Virtual Memory Alternatives for Transaction Buffer Management in a Single-level Store

by

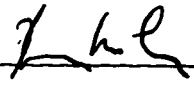
Dylan James McNamee

A dissertation submitted in partial fulfillment
of the requirements for the degree of

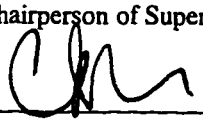
Doctor of Philosophy

University of Washington

1996

Approved by 

Chairperson of Supervisory Committee



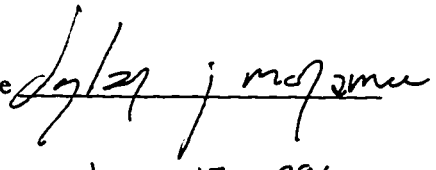
Chairperson of Supervisory Committee

Program Authorized
to Offer Degree Department of Computer Science & Engineering

Date December 17, 1996

Doctoral Dissertation

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to University Microfilms, 1490 Eisenhower Place, P.O. Box 975, Ann Arbor, MI 48106, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature 

Date December 17, 1996

University of Washington

Abstract

Virtual Memory Alternatives for Transaction Buffer Management in a Single-Level Store

by Dylan James McNamee

Chairpersons of the Supervisory Committee: Professor Edward D. Lazowska
Professor Henry M. Levy
Department of Computer
Science and Engineering

This dissertation addresses a key issue in the efficient execution of transaction systems in shared environments. Until recently, transaction systems typically ran on dedicated database server machines. Transaction systems are increasingly popular beyond the database domain, and thus are often run outside of the dedicated server environment: on client machines, personal computers on user desktops, or multi-use enterprise servers. The key difference between these environments is that server machines are dedicated to running one application (e.g., a database server) in isolation, while in the other environments the machine is shared by multiple applications (e.g., word processors, email programs, web browsers, and applications that use transactions). In these latter environments, the transaction system needs to cooperate with the operating system to effectively share the resources of the machine with all of the applications on the machine.

Virtual memory is the operating system facility that manages the physical memory of a machine to buffer (i.e., temporarily store) the data of the applications running on it. Similarly, a transaction system's buffer manager temporarily stores portions its data in memory, where it can be efficiently manipulated. In both the server and shared environments, transaction systems are run as applications on top of an underlying operating system. As a result, transaction systems' buffers are themselves buffered by the virtual memory system. In dedicated server environments the virtual memory system is not active because there are no other applications to compete with the transaction system for memory. On shared machines, however, competition from other applications may cause the operating system to remove memory used to store a transaction system's buffer. In current designs, virtual memory and transaction system buffer managers are oblivious to each other. Thus, when virtual memory paging occurs, the transaction buffer manager incurs extra disk reads and writes—without its knowledge—as it attempts to access its buffer memory. This phenomenon is called *double paging*.

Previous operating systems research has addressed the problem of double paging by providing transaction functionality within the operating system. This approach solves the problem by integrating transaction buffer management with virtual memory. It allows transaction systems to achieve excellent performance while allowing memory to be effectively utilized by all of the applications on the machine. Taking advantage of these results has not been feasible for commercial application designers, however, because these solutions require features of research operating systems that are not available in commercial operating systems. Further, this platform mismatch has made it difficult to project the performance results of these systems to commercial operating systems.

In this dissertation, we first describe a space of trade-offs between transaction performance in a multiprogrammed environment and the required degree of customized operating system support. We argue that the current alternatives represent the two possible extremes: poor performance with no additional support required, and excellent performance with extensive operating system support required.

Second, we present a new point in the space of trade-offs that is between these extremes. This technique improves performance over current practice by utilizing existing operating system facilities to integrate transaction buffer management with operating system virtual memory management.

Third, we describe an implementation of the new approach in a commercial operating system, Digital Unix, running on AlphaStation 250 workstations. We also present two more systems, built on the same platform, that are representative of the two existing alternatives. The implementation of the three systems on an identical platform enables, for the first time, a fair evaluation of the full spectrum of degrees of virtual memory support for transaction buffer management.

We conclude that current transaction buffer management designs perform poorly in the increasingly common memory-competitive environments. We argue that this poor performance is the result of fundamental structural decisions. We lend evidence to this argument by demonstrating significant performance improvements in a restructured transaction buffer management system that uses existing operating systems facilities. Finally, we demonstrate prototype implementations of transaction support that will be available in future extensible operating systems that provide further performance improvements.

Table of Contents

List of Tables

List of Figures

Acknowledgments

Completing a Ph.D. dissertation has been a daunting task. I could not have done it without the support provided by my advisors, Ed Lazowska and Hank Levy. Both of them have helped me to "raise the bar" of my expectations of myself. In spite of his chairmanship, Ed has given generously of his time, and continues to amaze me with the accuracy of his comments, no matter how busy he is. Hank's ability to see through complex problems (and poorly worded explanations) to the real problem at hand has helped me to improve my reasoning and to avoid many trouble-spots. I thank both of them for being such superb mentors. The other members of my committee, Brian Bershad and David Notkin, have also been very helpful.

The faculty and staff of the Computer Science and Engineering department at the University of Washington have assembled a near-optimal environment for graduate studies in our field. The camaraderie, the faculty's truly "open-door" policy, the excellent facilities and infrastructure, and the emphasis on quality research and presentations have all combined to create a positive and supportive culture. The aspect of this culture I have appreciated most is that the faculty treat students as colleagues and peers at all times. The department staff have all helped me throughout my graduate career. Nancy Johnson Burr, Jan Sanislo, David Becker and (of course) Frankye Jones have all saved me by pulling a rabbit out of a hat at an opportune time.

My fellow students have, in addition to being great friends, given me great amounts of help throughout my graduate career. In particular, Anthony LaMarca, Raj Vaswani, Neal Lesh, Ted Romer, Wayne Wong, Stefan Savage, Przemek Pardyak, Mark Fiuczynski, Terri Watson, Ed Felten, Mike Feeley, Ashutosh Tiwary, and Vivek Narasayya all carefully critiqued various papers and talks I have given, and helped me refine the research ideas that eventually turned into this dissertation. Jeff Chase, Mike Feeley, Vivek Narasayya and Ashutosh Tiwary all have key roles within the Opal project, and conversations with them greatly influenced the research in this dissertation. In addition, Ashutosh helped refine the ideas, and Vivek helped implement and evaluate the systems.

Our department is fortunate to have colleagues from industry who generously share their interests and expertise. I greatly value discussions I have had with Gregor Kiczales from Xerox PARC, Phil Bernstein from Microsoft, and John Wilkes from Hewlett Packard. My research was generously supported in part by a DARPA graduate fellowship.

Finally, I would like to thank my family, who have been amazingly patient as I seemed to spend a lifetime in school, realizing the importance of my goal. My mother's soaring heart gave me the strength I needed more than once in the past thirty years. The love of my father, Charles, Sandy, Aaron, Paula, Jennifer, Anne, Rebecca, Jan, Denny, Lauren, Lisa, Patrick, my grandparents, aunts, uncles and cousins (no kidding—each and every one) truly powered me through some low-energy times, and buoyed me higher when things went well. And last but most, thanks to Heidi for the twinkle in her eyes that keeps mine twinkling too.

Colophon

This dissertation was prepared using Frame software (from Adobe Corporation) running on a Macintosh Duo (both docked and in the Allegro Café). Microsoft Excel generated the graphs, and Tailor (from EnFocus Software) was used to prepare the graphs for inclusion in Frame. The text was set in Adobe Times Roman, a descendant of Monotype's Times New Roman. Times New Roman is well suited for electronic imaging because its flat stems, serifs and distinctive letterforms render text clearly at low resolutions. Until recently it was accepted that Times New Roman was first drawn in 1931 by Victor Lardent under Stanley Morison's direction at THE TIMES of London. The Monotype Corporation, Ltd. then cut the design for THE TIMES' exclusive use. Recent investigation by Mike Parker [Parker 94] reveals that the roman existed at Lanston Monotype in Philadelphia early in this century. Parker believes that circa 1904, W. Starling Burgess—well known naval architect, aircraft designer and creator (for Buckminster Fuller) of the Dymaxion automobile—commissioned Lanston to cut a custom typeface of his own design for a private project, but by 1918 found himself unable to pay for the work. In 1921 Lanston tried to interest the nascent TIME magazine in the orphan typeface, without success; the design eventually found its way to Lanston's sister company in London, where it was adapted for use in Morison's redesign of THE TIMES. We may better understand Times New Roman's continuing vitality after half a century of numbing popularity if it is indeed the work of a young genius for a personal project, rather than a contracted work intended for a newspaper.

Chapter 1

Introduction

This dissertation investigates virtual memory management alternatives for supporting transactions in single-level storage systems. *Transactions* are a programming construct that allows programmers to achieve *atomic* (i.e., "all or nothing") updates to data, even in the presence of failures. *Single-level stores* provide a uniform interface to data, allowing the same code to operate on data regardless of whether the data is in memory, on disk, or on a remote machine. Individually, transactions and single-level storage systems have proven to be important components of reliable, efficient and easy to build software systems. The *combination* of these facilities, provided by intermediate layers of software built on top of an underlying operating system, are an increasingly popular foundation for writing applications. However, this combination of features has conflicting implementation demands.

This dissertation examines two aspects of implementations of transactions in single-level stores. First, we examine the interactions between such intermediate software layers and the operating system. Second, we measure the effect that these interactions have on application performance, particularly in a multiprogrammed environment.

In this chapter we describe the structure of existing transaction and single-level storage system implementations, and explain why combining the two individual implementations can result in poor application performance. The goal of our research is to restructure these

systems so that they are compatible with existing operating systems and provide improved performance.

1.1 Transactions and Single-level Stores

The interaction between transaction systems (in particular, relational databases) and operating system virtual memory has been carefully examined by a number of researchers [Stonebraker 81, Traiger 82, Diel et al. 84, Chang & Mergen 88], who concluded that commercial operating systems are insufficient to support the transaction memory management needs of database systems. As a result, commercial database systems are typically structured to bypass the operating system's virtual memory facilities; instead, such systems implement the memory management required for transactions in privately managed *buffer pools*. This technique has been proven to be effective for relational databases running on dedicated server machines.

In separate research, single-level stores have evolved into broad ranging systems that efficiently support local and distributed computation. One use of single-level stores has been to provide infrastructure for building groupware-style applications that allow interactive collaboration among collections of users. A number of operating systems have been designed and prototyped that implement such single-level storage environments [Chase et al. 94, Rosenberg et al. 96]. In contrast to database system structures, these systems tightly integrate the management of the single-level store with virtual memory, because they extend the virtual memory service to include persistent and distributed data.

Transactions and single-level stores offer important complementary advantages: reliability benefits in the case of transactions, and increased ease of programming in the case of single-level stores. It is not surprising, therefore, that we have recently seen efforts to combine these two functions into a single infrastructure for application development, by means of application frameworks, object-oriented databases, and other application middleware. Systems that provide both functions must resolve the conflict between previous transaction systems' technique of bypassing virtual memory and previous single-level storage systems' technique of extending virtual memory. Existing systems supporting

transactions in a single-level store resolve this conflict by bypassing virtual memory in order to implement transaction buffer management. Therefore, existing systems must re-implement the single-level store on top of the bypassed virtual memory.

1.2 Motivation

Our research is motivated by the observation that the buffer management techniques used by systems providing transactions in single-level stores assume that the application is run on a dedicated server machine. The popularity of such systems as application-building infrastructure has meant that transaction systems are now often run outside of the dedicated server environment: on desktop machines, and in multi-use enterprise servers. The key difference between the dedicated server and the other environments is that server machines are dedicated to running one application (e.g., a database server) in isolation, while in the other environments the transaction system must compete for resources with other applications (e.g., word processors, e-mail programs, web browsers, and other applications that also use transactions). In competitive environments, the transaction system needs to cooperate with the operating system to effectively share the resources of the machine with all of the applications on the machine, or a significant performance penalty will be imposed.

In both the server and competitive environments, transaction systems are run as applications on top of an underlying operating system. As a result, the memory used by a transaction system to *buffer* (i.e., temporarily store) the transaction's data is itself buffered by the virtual memory system. When a machine is dedicated to running one application, the virtual memory system never has to *page* (i.e., remove) memory from that application. On shared machines, however, competition from multiple applications may cause the operating system to page out memory used to store a transaction system's buffer. Current operating systems force virtual memory and transaction system buffer managers to be oblivious to each other. Because of this lack of integration, when virtual memory paging occurs, the transaction buffer manager incurs extra disk reads and writes—without its knowledge—as

it attempts to access its buffer memory. This phenomenon is called *double paging* [Goldberg & Hassinger 74].

In contrast, a system that *integrates* transaction buffer management with virtual memory will not exhibit double paging because the virtual memory system cooperates with, or even implements, transaction buffer management. Figure 1.1 show the effect of increasing competition for memory on a visualization application implemented on top of two different single-level stores: one that is integrated with virtual memory, and another that uses a buffer pool technique to bypass virtual memory. The integrated solution is insensitive to memory competition—the application continues to perform well. The non-integrated buffer pool solution performs well at low levels of memory competition, which can be thought of as running on a dedicated server machine. However, as the percentage of memory consumed by competition increases, causing double paging to take place, performance degrades sharply, ultimately becoming worse by a full order of magnitude.

The performance degradation caused by double paging is so severe that, in practice, applications that use transactions in single-level stores must be extensively tuned (i.e., have an experienced analyst adjust system parameters) for each new installation, thus

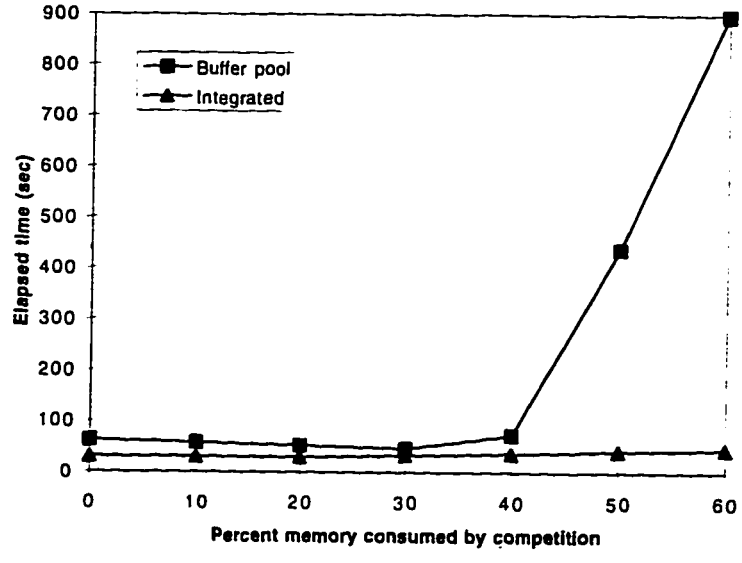


Figure 1.1: Elapsed time for visualization application to render one frame, as competition for memory increases.

greatly increasing their cost. Even when appropriately tuned, these applications can suffer from double paging unless the workload on the machine is constrained to well below its capacity, thus reducing the flexibility of the machines they are run on.

Previous research has addressed the problem of double paging by providing direct support for transactions within the operating system. Having the operating system provide such support solves the problem by integrating transaction buffer management with virtual memory. Taking advantage of these results has not been feasible for commercial application designers, however, because these solutions require features of research operating systems that are not available in commercial operating systems. Further, this platform mismatch has made it difficult to project the performance results of these systems to commercial environments.

This dissertation investigates alternate degrees of operating system support for transactions that provide the performance of an integrated single-level store. One alternative we examine is an approach that supports transactions in a single-level store without requiring the significant operating system redesign required by previous research solutions. The design, implementation and evaluation of this approach demonstrates that the growing number of applications using transactions in single-level stores will be able to achieve high performance without requiring manual tuning for each installation, and without requiring their users to significantly constrain the workload of the machines shared by these applications.

1.3 Thesis Statement and Contributions

My thesis is that the structure of current systems providing transactions in single-level stores is overly complex and inherently inefficient in memory-competitive environments. The complexity is a result of redundant memory management between the transaction system and the operating system. The lack of communication between the redundant layers results in poor performance in memory-competitive environments. This dissertation pre-

sents a range of alternatives for eliminating this redundancy and compares the performance of each alternative.

The primary contributions of this dissertation are:

- An analysis of the interactions between the memory management requirements of transactions and single-level stores.
- An analysis of the possible degrees of operating system support for systems that provide transactions in a single-level storage environment.
- A complete description of a transaction recovery mechanism that is integrated with virtual memory. This mechanism represents the application of general principles of transaction buffer management and operating system integration that can be implemented in most commercial operating system environments.
- An implementation of three approaches for operating system support of transaction recovery based on a common operating system infrastructure, which enables a fair comparison between them.
- A performance comparison of a representative of the current practice of transaction buffer management with the two approaches that eliminate buffer management redundancy: the first alternative requires custom operating systems, the second is our approach which is compatible with commercial operating systems.
- A demonstration that integrating transaction buffer management with the underlying virtual memory system results in significantly improved performance across a variety of workloads and degrees of memory competition.

1.4 Dissertation Outline

Chapter 2 describes how the memory management facilities provided by traditional operating systems have directly affected the structure of applications with specific memory

management requirements. It also describes how the structure of operating systems has evolved to accommodate these applications' requirements. Finally, it argues that these applications must be restructured to fully take advantage of the benefits of the new operating systems' structure.

Chapter 3 describes the tasks of applications that use transactions. Next, it describes the potential modes of interaction between transaction buffer management, logging, and the underlying operating system's virtual memory. Finally, it summarizes the log management, buffer policy and virtual memory integration decisions of a number of existing research and commercial database systems.

Chapter 4 extends the discussion in Chapter 3 by investigating the additional issues involved with implementing transactions in a single-level storage environment. Chapter 4 first discusses the attributes of single-level stores. Next it describes why many of these attributes impede the implementation of transactions using commercial operating system facilities. These difficulties have motivated current system structures that implement transactions by implementing buffer management in a privately managed *buffer pool*, thus bypassing—rather than integrating with—virtual memory management.

Chapter 5 describes a new buffer management mechanism, called user-level mmap shadow (UMS), that supports transactions in single-level stores. The buffer management provided by UMS is integrated with virtual memory *and* can be implemented without requiring modifications to existing commercial operating systems. Chapter 5 also describes the Recoverable Memory System (RMS), the transaction interface used by the applications in the experiments performed in the following chapter.

Chapter 6 first describes the implementations of two previous approaches for transaction buffer management: *buffer pool* systems that bypass virtual memory, and *kernel-based* approaches that rely on customized operating systems to integrate their buffer management with the virtual memory system. We compare the performance of the previous approaches to the performance of UMS using a variety of workloads and find that UMS is able to provide the performance advantages of the kernel-based approaches in memory-competitive environments, without relying on custom operating system support. Thus we find that UMS combines the benefits of both kernel-based and buffer pool approaches.

Finally, Chapter 7 summarizes the contributions of the dissertation, discusses some related work, and describes some future avenues of investigation in this area.

Chapter 2

Operating Systems Influence on Application Structure

The size and complexity of software systems has kept pace with the exponential improvements in hardware performance. Software systems have become sufficiently complex to cause researchers to revisit some of the basic principles of design [Parnas 72]. In particular, the principles of encapsulation and information-hiding are adapting to accommodate complex modular systems' need for flexibility and performance. The challenge has always been to design interfaces that hide unnecessary implementation details by abstraction, and expose those details that matter to clients of the interface. The dilemma that prompted researchers to revisit the issue is that, for a widely used interface, almost all implementation details are important to *some* client, thus no fixed set of abstractions is appropriate for all clients. To address this dilemma, programming languages [Shaw & Wulf 80, Kiczales 92] and operating systems [Bershad et al. 95, Engler et al. 94, Mogul et al. 87, Yokote et al. 89] have adopted new structures that offer client applications customizable "open implementations" of the primary interfaces provided by these systems.

An *open implementation* of an interface allows its clients to tailor its implementation via a separate controlling interface. Operating systems are an excellent example of the evolution of open implementations. Most operating system services — such as files, processes, virtual memory, communication — were originally provided via opaque interfaces that hide all implementation details from their clients. As applications evolved that had unique or high-performance service requirements, operating systems developed open implementations of individual services by providing additional interfaces that allow applications to provide advisory information or low-level control over resource management (e.g., `madvise` and `ioctl`, which control virtual memory and input/output devices, respectively). Recent operating systems research has moved toward generalizing this trend by developing uniform mechanisms to open the implementation of all operating system services. We call systems with open implementations of services *extensible operating systems.*

One benefit of extensible operating systems is that applications can achieve higher performance when they are able to customize the implementation of a service to their needs. Another benefit of extensible operating systems is that applications can take advantage of an open implementation of services to rid themselves of redundant resource management code, by customizing the resource management provided by the underlying system instead of replacing it. This structure centralizes the operating system's resource allocation, and at the same time allows applications to control the resource management policies. Allowing the operating system to control low-level allocation and mechanism decisions while providing applications the ability to manipulate high-level policy decisions achieves the goal of policy/mechanism separation set out in the Hydra system [Levin et al. 75].

The promise of extensible operating systems is great. Applications and operating systems can be better structured and more efficient. However, both the operating system and many of its client applications have to be redesigned in order for the full benefits of extensible operating systems to be realized. Many researchers are currently working on designing extensible operating systems (including SPIN [Bershad et al. 95], Aegis [Engler et al. 94], the Cache Kernel [Cheriton & Duda 94], Vino [Small & Seltzer 94], and Oberon [Mossenbock 94]). Much less research has been devoted to the influence on application

structure exerted by increasing degrees of system support and extensibility. This dissertation addresses a facet of this latter issue by examining various interactions between operating system memory management and the memory management needs of transaction systems.

In this introductory chapter we examine the interaction between operating system support and application structure. We first discuss the kinds of application demands that have motivated open implementations. Next we discuss the specific demands of different applications, and the varying degrees of extensibility provided by operating systems to meet these demands. Then we discuss operating system alternatives for providing application-specific memory management, and the effect these alternatives have on application structure and performance. Finally, we provide an overview of the remainder of the dissertation in terms of the issues described in this chapter.

2.1 Modular Systems and Open Implementations

Open implementations are motivated by an "impedance mismatch" between the client of a service and its implementation [Stonebraker 81, Anderson et al. 92, Kiczales 92]. This describes situations in which an implementation of an interface, or the interface itself, does not meet the application's needs, either in performance or functionality. For example, applications may have an impedance mismatch with their language system's memory allocator. Some applications want to specify the alignment of allocated memory because they want to control collisions in the processor's cache [LaMarca 96]. Other applications find that their pattern of memory allocation and deallocation causes excessive fragmentation in a particular memory allocator. Most memory allocators do not provide the control required by such applications. Applications with specific memory allocation needs therefore often request large blocks of memory from the language system's memory allocator, and sub-allocate "manually" according to their needs. Thus, the lack of an open implementation of memory allocation can cause client applications to reimplement customized memory allocation on top of the language system-provided memory allocation.

