is taking place in the system.

	Small	**Big**
module	6.6	24.3
total	33.0	121.5

Table 6.2. Database sizes (in megabytes)

6.3.2. Benchmark Operations

The experiments were performed using the T2A, T2B, and T2C traversal operations (see Section 5.1 for details). During each experiment, the traversals were run repeatedly at each client, so that the steady state performance of the system could be observed. Each traversal was run as a separate transaction. The client and server buffer pools were not flushed between transactions, so data was cached in memory across transaction boundaries.

6.3.3. Software Versions

We experimented with several of the QuickStore recovery software versions. Table 6.3 shows the names used to identify the different versions in the performance section. Each name generally consists of two parts. The first part identifies the scheme used for generating log records (PD, SD, or SL), and the second specifies the underlying recovery strategy that was used (ESM or REDO). In addition, the size of the recovery buffer given to the diffing schemes is sometimes appended to the name of these systems in the performance section. For example, PD-REDO-4 denotes a system using page-diffing with a 4 Mb recovery buffer in combination with redo-at-server recovery. In the case of whole-page logging the name has only one part (WPL).

Name	Description
PD-ESM	page diffing, ESM recovery
SD-ESM	sub-page diffing, ESM recovery
SL-ESM	sub-page logging(no diffing), ESM recovery
PD-REDO	page diffing, REDO recovery
WPL	whole page logging

Table 6.3. Software versions.

6.4. Performance Results

This section presents the performance results. We first present and analyze the results obtained using the small database and then turn to the results obtained using the big database.

6.4.1. Unconstrained Cache Results

This section presents results for experiments using the small database. All systems were given 12 megabytes of memory at each client to use for caching persistent data. For the systems that do diffing, 8Mb was allocated for the client buffer pool and 4Mb for the recovery buffer. This allocation of memory allowed all of the persistent data (modified and unmodified) accessed by a client to be cached completely in the client's main memory. In addition, since the small database was used, all of the data accessed by the clients could be cached in the server buffer pool as well.

Figures 6.6 and 6.7 show the response time and throughput versus the number of active clients for the T2A traversal[1] for several software versions. PD-REDO (page diffing using redo recovery) has the best performance overall, while WPL (whole-page logging) has the worst. WPL is just 22% slower than PD-REDO when one client is used, but its performance relative to the other systems steadily worsens as the number of clients increases. At five clients, WPL is 2.4 times slower than PD-REDO. WPL has slow performance in this experiment because T2A does very sparse updates, so WPL writes significantly more pages to the log than the other systems do.

Figure 6.8 shows the total number of pages (data and log) and the number of log record pages shipped from each client to the server on average during a transaction. The results in Figure 6.8 are labeled according to the underlying recovery scheme used, since that determined the number of pages sent for each system, e.g. PD-ESM and SD-ESM had the same write performance since when no paging occurs at the clients they always generate the

[1]T2A: update root atomic part of each composite part.

same number of log records and dirty pages. The main difference between PD-ESM and SD-ESM in this case is in the amount of data copied into the recovery buffer and diffed per transaction. The number of pages written to the log by the server was very close to the total number of pages shipped for WPL, and it was also close to the number of log pages shipped for the other systems. Figure 6.8 shows that WPL writes 435 pages back to the server on average during T2A, while PD-REDO writes just 5. Thus, the diffing scheme used by PD-REDO is very effective at reducing the amount work required at the server for recovery in this case.

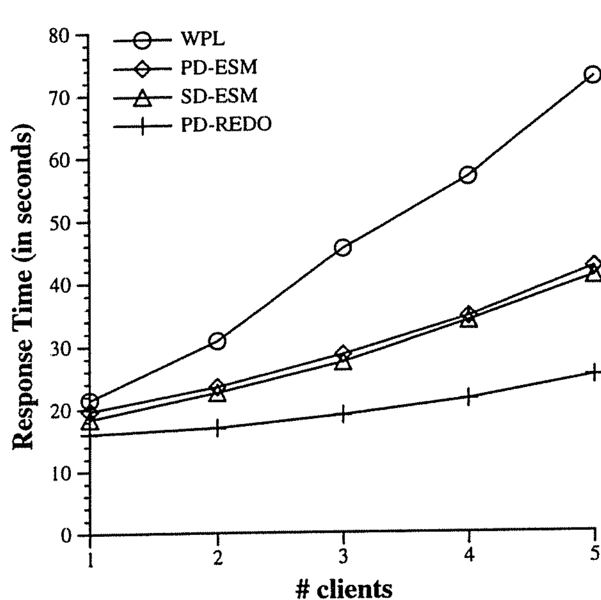

Figure 6.6. T2A, small database.

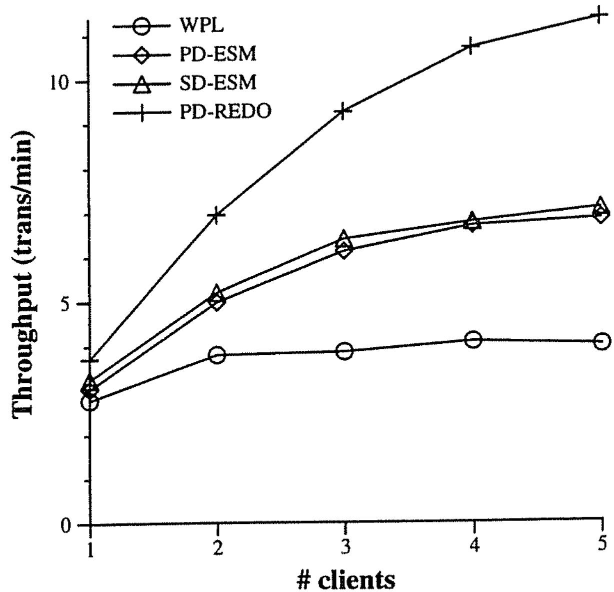

Figure 6.7. T2A, small database.

The performance of PD-ESM and SD-ESM lies between that of the other two systems in Figure 6.6. Surprisingly, the overall response time of SD-ESM is only slightly faster than PD-ESM (6.5% at 1 client, 3.3% at 5 clients), as the savings in CPU cost provided by SD-ESM were only a small part of overall response time in this experiment. The absolute difference in response time between SD-ESM and PD-ESM did not change significantly as the number of clients varied, and was roughly 1.3 seconds. This amounted to a savings of 3 milliseconds for SD-ESM for each page that was updated. The difference in CPU usage between PD-ESM and SD-ESM in Figure 6.6 was approximately 8% throughout, which was also quite low. We believe the small difference in CPU usage was caused partly by the CPU overhead at clients for shipping dirty pages back to the server. The response time for SL-ESM is not shown in Figure 6.6 since it was basically the same as SD-ESM. This was because the number of additional log pages generated by SL-ESM relative to SD-ESM was very small during this experiment.

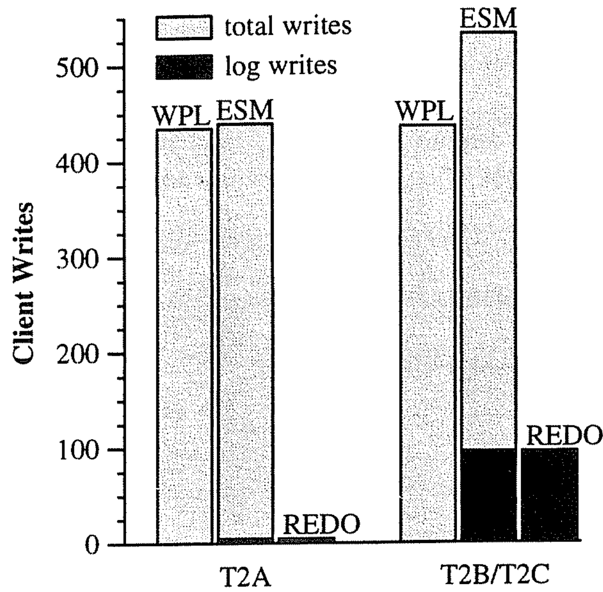

Figure 6.8. Client Writes, small.

The throughput results shown in Figure 6.7 mirror the response time results in Figure 6.6. Figure 6.7 shows that while the throughput increases with the number of clients for the systems that use diffing, WPL becomes saturated when more than two clients are used. The increase in throughput for PD-ESM is 56%, and for PD-REDO it is 67% as the number of clients increases from 1 to 5.

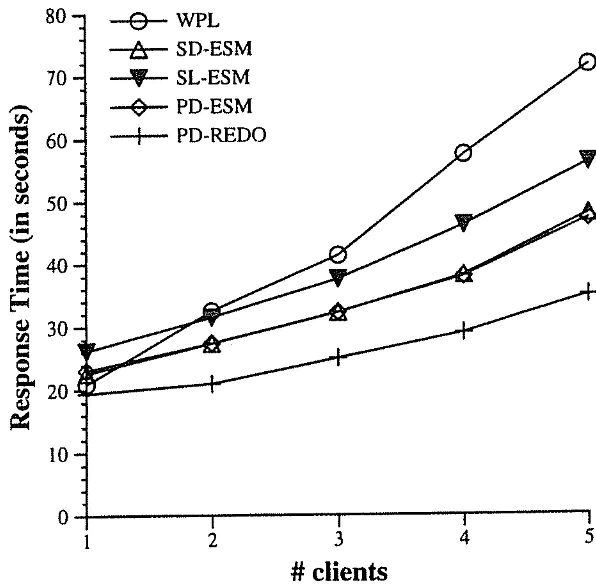

Figure 6.9. T2B, small database.

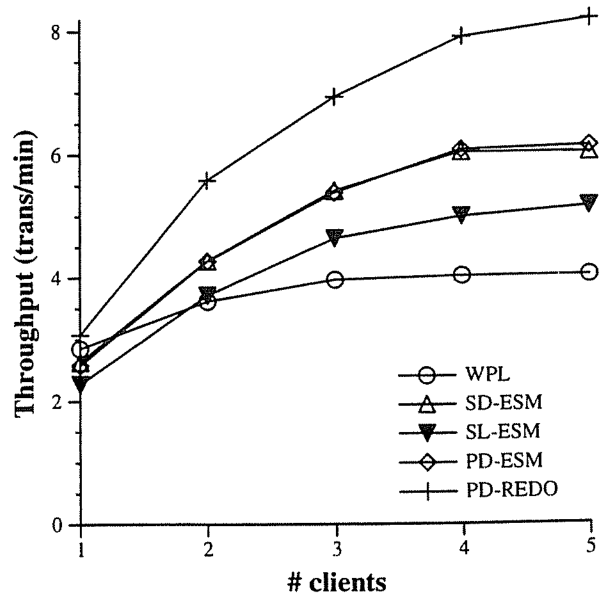

Figure 6.10. T2B, small database.

We turn next to Figures 6.9 and 6.10, which show the results of the T2B[2] traversal. Comparing Figure 6.9 with Figure 6.6 shows that the difference in performance between the systems is smaller during T2B, as T2B performs significantly more updates per page than T2A—slowing the performance of the diffing schemes. PD-REDO again has the best overall multi-user performance, however, the difference in performance between PD-REDO and WPL ranges from just 5% to 41% as the number of clients increases since PD-REDO must write a significant number of log records to disk per transaction (Figure 6.8). WPL is faster than the remaining systems when a single client is used, but its performance degrades more swiftly than the other systems since writing log records at the server is more of a bottleneck for WPL.

Interestingly, the performance of PD-ESM and SD-ESM is nearly identical during T2B due to the fact that the client CPU usage of the two systems was the same. The performance of SD-ESM is a bit worse relative to PD-ESM in T2B because T2B updates more objects on each page that is updated, causing SD-ESM to do more copy and diffing work. In addition, since more updates are performed, the cost of actually doing the updates is more of a factor for SD-ESM, since each update incurs the cost of a function call and other CPU overhead, as described in Section 6.2. The performance difference between SL-ESM and SD-ESM is approximately 14% in all cases in Figure 6.9, showing that it is indeed worthwhile here to diff the 64 byte blocks copied by the sub-page diffing scheme. Lastly, Figure 6.10 shows that the transaction throughput begins to level off after 4 clients for most of the systems, with WPL showing no increase in throughput beyond 3 clients.