Without access to an open implementation of a service, application writers often need to reimplement significant portions of the service in order to provide a variant of the service themselves. The open implementation solution is to have the memory allocator provide an auxiliary *meta interface* that allows applications to control the implementation of the service's *primary interface*. For example, a meta interface could allow programmers to control the alignment of allocated memory without having to reimplement a memory allocator themselves. Another meta interface could control the allocation policy to reduce fragmentation for various patterns of allocation and deallocation. Restructuring the service to provide an open implementation can improve both the structure and the performance of both the application and the service.

Figure 2.1 shows the structure of a language runtime system and its client applications both with and without an open implementation of memory allocation. With an open implementation of memory allocation, applications can become simpler because they no longer have to implement memory allocation to achieve the functionality they require. Without the meta interface, the service — memory allocation — is provided by the *combination* of the language runtime's memory allocator and the sum of the memory allocation code

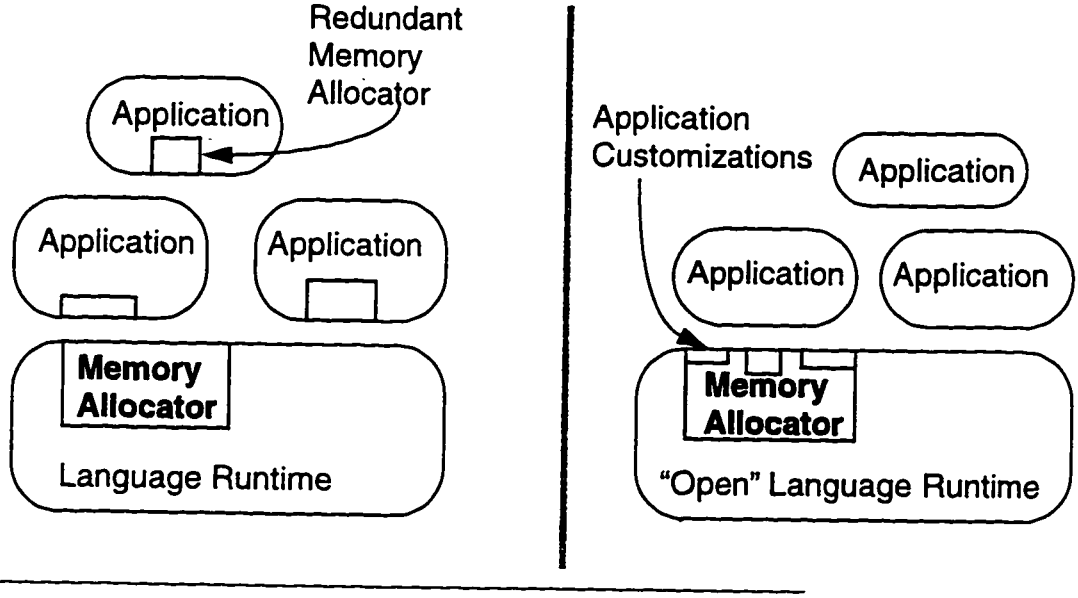


Figure 2.1: Comparison of inflexible vs. open implementations of memory allocation

within all of the client applications. Thus, an open implementation of memory allocation can reduce the overall complexity of the memory allocation service. The increased complexity of the system service required to provide the meta interface will often be outweighed by the simplifications to *each* of the applications that use the system service.

The next section describes three levels of open implementations of existing operating systems services and how they effect application structure and performance.

2.2 Operating System Support for Resource-Intensive Applications

The task of an operating system is to manage the machine's *resources* (e.g., the processor, memory and network hardware) and to provide a set of *services* to client applications (e.g., processes, virtual memory and sockets). A large portion of recent operating systems research has been devoted to supporting applications whose resource needs are not met by the original interface or implementation of a system service. Previous researchers have discussed the motivation for open implementations of operating systems [Rashid et al. 89, Anderson 92, Kiczales et al. 93, Draves 93]. David Keppel presented a taxonomy of different approaches to providing open implementations in general [Keppel 93]. This section examines three degrees of flexibility provided by existing operating systems and how they affect application structure and performance. First we describe *fixed interfaces* that provide no flexibility to applications, but are simple to understand and implement. Next, we describe how *adaptive interfaces* automatically adjust resource management according to individual application behaviors, and why they do not support applications with rapidly changing service requirements, or that need alternative interfaces. Then we describe *open implementations* of operating system services that allow the application to tailor the interfaces and implementations of system services to best suit their specialized needs. Finally, we describe why the open implementation approach enables the system to provide more efficient resource management than alternative structures by locating resource management where better decisions can be made.

2.2.1 Opaque, Fixed Interfaces

Operating system services were originally provided via interfaces that hid their implementation and allowed no flexibility to accommodate the varying needs of different applications. However, some implementation details have to be revealed — for even the simplest services — in order to provide acceptable performance and correct behavior. For example, the simplest definition of a stream input-output interface (e.g., to a disk file) is `open`, `close`, `read` and `write`. The description at this level of detail does not mention the existence of the buffers that are required to achieve high performance for reading from and writing to a stream. However, most applications that deal with streams are eventually affected by the presence of buffering. Programmers often have to insert explicit calls to flush a stream's buffers in order for output to be written to disk at a particular time (e.g., an ATM's deduction of $20 from an account balance during a withdrawal).

Breaches in "opaque" interfaces reveal the tension inherent in providing an efficient operating system service: operating systems have to provide services to many kinds of applications and no single implementation is likely to be the best for all of them. We classify these interfaces as fixed and opaque because the implementation's *details* are still hidden and because the service does not adapt to application behavior.

2.2.2 Adaptive Operating System Services

In cases where one implementation is unable to satisfy all clients, researchers instead make their systems adapt to applications' behavior either automatically or by providing advisory interfaces. The primary operating system services — processing, communication and storage — have been made adaptive in most current operating systems by adjusting resource management according to observed patterns of usage or to hints provided by applications. For example, the non-adaptive fixed priority, round-robin processor scheduling policy can cause interactive applications (those that frequently interact with user input) to suffer from poor response time in the presence of long-running compute-bound applications. Current operating systems use an adaptive scheduling policy that detects batch

applications and dynamically reduces their priority. By automatically separating the two classes of applications, the adaptive scheduling policy provides good response time to interactive applications and executes batch applications when the user is not interacting with the computer.

Another example of adaptive resource management was motivated by the poor performance resulting from the mismatch between standard virtual memory page replacement policies and some applications' access patterns. Standard replacement policies are designed to perform well for the common access patterns that exhibit spatial and temporal *locality*. Spatial locality is the motivation for the page-granularity of virtual memory systems — it means that an application that accesses any part of a page is likely to access most of the page. Temporal locality is the motivation for locality-based replacement policies — it means that a reference to a page at any time implies a high probability of another reference to the same page soon after. Thus a locality-based replacement policy, such as Least Recently Used (LRU), chooses to replace those pages that have not been accessed recently. Locality-based policies perform well for most applications, but some common access patterns can cause them to misbehave badly. For example, an application that reads a large file sequentially into memory can cause most of the pages on the machine to be replaced, regardless of whether they are being used by other applications. Some operating systems use an adaptive solution to address this problem. For example, Digital Unix adapts its replacement policy for pages that it detects are being accessed sequentially to prevent them from evicting more useful pages.[1] By dynamically adapting to application behavior, Digital Unix is able to provide good performance for a wider variety of workloads than would otherwise be possible.

Some applications — such as multimedia, databases and scientific computation — have resource needs that can not be effectively met even by adaptive systems. For example, an application's behavior may change too rapidly for the system to adapt, or it may have resource needs that would require a fundamental modification to the operating sys-

1. Sequential accesses activate Digital Unix's "sequential drain" mode, which eagerly discards sequentially accessed pages in favor of other pages that are accessed with locality.

tem's interface or implementation. In many cases, applications such as these reimplement the entire system service at the user-level in order to customize that service to meet their needs. For example, some applications implement user-level threads in order to customize scheduling, communication or synchronization. Database systems often implement user-level buffering so that they can control the buffer's size, replacement policy or recovery strategy. The needs of resource-intensive applications like these motivated current research into *extensible operating systems* that eliminate the duplicate resource management between the user-level and the operating system.

2.2.3 Extensible Operating Systems

An extensible operating system allows applications to customize parts of the implementation of a service in order to meet their special requirements. Since the system's implementation is shared among all applications, providing extensibility while preserving the safety guarantees of multiuser operating systems is a significant challenge to designers of extensible operating systems. Researchers have taken three approaches to designing safe extensible operating systems: embedded interpreters [Lee et al. 94, Mogul et al. 87], library-based services [Maeda & Bershad 93, Engler et al. 94] and safe dynamic linking of compiled code [Bershad et al. 95, Necula & Lee 96]. In spite of these systems' significantly different structures, they share the ability to let applications modify the implementation of a mismatched system service instead of requiring that they reimplement the service themselves.

Extensible operating systems have three benefits. First, by allowing applications to customize the implementation of services, extensible systems can provide those services with higher performance than is possible with either a fixed or adaptive implementation [Fiuczynski & Bershad 96]. Second, since applications can rid themselves of redundant resource management code, extensible systems can help applications achieve high performance with simpler implementations [Kiczales et al. 91].

The third benefit results from placing control of a resource where the most knowledge about that resource's utilization resides. Scheduler activations [Anderson et al. 92] provide

an example of this strategy for processor scheduling. For example, when a user-level thread system is built on traditional kernel threads, oblivious kernel scheduling can cause scheduling "convoys" when threads must wait for another thread that holds a user-level lock because it has been descheduled by the operating system. Since the operating system does not know about user-level locks, it does not know when it is a bad idea to deschedule a thread. Scheduler activations solve this problem by supporting scheduling at the user-level, where lock-management resides.

This dissertation examines a related situation for memory management. Most database systems implement buffer management at the user level to allow them to control their buffer's size, replacement policy and recovery strategy. However, the operating system has important knowledge about the memory usage of the other applications sharing the machine. Since user-level buffer managers are oblivious to other applications, they can make mistaken assumptions about memory availability that result in significantly reduced performance. Database designers have taken two approaches to their interaction with operating system memory management. The first approach is to implement user-level buffer management by bypassing the operating system's memory management. The second approach is to integrate database buffer management with the operating system's memory management by modifying the operating system to implement database functionality. We discuss the implications of these two approaches for memory management in the next section.

2.3 Application and Operating System Structure for Efficient Memory Management

There are two approaches to structure applications to take advantage of open implementations of an operating system service. The first approach is to use low-level interfaces to "take over" particular resources of the machine. This approach *replaces* operating system services, at a coarse granularity, with resource managers provided by the application. The second approach is to place control of the resource within the operating system and provide an auxiliary interface that allows applications to customize the resource management

in order to accommodate their particular needs. The first approach results in duplicate resource managers between the operating system and the application. The second approach integrates the application's resource management with the operating system's.

The memory and scheduling examples described in Section 2.2.3 demonstrate that an operating system can provide the support required to place resource management either within an application or within the operating system, depending on the resource or its management requirements. The challenge for the design of an effective operating system/ application interaction is to determine where the management of each resource belongs; within the application or within the operating system.

The degree to which a resource is multiplexed with other applications influences whether operating system-level or application-level control over that resource is more efficient. In some cases, resources can be dedicated to a fixed number of applications. Database servers and video game machines are two examples of such an environment. In this environment, resources (e.g., memory) can be statically partitioned among applications, and the system can efficiently give control over resources to the applications that require it. General purpose desktop machines, however, have dynamic workloads with multiple applications that compete for all of the machine's resources. In these environments the operating system frequently reallocates resources between applications. In this case, placing resource management within the operating system may be more efficient.

Database and operating systems researchers have long been at odds over the level at which to implement memory management. Database researchers have made convincing arguments that operating systems' memory management facilities are inadequate for supporting database buffer management [Stonebraker 81, Traiger 82]. As a result, database researchers developed techniques to bypass operating systems' memory management in order to implement buffer management entirely within the database system. These techniques have proven very effective in the database server environments for which they were intended. Currently, however, databases are often used outside the server environment because of their use in popular application-building infrastructure, or *middleware*, such as object oriented databases. Further, server machines are being configured to run dynamic workloads, such as corporate "enterprise servers" that manage databases, web servers and

file servers at the same time. Even though they now find themselves running in memory-competitive environments, databases still use the buffer management techniques developed for dedicated servers.

When a resource is managed by applications in environments where resource competition results in frequent operating system management as well, the resulting simultaneous levels of resource management can result in extremely poor performance. The first observed example of this phenomenon was for memory management, and is called the *double paging anomaly* [Goldberg & Hassinger 74]. Since *paging* describes the task of buffer management, double paging describes the situation of two conflicting levels of buffer management. Databases in memory-competitive environments can suffer from double paging, which results up to three times the amount of disk traffic, which is usually the performance bottleneck in these systems. One way to eliminate double paging is to integrate database memory management with the underlying operating system memory management, thus eliminating one of the levels of paging.

Operating systems and database researchers have built operating systems with customized support for databases (e.g., Camelot's use of Mach [Spector 91], the Bubba operating system's support for databases [Boral et al. 90], and IBM's various database machine/operating systems, including DB2 [Cheng et al. 84], System R [Gray et al. 81], and 801/CPR [Chang & Mergen 88]). The problem with using these systems to solve double paging in applications is that users' desktop machines run commercial operating systems, whose structure and interfaces are significantly different from the structure and interfaces of database operating systems.

The current research into extensible operating systems offers great promise: these systems will allow applications to implement the functionality of the research database-specific operating systems (via system service extensions), while also providing commercial operating system interfaces. Until these extensible systems are deployed, however, applications are constrained to using existing commercial operating systems' facilities and to experiencing poor performance in memory-competitive environments.

The new system described in this dissertation provides a bridge to these applications that allows them to be structured as they would be in an extensible operating system, but by using currently available facilities.

Chapter 3

Transaction Buffer Management and Virtual Memory

This chapter sets the technical stage for the dissertation by describing existing approaches to transaction buffer management, and the interaction of these approaches with existing operating system virtual memory management mechanisms.

Transactions have proven to be a key component of reliable, efficient and maintainable applications that manage distributed and/or persistent data. Transactions are delimited sequences of data accesses ending in a call to a system *commit* primitive. Transaction commits are *persistent* and *atomic*. Persistence is the ability to retain data across non-catastrophic failures and is usually provided by storing data on magnetic disks. Atomic updates happen all at once or not at all, even in the presence of failures. The access time of modern disks is five orders of magnitude slower than non-persistent main memory (5 milliseconds for disks vs. 50 nanoseconds for memory). As a result, the technique of temporarily *buffering* persistent data in memory is critical for high performance transactions. This chapter describes the interaction between transaction system buffering and the buffering done by the underlying operating system.

A key function of a transaction system is supporting modifications to persistent data. Allowing applications to modify data in memory buffers is critical for high performance. Since buffers are not persistent, transaction systems must carefully manage the movement of data to and from buffer memory. Transaction system researchers have developed buffer management techniques that make updating persistent data efficient. These techniques assume that the transaction system can manage the machine's main memory to implement buffer management. However, this assumption may be violated when a transaction system is run within virtual memory — a situation that can result in significantly degraded performance.

Virtual memory is a facility provided by operating systems that manages the main memory of a machine to provide applications with the illusion of a large amount of memory. Since the total amount of data accessed by applications can greatly exceed the amount of main memory, operating systems store virtual memory data on disks. In a manner similar to transaction system buffer management, operating systems buffer regions of virtual memory in main memory in order to minimize access times. Unlike transaction systems, virtual memory does not provide persistence. Because operating systems view transaction systems as normal clients of virtual memory, transaction system buffers are in virtual memory, thus potentially violating the transaction system's assumption that buffer memory is always efficient to access. Thus transaction systems conflict with operating systems over the control of memory management.

Transaction systems can resolve this conflict by running on dedicated server machines. System administrators control the workload on servers to ensure that the total amount of virtual memory used by applications and the transaction buffer manager does not exceed the main memory of the machine. When the amount of virtual memory can be accommodated in main memory the virtual memory system does not need to perform buffer management, thus the transaction system's buffers are always in main memory.

Transaction systems are increasingly being used outside the server environment: on client machines or personal computers on users' desktops. On desktop machines applications are not as carefully controlled, so the virtual memory system can cause transaction buffer pages to reside on disk. Thus the techniques previously used in server-based trans-

action systems are no longer an efficient solution to the conflict between transaction systems and operating system virtual memory.

This chapter explores the conflicting memory management demands of transaction systems and virtual memory. It first describes the transaction properties and the memory management techniques — buffering and logging — used to implement them. Then it describes operating system virtual memory management and the variety of ways a transaction system's buffering and logging can interact with virtual memory. This chapter concludes by summarizing the interaction between buffer management and virtual memory for a number of transaction systems and their underlying operating systems.

3.1 Transaction Properties

Transactions are a formalization of the intuitive notion of "all or nothing" modifications to important data. For example, a bank wants its account data to be accurate in the presence of power failures, system crashes, as well as simultaneous activity to a single account. Transactions have been key components of a wide variety of distributed systems. The designers of these systems have found transactions to be invaluable toward making their designs tractable, correct and efficient. The properties provided by transaction systems are summarized by the "ACID" mnemonic [Haerder & Reuter 83]. These properties are:

- *Atomicity.* Transactions are delimited by *begin_transaction* and *commit* statements. When *commit* has successfully completed, the actions performed during the transaction are installed all at once. If the system fails or if the transaction is manually aborted, none of the transaction's changes are made.

- *Consistency.* As long as the other properties hold, consistency is guaranteed by the applications themselves performing only valid manipulations of the data. An example of an inconsistent transaction would be a bank transfer where $20 was deducted from a client's account, but $40 was dispensed from a cash machine.

- *Isolation.* In the presence of multiple concurrent transactions, potentially attempting to modify the same data, a transaction system must assure that the

results are equivalent to some sequential ordering of the same actions. This notion is formally referred to as the *serializable* property.

- *Durability.* The state of a correct transaction system is preserved across a defined set of failures. Durability is implemented by saving the state of committed transactions to stable (i.e., failure-resistant) storage. *Persistence* is the non-mnemonically convenient name for this property.

The transaction system controls the accesses and updates to the database contents in order to provide the transaction properties. The core components of a transaction system are the transaction manager, the log manager, the buffer manager and the lock manager. The transaction manager coordinates the actions of the other components to implement transactions. The transaction system's *recovery mechanism* is provided by a combination of the buffer manager and the log manager, which together are responsible for ensuring that database contents can be restored to a transaction-consistent state after a failure (i.e. a state that obeys the ACID properties). This rest of this chapter addresses the interaction between the transaction system's recovery mechanism and the underlying operating system.

3.2 Buffer and Log Management in Transaction Systems

The durability requirement of transaction systems implies that data is kept on *stable storage*. Magnetic disks are the medium most often used as stable storage. The time required to access data on today's disks is between 5 and 8 milliseconds. This is five orders of magnitude slower than the time required to access main memory (approximately 50 nanoseconds). Therefore transaction systems buffer frequently used data in main memory, where it can be accessed much more efficiently. In doing so, the time spent to read data from disk is amortized over the number of accesses to that data.

Handling updates in a transactional manner complicates buffer management; both atomicity and durability can be compromised by buffering data without taking extra mea-

sures. Atomicity can be violated by buffer management mechanisms that can write portions of uncommitted transactions to the database. If transaction modifications are buffered, durability can be violated because main memory buffers generally do not survive software or hardware failures.[1]

The combination of buffer management and logging provide the durability and atomicity transaction properties. The rest of this section describes buffer and log management and the interactions between the two.

3.2.1 Buffer Management Alternatives

The goal of the buffer manager is to keep frequently accessed data in memory buffers. The amount of data stored in a database usually greatly exceeds the amount of memory in most machines. Even though the amount of memory available in desktop machines is growing, the sizes of the databases users wish to access remain larger than memory sizes, and appear to be growing at least as fast. As a result, once buffer memory is consumed, the buffer manager must remove data from the buffer to accommodate new requests. The policy used to choose what data to be removed is the buffer manager's *replacement policy*. If the data being replaced has not been modified, or if modifications have been saved in auxiliary storage, data can be replaced from the buffer by simply overwriting it with new data to be buffered. If the data has been modified, however, the buffer manager must save the modifications somewhere before replacing the data, and must remember to retrieve the modified data upon subsequent requests. How these buffer management actions interact with transaction processing are summarized by their "steal" and "force" characteristics.

1. The Rio file cache [Chen et al. 96] demonstrates a method of improving the durability of DRAM buffers across software failures, which may be sufficient for some applications. This could be used to greatly improve the performance of transaction systems, though more experience with the technique is needed.

Steal vs. no steal

Steal and no steal buffer managers differ in how they interact with the database's *before image* on stable storage during buffer replacement. The before image is the database contents before the modifications of any uncommitted transactions. A steal buffer manager can, at buffer replacement time, overwrite the before image with uncommitted transaction modifications. In this case the transaction manager must ensure the before image can be recreated (e.g. with information about how to undo the modification) in the event of a transaction abort or a system failure. The alternative buffer management policy, no-steal, ensures that the before image remains in the database until the transaction commits. A no steal buffer manager can be implemented by either *pinning* the modified data in buffers for the duration of the transaction (i.e., making them ineligible for replacement for that duration), or by diverting pageouts to a temporary region of stable storage.

The steal policy has a number of advantages over no steal in most environments. In a no steal buffer manager, when buffer replacement is unavoidable, the writes of modified pages to a temporary area (resulting from page replacement) do not contribute toward the total number of disk writes required to commit a transaction, because the commit procedure must perform the modifications to the database or to the log. In contrast, buffer replacement in a steal buffer manager updates the database itself, which may obviate some of the updates to the database during commit. A no steal buffer manager may instead pin modified data in the buffer. Pinning data limits the amount of data a transaction can modify to the amount of data that can fit in the buffer, which can be unacceptable. Further, as the amount of pinned data grows, the proportion of the buffer that can be used to buffer newly accessed data shrinks. This constraint on the buffer replacement policy results in reduced buffer hit rates which reduces performance. Figure 3.1 depicts an example of buffer replacement for steal and no steal buffer managers.