Figure 6.11 shows the response time results for the T2C[3] traversal. The response time for PD-ESM, PD-REDO, and WPL did not change significantly relative to T2B because these systems allow applications to update objects by dereferencing normal virtual memory pointers. The performance of both SD-ESM and SL-ESM was 3.5 seconds slower, independent of the number of clients, due to the higher cost of performing updates in these systems—the overhead for performing the updates themselves is significant during T2C since a total of 1,049,760 additional updates are performed per transaction relative to T2B. Thus, the performance difference between SD-ESM and PD-ESM ranges from 12% down to 6% as the number of clients increases in Figure 6.11. The throughput results for T2C are not shown since they were also similar to those for T2B and can be deduced from the response time results of Figure 6.11. In addition, the number of pages (total pages and log record pages) sent from the client

[2]T2B: update all atomic parts of each composite part.

[3]T2C: update all atomic parts of each composite part four times.

to the server was the same as during T2B (Figure 6.8).

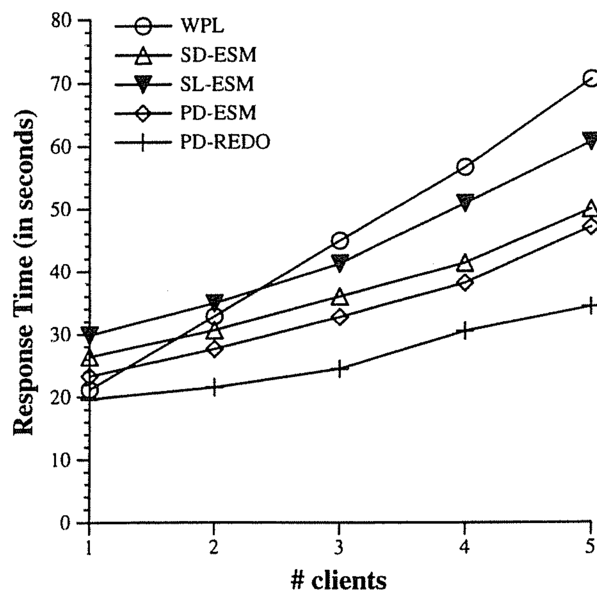

Figure 6.11. T2C, small database.

6.4.2. Constrained Cache Results

This section presents results obtained by running the various systems with a more restricted amount of memory at each client. As in the previous section, the small database is used, but each client is given only 8 megabytes of memory to use for caching persistent data here. For the systems that do diffing, 7.5Mb was allocated for the client buffer pool and .5Mb for the recovery buffer. This allocation of memory results in a client buffer pool that is large enough to avoid paging. So, for example, the performance of WPL here is the same as in the previous section. However, now there is insufficient space in the recovery buffer to hold all of the data required by the diffing schemes until commit time.

Figures 6.12 and 6.13 show the response time and throughput for traversal T2A in the constrained case. SD-ESM has the best performance in Figure 6.12. SD-ESM is 31% faster than PD-ESM and 40% faster than WPL at five clients. PD-ESM is slower than SD-ESM in this case because it experiences more contention in the recovery buffer. This causes PD-ESM to generate 4 times as many pages of log records on average per transaction as did SD-ESM (see Figure 6.16). WPL has competitive performance when the number of clients is low, but it has the worst performance when three or more clients are used. The performance of WPL degrades faster than the performance of the other systems, as before, because server performance is more of a limiting factor for WPL. On the

other hand, the diffing schemes benefit from their ability to perform more work at the clients, which allows them to scale better.

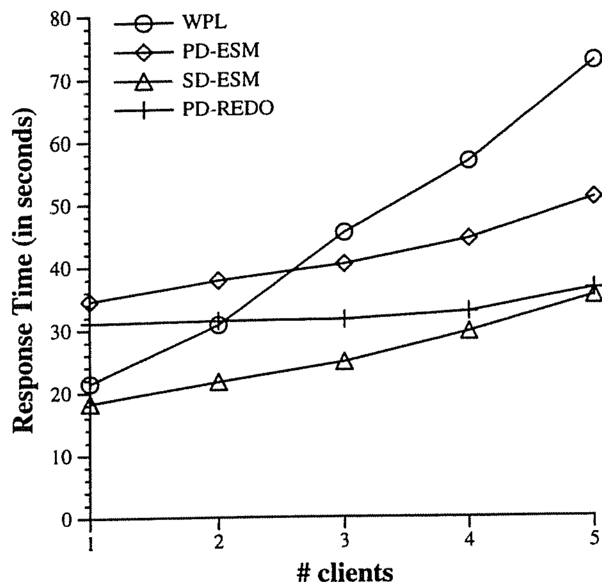

Figure 6.12. T2A, small, constrained cache.

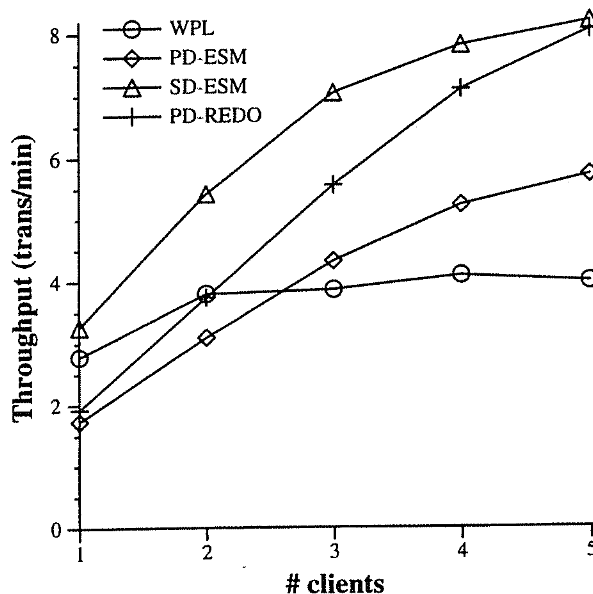

Figure 6.13. T2A, small, constrained cache.

The performance of PD-REDO appears to be approaching that of SD-ESM as the number of clients increases. PD-REDO scales better than the other systems here because it doesn't suffer from the overhead of shipping dirty pages back to the server. PD-REDO thus has the second best performance when two or more clients are used and is only 3% slower than SD-ESM at five clients. The throughput results shown in Figure 6.13 show that transaction throughput increases for all of the systems, except WPL (which becomes saturated), as the number of clients increases. SD-ESM and PD-REDO show the biggest increases in throughput. The throughput of SD-ESM increases by 2.5 times when the number of clients is increased, while the throughput for PD-REDO increases by a factor of 4.2.

We now turn to Figures 6.14 and 6.15, which show the response time and throughput, respectively, for traversal T2B. WPL has the best performance at one client, where it is 27% faster than SD-ESM (which is the next best performer). Beyond two clients, however, SD-ESM has the best performance, and at five clients, SD-ESM is 30% faster than WPL. SD-ESM scales better than WPL because it performs most of its recovery work at the clients, while WPL relies more heavily on the server. The page diffing systems, PD-ESM and PD-REDO, have the worst performance in Figure 6.14. The reason for this can be seen in Figure 6.16 which shows the number of writes per transaction for the systems. In Figure 6.16, PD-ESM produces 2.2 times as many log pages per transaction as does SD-ESM during T2B. In addition, the number of log pages produced by PD-ESM is approaching the number of

pages written to the log per transaction by WPL due to the high level of contention in the recovery buffer for PD-ESM.

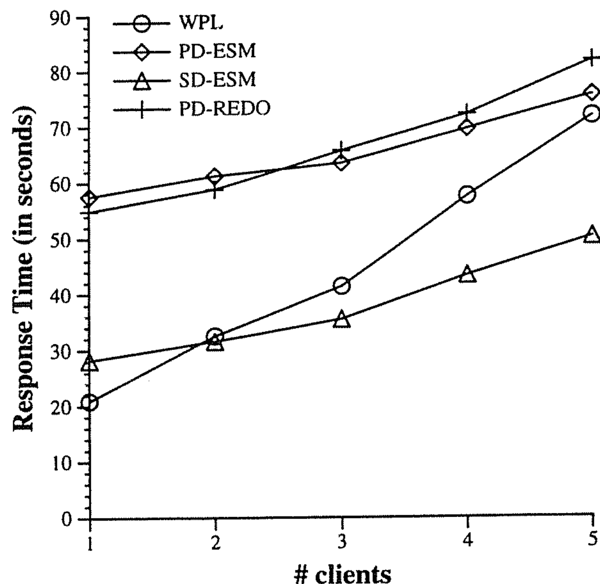

Figure 6.14. T2B, small, constrained cache.

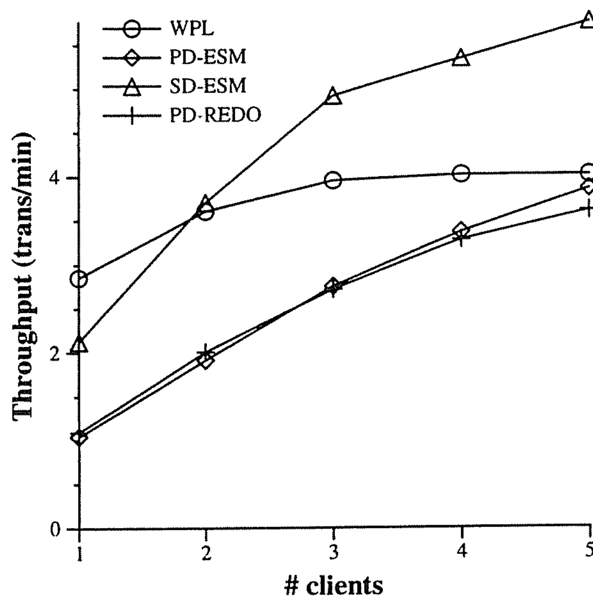

Figure 6.15. T2B, small, constrained cache.

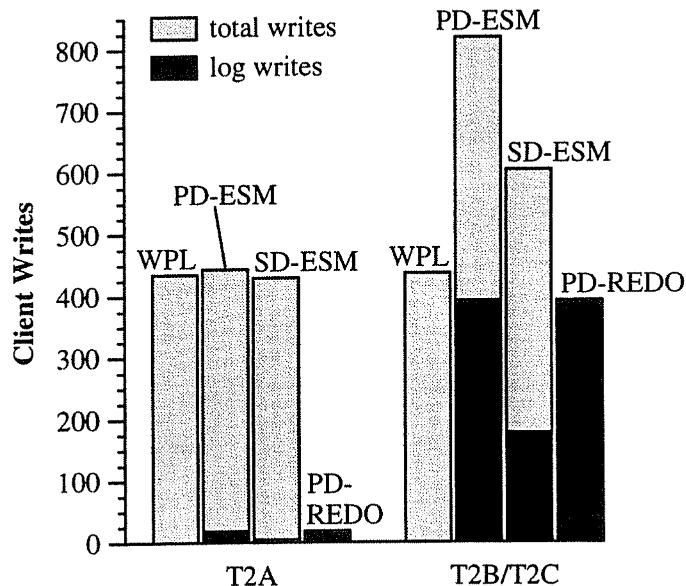

Figure 6.16. Client Writes, small, constrained cache.

PD-REDO is slightly faster than PD-ESM when the number of clients is low, but for more than three clients, PD-ESM has better performance. PD-REDO doesn't scale as well as PD-ESM in Figure 6.13 because of the CPU overhead of applying log records at the server for PD-REDO. The results for T2C are not shown for this experiment because they were similar to the results for T2B. The only difference in the times for T2C was that SD-ESM

was consistently slower by a few seconds relative to its times for T2B.

6.4.3. Big Database Results

This section contains the results of the experiments that were run using the big database. In these experiments, all of the systems were given 12Mb of memory at each client to use for caching persistent data. For the systems that do diffing, two alternative strategies for partitioning the memory between the client buffer pool and the recovery buffer were explored. Some of the systems were given 8Mb of memory to use as the client buffer pool, and the remaining 4Mb was used for the recovery buffer. Others were allocated 11.5Mb for the buffer pool and just .5Mb for the recovery buffer.