Force vs. no force

The force characteristics of a buffer manager determine what happens to the database when the transaction commits. At commit time, a *force* policy updates the database with

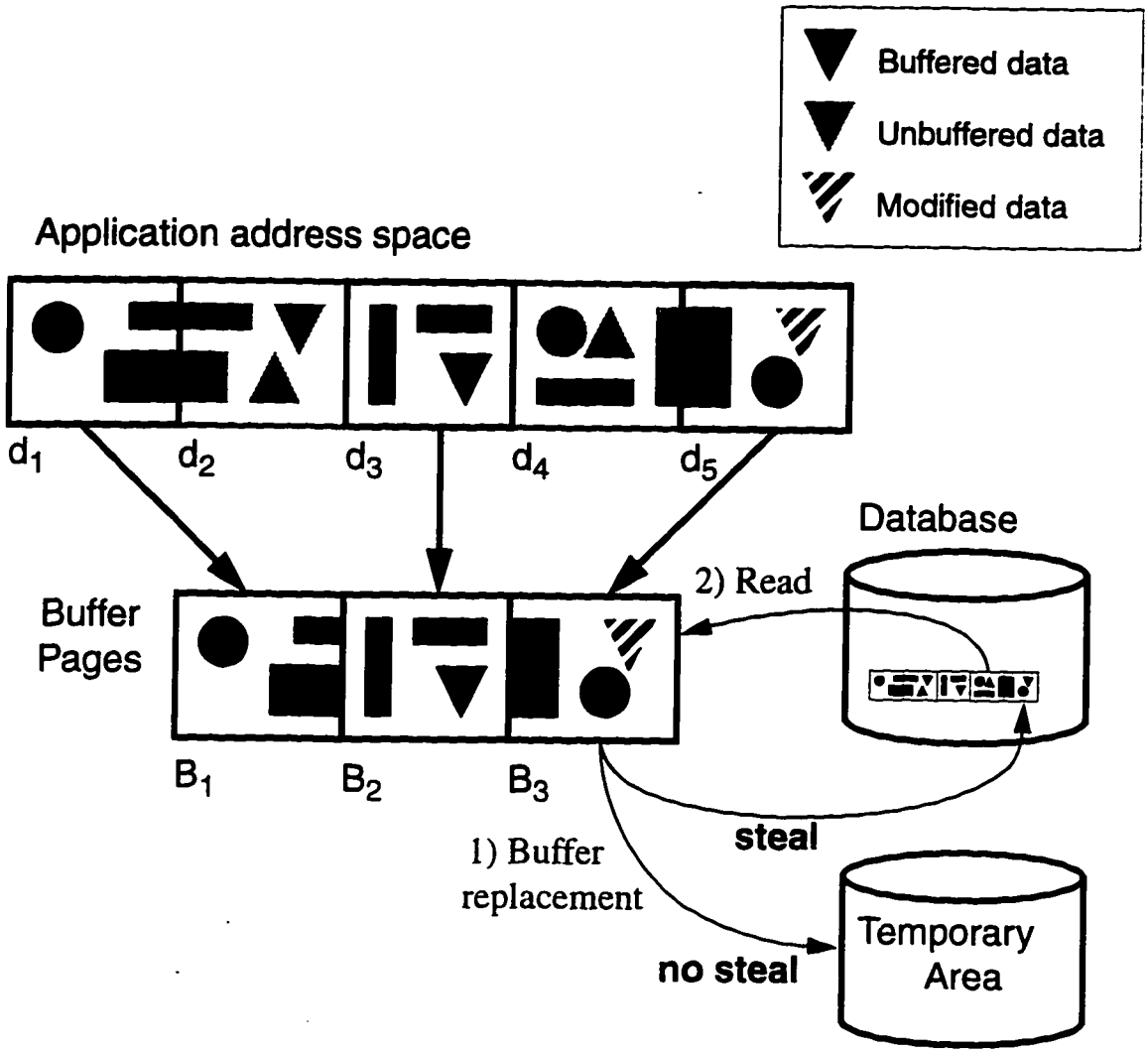


Figure 3.1: Buffer replacement for steal vs. no steal buffer management.
In this example, the database pages d_1, d_3 and d_5 are buffered by buffer pages B_1, B_2 and B_3 respectively. An access to an unbuffered database page (e.g., d_4) would be a buffer miss. Since the buffer is full, the buffer manager would choose a buffer page to replace. If the chosen page (e.g., B_3) has been modified, a steal buffer manager writes the data back to the database at position d_5. A no steal buffer manager writes the data to a temporary area, leaving the before image of d_5 in the database. After the data is written, or if the data had not been modified, the B_3 buffer page can be used to buffer d_4 by reading d_4 from the database into B_3 and associating the buffer page with the database page's location in the application.

the transaction's modifications. A *no force* policy does not update the database itself, but rather stores the transaction's modifications in a log record. Figure 3.2 describes the operation of *commit* for force and no force buffer managers. Since modifications are not

written directly to the database, as transactions proceed the current contents of the database diverge from the database image on stable storage. This means that to satisfy a buffer miss, the transaction system must first read the database image, then apply the updates to the data from the log. Recovery time is similarly delayed by long logs. Both operations can take time proportional to the length of the log. The system can bound the size of the log by periodically recording the locations of the transaction-consistent copies of all database objects. This operation is called *checkpointing* the database.

The steal and force characteristics of a transaction system summarize how the buffer manager handles pageout and commit, respectively. A system could implement any combination of steal and force for its buffer management. However, since data is made persistent at different times in each policy (e.g., a no force buffer manager does not update the database at commit), the transaction system needs to maintain auxiliary storage to provide the transaction properties. This storage is managed by the logging manager, which is described in the next section.

3.2.2 Log Management Alternatives

The task of the buffer manager — to keep frequently used data in fast but volatile memory — can be at odds with the atomicity and durability properties of a transaction system. The log manager is the component of a transaction system that stores the auxiliary data required to provide atomicity and durability in the presence of buffering. The design decisions made for buffer management influence the types of logging information required. Logs can contain the information required to undo a modification, to redo a modification, or both. The information kept in the log may describe the operation performed (operational logging), or the values of the data before or after the operation (value log-

ging). Finally, for value logging systems, the granularity of logging — the amount of data

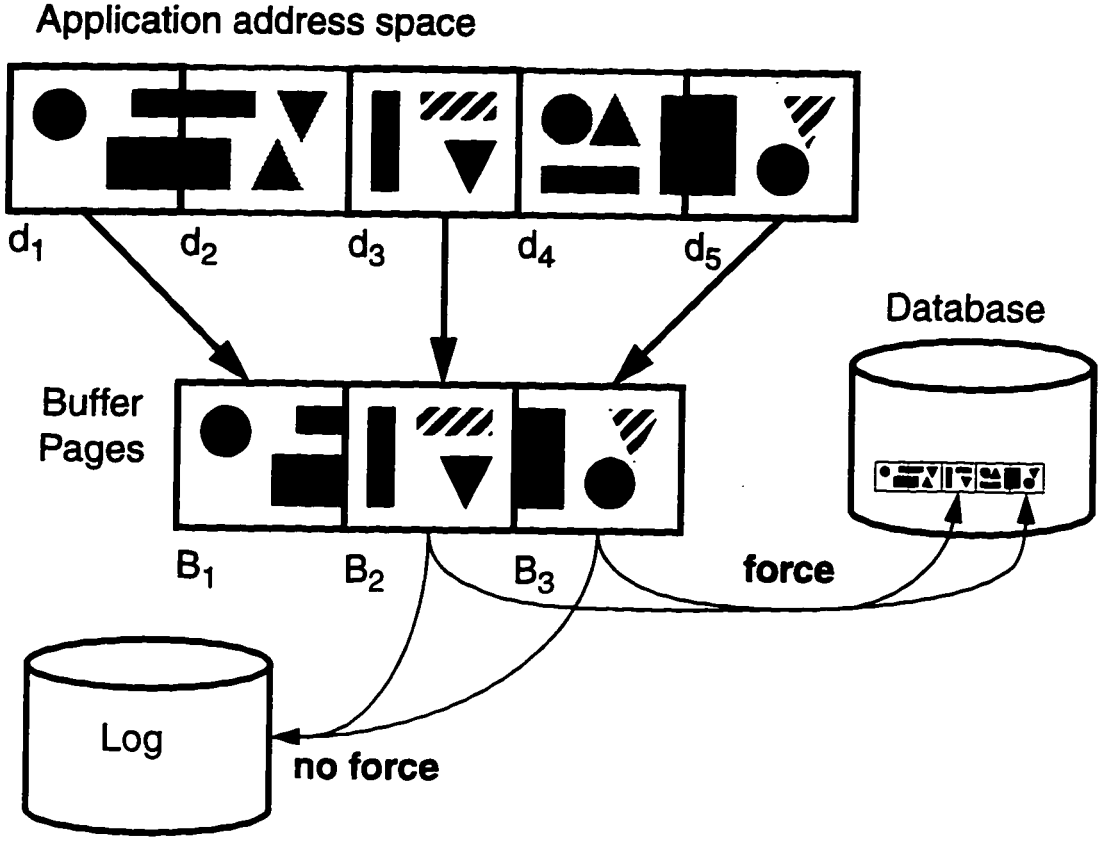


Figure 3.2: Transaction commit for force vs. no force buffer management.
In this example, the database pages d_1, d_3 and d_5 are again buffered by buffer pages B_1, B_2 and B_3 respectively. The application has modified objects on d_3 and d_5. In order to atomically commit the modifications to stable storage, a force buffer management policy must first save undo information (e.g., to the log), then updates the database directly. A no force buffer management policy reflects updates only to the log, by saving redo information. In the no force case, buffer misses are more complex to satisfy: the buffer manager fetches the page from the database, then applies all of the redo information affecting the fetched page. No force buffer managers can bound the amount of work required for installation reads by periodically *checkpointing* the database, which collects on disk a more recent state of the database.

in each log entry — may be the same size as a buffer page, or may correspond to the amount of data modified by the transaction.

Undo vs. redo logging

The type of information logged (undo vs. redo) is affected by the choice of buffer management policy. For example, a steal buffer manager requires that undo information be saved to stable storage before buffer replacement can occur. This is because the buffer manager may modify the before image before a transaction has committed. A no steal buffer manager removes this requirement, but incurs the disadvantages associated with the no steal policy. In addition to the choice of steal vs. no steal, the force attributes of the buffer manager impacts logging. A no force buffer manager requires that redo information is kept, because at transaction commit, a no force buffer manager does not update the database. Therefore the transaction's modifications must be stored in the log. Using a force buffer manager removes this requirement for redo log information. From the constraints described in this section, it may seem that a no steal, force buffer manager requires no logging information at all. The constraints of implementing atomic commit, however, do require logging information even with a no steal, force buffer manager. This is because the commit procedure must atomically perform a series of modifications to the database. Since a series of modifications is not an atomic operation, some form of redo or undo information must be on stable storage before the commit operation can modify the database.

Operational vs. value logging

One type of information that may be stored in a log is a description of the operation performed by the transaction. This has the advantage that a single operation description may be enough to represent both undo and redo information. For example, the description "move $20 from account x to y" says how to redo the transaction, and also provides enough information to undo the transaction.[2] One disadvantage of operational logging is that *each* operation within a transaction must be logged. Long-running transactions that

modify the same data repeatedly may write more information using operational logging than if all of the operations were summarized in one log entry. In addition, the logging system must be aware of the semantics of the operations being logged, which requires significant support from the application or the language system.

The alternative to operational logging is value logging. Value logging ignores the semantics of the operations and instead deals with the contents of the database before and after the transaction. A value undo log is the set of before images of any data modified within the transaction. Similarly, a value redo log is the state of the data after it has been modified. One advantage of value logging is that it can succinctly capture the effect of a large number of operations to the same data. One disadvantage of value logging is that undo and redo information must be gathered separately. Another disadvantage is that a value logging system must choose the *amount* of data to save in each log record, and this choice involves a number of trade-offs.

Logging granularity

Transaction systems that use value logging must choose the granularity of logging — the amount of data stored in each log record. Using the same granularity as the buffer manager allows the buffer manager to implement logging, which can simplify the system considerably. However, using buffer pages as the logging granularity may result in significant amounts of logging activity for unmodified data. The alternative is to use fine-grain logging. Fine-grain logging systems adjust the amount of data logged according to the amount of data modified by the transaction. Fine-grain logging can be done at many levels of a transaction system. The buffer manager can perform page differencing [White & DeWitt 94], the language can incorporate logging information within modifications [Liskov et al. 94], specialized hardware support can be used to gather sub-page grain update information [Chang & Mergen 88], or the programmer can manually bracket modifications with logging directives [Satya et al. 94]. The designers of systems that utilize

2. Some types of operations cannot be undone (such as dispensing cash at an ATM, raising a traffic barrier, etc.). In these cases, the operation is performed as a part of the commit action.

page-grain logging [Lamb et al. 91] justify their decision by stating that careful clustering of objects within a page results in little wasted logging activity. Clustering gathers objects that are likely to be modified together on to the same page. Clustering can be very effective for specific workloads, but other operations on the same data may benefit from different clustering patterns, which can decrease its overall effectiveness.

Write-ahead logging vs. shadowing

The logging techniques discussed so far in this section have all been variants of *write-ahead logging*. Write-ahead logging refers to the technique of writing, to the log on stable storage, the information required to redo and/or undo a modification before modifying the database itself. In value logging systems, this undo information is just a copy of the data already in stable storage in the database itself. Shadow files are an alternative to write-ahead logging that avoids the need to write undo log information by preserving the original page of the database on stable storage until the transaction commits. Unlike force, no steal buffer management, shadow files remove the need for logging in order to implement atomic commit. Figure 3.3 shows how atomic metadata updates are used to atomically install transaction modifications.

The main advantage of shadow files is that they obviate undo logging. For some workloads this results in a great reduction in writes to stable storage. However, the shadow file approach has two significant disadvantages. First, it is inherently page-grained; therefore it is not well suited for situations where page granularity is inefficient. Second, even if page granularity is not a problem, as updates occur the shadow technique can scatter the database between the two files. This defeats inter-page clustering optimizations, and can result in significant performance degradation [Mohan et al. 92].

3.3 Transaction Buffer Management and Virtual Memory

This section discusses how transaction system buffer management interacts with virtual memory in the presence of paging. First, we describe how virtual memory works and compare it to transaction system buffer management described in Section 3.2.1. Then we describe three ways existing transaction system buffer managers interact with virtual memory.

3.3.1 Virtual Memory Buffer Management

Virtual memory is the memory service provided by most modern operating systems. It manages the physical memory of a machine to buffer the data being used by all of the applications on that machine. The task of virtual memory is similar to that of a transaction system buffer manager: to use a limited amount of memory to buffer a potentially much larger amount of data. Transaction systems can use virtual memory as a buffer manager, when combined with a lock manager and log manager. As described in Section 3.2.2, the type of logging information required depends on the steal attributes of the buffer manager.

As a component of a transaction system, virtual memory can implement either steal or no steal buffer management. No steal buffer management is provided when the application uses the filesystem to read the contents of a database (or file) into virtual memory. By default, the virtual memory system uses its own *paging file* as the storage for buffer replacement. Since buffer replacement causes data to be written to the paging file rather than the database file, in this case it implements no steal buffer management. When the application first reads the database into virtual memory, all of the virtual memory buffers are *dirty* (i.e., modified with respect to the virtual memory system), thus the virtual memory system must write each such page to its paging file the first time it is replaced from the buffer.

Virtual memory systems can also provide steal buffer management, using *mapped files*. Mapped files originated in Atlas [Kilburn et al. 62] and were central in the design of

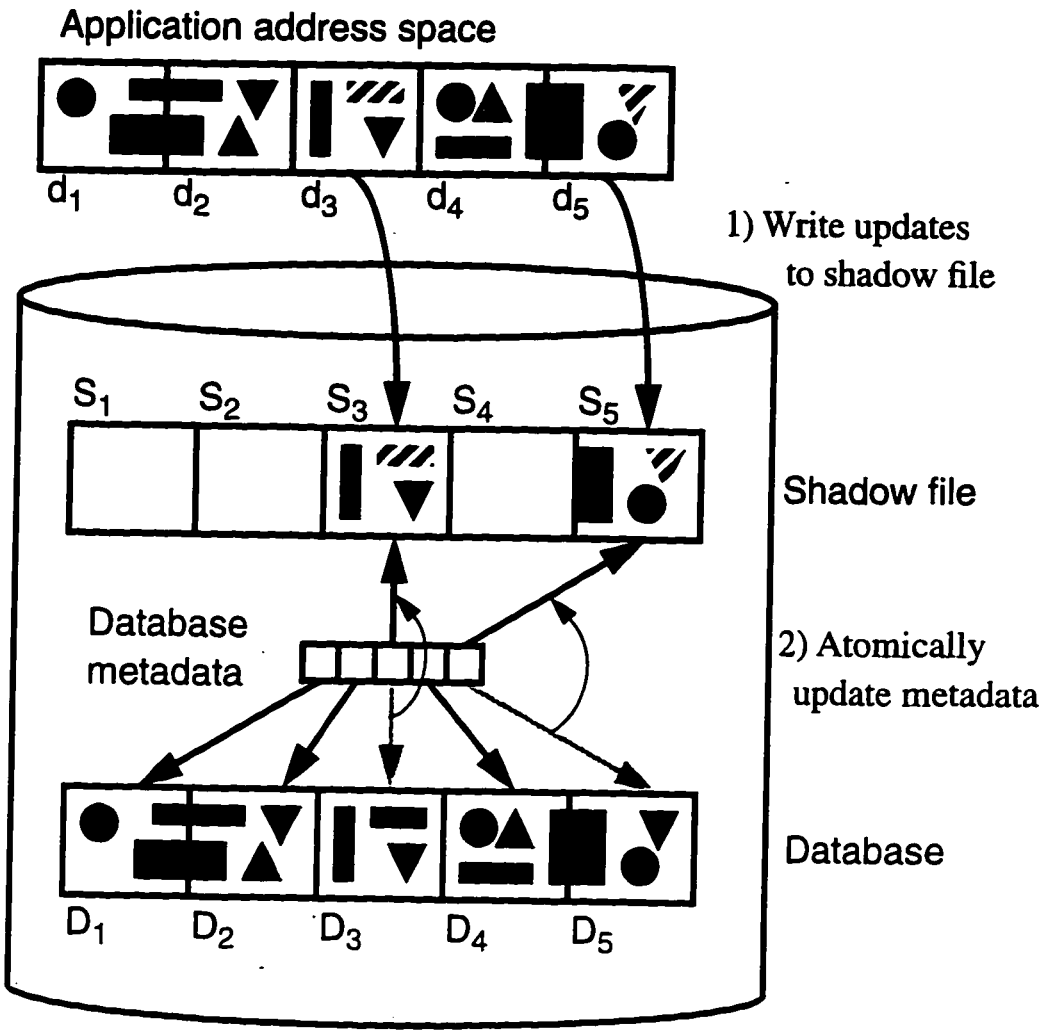


Figure 3.3: Operation of commit in a shadow file transaction system.

This example shows a database file, its shadow file, and the metadata. The metadata describes which pages from each file comprise the current state of the database. The buffers have been omitted for clarity. Before the transaction commits, the metadata consists of pointers to pages [D_1, D_2, D_3, D_4, D_5]. The application has modified data of pages d_3 and d_5 of the database. The commit procedure first writes the new data to the shadow file locations for those pages: S_3 and S_5. The commit completes by atomically updating the metadata to consist of [D_1, D_2, $\mathbf{S_3}$, D_4, $\mathbf{S_5}$]. The shadow vs. database status of each page toggles after each transaction, so a subsequent modification to d_3 would first write the data to D_3, then update the metadata to consist of [D_1, D_2, $\mathbf{D_3}$, D_4, S_5].

Multics [Bensoussan et al. 69]. When an application maps a database into its virtual memory, it can access that memory just as if it had read it from the filesystem. Since the buffer management of mapped files is steal, however, modifications to that memory are written back to the database whenever the virtual memory system replaces a dirty (i.e., modified) page. In contrast to using virtual memory as a steal buffer manager, fetched pages of mapped files are not dirty to the virtual memory system, so they do not have to be written during replacement unless they have been modified by the application. The remainder of this section discusses various ways transaction system buffer management can interact with virtual memory as both a steal and no steal buffer manager.

3.3.2 Using Virtual Memory as a No Steal Buffer Manager

The simplest interaction between virtual memory and transaction system buffer management, taken by systems such as Texas [Singhal et al. 92], Thor [Liskov et al. 94] and RVM [Satya et al. 94], is to use virtual memory as a no steal buffer management system. Since all buffer pages start out dirty when virtual memory is used as a no steal buffer manager, the first buffer replacement of each page requires a write to the paging file. These writes add wasted traffic to the disk and memory systems which are usually already the performance bottlenecks. When the amount of data accessed does not cause paging, however, this is a fairly efficient solution because lack of paging results in no wasted writes. The buffer manager may choose to use force or no force, depending on the type of logging used.

Commercial transaction systems do not adopt this approach because they need to support accesses to databases that are much larger than memory, which would result in large numbers of extraneous writes to the paging file. For example, a read-only access of an entire database that is ten times larger than memory would cause 90% of the database to be written to the paging file. Furthermore, since the paging file is a bounded system resource, this approach limits the amount of data that can be accessed by a single application to be less than the size of the paging file. Finally, transaction system designers have found that traditional virtual memory replacement policies can be inefficient for some

access patterns. In particular, many relational database structures are accessed sequentially, and traditional virtual memory replacement policies, which assume temporal access locality,[3] perform poorly with such access patterns. In the next section we describe the technique that current transaction systems use to avoid these problems.

3.3.3 The Buffer Pool Approach: Bypassing Virtual Memory

To avoid the problems associated with buffering the database in unbounded amounts of virtual memory, commercial transaction systems use the *buffer pool* technique to attempt to bypass virtual memory. The transaction system first determines how much memory to use for the database buffer. It handles accesses to unbuffered data by reading the data from the file into a free page of the buffer pool, and referring the application to the buffered data.[4] When the free list is exhausted, the buffer manager uses *its* replacement policy in the buffer pool.

The buffer pool approach gives the buffer manager control over page size, data placement, replacement policy, prefetching, and the choices between steal and no-steal and between force and no force buffer management. On dedicated server machines this is an effective buffer management technique. Using a buffer pool limits the amount of virtual memory used to buffer the database. As long as the *total* virtual memory consumed on the machine can be accommodated by the available physical memory, no virtual memory paging occurs, thus the virtual memory system is bypassed. Administrators of transaction system servers carefully manage the applications running on the server to ensure this condition.

On desktop machines, however, applications are not so carefully controlled, so the virtual memory system can cause buffer pages to be paged out. This results in *double paging*

---

3. Temporal access locality means that once a data is accessed the first time, it will be accessed a repeatedly in a short amount of time. Replacement policies tailored for temporal access locality give recently accessed pages priority over less recently accessed pages.

4. There are a large number of ways the buffer manager can provide a buffer page to the application — it may map the buffer page into the application's address space, or use software indirection to refer accesses to buffered data. To allow buffer replacement, any chosen method must allow pages to be revoked.

— two layers of buffer management that are simultaneously active and oblivious to each other. Figure 3.4 presents an example of double paging and describes why double paging reduces performance. Where memory competition is unavoidable, such as on desktop machines, transaction system buffer managers must cooperate with virtual memory in order to avoid double paging and its resulting performance degradation.

3.3.4 Cooperation Between Transaction Buffer Management and Virtual Memory

A number of transaction systems have solved the double paging problem by integrating their buffer management with virtual memory. These systems have supported both steal and no steal buffer management.