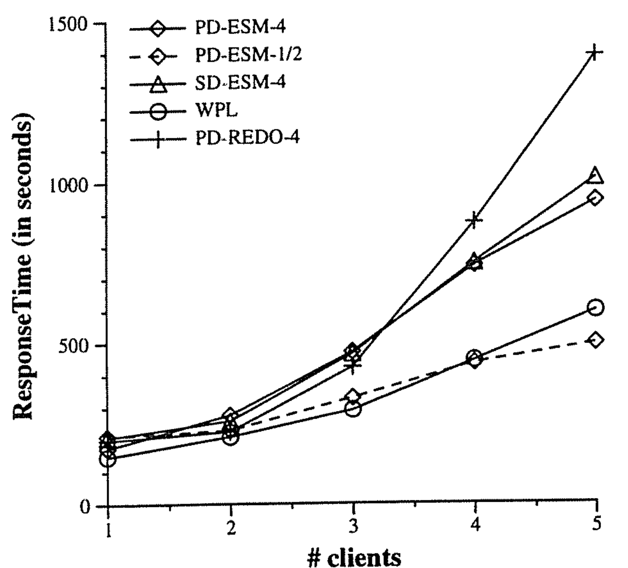

Figure 6.17. T2A, big database.

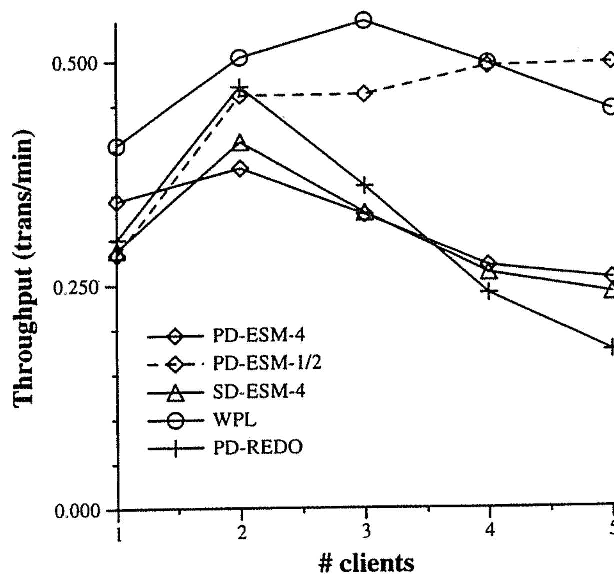

Figure 6.18. T2A, big database.

Figure 6.17 shows the average response time for each system when performing traversal T2A. WPL has the fastest response time when the number of clients is less than four; however, when four or more clients are used, PD-ESM-1/2 has the best performance. For example, PD-ESM-1/2 is 17% faster than WPL for five clients. Overall, WPL has fast performance in this experiment because it is able to devote all of available memory at the clients to the buffer pool; thereby, decreasing the amount of paging in the system and lessening the burden placed on the server. However, the server log disk becomes a bottleneck for WPL as the number of clients increases and eventually PD-ESM-1/2 performs better. As in previous experiments, the diffing scheme (PD-ESM-1/2) appears to scale better since it makes use of the clients' aggregate processing power to lessen the burden placed on the server's log disk and CPU.

Comparing the performance of PD-ESM-1/2 and PD-ESM-4 in Figure 6.17 shows the importance of choosing a good division of memory between the client buffer pool and the recovery buffer. PD-ESM-4 has better performance than PD-ESM-1/2 when one client is used, but for multiple clients, when paging between the client and the server becomes relatively more expensive, PD-ESM-1/2 is always faster. Indeed, as Figure 6.18 shows, the systems that were given smaller client buffers pool begin to thrash significantly when more than two clients are used, while the throughput for PD-ESM-1/2 continues to increase as the number of clients increases (although the increase is very small when more than two clients are used).

Interestingly, there is little difference in performance between PD-ESM-4 and SD-ESM-4, in Figure 6.17. This is because there is a significant amount of paging going on in the client buffer pool in Figure 6.17 (paging in the buffer pool is exactly the same in both systems), and each time a modified page is replaced in the client buffer pool the same number of log records are generated by both PD-ESM-4 and SD-ESM-4. Thus, sub-page diffing doesn't save nearly as much in terms of the number of log records generated in Figure 6.17 as it did in the small, constrained experiment (Figure 6.12). In fact, SD-ESM-4 only generates 10% fewer log records than does PD-ESM-4 in Figure 6.17, as opposed to generating 75% fewer log records in Figure 6.12. Finally, we note that although PD-REDO-4 is competitive with PD-ESM-4 when fewer than three clients are used; its performance grades more quickly than the other systems when the number of clients increases beyond three. This is because a larger number of pages must be reread from disk at the server by PD-REDO-4 as the number of clients increases, so that log records describing updates to the pages can be applied.

Comparing the results shown in Figure 6.17 to those of Figure 6.19, we see that the relative performance of the systems during T2B (dense updates) and T2A (sparse updates) is similar for the big database. WPL does somewhat better relative to PD-ESM-1/2 during T2B, as T2B updates a much larger number of objects per page, thus causing PD-ESM-1/2 to generate more log records. However, PD-ESM-1/2 scales better than the other systems (including WPL) and appears to surpass WPL in terms of performance when more than five clients are used. The performance of SD-ESM-4 and PD-ESM-4 is almost identical in Figure 6.19 (as it was in Figure 6.17). Again, this is because SD-ESM-4 generates almost as many log records as PD-ESM-4, and because the savings in client CPU cost provided by SD-ESM-4 for performing less diffing and copying work does not provide any noticeable benefit in terms of performance. The scalability of the REDO algorithm is again the worst overall in Figure 6.19 as it was in the sparse update case (Figure 6.17). When five clients are used PD-REDO-4 is 25% slower than PD-ESM-4

Figure 6.20 shows the throughput for the dense traversal run on the big database. Not surprisingly, the throughput results for the dense traversal are similar to those for the sparse traversal (Figure 6.18). Figure 6.20 highlights the fact that the systems that were given a larger client buffer pool (WPL, PD-ESM-1/2) scale better than the systems that were given smaller buffer pools (SD-ESM-4, PD-ESM-4, PD-REDO-4). This is because avoiding paging between the client and the server when using the big database is more important for achieving good performance than avoiding the generation of additional log records as is done by PD-ESM-1/2 and WPL. Even though WPL was given a large (12Mg) client buffer pool it begins to thrash for more than three clients, as the server becomes more of a performance bottleneck.