Virtual Memory Support for Buffer Pools

Many operating systems provide memory-critical applications — such as transaction systems — with control over whether individual pages of virtual memory should be considered during replacement decisions. This allows transaction systems that use virtual memory as a no steal buffer manager to avoid double paging by *pinning* the buffer pool in physical memory (i.e., making buffer pages unavailable for replacement). Pinning buffer memory prevents double paging but results in added pressure on memory, since there are fewer pages available to buffer data. The buffer manager can reduce this effect by sizing its buffer pool to be a small fraction of available memory, but this results in unnecessarily poor performance in times of light competition for memory. Therefore this approach is best used on server machines as an added measure to prevent double paging.

Even with operating system support, there are two problems with the buffer pool approach in the presence of virtual memory paging. The first problem is that buffer pools in virtual memory provide no steal buffer management, which is inefficient in the presence of frequent page replacement. The transaction system may perform steal buffer management of *its* buffers, but when the virtual memory system replaces pages, its writes go to the

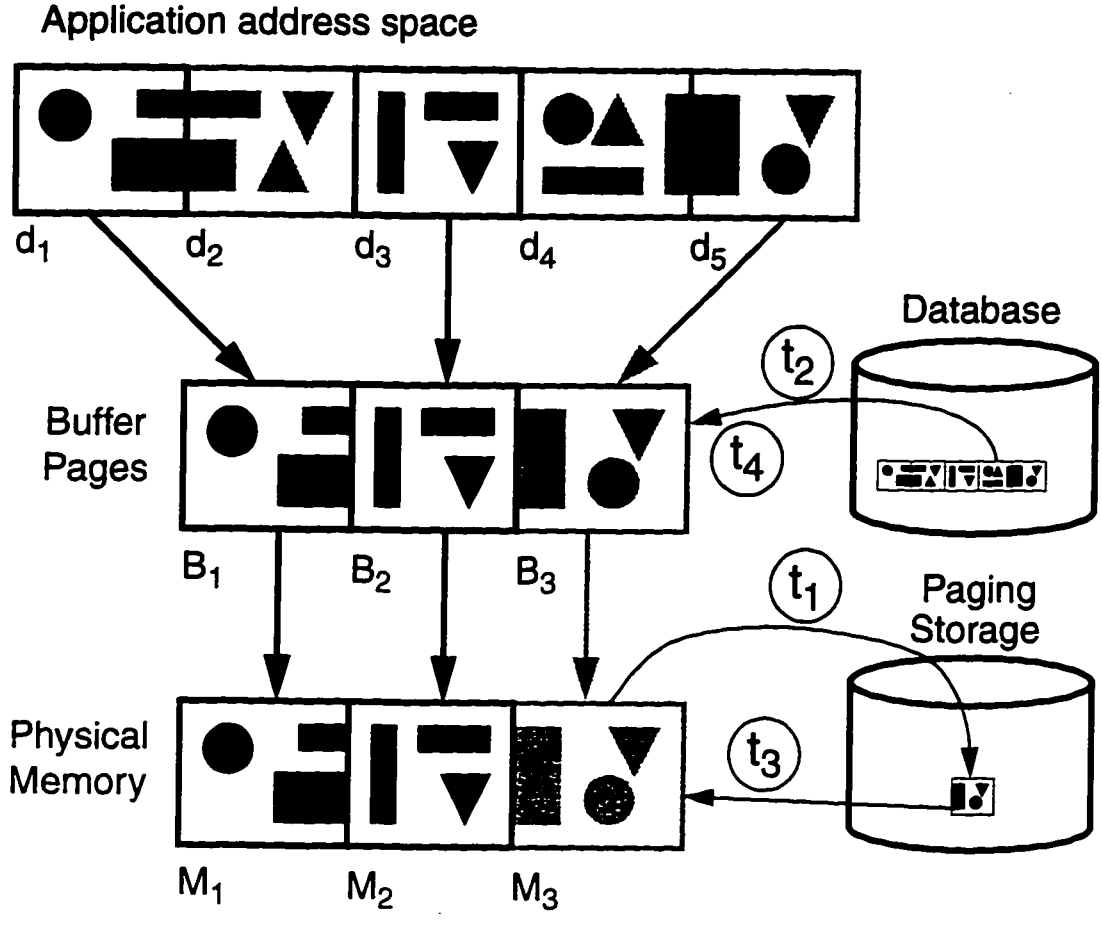


Figure 3.4: Double paging between a transaction system buffer pool and virtual memory

In this example the transaction system buffer pool is in virtual memory. Database pages d_1, d_3 and d_5 are buffered in virtual memory pages B_1, B_2 and B_3. At time t_1 the virtual memory system replaces page M_3, writing its data to paging storage. At time t_2 the buffer manager receives a request from the application for page d_4. The buffer manager attempts to satisfy the request by reading d_4 from the database into buffer B_3. Since B_3 is not resident, the read operation causes a virtual memory page fault. At time t_3, the virtual memory system satisfies the page fault by reading M_3's data from paging storage. At time t_4, the buffer read resumes, and overwrites B_3 with the contents of d_4. The oblivious operation of two levels of buffer management (the transaction system's and virtual memory's) results in a wasted disk read (at time t_3) on the critical path of satisfying the application's request for data.

paging file. Writes to the paging file do not contribute to the writes required to commit the transaction, which need to be either to the database or to the log. Thus the writes performed by the virtual memory system only add to the total writes required to commit the transaction. The second problem is that each time a buffer pool reads data into a buffer page, that page is considered to be dirty by the virtual memory system. This results in writes to the paging file any time the virtual memory system needs to replace buffer pool pages. Both of these problems are addressed by integrating the virtual memory system with transaction system buffer management.

Integrating Transaction System Buffer Management with Virtual Memory Management

As described in Section 3.3.1, mapped files are an interface to virtual memory that provides steal buffer management. Section 3.2.2 described why steal buffer management requires undo information (either in a log or a shadow file). Unfortunately, current virtual memory systems do not collect undo information. Undo logging can be implemented at the application or language level, but those approaches rely on support that was not required by the buffer pool approach. Requiring support from applications burdens programmers with the need to correctly insert transaction directives. Relying on language or compiler support reduces the portability of client applications and constrains programmers' choice of languages. This approach is also ill-suited to language-heterogeneous applications. The rest of this section describes a variety of operating systems that provide language-independent support for implementing transactions in mapped memory.

Computer system vendors have designed custom hardware and operating systems specifically to efficiently support transaction buffer management. The IBM CPR operating system [Chang & Mergen 88] on the 801 processor is an example of this approach. The CPR system uses the 801's support for fine-grained redo logging and controls all of the system's memory management, thus avoiding double paging. CPR's choice of redo logging necessitates no steal buffer management, however, which reduces its efficiency during frequent buffer replacement.

An alternative approach, used by TABS [Spector et al. 85], Camelot [Spector 91], Bubba [Boral et al. 90], and DB2 [Crus 84], is to provide support for transaction buffer management in customized operating systems that run on commodity hardware. A technique developed in the TABS prototype and refined in the Mach operating system is to extend the virtual memory system by providing an external memory management (XMM) interface [Young 89] to applications that have special virtual memory needs. The Camelot distributed transaction system uses Mach's XMM interface to integrate its buffer management with virtual memory. The XMM interface allows Camelot to control the source of data for page faults, and the destination of data during page replacement. Camelot uses this facility to implement a pageable no steal buffer manager by directing pageouts of dirty data to a temporary file until transactions commit.

Transaction system designers have not been able to use these research results in commercial products because they either rely on hardware support (e.g., CPR/801 [Chang & Mergen 88]), or operating system modifications (e.g., Camelot [Spector 91], Mach [Rashid et al. 89], TABS [Spector et al. 85], Bubba [Boral et al. 90], and DB2 [Cheng et al. 84]) that are not available in commercial operating systems such as Unix, Windows, DOS or MacOS.

3.4 Summary of Existing Systems

Table 3.1 summarizes the buffer and log management decisions of a number of existing transaction systems. The table describes the granularity of logging information (either page grain or finer than page grain), the type of recovery information stored (redo, undo, or shadow), the type of buffer management used (steal vs. no steal and force vs. no force), and finally, whether transaction buffer management is integrated with virtual memory.

The top half of the table represent systems that are not integrated with virtual memory. Most of these systems derive their recovery mechanism from the ARIES system [Mohan et al. 92], which innovated write-ahead redo/undo-logging with orthogonal granularity for logging and locking. ARIES' design provides a choice of steal or no steal buffer management. The granularity of logging is usually finer than page-grain in ARIES-derived sys-

tems, but the mechanisms for achieving this vary. For example, QuickStore [White & DeWitt 94] computes differences between before-images and after-images of modified data to achieve fine-grained redo logging information. Because it does not log undo information, QuickStore's buffer management is no steal. Thor uses information gathered from the compiler to generate its redo logging information. ObjectStore, on the other hand, gathers page-grain undo logging information from reference and modify events provided to the logging manager by the buffer manager. POMS [Cockshot et al. 84] does not use write-ahead log based recovery at all, but rather provides recovery by using the shadow file technique.

Table 3.1: The log and buffer management decisions of transaction systems

System	Log grain		Recovery Information			Steal	Force	Integrated w/ OS VM
	Page	Fine	Undo	Redo	Shadow			
ARIES		√	√	√		√		
QuickStore		√		√				
Thor		√		√				
ObjectStore	√			√				
POMS		√			√	√	√	
System R	√				√	√	√	√
801/CPR		√		√				√
BubbaOS	√			√				√
TABS	√		√	√				√
Camelot	√		√	√				√

The bottom half of the table represents systems that integrate transaction buffer management with virtual memory. IBM's System R [Gray et al. 81] includes a custom operating system with a shadow file recovery mechanism to integrate recovery with virtual memory. Since System R uses shadow files instead of logging, commit consists of forcing modified pages to stable storage, then atomically updating the metadata to reflect the modifications. The designers of the recovery mechanism for the 801/CPR system claimed that System R's use of shadow files results in decreased performance due to loss of clustering.

The CPR system instead uses cache-line grain (128 bytes) redo information provided by the 801 hardware to implement recovery. BubbaOS is a prototype operating system designed to provide virtual memory support for no steal buffer management.

All of the non-integrated transaction systems are no steal with respect to virtual memory paging. Even a system that performs steal buffer management in its buffer pool (e.g. ARIES), is no steal when the virtual memory system replaces a buffer pool page, because virtual memory writes those pages to its paging file rather than the database. As discussed in Section 3.3.1, this situation results in double paging, which greatly decreases performance. Some of the systems use special support from the operating system to prevent double paging by pinning buffer pool pages in memory during transactions. This solution eliminates double paging, but still provides no steal buffer management. To date, all of the systems that provide steal buffer management that is integrated with virtual memory use shadow file based recovery mechanisms instead of write-ahead logging. Unfortunately, the shadow file mechanism causes the database to lose any optimized disk clustering, which has been found to significantly degrade performance [Mohan et al. 92].

The next chapter describes the additional issues and trade-offs related to implementing transaction buffer management in single-level storage systems.

Chapter 4

Transaction Recovery in Single-level Storage Systems

Single-level storage systems provide the convenient abstraction of persistent memory that is as large as secondary storage and as fast as main memory. The Atlas system's single-level store [Kilburn et al. 68] inspired modern mapped files, which were first provided by Multics [Bensoussan et al. 69]. The key attribute of a single-level storage system is a uniform interface to data, regardless of its location in the memory hierarchy. Mapped files present this interface in an environment that coexists with other interfaces to persistent data that provide explicit data movement operations (e.g., filesystems' *read* and *write* interface to persistent data). In contrast, single-level stores and mapped files provide an *implicit* interface to persistent data. The storage manager reads and writes data invisibly to the application, in response to the application's memory accesses. By using a single-level store, application-writers can unify the code that deals with volatile and persistent data structures, as well as eliminate the often complex conversion between in-memory data formats and their on-disk counterparts.

Various extensions to single-level stores have broadened the scope of uniform addressing from local persistent storage to a local network or beyond (e.g., Opal [Chase et al. 92], Monads [Rosenberg 92] and Hemlock [Garrett et al. 92]) and added new semantics such as fine-grained protection (e.g., Psyche [Scott et al. 90], Opal, PA-RISC [HP 90] and the PLB [Koldinger et al. 92]). These extensions have improved the utility, robustness and performance of single-level stores. The development of persistent object storage systems and object-oriented databases in the mid 1980's provided the benefits of transactions in single-level storage environments (e.g., POMS [Cockshot et al. 84], GemStone [Maier & Stein 86], and ObjectStore [Lamb et al. 91]).

One of the main benefits of single-level stores is that they provide the opportunity to integrate transaction systems' buffer management with virtual memory. Such integration provides several previously described benefits, including more efficient access to persistent data, and easier data sharing [Shekita & Zwilling 92, Eppinger 89, Chase 95]. As described in Section 3.3, integrating transaction buffer management with virtual memory also improves performance in memory-competitive environments by avoiding double paging.

The most serious drawback of single level stores is that their implicit interface hinders applications that want to control data movement, such as transaction system recovery managers. For example, a system that maps a single level store (e.g., using a mapped file) cannot control when modifications are written to the store, nor where they are written. Because of this limitation, none of the existing commercial object-oriented database systems' transaction buffer managers are implemented on top of the single-level storage interface provided by mapped files. Instead, these systems implement the single-level storage interface provided to their clients using buffer pool buffer management and the filesystem's explicit interface to data.

Single-level stores implemented with buffer pool transaction managers sacrifice one of the main benefits of the single-level storage structure: avoiding double paging by integrating buffer management with virtual memory. Research operating systems have addressed this problem by providing the additional support required to implement transactions in a single-level store [Chew et al. 93, Spector 91, Boral et al. 90]. These systems provide an

auxiliary interface that provides explicit control over the single level store's data movement. Unfortunately, commercial object-oriented databases cannot take advantage of these techniques because their client applications need to run in commercial operating system environments.

This chapter describes some of the issues and trade-offs that affect log and buffer management in a transactional single-level storage system. Section 4.1 describes important attributes of single-level storage systems. Section 4.2 describes the interactions between single-level stores and transaction logging and buffer management. Section 4.3 summarizes the issues discussed in this chapter, and describes the motivation for the design of a new recovery mechanism for a single level store that we present in Chapter 5.

4.1 Attributes of Single-level Stores

The attributes of single-level stores that most affect their implementation are transparent persistence, the size of the store, and the implicit interface to persistent data. This section describes each of these aspects and discusses some of the implementation alternatives for addressing each of them.

4.1.1 Transparent Persistence

One of the primary benefits of a single-level store is transparent persistence — the ability for applications to manipulate data structures without regard to whether they are volatile or persistent. Single-level storage systems can provide varying degrees of transparency to programmers. For example, some systems automatically determine which objects are persistent via reachability from a persistent "root" (e.g., POMS). In these reachability-based systems, after specifying the persistent root, the programmer can be totally oblivious to the persistent status of objects because it is managed by the system. Another approach is to require that persistence be specified at an object's allocation time (e.g., RVM, Opal) by providing both persistent and volatile memory regions from which objects can be allocated. Systems such as the reachability- and allocation-based systems, which allow code

to be independent of the persistence of objects it operates upon, are said to provide *orthogonal persistence* [Cockshot et al. 84, Maier & Stein 86]. An alternative is to make persistence be an attribute of type [Carey et al. 86], but this does not provide orthogonal persistence (i.e., programmers have to define separate classes and methods for persistent and volatile objects that are otherwise identical). Systems that define persistence by reachability require support from the language or compiler, because determining object reachability requires a high degree of knowledge about types and object layout. In contrast, systems that define persistence at allocation time permit the storage manager to be ignorant of object storage details, thus avoiding dependence upon a particular language system.

4.1.2 Size of the Store

On 32-bit machines an address space spans 4 gigabytes. The inevitable loss of some of that address space to the application's code and volatile memory regions, the operating system, etc., significantly limits the size of single-level stores that use machine-native pointers as object references. Swizzling was devised in order to overcome this limitation [Singhal et al. 92]. A swizzling persistent object store can have an arbitrarily long "persistent name" for an object. An object's persistent name could be translated to its location in the buffer upon each reference, but doing this would severely reduce performance. Instead, a swizzling system replaces the persistent names of buffered objects with their addresses in the buffer. Subsequent accesses occur as fast as a normal pointer traversal. ObjectStore, QuickStore and Texas are examples of swizzling persistent object systems.

In the early 1990's, the size of virtual addresses in many microprocessors increased from 32 to 64 bits. These "wide address space" processors are available from Digital [Sites 92], Hewlett-Packard [HP 90], IBM [Sys/38], and more are introduced regularly. For all but the most wide-ranging address spaces, these processors can obviate swizzling because 64 bits is sufficient to address all of the magnetic disks likely to ever be made. It is certainly sufficient for a local area network, which is the scope of tightly-collaborative applications that persistent object systems most often support [Zdonik & Maier 90]. Other

architectural advances, such as inverted page tables [HP 90, Kane & Heinrich 92, Mot 93, Talluri et al. 95], have contributed to making direct support of wide, sparse address spaces feasible and efficient. Using machine-native pointers as object references has many advantages, which were thoroughly explored in the Opal project [Chase 95]. In addition to avoiding the overheads of swizzling, using machine pointers as object references allows clients on the same machine to share buffers. In contrast, in order to share buffered data in a swizzling system, each application that potentially wants to share data with another would have to coordinate the placement of swizzled data. The complexity added by this constraint has meant that all swizzling systems as of 1996 [Singhal et al. 92, Lamb et al. 91, White & DeWitt 94] maintain private swizzled buffer pools for each application on a machine. Thus, in these systems, each application accessing the same data has its own private copy of the data in its buffer.

4.1.3 Memory Interface to Application Objects

Explicit interfaces to persistent data, such as provided by filesystems and relational databases, require that applications call procedures to save and fetch data to and from persistent storage. In contrast, single-level stores provide an *implicit* interface to persistent data, in which load and store operations on application objects in memory are automatically satisfied by the underlying system.

The most general form of data structures in a single-level store are graphs: nodes are application objects and edges are references between them. Applications operate on these data structures by traversing the intra-object references and manipulating the contents of objects. Therefore the efficiency of traversal, data access, and manipulation are crucial for good performance. Because of this requirement, many persistent object systems — including Camelot, Cricket, ObjectStore, QuickStore, μDatabase [Buhr et al. 92] and BubbaOS — provide applications with memory-mapped access to database buffers.[1] Mapping buff-

1. Mapping buffers provides an implicit interface to persistent data, but it does not integrate buffer management with virtual memory — these systems still manage private buffer pools.

ers into an application's address space allows applications to access and modify objects as well as traverse references by using machine-native memory operations. The alternative — taken by POMS, Objectivity, GemStone, and Thor — is to add a layer of indirection between an application's accesses and the underlying buffers. The most common way to achieve this is by overriding the dereference operation to interpose buffer management code. Adding this layer of indirection increases the system's control during buffer misses and replacement, but at a significant cost to the performance of buffer hits, which are more critical for high performance in these systems' workloads [Eppinger 89, Shekita & Zwilling 92, Lamb et al. 91]. In addition, the memory mapping approach enables — but does not automatically provide — integration with virtual memory. For example both ObjectStore and QuickStore, which map database buffers into the application's address space, are not integrated with virtual memory because their buffers are in the databases' private buffer pools. As a result, these systems can suffer from double paging when competition for memory from other applications causes the virtual memory system to replace some of the database's buffers.

4.2 Transaction Management in Single-level Stores

The implicit interface to persistent data provided by single-level stores presents challenges for implementing transaction recovery mechanisms. This section describes why the buffer management and logging components of a recovery mechanism are difficult to implement in a single-level store, and how previous systems have used special operating system support to overcome these difficulties.

4.2.1 Control Over Buffer Management in a Single-level Store

The aspect of buffer management that most affects the implementation of single-level storage systems is the trade-off between transaction manager control and virtual memory integration. As discussed in Section 3.2, a transaction system coordinates buffer management and logging in order to implement recovery. A persistent storage system may also wish to

control the buffer replacement policy in order to improve performance. For example, relational database researchers have devised replacement policies that greatly outperform standard virtual memory policies for some workloads (e.g., DBMIN [Chou & DeWitt 85]). All of the research and commercial single-level storage systems listed in Table 3.1 use traditional locality-based replacement policies, such as LRU or its approximations (e.g., clock [Corbato 69] or FIFO with second chance [Turner & Levy 81]). Thus the primary demonstrated need for buffer management control in a persistent object store results from the constraints of implementing recovery.

Managing a private buffer pool provides full control over buffering at the cost of reduced integration with virtual memory and the attendant problems of double paging. In contrast, mapping persistent data fully integrates buffer management with virtual memory by yielding all control over buffer management to the virtual memory system. A succession of research operating systems have provided control over some aspects of virtual memory buffer management to applications that require more control than is provided by virtual memory's implicit interface. For example, Mach external pagers [Young 89], which were inspired by the operating system modifications required in the TABS system [Spector et al. 85], allow applications to control the source for fetching mapped data, and the destination for pageouts. This control is used by various systems to implement no steal transaction buffer management that is integrated with virtual memory (e.g., Cricket [Shekita & Zwilling 92] and Camelot [Spector 91]).

Some research systems provide control over virtual memory replacement policies [McNamee & Armstrong 89, Harty & Cheriton 92, Sechrest & Park 91], but these facilities have not yet been used to implement transaction recovery mechanisms. The high overheads of communication between applications and the virtual memory system incurred by these systems is being addressed by efforts to restructure operating systems to allow applications to directly manage their own virtual memory, either by linking application-specific code into the kernel, as in SPIN [Bershad et al. 94] and HiPEC [Lee et al. 94], or by pushing the implementation of virtual memory to user-level libraries where it can be modified by interested applications, as in Aegis [Engler et. al 95].

The additional control provided by research operating systems could be used to prevent double paging in a buffer pool system, by allowing a buffer manager to disallow page replacement for buffer pool pages [Chew & Silberschatz 92]. This approach could reduce or avoid double paging, but it would not provide the other benefits of integrated buffer management, such as more efficient access to persistent data and ease of sharing buffered data. Memory may also be less efficiently utilized by this approach because it is partitioned between the buffer pool and the virtual memory system, making it more difficult to effectively shift memory between them to adapt to varying demand [Ousterhout et al. 88, Nelson et al. 93].

4.2.2 Logging in a Single-level Store

Logging and buffer management combine to provide the recovery mechanism for a transaction system. The transparent persistence provided by single-level storage systems significantly constrains the methods of gathering logging information. Many systems gather logging information via hooks in a persistent programming language. This approach is taken by the Thor system's Argus language [Liskov 88], and by POMS with PS-Algol [Cockshot et al. 83]. Another approach is to integrate logging with buffer management. For example, QuickStore [White 94] keeps before-images of modified data in memory, and logging differences between them and the modified data at commit. A system that does not provide transparent logging is the Recoverable Virtual Memory system (RVM) [Satya et al. 94]. RVM applications must ensure that any data modifications within a transaction are preceded by calls to a `set_range` system primitive, which performs the logging. This approach could made transparent to programmers with language support to automatically insert the appropriate `set_range` calls. However, RVM requires large contiguous ranges to achieve high performance, and compiler support to achieve this without significant programmer direction has not yet emerged.

4.2.3 Operating System and Hardware Support

The degree of operating system support required by a single-level store's recovery mechanism often involves a trade-off between performance and portability. Currently, all of the systems that integrate transaction buffer management with virtual memory do so by taking advantage of specialized features available only in research operating systems. For example, Camelot and Cricket both provide integrated buffer management by using the Mach operating system's external pager facility to achieve no steal buffer management for mapped files (which normally provide steal buffer management). In addition to the performance drawbacks of no steal buffer management in a paging environment, these systems' use of the external pager interface prevents designers from using their techniques in other operating system environments.[2] Other systems implement transaction management in single-level storage systems by making custom modifications to the operating system [Boral et al. 90], relying on custom hardware [Cheriton & Duda 95, Herlihy & Moss 93], or both [Chang & Mergen 88]. In contrast, today's systems that use commonly available features of hardware or operating systems do not integrate their transaction buffer management with virtual memory, resulting in double paging (and hence poor performance) in memory-competitive environments.

4.3 Discussion

This chapter described some implementation techniques that existing systems use to provide the key attributes of a single-level store: transparent persistence, a large store and a memory interface. We also discussed the difficulties of implementing transactions on top of a single-level storage interface, and described the special facilities provided by some research operating systems to support transactions in this environment.

At the beginning of this chapter we noted that commercial single-level storage systems do not have access to the research operating system facilities that enable transaction recov-

2. Even the operating systems derived from Mach, such as Digital Unix, NeXT's OS and MkLinux have chosen not to provide the external pager interface.

ery in virtual memory-integrated single-level storage. As a result, these object-oriented databases do not implement the single-level storage interface that they provide to their clients by using the single-level storage provided by the operating system (i.e., mapped files). Instead, they implement private buffer pools on top of the explicit filesystem interface. Because they are not integrated with virtual memory, these systems can suffer from double paging in memory-competitive environments.

In the next chapter we describe a new recovery mechanism that supports single-level stores that are integrated with virtual memory, enabling them to perform well in memory-competitive environments. Further, this system uses facilities provided by commercial operating systems, so it may be used by commercial application designers. Finally, it provides steal buffer management, unlike previous research integrated buffer managers, which should further improve its performance in memory-competitive environments.

Chapter 5

A Recovery Mechanism for Single-level Storage Systems

This chapter describes the design and implementation of a recovery mechanism that fulfills a growing need: efficient transaction support for single-level storage systems in memory-competitive environments. This system, called user-level mmap_shadow (UMS), uses facilities provided by commercial operating systems to implement a steal / force transaction recovery policy that is integrated with virtual memory. Thus UMS combines the compatibility of today's non-integrated systems with the high performance in memory-competitive environments of research systems that integrate transaction buffer management with virtual memory.

In Section 5.1 we describe the decisions made in the design of UMS to address the issues, described in Chapter 4, related to building a recoverable single-level storage system. Next, Section 5.2 describes the operating system requirements and the implementation of UMS. In Section 5.3 we describe a single-level storage interface, called the Recoverable Memory Service (RMS), that we built on top of UMS's recovery mechanism.

Finally, Section 5.4 summarizes the benefits and attributes of UMS's support for recovery in a single-level store.

5.1 Design Decisions for Recovery in a Single-level Store

Since RMS provides recovery in a single-level storage environment, it faces all of the challenges of such systems that were discussed in Chapter 4. As a single-level storage system, it must provide access to a large persistent store via a memory interface, and allow applications to manipulate persistent data in the same manner as volatile data. As a recovery mechanism in a single level store, it must also coordinate logging with the operating system's virtual memory buffer management. Finally, it must satisfy these criteria while meeting our goal of compatibility with commercial operating systems. This section summarizes the design features of UMS and RMS that address each of these issues.

Support for the Single-level Storage Interface

Single-level stores provide a memory interface to their clients. To achieve this, RMS provides persistent segments to which applications can *attach*, and subsequently access by referencing the segments' memory locations. RMS implements the attach operation by mapping a file that stores the segment data into the client's address space, thus the memory interface for reads is implemented by the virtual memory system. RMS persistent segments can coexist with volatile segments provided by a language runtime heap. This arrangement permits applications to create persistent or volatile objects of any type by specifying the segment from which the object is allocated. RMS does not control references between persistent and volatile objects, so applications must ensure that the objects referenced by a persistent object are also persistent (to avoid "dangling pointers" caused by pointing at volatile objects that subsequently disappear).

RMS takes advantage of the large address spaces provided by modern 64-bit processors to implement a single-level store without needing to swizzle persistent references into machine pointers to buffered data. This decision simplifies our implementation as well as

making it more efficient, by avoiding the run-time costs of swizzling. Finally, by avoiding swizzling, we are able to isolate the interaction between recovery management and virtual memory from the significant impact that swizzling has upon virtual memory management [Narasayya et al. 96].

Support for Recovery

Previous single-level stores that are integrated with virtual memory have used facilities such as Mach external pagers to provide no steal buffer management by diverting virtual memory pageouts to a temporary file during transactions. As we discussed in Chapter 3, no steal buffer management is not as efficient as steal buffer management in the presence of frequent buffer replacement. UMS provides steal buffer management for recovery in persistent segments by mapping segment files into clients' address spaces with read-only protection. As we discussed in Section 3.2.2, steal buffer management requires that the log manager collect undo log information because a steal buffer manager can overwrite the before-images of data before a transaction is committed. UMS collects page-grain undo-logging information by using the "signal handler" facilities provided by commercial operating systems to write log records before each page can be modified. We describe the details of how UMS supports recovery in the next section.

5.2 The Design and Implementation of User-level mmap_shadow (UMS)

This section describes the design and implementation of a transaction system recovery mechanism, called user-level mmap_shadow (UMS), that provides write-ahead logging and steal buffer management that is integrated with virtual memory. Section 5.2.1 describes the operating system facilities required to implement UMS, and Section 5.2.2 shows how these facilities are used to maintain undo-logging information for updates in a single-level store.

5.2.1 Operating System Facilities Required by UMS

Previous transaction systems have provided efficient buffer management in memory-competitive environments by using special support provided by research operating systems. Since users want to run shrink-wrapped applications, their desktop machines run commercial operating systems instead of the customized operating systems that can be run on server machines. Therefore UMS implements buffer management using the following facilities that are provided by most commercial operating systems, yielding a transaction system that is both feasible and efficient in modern memory-competitive environments:

- *mapped files.* Mapped files are available in all current variants of Unix using the `mmap` system call. Similar functionality is available in Windows (95 and NT) [Custer 93], and will be available in Macintosh OS version 8.

- *virtual memory protection.* Most operating systems allow applications to control the access permission for regions of virtual memory. Virtual memory protection controls three actions: read, write, and execute. Some hardware does not allow some combinations (e.g., allowing execute but not read is uncommon). All that is required to implement UMS is the ability to specify between read/write and read-only permissions. When a write is attempted to read-only memory, the processor raises a protection-violation trap in the operating system, which forwards the trap to the UMS logging code via a software signal, described below. Applications can control virtual memory protection in Unix systems using the `mprotect` call. NT provides similar functionality with the `VirtualProtect` call.

- *software signal handlers.* These application-supplied procedures are invoked as a result of an operating system event, such as an attempt to write to a region of memory that has read-only protection. Unix applications provide signal handlers to the operating system using the `sigvec` system call. In NT, applications can define *exception handlers* to achieve a similar effect. In unprotected operat-

ing systems, such as Windows 3.1, Windows 95 and the Macintosh OS, this functionality is often achieved by patching the operating system using "hooks" provided for this purpose.

The next section describes how UMS uses these operating system facilities to gather logging information for updates in a single-level store.

5.2.2 Implementation Details

We implemented UMS in Digital Unix as a link-library. A client attaches to the persistent segment by calling the `mmap_shadow` library routine, passing as arguments the file to be mapped, the amount of data to be mapped, and the files to be used for the log and the metadata. This call maps the database into the client's address space with read-only protection, and installs a write-protection violation signal handler. After the client has attached, it can access the database as in a single-level store; by reading from and writing to memory locations.

The operating system's virtual memory mechanism satisfies reads to mapped UMS regions exactly as it would for a normal mapped file. Accesses to resident (i.e., buffered) data are satisfied at full memory speed without any added overhead. Access to non-resident data results in a virtual memory fault which the operating system satisfies by reading the page containing the data from the mapped file into the filesystem's buffer, and mapping that buffer into the application's address space. Our use of mapped memory avoids the overhead of the memory copy from the filesystem's buffer to the transaction system's buffer incurred by the buffer pool approach.

Writes to a UMS region result in a write-protection violation fault to the virtual memory system. Figure 5.1 shows the operation of the signal handler in response to write-protection violation signals. First, the signal handler writes the before-image of the page about to be modified to the log file. It also maintains a metadata file that stores the association of log file before images to database pages. Second, the handler synchronously writes the log and the metadata to disk to ensure they are persistent before the virtual

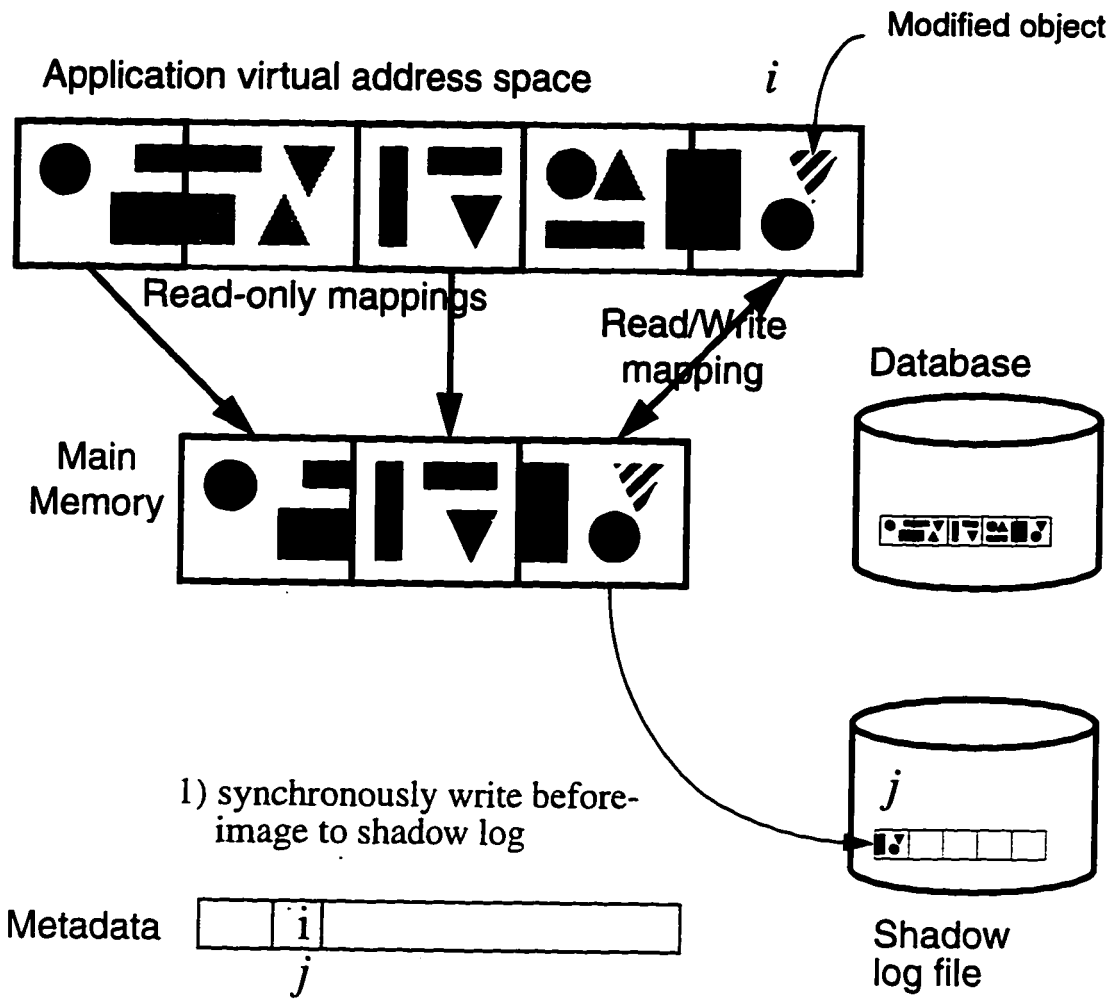


Figure 5.1: UMS write-fault processing for a fault on page *i*.

memory system could possibly overwrite the database page. Third, once the disk writes are complete, the signal handler calls `mprotect` to enable writes to the page, and returns from the signal handler which causes the operating system to resume the application.

This protocol ensures that a before-image of a page is written to disk before the page can be modified by virtual memory pageouts. Thus a transaction can be aborted by copying each of the pages saved by the signal handler back to the database file.

UMS writes the log file sequentially to avoid the high cost of disk seeks. In addition, we place the metadata on a separate disk to prevent log writes and metadata writes from interfering with each other. We could embed metadata into the log file to achieve sequential writes with one fewer disk [Rosenblum & Ousterhout 91], at the cost of some implementation complexity.

The next section describes how we used the functionality provided by UMS to implement a recoverable single-level storage system.

5.3 The Recoverable Memory Service: A Transaction System Built on UMS

The recovery mechanism provided by UMS is sufficient to implement atomic and durable transactions, but it lacks the operations expected by clients of a transaction system (e.g., *begin-transaction*, *commit*, *abort*). We built the Recoverable Memory Service (RMS) on top of UMS to support applications and benchmarks that require a transaction system interface. This section describes the interface provided by RMS, and describes how it uses UMS to implement atomic transactions.

5.3.1 The Structure of RMS

RMS is a user-level library that is linked into client applications. The RMS library is modularly structured to allow us to experiment with different approaches to buffer management. The structure of RMS is shown in Figure 5.2.

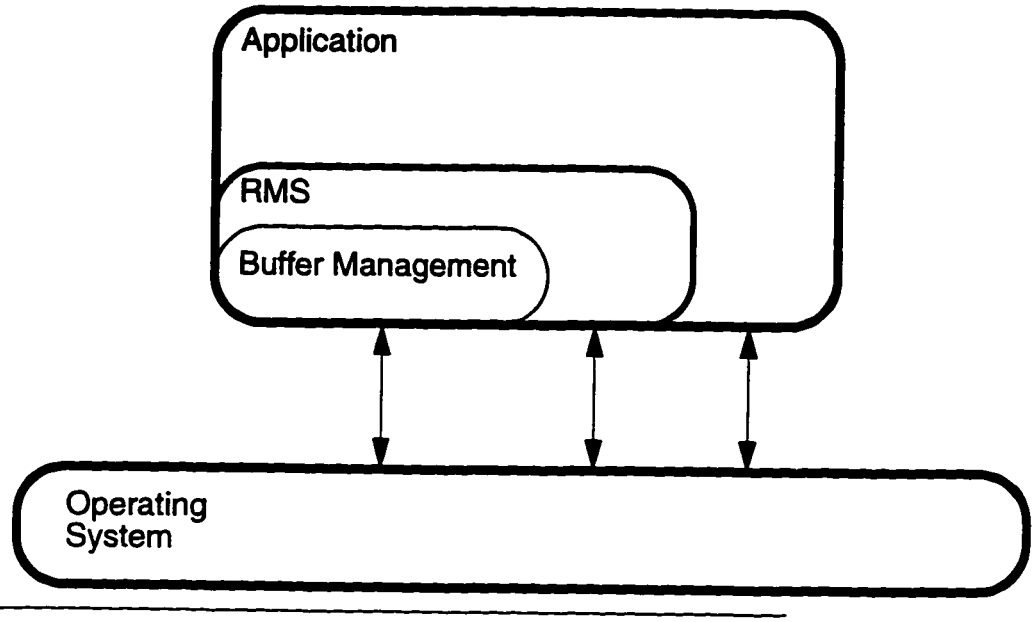


Figure 5.2: The Relationship between Applications, RMS and the Operating System

The figure shows that the RMS library isolates the buffer management module from the application. This allowed us to replace the buffer management module without changing the application. The figure also shows that all three components of the application — RMS, its buffer management, and the application itself — are able to access the resources of the underlying operating system.

5.3.2 The RMS Interface

Table 3.1 presents the application programmer's interface to RMS. Applications typically attach to the database during initialization (using `rms_attach`), then perform a series of transactions (delimited by `rms_begin_transaction` and `rms_commit`), and finally detach from the database (`rms_detach`) before exiting. Applications may cause a transaction to be aborted directly (`rms_abort`). Any uncommitted transactions are also aborted implicitly by the restore operation (`rms_restore`) after a system failure.

Table 5.1: RMS Application Programming Interface

RMS Routine	Description
`void * rms_attach (char *fileName)`	Maps a persistent segment's file into the application's address space and installs the logging signal handler for write accesses. A segment only needs to be attached once per program invocation. If the system previously crashed, `rms_attach` also restores the state of the segment by calling `rms_recover`. Returns the address of the mapped segment.
`void rms_detach (char *fileName)`	Removes any mappings from the named segment in this application. Any transactions on the database should be terminated (by calling either `rms_commit` or `rms_abort`) before `rms_detach` is called.
`void rms_begin_transaction (char *fileName)`	Starts a transaction within the named segment. A transaction is concluded by calling `rms_commit`
`boolean rms_commit (char *fileName)`	Commits a transaction. Any modifications to the segment between a call to `rms_begin_transaction` and `rms_commit` will happen atomically with respect to the database. Returns `true` on successful commit. If the commit fails, this call returns `false`. The caller should call `rms_restore` after a failed commit.
`boolean rms_abort (char *fileName)`	Aborts a transaction. Atomically discards any modifications to the segment since `rms_begin_transaction`, restoring the state to that before the transaction began. Returns `true` on successful abort. If the abort fails, this call returns `false`. The caller should call `rms_restore` after a failed abort.
`boolean rms_restore (char *fileName)`	Restores the segment to a consistent state by rolling back the modifications made by any uncommitted transactions. Returns `true` on successful restore. If the restore fails, this call returns `false`. The caller should call `rms_restore` again after a failed attempt.

5.3.3 RMS Implementation Details

This section describes the implementation of the RMS library routines. The first three perform bookkeeping operations. `Rms_attach` first calls `mmap_shadow`, which maps the file with read-only protection and installs the signal handler that performs undo logging for modifications to the file. Second, `rms_attach` calls `rms_recover` to recover from a previous crash, if necessary (`rms_recover` is a null operation if no recovery is needed). `Rms_detach` calls `munmap` to remove the database from the application's address space. `Rms_begin_transaction` initializes the metadata for the transaction.

The `rms_commit`, `rms_abort` and `rms_recover` procedures all interact with the buffer and log in order to perform their respective functions. The implementation of `rms_commit` is depicted in Figure 5.3. At the time an application calls `rms_commit`, modified pages of the current transaction may have been written to the database or may be buffered in memory. In addition, the log contains copies of the unmodified version of each page that has been modified. The goal of `rms_commit` is to atomically update the database and reset the log. To do this, `rms_commit` first writes each modified buffer page to the database file. After all of the writes complete, `rms_commit` commits the transaction by resetting the metadata. If the entire metadata structure is small enough to be atomically written to disk (we assume this size to be a virtual memory page, but may be larger or smaller), then it suffices to clear the metadata in memory and write it atomically to disk. If the metadata spans multiple pages, then we reset the metadata using two-phase commit [Gray 78], as follows. The metadata structure includes a flag, `being_cleared`. When the flag is true the metadata is considered empty. The operation of restoring the database after a crash checks the `being_cleared` flag, and if set, clears the metadata. Using this recovery protocol reduces transaction commit to the process of atomically setting the `being_cleared` flag. Once the `being_cleared` flag is written to disk, the other pages of the metadata can be cleared on the disk one at a time.

Figure 5.4 shows how `rms_abort` and `rms_recover` restore the state of the database to that of before any uncommitted transactions. This is done by copying the before-images (from the log) of each modified database page back to the database file. Because the data-

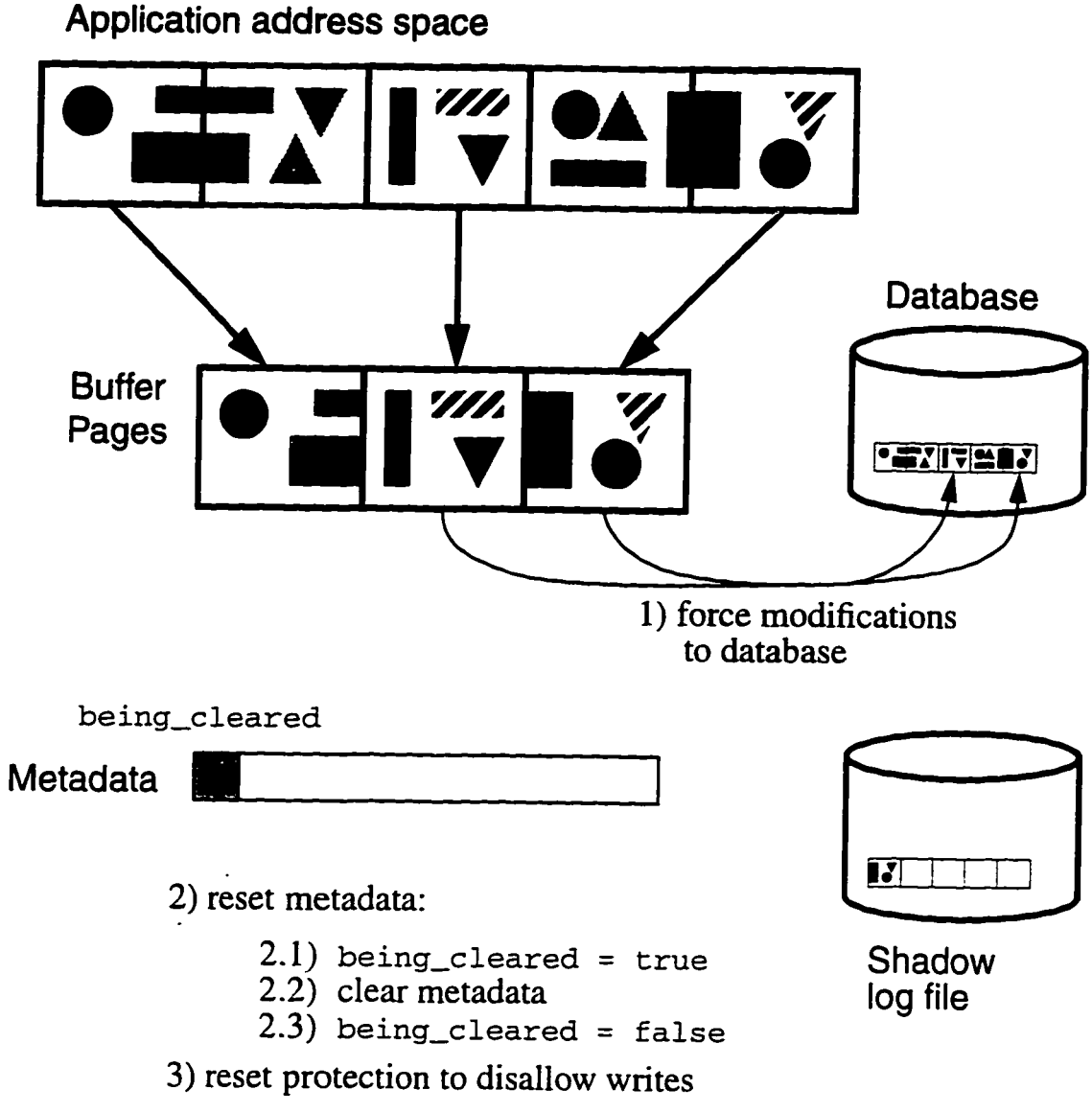


Figure 5.3: Operation of `rms_commit`.