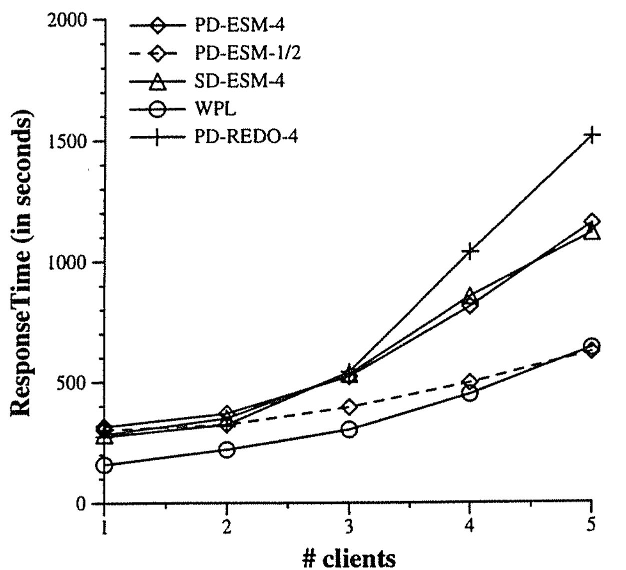

Figure 6.19. T2B, big database.

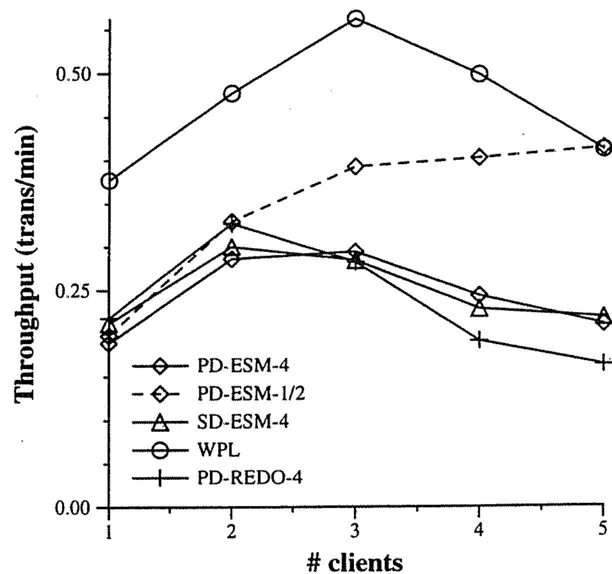

Figure 6.20. T2B, big database.

6.5. Related Work

This section discusses related work that has been done on the design and implementation of recovery algorithms for client-server database systems. We also compare the study presented here with other studies that have dealt with the issue of recovery performance.

[Frank92] describes the design and implementation of crash recovery in the EXODUS Storage Manager (ESM). In addition, [Frank92] discusses the issues that must be addressed when implementing recovery in a client-server environment. The recovery algorithm described in [Frank92] is based on ARIES [Mohan92], and supports write-ahead-logging and a STEAL/NO FORCE buffer management policy at the server. However, the ESM currently requires that all dirty pages be shipped from a client to the server before a transaction commits. This could be termed a force-to-server-at-commit policy. We note here that while ESM addresses the recovery issues raised by

a client server architecture, it does not address the issues discussed in Section 6.1, i.e. issues that are specific to object-oriented systems and memory-mapped stores.

More recently, [Mohan94] has presented an algorithm termed ARIES/CSA which also extends the basic ARIES recovery algorithm to the client-server environment. ARIES/CSA differs from ESM in that it supports fine-granularity locking which ESM currently does not. ARIES/CSA also supports unconditional undo. Another difference between the two approaches is that clients in ARIES/CSA take their own checkpoints in addition to checkpoints taken by the server, while in ESM checkpoints are only performed at the server. In principle the recovery techniques described in Section 6.2 that use ESM could also be used in conjunction with ARIES/CSA. However, it is not clear how features such as fine-granularity locking that are supported by ARIES/CSA would be used by a memory-mapped storage system such as QuickStore since the memory-mapped approach is inherently page-based.

A related study on recovery performance appears in [Hoski93]. The study presented in [Hoski93] differs from the study presented here, in that it is only concerned with alternative methods for detecting and recording the occurrence of updates. All of the schemes described in [Hoski93] use differencing to generate log records. We consider these issues as well, and also examine different strategies for processing log records in a client-server environment, i.e. ARIES-based schemes vs. whole-page logging. An interesting note about the study presented here is that our results differ substantially from the results presented in [Hoski93]. Some of the variation in the results is undoubtedly due to architectural differences between the systems examined in the two studies. For example, in [Hoski93] the transaction log is located at the client machine instead of at the server as in ESM. The advantage of locating the log at the server as ESM does is that it increases system availability, since if a client crashes, the server can continue processing the requests of other clients. Placing the log at the server impacts performance because log records must be sent from clients to the server (usually over a network) before they are written to disk. Thus, differences in alternative techniques for generating log records will be smaller in a system in which the log is located at the server than in a system where log records are written to disk at the client.

Other reasons for differences in the results have to do with implementation details. For example, both [Hoski93] and our study examine the tradeoffs involved in detecting the occurrence of updates in software versus using virtual memory hardware support. [Hoski93] uses a copy swizzling approach (see Chapters 2 and 3 for details concerning copy swizzling), that copies objects one-at-a-time into the object cache. The hardware-based recovery scheme used in [Hoski93] requires that the virtual memory page in the object cache that will hold an object be unprotected and then reprotected each time an object is copied into the page—producing a substantial amount of

overhead (up to 100%), even for read-only transactions. Under the approach used in QuickStore (in-place access, page-at-a-time swizzling) a page's protection is only manipulated once, when the first object on the page is updated. Thus, our results show that the hardware-based detection scheme does much better than do those presented in [Hoski93]. In particular, the scheme used in QuickStore does not impact the performance of read-only transactions.

[Hoski93] also examines several schemes (termed *card marking*) that are similar to the sub-page approach examined here. The results presented in [Hoski93] show that the size of the sub-page region used for recovery has a great impact on performance. The difference in performance reported in [Hoski93] is due to the fact that using a small sub-page unit for recovery can reduce diffing costs when updates are sparse. The results presented in this study, on the other hand, show that the size of the sub-page blocks is important, not because of savings in CPU costs when the block size is small, but because smaller block sizes can result in the generation of fewer log records when space in the recovery buffer is tight. [Hoski93] does not consider this case. We believe the differences in the results are due to the fact that we use an in-place swizzling scheme while [Hoski93] does copy swizzling, and to the cost for supporting concurrency control (the system examined in [Hoski93] is single user system that don't support concurrency). The study presented here also differs from [Hoski93] in that we examine the scalability of the various recovery schemes as the number of clients accessing the database increases. We also examine the performance of the different recovery techniques when a large database is used and a significant amount of paging (object replacement) is taking place in the system. [Hoski93] does not consider these issues.