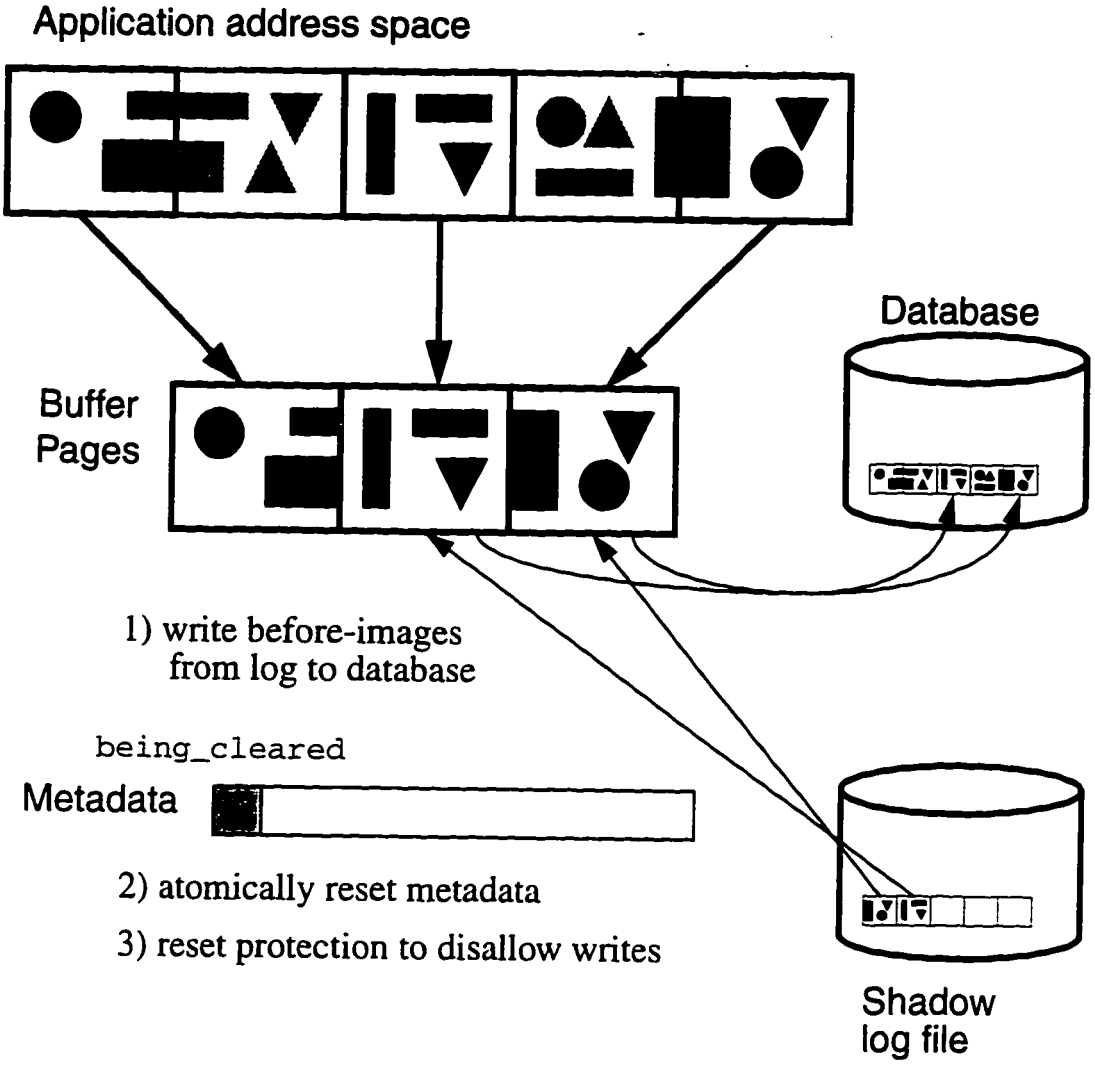


Figure 5.4: Operation of `rms_recover` and `rms_abort`.

base is mapped, the file writes of the log before images to the database restores the in-memory buffers to the pre-transaction state in addition to the database file. Once this is done, the database contents have been restored, and the procedure is completed by atomically resetting the metadata, which is done as described in the `rms_commit` procedure.

5.4 Summary

This chapter described the design and implementation of a new recoverable single level storage system. The system consists of two components. The first component is UMS, User-level Mmap-Shadow. UMS provides recovery information for updates in mapped memory. The second component is RMS, the Recoverable Memory Service. RMS provides a single-level storage interface on top of UMS's recovery mechanism.

The contribution of our design is that it combines the performance benefits of special support provided by research single-level storage systems, with the applicability to commercial environment of buffer pool systems. We demonstrated the latter point by implementing UMS and RMS on top of one commercial operating system, Digital Unix, and by isolating the operating system features it requires and determining their availability in other commercial operating systems.

We will demonstrate the former point—high performance—in the next chapter. We compare the performance of versions of RMS that are built with three different buffer management structures: UMS, buffer pool, and specialized kernel support. For read-only workloads, the design of UMS should allow it to provide identical performance to the systems that rely on specialized kernel support because they both integrate transaction buffer management with virtual memory. The performance of the integrated approaches should be better than the buffer pool approach for workloads without memory competition, due to avoiding the extra data copying caused by two levels of buffer management. With competition for memory, both integrated approaches will *significantly* outperform the buffer pool approach due to avoiding double paging. The performance of the three systems for read/write workloads will be closer, since all three systems must write modified buffers before

reclaiming them, as depicted in Figure 3.1, but will vary according to each system's overheads associated with handling updates (logging and commit).

The performance measurements in the next chapter confirm our expectations that UMS outperforms the buffer pool approach for a variety of workloads, and that it comes close to the update performance of systems that take advantage of special transaction management facilities provided by research operating systems.

Chapter 6

Evaluating Alternatives for Recovery in a Single-level Store

This chapter evaluates virtual memory alternatives for recovery in single-level storage systems. To do this, we built three representative buffer management alternatives for implementing recovery in a single-level storage environment. The platform we used was a Digital AlphaStation 250 workstation running Digital Unix 3.2. Currently this is a high-end workstation, but we expect similar systems to be common on users' desktops in a few years.

Our goals for the performance evaluation were to determine the performance impact of integrating buffer management with virtual memory, and to measure the further benefit of extensive kernel support for transaction recovery in single-level storage systems. To achieve these goals, we complemented the implementation of UMS described in the previous chapter with two additional implementations that represent the current extremes for recovery in a single level store. The first system implements user-level buffer-pool style buffer management. The second system implements a representative of "extensive kernel support" for mapped transactions. The workload we used to compare the three systems is

OO7 [Carey et al. 93], a standard benchmark for object-oriented databases that simulates the workload of an engineering design application.

This chapter first describes the two systems we compared against UMS, then describes the OO7 benchmark, and finally presents the performance results of the three systems with a variety of workloads. The results in this chapter demonstrate that UMS combines the performance of integrated research systems with the compatibility of user-level buffer pool systems.

6.1 Support Alternatives for Transactions in Single-level Storage Systems

This section describes the two systems we built to compare the efficiency of recovery support provided by the UMS mechanism described in Chapter 5. Existing alternatives for supporting transactions in a single-level store can be divided into two categories. The *kernel-based* approach utilizes significant support provided by the operating system to integrate transaction buffer management with the operating system's virtual memory. This approach is taken by many research systems, including Camelot, Cricket, Grasshopper, and RPVM. Utilizing operating system support can provide high performance, but it also has the disadvantage of not being available to application writers who are constrained to using facilities provided by commercial operating systems.

The *buffer pool* approach is for applications to privately manage transaction buffers using the facilities provided by commercial operating systems. This is the approach taken by existing single-level storage systems that provide transactions on commercial operating systems, including QuickStore [White & DeWitt 94], ObjectStore [Lamb et al. 91], O2 [O2 91], Gemstone [Maier & Stein 86], and Objectivity. Buffer pool systems' primary advantage is that they are compatible with commercial operating systems. Another advantage of the buffer pool approach is that it provides the application with full control over transaction buffer management, including buffer page size, replacement policy and recovery policy. One disadvantage of buffer pool systems is less efficient access to data than the integrated approach, because buffer pool systems incur an additional copy between the file

buffer and the buffer pool. Another disadvantage is that they provide worse performance in memory-competitive environments resulting from double paging.

We built representatives of both approaches to allow us to compare the performance of the two existing approaches for supporting recovery in a single-level store to our mechanism, UMS, which was described in Chapter 5. The remainder of this section is structured as follows: Section 6.1.1 describes the buffer pool system, which interacts with virtual memory in a manner similar to current state-of-the-art object-oriented database systems. Section 6.1.2 describes Recoverable Persistent Virtual Memory (RPVM [Chew et al. 93]), a system representative of extensive kernel support for recoverable memory in a single-level storage environment.

The three systems compared in this chapter — UMS, buffer pool and RPVM — can be characterized by how they fit into two overlapping categories: operating system support and integration with virtual memory, as shown in Figure 6.1. The performance evaluation in this chapter demonstrates that UMS combines the advantages of existing approaches: it offers the high performance of integrated systems as well as the compatibility of buffer pool systems.

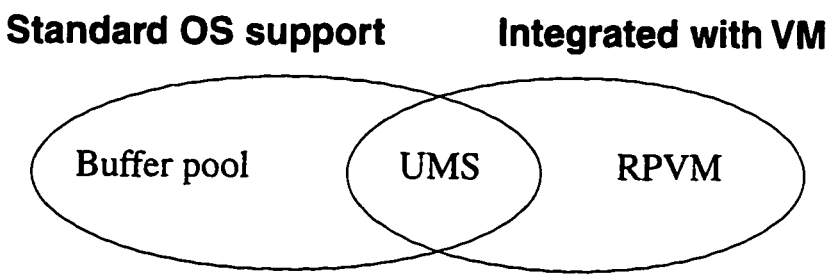


Figure 6.1: Classifications of the three buffer management systems

6.1.1 An Implementation of a Buffer Pool System

This section describes the implementation of the buffer pool system we used to compare against the other buffer management approaches. Current state-of-the-art page-grain

object database systems, such as QuickStore or ObjectStore, manage a private page-grain buffer pool at the application-level. We did not use these systems themselves for two reasons. First, commercial object database companies do not permit benchmark results to be published without considerable oversight [Carey et al. 93]. The second reason results from our goal of isolating the impact of the different systems' interactions between transaction memory management and virtual memory. This goal necessitates that we hold constant between the subject systems such issues as swizzling and recovery policy. For these reasons, we designed and built a buffer-pool single-level storage system based on the buffer management of state-of-the-art page-grain object databases. To hold other implementation variables constant, our buffer pool system uses the Alpha processor's 64-bit pointers as object references (instead of swizzling), and uses the same recovery policy as UMS.

The buffer pool management module we implemented is a plug-in replacement for UMS within our RMS library. In order to implement accesses and updates within an RMS persistent segment, the buffer pool database maps *buffer pages* that contain copies of pages of the segment, rather than mapping the segment itself (as UMS does). In our implementation, we represent the buffer pages as pages of a file, so that the buffer manager can name them.[1]

At initialization time (i.e., as part of the `rms_attach` procedure), the buffer manager creates the file used to name its buffers. The size of the file is chosen to be the size of the buffer pool, which is an initialization-time parameter. Next, the buffer manager provides to the operating system a signal handler for virtual memory protection violations. The handler is invoked for both read- and write-protection violation events. The write protection handler is similar to UMS's, described in Section 5.1, because the buffer pool system uses the same recovery policy. In addition to write-protection violation signals, the buffer pool system must also handle signals caused when the application attempts to access unbuffered data.

1. There are two ways for a user-level application to name portions of memory: by file name and offset for mapped files, or by virtual address for the remainder of the address space. Since the buffer pool system needs to refer to the buffers independent of their virtual memory address, it must use a file name and offsets within the file.

When a client of a mapped file references an unbuffered location, the virtual memory system's read-fault handler fetches the data from the mapped file and resumes the faulting application. In buffer pool systems, however, unbuffered pages have no valid mapping, thus the virtual memory system is unable to handle the fault. Thus buffer misses cause the virtual memory system to pass the fault on to the buffer manager's signal handler. The signal handler for a buffer miss resolves the fault by finding an unused page of its buffer pool, mapping it to the faulted location, reading the data from the segment's file into the new location, then resetting the protection to read-only so that writes to the page will be logged by the write fault handler.

If the buffer pool system can find no available pages to satisfy the fault, it reclaims a set of pages as described in Table 6.1:

Table 6.1: Page reclamation procedure

```
for (i = 0; i < pages_to_reclaim; i++)
begin
    p <- choose the next page to reclaim
    if p is dirty then write it to the segment file
    add p to the free page list
end for
```

The buffer pool system can use any of a large number of policies to choose the next page to reclaim. The policies we implemented for this study were Random, FIFO and FIFO with second chance (FIFO-SC) [Turner & Levy 81]. We discuss these policies in depth in Section 6.2.4

6.1.2 An Implementation of Extensive Kernel Support for Mapped Transactions

The other class of systems we want to evaluate is the class that utilizes extensive operating system support to implement transaction recovery in single-level stores. In contrast to the buffer pool approach, these systems allow virtual memory to perform the buffer management for the single-level store. These systems rely on extensions to virtual memory in

order to provide the transaction properties. Recall from Section 3.2 that when buffer replacement occurs during a transaction, *steal* buffer managers can overwrite the before-image with uncommitted data, and *no steal* buffer managers must either avoid replacing uncommitted data, or write uncommitted data to temporary storage. Section 3.2.1 described why *steal* buffer management is most efficient in the presence of frequent buffer replacement. Most research operating system support for transactions in a single-level store support *no steal* buffer management. The RPVM system [Chew et al. 93] is one system that support steal buffer management in a single-level store. RPVM is a set of modifications to the Mach [Rashid et al. 89] microkernel that support a wide range of transaction recovery mechanisms. We wanted to compare the other buffer management mechanisms to a steal recovery policy that was built upon the original implementation of RPVM, however, the source code to RPVM is not available. Further, even if it were, we would not be able to isolate the performance differences that were caused by the different operating systems (Digital Unix vs. Mach) and the issue we are interested in: the interaction between buffer management and virtual memory. Therefore, we designed a kernel mechanism based on RPVM's functions and implemented it in Digital Unix. The RPVM project did not implement a recovery mechanism using their system and thus could not compare their system to existing buffer management approaches; our comparison in Section 6.3.2 of RPVM's support for recovery in a single-level store with the buffer pool approach is therefore a contribution of this dissertation.

This section first describes the RPVM interface and then describes how it can be used to implement an improved version of the steal/force recovery policy provided by the UMS and the buffer pool systems. Finally, we describe the simplifications to the version of RPVM we built for our experiments.

Recoverable Persistent Virtual Memory

The RPVM system consists of extensive modifications to the virtual memory manager of Mach version 3.0, a research operating system from Carnegie Mellon University. RPVM provides an interface to transaction system recovery managers that allows them to control

which pages are written to persistent storage by the virtual memory page replacement mechanism, and *when* they are written. This control is expressed by allowing the application to provide *flush-before* rules between pages and to acquire and release *flush locks* on individual pages. Acquiring a flush lock on a page prevents that page from being *propagated* (i.e., paged out to the mapped file) until the flush lock is released. Flush rules constrain the ordering of pageouts between pages. For example, for pages A and B, the rule $A \leq_p B$ constrains A's propagation to occur before, or atomically with, B's propagation.

Flush locks can be used to implement no steal buffer management for transaction updates in a single-level store. A single-level storage system could be implemented by mapping the store into an application's address space with read-only protection, and providing a handler for write-protection violation signals. The signal handler would acquire a flush lock on the faulted page, then set the page's protection to allow the write. To implement the *commit* operation, the transaction manager would perform the log writes (or use the shadow page technique) required to install the modifications atomically.

Since we want to compare RPVM to UMS, which provides steal buffer management, we designed a recovery mechanism that uses RPVM's kernel support to provide steal buffer management. Figure 6.2 depicts the processing of a write-protection violation fault in our RPVM-based recovery mechanism. The segment is initially mapped with read-only protection, and the buffer manager provides a signal handler for write-protection violation signals. Thus when an application first writes to a page during a transaction, the read-only protection causes the signal handler to be invoked. The signal handler first copies the before-image of the page about to be modified to the buffer memory of the log file. Second, it updates the metadata in memory, to store the page's "logged" status for this transaction. Third, it adds two flush-before rules between the data file's page j, log page l, and metadata page m. The flush-before rules are $l \leq_p j$ and $m \leq_p j$. Given these rules, the RPVM kernel ensures that both the log page and metadata page are propagated before the database page's before-image is overwritten. This guarantee is sufficient to provide the persistent and atomic transaction commit provided by RMS, as described in Section 5.2. Fourth, it sets the protection on the data page to allow the update, and returns from the signal handler to resume the application.

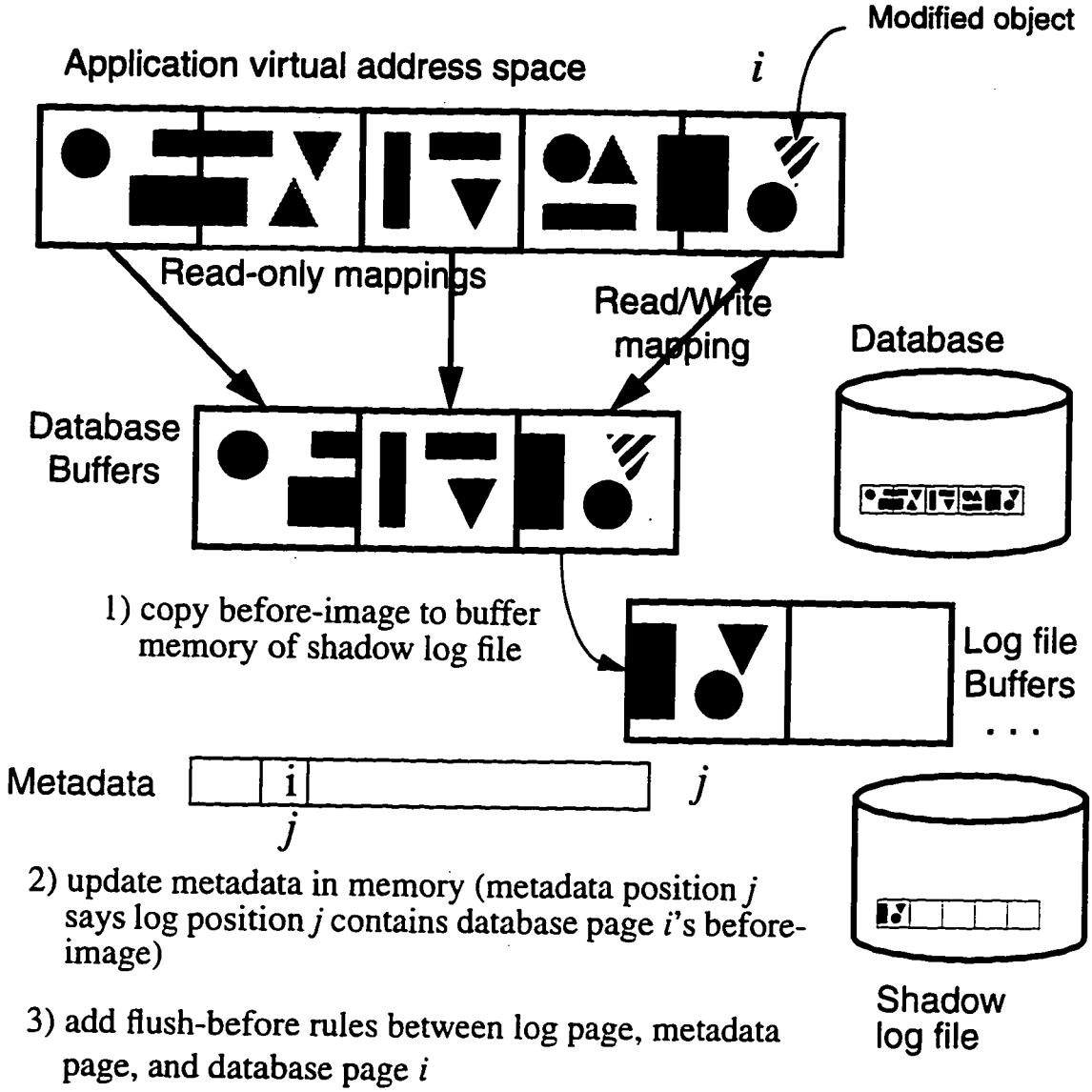


Figure 6.2: Write-fault processing for a fault on page *i* with RPVM kernel support for no steal buffer management

The key difference between our RPVM recovery mechanism and that of UMS is that RPVM's copy of the before-image to the shadow log is a memory-to-memory copy rather than a synchronous disk write as in UMS. Similarly, RPVM updates the metadata in memory rather than writing it synchronously to disk as UMS does. UMS needs to write the before-image and metadata directly to disk, because it needs to be assured that they are persistent *before* the before-image could possibly be overwritten by a virtual memory pageout. The RPVM operating system provides this assurance by enforcing the flush-before rules. This assurance means that both of the pages that have to be written to disk synchronously by UMS can instead reside in memory in an RPVM system, because the operating systems guarantees they will be paged out before the before-image can be overwritten by modified data.

Our RPVM recovery mechanism does not require all of the facilities provided by the original RPVM operating system interface. We took advantage of this to design a simplified version of RPVM for our experiments, which we describe next.

Simplified RPVM

We found that we could support our recovery mechanism by implementing a specialized subset of RPVM's functionality. This made the system substantially less complex, thus easier to implement. Table 6.2 summarizes the simplifications we made to RPVM in our implementation. The first simplification, which simplifies our implementation as well as improves its performance in comparison to the original RPVM, is that we do not allow flush rules to be transitive. Transitivity is not required for our recovery mechanism because flush rules are only inserted between database pages and log pages. Not implementing transitivity significantly reduces the complexity of virtual memory page replacement, because the number of rules constraining any individual page replacement decision is known without searching through a list, and can actually be strictly bounded.