Although several OODBMSs are commercially available, very little has been published concerning the recovery algorithms they use. The O_2 system [Deux91] also uses an ARIES-based approach to support recovery. O_2 differs from ESM and ARIES/CSA in that it uses shadowing to avoid undo. The most popular commercial OODBMS is ObjectStore [Lamb91] which like, QuickStore, uses a memory-mapping scheme to give application programs access to persistent data. As was mentioned in Chapter 3, ObjectStore uses a technique which we term whole-page logging to support recovery. The basic idea of this approach is that dirty pages are shipped from a client to the server and written to the log before a transaction is allowed to commit. This differs from the ARIES-based schemes mentioned above which only require that the log records generated by updates be written to disk at commit time. Finally, we note that the whole-page logging approach to recovery was first described in [Elhar84] which presents the design of the *database cache*.

6.6. Conclusions

This chapter has presented an in-depth comparison of the performance of several different approaches to implementing recovery in QuickStore, a memory-mapped storage manager based on a client-server, page-shipping architecture. Each of the recovery algorithms was designed to meet the unique performance requirements faced by QuickStore.

The results of the performance study show that using diffing to generate log records for updates at clients is generally superior in terms of performance to whole page logging. The diffing approach is better because it takes advantage of the aggregate CPU power available at the clients to lessen the overall burden placed on the server to support recovery. This provides much better scalability, as it prevents the server from becoming a performance bottleneck as quickly when the number of clients accessing the database increases.

The study also compared the performance of two different underlying recovery schemes upon which the diffing algorithms were based. These included the recovery algorithm used by the EXODUS Storage Manager and a simplified scheme termed redo-at-server (REDO). While the REDO approach provided significant performance benefits in some cases (when using a small database, while producing only a moderate number of log records per transaction), it failed to perform well when the database was bigger than the server buffer pool, and when the volume of log records sent to the server per transaction was high. The results presented in the study show that REDO can suffer from both disk and CPU bottlenecks at the server. System builders will have to decide whether the simplifications in system design and coding are worth the poor scalability of REDO in certain situations.

Finally, the study compared the performance of page-based and sub-page diffing techniques. Surprisingly, the sub-page diffing techniques provided very little advantage in terms of performance over the page-based approach. This is apparently due to the fact that diffing is a relatively inexpensive operation compared to the other costs involved in the system, such as network and disk access costs. The sub-page diffing approach did pay off in one situation, i.e. when the amount of memory that could be devoted to recovery was very low. System designers will have to decide whether this situation is likely to arise often enough in practice to justify using sub-page diffing. In addition, it was shown that sub-page diffing can have worse performance than page-diffing when updates are performed repeatedly. This fact must be weighed against the mild advantages of the technique when deciding on an implementation strategy. Finally, the results showed that diffing is even worthwhile when a sub-page granularity is used for recovery. Systems that used a sub-page granularity, but which didn't use diffing always had comparable or worse performance than the systems that used diffing.

CHAPTER 7

SUMMARY

This thesis has presented a detailed description of the design and implementation of several alternative pointer swizzling techniques for object-oriented database systems. The pointer swizzling techniques examined in the thesis included several software swizzling techniques that were implemented in the context of the E programming language and the memory-mapped pointer swizzling approach of QuickStore. Both QuickStore and E were implemented using the EXODUS Storage Manager to make direct comparisons meaningful.

The major contribution of the thesis is the detailed and accurate comparison that was made of the performance of the various swizzling techniques. The comparison of the hardware and software-based swizzling techniques was particularly interesting because they represent radically different approaches to implementing persistence. The performance results contained in the thesis compare the overall performance of the swizzling techniques over a range of traversal and query tests using the OO7 and OO1 database benchmarks. In addition, the thesis presents a detailed analysis of the overall performance when necessary, to better understand the underlying costs and benefits involved in pointer swizzling.

The thesis also investigates the issue of providing crash recovery in systems that provide full support for pointer swizzling, since the approach used to implement swizzling can significantly impact the opportunities that are available to perform recovery. Several recovery algorithms that are appropriate for client-server OODBMSs such as QuickStore are described. The recovery schemes described in the thesis include: page-diffing, sub-page diffing, whole-page-logging, and redo-at-server. All of these recovery schemes were implemented in QuickStore/EXODUS and compared; again using the OO7 benchmark, using several multi-user workloads.

In the future, we would like to explore variations of the object caching and page caching schemes studied in Chapter 3 in the context of EPVM 2.0 to see if an approach combining their relative strengths can be found. It would also be interesting to examine the issue of hardware vs. software-based pointer swizzling in the context of heterogeneous OODBMSs, i.e. OODBMSs where objects are stored using multiple different hardware platforms and where the same persistent objects are accessed by applications written in different programming languages. Finally, we would like to explore improvements to the recovery schemes based on diffing to see if the diffing

approach could be enhanced so that it adapts better to workload changes.

REFERENCES

[Atkin82] M. Atkinson, K. Chisholm, and P. Cockshott, "PS-Algol: an Algol with a Persistent Heap," *ACM SIGPLAN Notices*, Vol. 17, pp. 24-31, July 1982.

[Atkin83] M. Atkinson, K. Chisholm, and P. Cockshott, "Algorithms for a Persistent Heap," *Software Practice and Experience*, Vol. 13, No. 3, pp. 259-272, March 1983.

[Ball92] T. Ball, J. Larus, "Optimally Profiling and Tracing Programs", *POPL 1992*, pp. 59-70, January 1992.

[Butter91] P. Butterworth, A. Otis, J. Stein, "The Gemstone Object Database Management System", *Communications of the ACM*, Vol. 34, No. 10, October 1991

[Carey89] M. Carey et al., "Storage Management for Objects in EXODUS," in *Object-Oriented Concepts, Databases, and Applications*, W. Kim and F. Lochovsky, eds., Addison-Wesley, 1989.

[Carey93] M. Carey, D. DeWitt, J. Naughton, "The OO7 Benchmark", *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Washington, DC, May 1993.

[Carey94] M. Carey, D. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, S. White, M. Zwilling, "Shoring Up Persistent Applications", *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Minneapolis, MN, May 1994.

[Catte91] R. Cattell, "An Engineering Database Benchmark," in *The Benchmark Handbook For Database and Transaction Processing Systems*, Jim Gray ed., Morgan-Kaufman, 1991.

[Cock84] P. Cockshott et al., "Persistent Object Management System," *Software Practice and Experience*, Vol. 14, pp. 49-71, 1984

[Codd70] E. F. Codd, "A Relational Model for Large Shared Data Banks", *Communications of the ACM*, Volume 13, Number 6, (June 1970), pages 377-387.

[DeWitt90] D. DeWitt, P. Futtersack, D. Maier, F. Velez, "A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems", *Proceedings of the 16th International Conferece on Very Large Data Bases*, Brisbane, Australia, August, 1990.

[Deux91] O. Deux et al., "The O2 System", *Communications of the ACM*, Vol. 34, No. 10, October 1991

[Elhar84] K. Elhardt, R. Bayer, "A Database Cache for High Performance and Fast Restart in Database Systems", *ACM Transactions on Database Systems*, 9(4):503-525, December 1984.

[Exodu93] "Using the EXODUS Storage Manager V3.0", technical documentation, Department of Computer Sciences, University of Wisconsin-Madison, April 1993.

[Frank92] M. Franklin et al., "Crash Recovery in Client-Server EXODUS", *Proceedings of the ACM SIGMOD International Conference on Management of Data*, California, 1992.

[Frank93] M. Franklin, "Caching and Memory Management in Client-Server Database Systems", Ph.D. thesis, University of Wisconsin-Madison, technical report #1168, July 1993.

[Gray78] J. N. Gray, "Notes on Data Base Operating Systems", in *Operating Systems: An Advanced Course*, R. Bayer, R. M. Graham, G. Seegmuller (Editors), Springer-Verlag, Berlin, Germany, pages 393-481, 1978.

[Haerd83] T. Haerder, A. Reuter, "Principles of Transaction Oriented Database Recovery - A Taxonomy", *Computing Surveys*, Vol. 6, No. 1, February 1988.

[Hoski93a] A. Hosking, J. E. B. Moss, "Object Fault Handling for Persistent Programming Languages: A Performance Evaluation", *Proceedings of the ACM Conference on Object-Oriented Programming Systems and Languages (OOPSLA)*, pp. 288-303, 1993. Proceedings of the 16th International Conferece on Very Large Data Bases

[Hoski93b] A. Hosking, E. Brown, J. Moss, "Update Logging in Persistent Programming Languages: A Comparative Performance Evaluation", *Proceedings of the 19th International Conferece on Very Large Data Bases*, Dublin, Ireland, 1993.

[Jagad94] H. Jagadish, D. Lieuwen, R. Rastogi, A. Silberschatz, "Dali: A High Performance Main Memory Storage Manager", to appear in *Proceedings of the 20th International Conferece on Very Large Data Bases*, Santiago, Chile, September 12-15, 1994.

[Khosh86] S. N. Khoshafian, G. P. Copeland, "Object Identity", *Proceedings of the ACM Conference on Object-Oriented Programming Systems and Languages (OOPSLA)*, pages 408-416, November 1986.

[Kemper93] A. Kemper, D. Kossmann, "Adaptable Pointer Swizzling Strategies in Object Bases", *Proceedings of the International Conference on Data Engineering*, pages 155-162, Vienna, Austria, April 1993.

[Lamb91] C. Lamb, G. Landis, J. Orenstein, D. Weinreb, "The ObjectStore Database System", *Communications of the ACM*, Vol. 34, No. 10, October 1991

[McAul94] M. McAuliffe, M. Solomon, "A Trace-Based Simulation of Pointer Swizzling Techniques", to appear in *Proceedings of the International Conference on Data Engineering*, Taipei, Taiwan, March 6-10, 1995.

[Mohan92] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, P. Schwartz, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging", *ACM Transactions on Database Systems*, Vol. 17, No. 1, March 1992.

[Mohan94] C. Mohan, I. Narang, "ARIES/CSA: A Method for Database Recovery in Client-Server Architectures", *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Minneapolis, MN, 1994.

[Moss92] J. Eliot B. Moss, "Working with Persistent Objects: To Swizzle or Not to Swizzle", *IEEE Transactions on Software Engineering*, 18(8):657-673, August 1992.

[Objec90] Object Design, Inc., "ObjectStore User Guide", Release 1.0, October 1990.

[Objy92] Objectivity, Inc., "Objectivity Reference Manual". 1992.

[Ontos92] Ontos, Inc. "Ontos Reference Manual", 1992.

[OO7] UW OO7 Benchmarking Team, *personal communication*, August 1994.

[Orens92] J. Orenstein, personal communication, May 1992.

[Rich93] J. Richardson, M. Carey, and D. Schuh, "The Design of the E Programming Language", *ACM Transactions on Programming Languages and Systems*, Vol. 15, No. 3, July 1993.

[Schuh90] D. Schuh, M. Carey, and D. Dewitt, Persistence in E Revisited---Implementation Experiences, in *Implementing Persistent Object Bases Principles and Practice, Proceedings of the Fourth International Workshop on Persistent Object Systems*, Martha's Vineyard, MA, Sept. 1990.

[Shek90] E. Shekita and M. Zwilling, "Cricket: A Mapped Persistent Object Store", *Proceedings of the Fourth International Wor kshop on Persistent Object Systems*, Martha's Vineyard, MA, Sept. 1990.

[Ultrix90] Ultrix 4.0 General Information (commands), Digital Equipment Corporation, vol. 3b, June 1990.

[Versan92] Versant, Inc., "Versant Reference Manual", 1992.

[Wilso90] Paul R. Wilson, "Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Spaces on Standard Hardware", Technical Report UIC-EECS-90-6, University of Illinois at Chicago, December 1990.

[Zdon90] S. Zdonick, D. Maier, "Fundamentals of Object-Oriented Databases", in *Readings in Object-Oriented Database Systems*, Zdonick and Maier, eds., Morgan Kaufmann, 1990.