The second simplification is that individual pages may have only up to two flush-before rules. In other words, a page x can only be involved in at most two rules of the form $i \leq_p x$. This simplification, combined with the third — that rules are only valid between

resident pages — further streamlines the processing required at page replacement time by allowing rules to be stored in the virtual memory system's resident page descriptor, `vm_page_t`.[2] The two-rule constraint is appropriate for our recovery mechanism because a database page has only two flush-before rules inserted per update: one for the log page, the other for the metadata page. Allowing flush-before rules only between resident pages is also appropriate, because the three pages, *l*, *m* and *d* are first accessed by the signal handler (the copy from *d* to *l*, and the update to *m*), so they are resident immediately before the flush rules are added, thus it is unlikely that any of them would be paged out in the intervening time. Correctness is never compromised, regardless of when pages are replaced, because `add_flush_rule` returns an error code when it is called on a nonresident page, thus allowing the signal handler to re-touch the pages and attempt to add the flush-before rule again.

The fourth simplification, not supporting flush locks, was enabled by our steal buffer management policy, which does not require flush locks. Our final simplification was to not implement RPVM's shared memory interface. Avoiding the shared memory interface simplified our implementation, as well as made it more robust (a read/write shared memory interface between user-level and operating system modules violates common system structuring rules). Finally, the results in Section 6.3 show that the granularity of page-based events resulted in reasonable overheads for a system-call based interface.

Table 6.2: Summary of simplifications to RPVM

RPVM	Simplified RPVM
Rules may be transitive	Transitive rule chaining not supported
Unbounded rules out of and into a page	2 rules out, unbounded rules into a page
Rules persist across pagein and pageout	Rules only valid for resident pages, cleared on pageout
Flush locks for no steal buffer management	No flush locks
Shared memory and system call interfaces	System call interface only

2. RPVM normally maintains a separate hash table, which is consulted during both pagein and pageout, adding 100 to 200 μsec per operation. The performance impact, however, was not as important to us as the complexity of introducing a new kernel data structure.

6.2 Experimental Methodology

Ideally, we would be able to compare the systems under consideration by incorporating them into a set of real-world applications and run them in realistic multiprogrammed environments. Unfortunately, a "standard suite" of applications that use a single-level store is not available. In fact, it has proven difficult to find the source code to *any* such applications that are in regular use. The kinds of applications needed for our comparison are common in industrial settings, but we found that they are important assets of the companies that use them, and thus are not freely distributable. Therefore we had to use a synthetic workload for most of our experiments. We chose OO7 [Carey et al. 93], a synthetic benchmark that has gained wide acceptance as a vehicle to compare object-oriented databases. To augment these results, we also compared the systems using an interactive rendering application [Chamberlain et al. 96]. This section describes the OO7 benchmark and the rendering application, the equipment they were run on, and the experiments we used to compare the buffer management and recovery alternatives under consideration.

6.2.1 The OO7 Benchmark

The OO7 benchmark is intended to simulate the behavior of a class of object-oriented database applications that includes Computer Aided Design (CAD), Computer Aided Manufacturing (CAM), and Computer Aided Software Engineering (CASE). The benchmark consists of a configurable database and a set of *traversals*. Each traversal is an application that accesses the database with a particular access pattern and update behavior.

Figure 6.3 shows the structure of a OO7 database. The OO7 database consists of a hierarchy of objects. The root of the hierarchy is the database itself, and it contains one or more *modules*. Modules are meant to represent something like a complex circuit assembly being designed with a CAD tool. Each module consists of a tree of *complex assemblies*, which are themselves sub-modules. Complex assemblies have pointers to the sub-modules it contains, until finally, at the leaves of the tree, they point to a set of *base assemblies*. A

base assembly contains a graph of *atomic parts*. Atomic parts are the basic units (such as resistors, capacitors, wires, etc., in an electrical circuit) that are used to build the module.

We set the size of the database from 24 to 192 Megabytes by varying the number of atomic parts per base assembly.

We linked the OO7 application, which was written in C++, to each of the three versions of our RMS library (one per buffer management technique). Thus the OO7 code was identical in each of the applications being measured and only the buffer management code differed between them. OO7 consists of a set of traversals that have different access and update behaviors. We chose the T1 and T2b traversals of OO7 as the basis of our evaluations because they best highlight the differences between the buffer management techniques used. The T1 traversal examines all composite and atomic parts in the database using a depth-first traversal. Thus, T1 measures the efficiency of read-only database accesses. T2b also traverses the tree depth-first, however it modifies each atomic part as it is visited. T2b was therefore used to measure the performance of the systems in response to frequent updates.

6.2.2 The Render Application

In addition to OO7, we compared the systems by comparing their performance running an interactive rendering application [Chamberlain et al. 96]. This application does not modify the data, thus it is similar to the OO7 T1 traversal, however its computation, access pattern and data layout are significantly different from OO7. Our experiments with this application measured the amount of time required to render a single frame, which consists of traversing an octree that represents the scene being viewed. The scene is represented hierarchically by inscribing it within the octree. Each non-empty cell of the octree is successively subdivided into a sub-octree, until the geometric contents of a cell are simpler than a fixed threshold. To render a frame, the octree is traversed to select the cells that are suitable to be drawn based on their depth in the viewer's field-of-view; the portions of the scene that are closer to the viewer require more detail, thus deeper traversal, than portions that are further away.

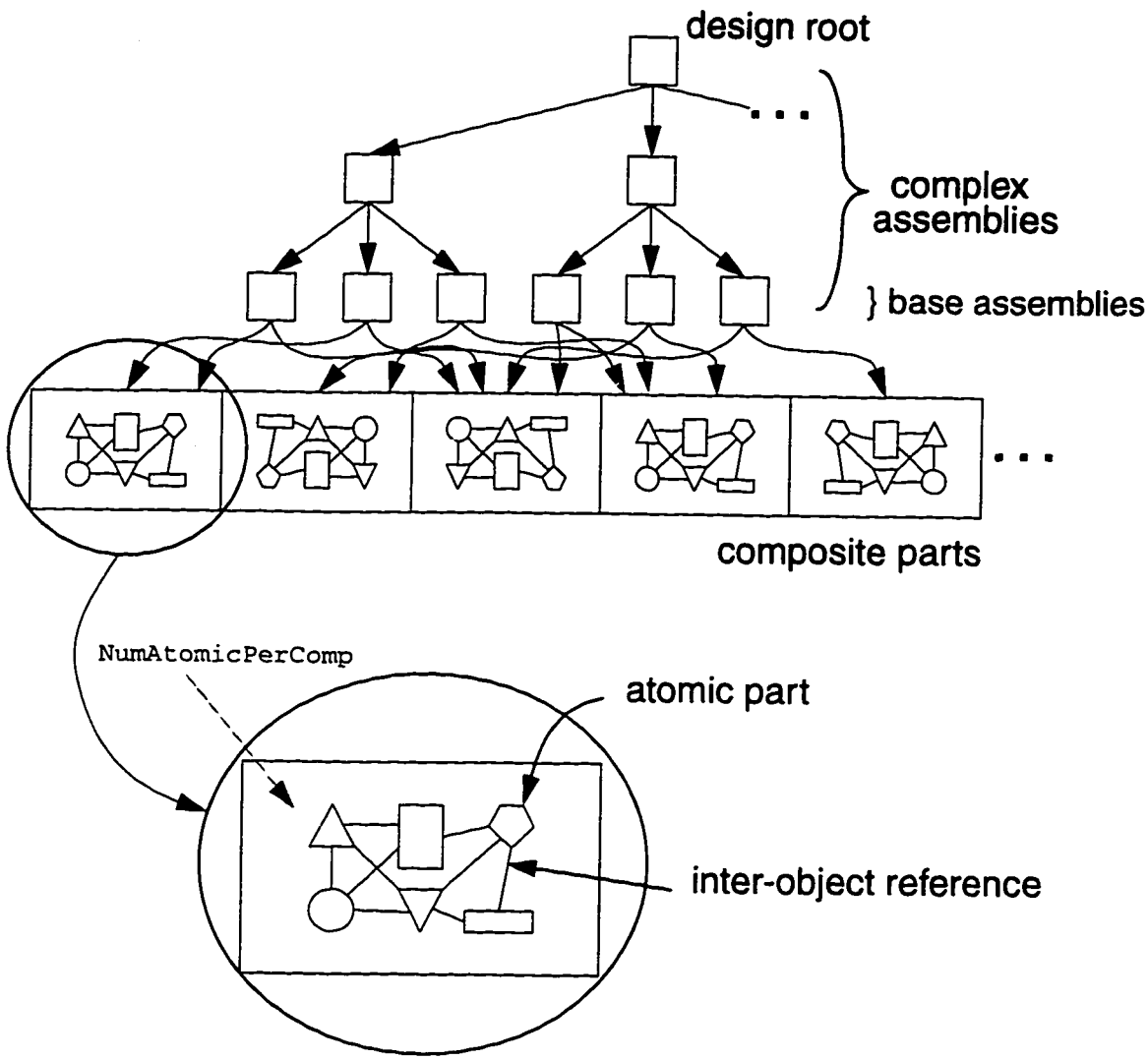


Figure 6.3: OO7 database structure

6.2.3 Hardware and Software Configuration

The experiments were run on Digital AlphaStation 250 workstations. These machines have Alpha 21064A processors running at 266 MHz and were configured with 128 MB of RAM and four hard disk drives. The operating system was Digital Unix version 3.2. The virtual memory page size was 8K bytes. We used version 2.6.3 of the GNU C/C++ compiler to compile the application and RMS libraries.

We found we had to modify two of Digital Unix's virtual memory parameters because both UMS and the buffer pool systems use large, sparse mappings and discontiguous regions of virtual memory protection. The first parameter we had to change was the maximum number of entries allowed in an address space. This parameter, `MAPENTRIES`, is normally 200. Since each resident page can potentially have its own mapping, we found we had to increase this limit to the maximum number of pages in the buffer pool (16,000 for our experiments). The second parameter we had to change was the maximum size of the `vpage` array. The `vpage` array stores the virtual memory protection of mapped pages. Thus the `vpage` array bounds the maximum size of a single mapping. Since UMS maps an entire database at attach time, the `vpage` array had to be increased from its original 16*1024 page limit (128 MB, since each page is 8Kbytes) to 24*1024 pages in order to accommodate our 192 MB database.

The last system-level optimization used for these experiments is that both the buffer pool and UMS systems used *raw disk* partitions for their logs. Raw disk partitions are regions of a disk that are not managed by the normal filesystem, but rather are accessed directly by the application. Using raw disk for their logs allows these systems to bypass the file buffer and avoid disk seeks otherwise incurred by synchronous log writes. Bypassing the file buffer allows more data pages to be buffered, thus increasing the buffer hit rate. Writing the log to raw disk also avoids a disk seek that would otherwise be needed to update the filesystem's metadata for the log. Since our recovery mechanisms maintain their own log metadata, this seek is needless overhead. These optimizations significantly improve the performance of these systems for large database accesses. We feel that using raw disk is a realistic optimization because recent commercial filesystems provide the

same functionality within the filesystem interface (e.g., SGI's XFS direct I/O facility [Sweeney et al. 96]), and future filesystems should include similar facilities. The RPVM system did not need to use raw disk for its log, since its location in the kernel allowed it to bypass the file buffer and avoid seeks using normal files.

6.2.4 Parameter Settings for the Buffer Pool System

Because they replace the operating system's buffer manager with their own, buffer pool transaction systems do not specialize an existing buffer manager in order to implement the buffer management policy they need. Instead, buffer pool systems implement an entire buffer management system, which entails making a number of decisions that are otherwise peripheral to the transaction buffer management policy (the steal / no force recovery policy used by all of the systems). Even though these decisions do not affect the recovery policy, setting them properly is critical to providing high performance. This section describes the procedure we used to choose the most efficient data granularity (buffer page size), the buffer replacement policy and the size of the buffer pool.

Buffer Granularity

A natural choice for buffer granularity is to use the same granularity as the virtual memory system. Digital Unix uses an adaptive granularity for data fetches, depending on the memory access pattern. By default, memory faults are serviced by reading single 8 kilobyte pages at a time. When the system detects sequential accesses, however, it uses clustered read-ahead of 64 kilobytes at a time. Therefore, the access pattern of our OO7 application affects the granularity of virtual memory reads. To choose the most efficient granularity for our buffer pool database, we compared the run-time of the OO7 T1 traversal for both 8 kilobyte and 64 kilobyte buffer pages, which are the two units of transfer used by Digital Unix. The results are summarized in Figure 6.4. For small databases, page size does not drastically affect performance. For larger databases, large pages provided better perfor-

mance because they amortize signal handling and fixed disk costs over more data transferred. These results prompted us to use 64 kilobyte buffer pages.

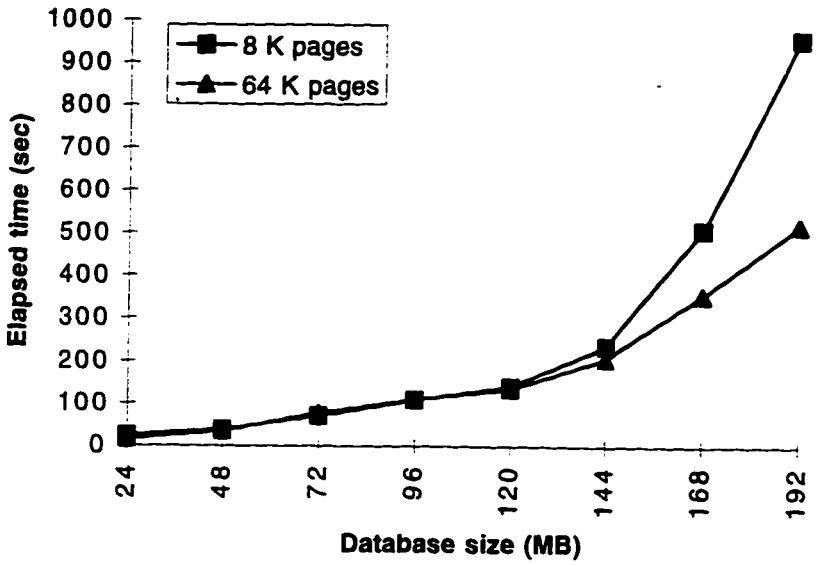


Figure 6.4: Buffer pool page size comparison for OO7 T1 traversal
For database sizes that allow the filesystem buffer cache to coexist with the transaction system's buffer pool (which was set to 96 MB), the filesystem's clustering optimizations enable the 8 K page system to perform as well as the 64 K page system. For larger databases, the size of the buffer pool interfered with the filesystem's optimizations, allowing the manual clustering performed by the 64 K page system to perform better.

Buffer Replacement Policy

We implemented three replacement policies for the buffer pool system: FIFO, FIFO with Second Chance (FIFO-SC) and Random. As their names suggest, FIFO replaces pages "first-in, first-out," and Random chooses pages to replace randomly. FIFO-SC is an approximation to the "least recently used" (LRU) policy, which takes pages' reference patterns into account when making replacement decisions. Specifically, LRU chooses as the next page to replace the page that was last referenced the longest time in the past. The ideal implementation of this policy sorts pages by their last time of reference and picks the

"oldest" page. Since hardware has not yet provided timestamps for memory references,[3] most virtual memory page replacement policies implement approximations to LRU. FIFO-SC is an approximation to LRU that was invented to accommodate the VAX architecture's lack of a reference-bit. This policy is also convenient for user-level buffer pool systems because most user-level applications do not have access to reference bits regardless of whether the hardware provides them. FIFO-SC is also Digital Unix's default page replacement policy.

The FIFO-SC policy keeps pages on one of three FIFO queues. The first queue, the *active list*, contains pages with full access protection enabled. When a page reaches the head of the active list, the buffer manager (at the user-level or in the kernel) protects the page to disallow all accesses and adds it to the tail of the *inactive list*. When a page reaches the head of the inactive list, it is paged out (if it is dirty) and added to the *free list*. If an application references a page while it is protected and on the inactive list, the page gets a "second chance" at remaining resident by having its protection reset to the original state and being put back on the active list. This policy ensures that frequently accessed pages are kept in memory because they get referenced while on the inactive list, while pages no longer in use get paged out after working their way through the two queues. The attribute of FIFO-SC that most affected its performance in these experiments is that while pages are on the inactive list, accesses to them cause virtual memory page faults, resulting in additional inefficient communication between the operating system and the buffer pool system.

The other policies we implemented, Random and FIFO, do not incur additional communication while pages are buffered. Instead, the Random and FIFO policies keep buffer pages on a single list, and choose the next page to replace via random selection or first-in, first-out, respectively.

The question we wanted to answer was whether the implementation overheads of FIFO-SC are overcome by the benefits of its approximation to LRU for OO7's access patterns. Figure 6.5 compares the elapsed time for the OO7 T1 traversal for each of the three

3. Software TLB handlers [Kane & Heinrich 92, Sites 92] can be used to provide fairly accurate reference timestamps. Even so, the approximations to LRU have proven effective enough that the extra bookkeeping a TLB-based time stamp approach would require has deterred its use for virtual memory page replacement.

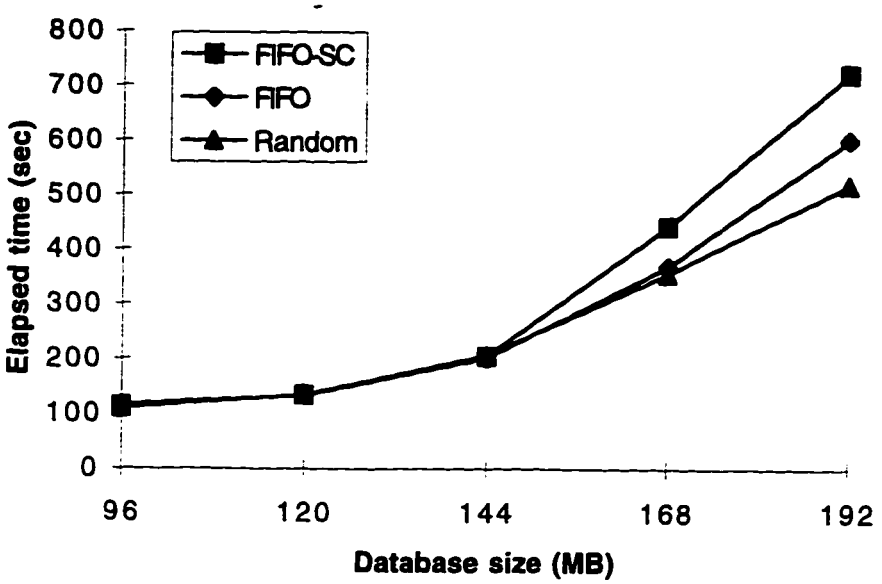


Figure 6.5: Buffer pool replacement policy comparison for OO7 T1 traversal
For databases smaller than 144 megabytes, which do not cause frequent replacement in the buffer pool, different replacement policies perform approximately the same. When paging does occur, Random performs better than FIFO and FIFO-SC because it achieves reasonable buffer hit rates with low signal-handling overheads.

replacement policies for database sizes ranging from 96 megabytes to 192 megabytes. The results for T2b traversals were similar. As we expect, replacement policy does not matter for database sizes that do not cause significant amounts of buffer replacement. When buffer replacement does occur, Random outperforms FIFO because it incurs fewer virtual memory faults (FIFO's buffers were paged out due to interactions with Digital Unix's replacement policy). Random avoids the overheads incurred by FIFO-SC resulting from the added signals used to detect references. For the OO7 T1 traversal the reference information gathered by FIFO-SC only slightly reduced the number of buffer misses. Those savings were outweighed by the increased overheads of FIFO-SC. Therefore, we used Random for the subsequent experiments.

Buffer Pool Size

The final tuning parameter is the size of the database's buffer pool. To determine the optimal buffer size, we measured the elapsed time of the OO7 T1 traversal for databases between 24 and 192 megabytes while varying the size of the buffer pool from 16 megabytes to 128 megabytes. The buffer page size was 64 kilobytes and the replacement policy was Random for all configurations. The results of this comparison are shown in Figure 6.6. We found that performance was best for moderate-to-large buffer pools (i.e., 96 to 128 megabytes). We chose the smallest of these acceptable buffer pool sizes (96 megabytes), because in the experiments to come, particularly those with competition for memory, the smallest possible buffer pool sizes are advantageous.

6.3 Performance Results

We measured the performance of the three systems both with and without memory competition of the read-only and the write-intensive OO7 workloads. As a result of our tuning experiments, we used the Random buffer replacement policy with 64 kilobyte buffer pages and a buffer size of 96 megabytes (12,000 machine pages). There are two axes of comparison in this section, corresponding to the two overlapping classifications shown in Figure 6.1: integration with virtual memory and operating system support.

This section first presents experiments that use the OO7 T1 read traversal with and without competition for memory in order to distinguish between the performance of the two integrated systems[4] and the buffer pool system. These read-only experiments isolate the performance benefits of integrating transaction buffer management with virtual memory. Next, we use the T2b write traversals to highlight the efficiency of updates between each of the three systems. This latter comparison measures the performance benefits of providing significant operating system support for transaction buffer management.

4. UMS and RPVM perform identically for read-only workloads such as the T1 traversal because they differ only in how they manage updates.

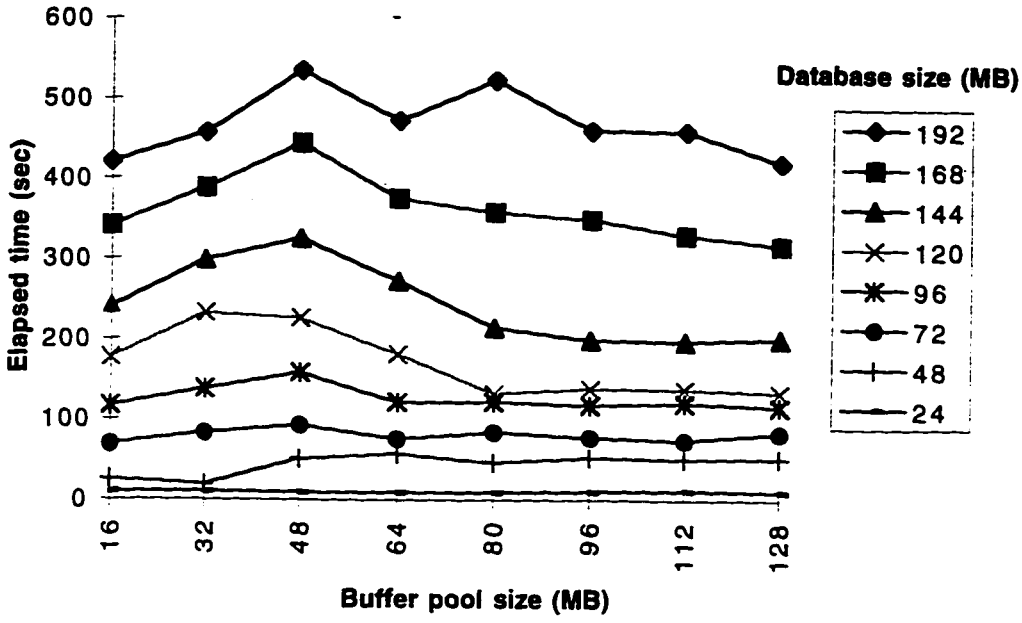


Figure 6.6: Buffer pool size comparison for OO7 T1 traversal

For small databases (the lines near the bottom of the graph), buffer pool size has little effect on performance. For larger databases, small buffer pools (less than 48 megabytes) allow the filesystem buffer to coexist with the transaction system's, thus allowing the filesystem's own buffer management to provide high buffer hit rates. The "hump" in the middle of the graph results from competition between the buffer pool and the filesystem's buffer. The medium-sized buffer pools (48 to 80 megabytes) are too small to provide high hit rates, and they force the filesystem's buffer to be too small to provide high hit rates, thus resulting in poor performance. Finally, moderate to large buffer pools (96 to 128 megabytes) are large enough to provide high hit-rates and corresponding high performance. (The anomalous performance of the 80 MB buffer pool on the 192 MB database appears to result from higher read costs due to seeks caused by an interaction between the access pattern, the replacement policy and the buffer pool size.)

6.3.1 Read Efficiency With and Without Competition for Memory

We compared the performance of the systems running the OO7 T1 read-only traversal with various amounts of available memory. This workload is intended to mimic the behav-

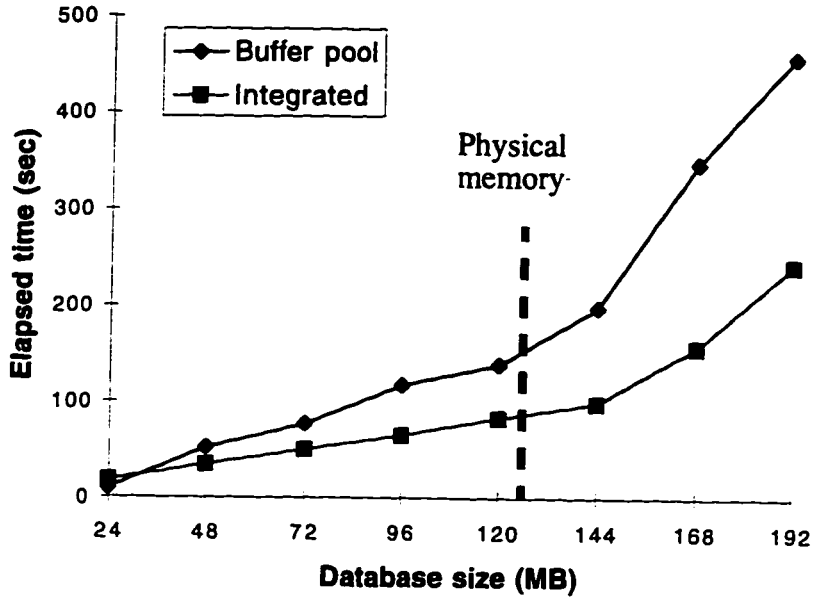


Figure 6.7: OO7 T1 traversal of 24 to 192 megabyte databases with buffer pool and integrated buffer management

ior experienced by a user using an application to browse a large database, concurrently with running a varying number of other memory-consuming applications. We simulated the competition workload, rather than run actual applications in the background, because we wanted to isolate the memory effects of such competition. Using applications to provide memory competition would make it difficult to isolate the memory component of application competition from the other overheads they impose, such as competition for I/O and for the CPU.

Our first comparison is between the buffer pool system and the integrated systems for the OO7 T1 traversal of databases ranging from 24 megabytes to 192 megabytes without memory competition. Figure 6.7 shows the result of this comparison. The results demonstrate the performance impact of the higher overhead for buffer misses in the buffer pool system. The higher overhead is reflected in a steeper slope for the buffer pool system than the integrated system. Both systems incur more frequent buffer misses (and correspondingly steeper slopes) for the databases that do not fit in available memory, larger than 120

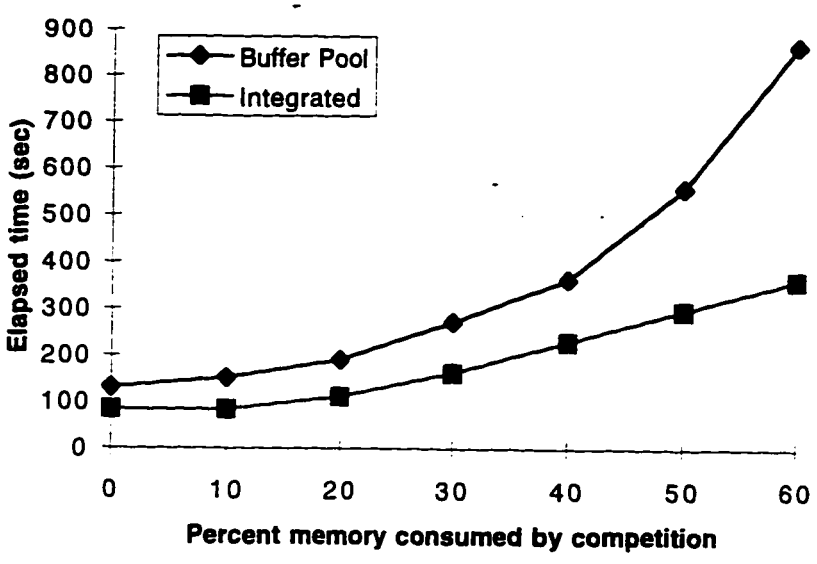


Figure 6.8: OO7 T1 traversal for 120 megabyte database with increasing memory competition

megabytes, but the figure shows the buffer pool's slope remains steeper than the integrated system's slope. This figure shows the efficiency advantages of memory-mapped access to persistent data, independent of issues of memory competition.

Our second experiment compares the buffer pool and integrated systems running the OO7 T1 traversal with a 120 megabyte database, as competition for memory varies from 0% to 60%. Figure 6.8 shows the result of this experiment. The left-most points, those with 0% competition, correspond to the performance of the two systems in Figure 6.7 for the 120 MB database. As competition increases, we see that the two systems' performance degrades differently. The integrated systems' performance degrades linearly, from increased buffer misses, once the database's working set no longer fits in available memory. The buffer pool system, on the other hand, degrades worse than linearly as competition grows, resulting from significantly increased virtual memory page faults due to double paging.

We repeated these read-only experiments using the render application. The dataset size of this application is not as easily controllable as OO7, so we compared the systems with only two datasets: a 50 megabyte scene and a 200 megabyte scene. The results without

competition are summarized in Table 6.3. The results as competition for memory

Table 6.3: Render application results without competition for memory

Scene size	Buffer pool	Integrated
50 megabytes	61.8 sec	29.6 sec
200 megabytes	101.5 sec	62.5 sec

increases for the two dataset sizes are shown in Figure 6.9 and Figure 6.10. The performance of the render application is similar to the results of OO7 for the same experiments, except that the integrated system's performance is affected by competition less than it was for OO7. Because render's viewer-adaptive traversal accesses a smaller proportion of the database than the OO7 traversals do, the decreased amount of memory as competition increases does not increase the buffer miss rate as much as it does for OO7. This does not translate into similar improvement for the buffer pool system. Memory accesses in the buffer pool system are more affected by the buffer allocation and replacement policies, which remain the same between the OO7 and render experiments. Therefore, we see

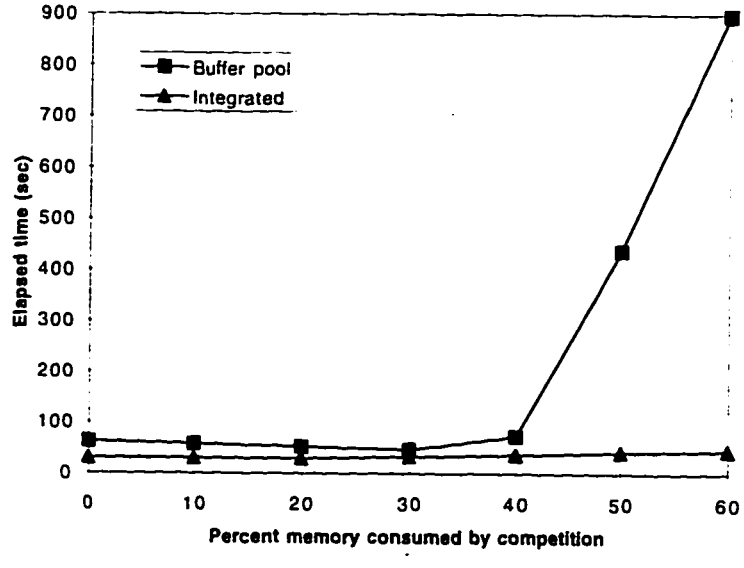


Figure 6.9: Performance of render application on small (50 megabyte) dataset, with increasing competition for memory.

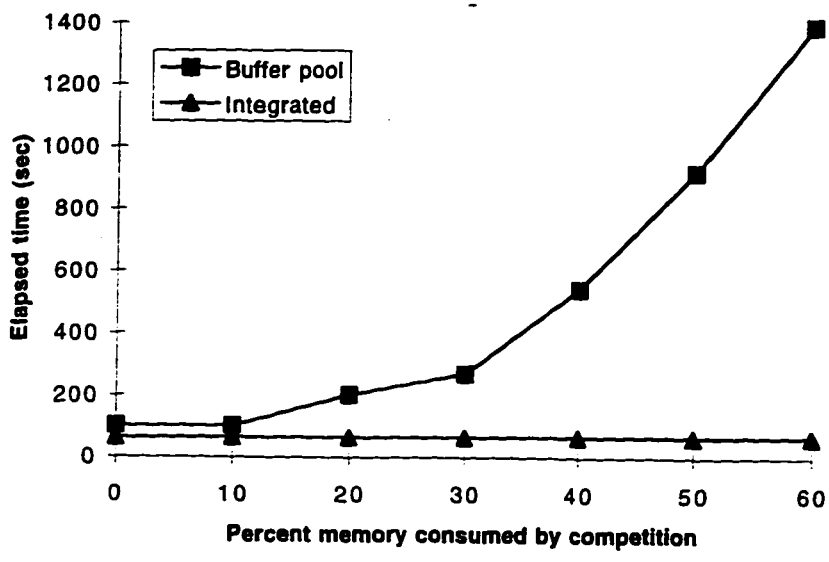


Figure 6.10: Performance of render application on large (200 megabyte) dataset, with increasing competition for memory

extreme performance degradation as memory competition causes the buffer pool system to suffer from double paging.

6.3.2 Efficiency of Support for Recoverable Updates

The previous experiments distinguished the performance of the integrated and the buffer pool systems for read accesses. The next experiment uses the write-intensive OO7 T2b update traversal to evaluate the efficiency of support for recovery of all three systems. This workload causes the most recovery processing of all of the OO7 traversals, thus magnifying the differences in the efficiency of updates in the three systems.

Figure 6.11 shows the performance of UMS, buffer pool and RPVM for the T2b update traversal for database sizes between 24 and 192 megabytes. For the database sizes that can be accommodated in memory — those smaller than 120 megabytes — RPVM performs significantly better than either the UMS or buffer pool systems. RPVM's advantage is a result of not having to synchronously write before-images (or metadata) to the log

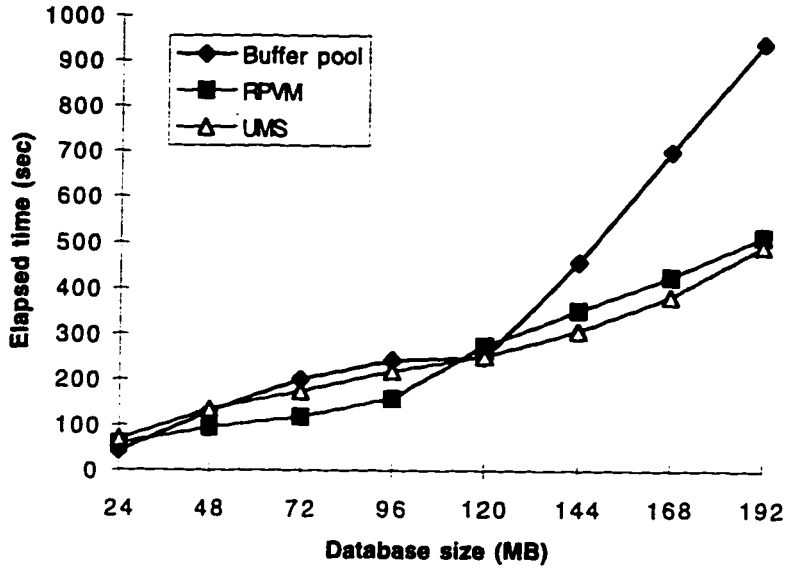


Figure 6.11: OO7 T2b traversal of 24 to 192 megabyte databases with buffer pool, UMS and RPVM

before allowing updates. By delaying such updates, RPVM can overlap writing of before images with processing and reading data, thus improving its efficiency. UMS and the buffer pool system perform very similarly for these database sizes because their behavior for these workloads is identical: the first write to each page causes the operating system to execute their signal handler, which writes the page's before-image to the log and updates the metadata.

For databases that do not fit in memory, the buffer manager must write the dirty buffer pages to backing storage in order to reclaim buffers. In this case, UMS is more efficient than the buffer pool system, because it is able to take advantage of the operating system's page cleaning optimizations that are not available to user-level buffer managers. In addition, the buffer pool system must write modified buffers through the filesystem interface, which adds data-copying overheads as well as increases competition for memory. Somewhat surprisingly, UMS also outperforms RPVM for large databases. To understand this behavior, consider the difference of these systems' management of log pages. RPVM's advantage for small databases is that log pages are not synchronously written to disk, but

instead are copied to memory. For large databases, RPVM's buffered log pages consume memory that could otherwise be used to buffer the database and reduce buffer miss rates. In contrast, UMS writes its log pages directly to disk, so log records never occupy buffer space. Figure 6.11 shows that for large databases, RPVM's benefits of delaying log writes are overcome by the costs of increased buffer misses. An RPVM system could improve performance by adapting its logging mechanism according to database size or paging load. We explore this approach in the next chapter.

6.3.3 Summary

In this chapter we describe, for the first time, a "level playing field" performance comparison among three approaches to supporting transaction buffer management in single-level storage systems.

The first approach is RPVM, Recoverable Persistent Virtual Memory, which requires the extensive customized kernel support provided by research operating systems but not available in commercial operating systems. We expect, and our experiments confirm, that this approach provides good performance, at the price of limited applicability.

The second approach is buffer pools, which involves application-level buffer management—the approach common in practice today, because prior to our work it was the only viable approach on commercial operating systems. We expect, and our experiments confirm, that this approach suffers significant performance penalties in a multiprogrammed environment.

The third approach is the system architecture described in Chapter 5, User-level Mmap-Shadow (UMS). This approach uses facilities available in commercial operating systems to integrate transaction buffer management with virtual memory. We expect, and our experiments confirm, that this approach provides good performance a wide variety of workloads, including multiprogrammed environments.

To summarize, the contributions of this chapter are:

- A performance comparison of the three systems—UMS, RPVM and buffer pool—across a variety of workloads, including read-only and read/write, and with varying degrees of competition for memory.
- An implementation of all three buffer management approaches in a consistent environment, thus minimizing the performance differences attributable to reasons other than the interaction between buffer management and virtual memory. This implementation provides a "level playing field" for comparing the systems.
- The implementation and performance measurements of UMS, which provide an existence proof that the compatibility benefits of buffer pool systems can be combined with many of the performance benefits of systems that use extensive kernel support to integrate transaction buffer management with virtual memory.

Our experience with these systems shows that integrating buffer management with virtual memory can simplify the design of such systems by allowing the virtual memory system to provide buffer management in lieu of an application-level buffer manager. The performance evaluation in this chapter demonstrates that the integrated systems perform as well or better than the buffer pool system for read-only workloads without memory competition, and perform significantly better for workloads with memory competition. For write-intensive workloads, the integrated systems again outperformed the buffer pool system. Further, the extensive kernel support provided by RPVM provides significant performance improvements for small databases, but for large databases UMS outperforms RPVM because it does not buffer log pages. Thus we conclude that UMS provides effective support for transactions in a single-level store which combines the performance benefits of previous research systems with the compatibility benefits of existing buffer pool approaches.

In the next chapter we describe some related work, discuss some possible performance and functionality enhancements to UMS, and offer some concluding remarks.

Chapter 7

Contributions, Related Work and Future Directions

7.1 Contributions

Until recently, transaction systems were almost exclusively deployed as components of large server systems, such as relational database servers. However, transactions and single-level stores are increasingly used as application development aids, which has resulted in an increasing number of transaction systems that are run outside the dedicated server environment: on desktop machines and multi-use enterprise servers. The implementation techniques of current transaction systems make the assumption that the application has dedicated memory. Competition for memory outside the dedicated server environment violates the assumption of dedicated memory, resulting in poor performance for the many applications that use embedded transaction systems.

In this dissertation, we demonstrated that the performance of transactions in memory-competitive environments can be greatly improved by integrating transaction buffer management with virtual memory. We have explored two degrees of operating system support required to achieve integrated transaction buffer management:

- The existing approach: employing extensive kernel support from research operating systems, in which the virtual memory replacement policy is modified to take into account the committed status of modified transaction data and the relationship between modified pages.
- Our new approach: a user-level system that uses facilities provided by existing commercial operating systems to manage transaction logging at the user-level, leaving the virtual memory system unmodified.

The primary contributions of this dissertation are:

- An analysis of the interactions between the memory management requirements of transactions and single-level stores.
- An analysis of the possible degrees of operating system support for systems that provide transactions in a single-level storage environment.
- A complete description of a transaction recovery mechanism that is integrated with virtual memory. This mechanism represents the application of general principles of transaction buffer management and operating system integration, and can be implemented in most commercial operating systems environments.
- An implementation of three approaches for operating system support of transaction recovery based on a common operating system infrastructure, which enables a fair comparison between them.
- A performance comparison of a representative of the current practice of transaction buffer management with the two approaches that eliminate buffer management redundancy: the first alternative requires custom operating systems, the second is our approach which is compatible with commercial operating systems.

- A demonstration that structuring transaction buffer management to be integrated with the underlying virtual memory results in significantly improved performance across a variety of workloads and degrees of memory competition.

The rest of this chapter is structured as follows: Section 7.2 discusses some of the work related to the transaction recovery mechanism of UMS and the operating system support it requires. Section 7.3 describes some future directions of research suggested by the transaction buffer management techniques described in this dissertation. Finally Section 7.4 ends the dissertation by offering some concluding remarks.

7.2 Related Work

Specific related work has been discussed throughout this dissertation. More generally, however, we build upon results in three areas: transaction management, operating systems support for transactions, and single-level storage systems. The foundational contributions in these areas are discussed in the subsections that follow.

7.2.1 Transaction Management

Transaction management is a rich field of study with a depth of academic research as well as commercial experience that can be daunting to a newcomer. The excellent textbook, *Transaction Processing: Concepts and Techniques* [Gray & Reuter 93] serves as a trustworthy guide to many of the issues of transaction processing and buffer management. The emphasis in this book is on transactions on relational data in server environments. This dissertation extends the lessons from this book to include support for transactions in both memory-competitive environments and single-level storage systems. In addition to the synthesizing work of the book, two specific areas of transaction processing are related to the work in this dissertation: transaction recovery policies, and operating system support for transaction buffer management.

Transaction Recovery Policies

The ARIES recovery mechanism's sheer flexibility and high performance represents a culmination of decades of ground-breaking work on transaction management at IBM. Similar to Multics, much subsequent research has consisted of applying subsets of ARIES' functionality to new environments. This dissertation is one such example. The steal / force recovery policy of UMS is one setting of the many variables provided by ARIES. Unlike this dissertation, the ARIES researchers were not concerned with the interactions between buffer management and the underlying operating system. Further, they do not consider the added opportunities or constraints involved in supporting transactions in a single-level store.

In contrast to ARIES, all of the systems that use operating system support for transactions use no-steal buffer management. For example, TABS [Spector et al. 85], Camelot [Spector 91], and a proposed virtual-memory integrated version of RVM [Satya et al. 94] all take advantage of external pager-like functionality to re-route pageouts of dirty transaction data to a temporary file. The choice of no-steal buffer management for these systems was logical: without using UMS's technique of writing undo log records before data is modified, no-steal buffer management requires significant additional support from the operating system. Thus UMS extends the previous work in this area by exploring operating system-support for steal buffer management that is integrated with virtual memory.

Operating System Support for Transaction Buffer Management

Since Stonebraker's landmark paper on operating system support for database management [Stonebraker 81], operating systems researchers have been striving to make up for the shortcomings he pointed out. One claim of this dissertation is that these efforts are now nearing fruition, and it is time for transaction systems to more directly take advantage of operating systems' services. This section summarizes the operating systems research that has brought us to this point.

The TABS system's modifications to the Accent operating system [Rashid 84] were among the first efforts to modify a general-purpose operating system to support transac-

tions. These modifications influenced the external memory management facilities provided by Accent's successor, Mach [Young 89], which has been used by a number of transaction systems (e.g., Camelot [Spector 91] and Cricket [Shekita & Zwilling 92]). Neither Accent nor Mach provided control over virtual memory page replacement, however, which would be required to implement steal buffer management with buffered logs. A number of projects have investigated interfaces and mechanisms for allowing applications to control page replacement policies, including PREMO [McNamee & Armstrong 90], PODS [Sechrest & Park 91], V++ [Harty & Cheriton 92], HiPEC [Lee et al. 94] and AVM [Engler et al. 95]. However, none of these systems have yet been used to control page replacement in order to support transactions.

RPVM [Chew et al. 93] provides a rich set of controls over virtual memory management specifically tailored to supporting transaction recovery in virtual memory. Our implementation of RMS on RPVM is the first description of a steal recovery policy that uses the primitives provided by RPVM. In addition, the previously available performance results for RPVM were "microbenchmarks" that timed the cost of adding and processing flush rules. Thus our results in Chapter 6 are the first full-system measurements of the transaction support provided by RPVM.

Finally, our work is related to the recent extensible operating systems research [Bershad et al. 95, Engler et al. 94, Cheriton & Duda 94, Small & Seltzer 94]. By enabling applications to customize the management of all resources, each of these systems can potentially provide more complete support for transactions than is provided by existing operating systems. No research has yet studied how to implement transactions using the facilities provided by extensible systems or compared the resulting performance to previous approaches. Our experimental platform can be viewed as a prototyping infrastructure that allowed us to experiment with how applications could implement transactions using the facilities that will be available in extensible operating systems. Our goal was to compare the memory management behavior of user-level transaction buffer management with the buffer pool and kernel-centric approaches. Given this goal, our infrastructure on Digital Unix is better than using an extensible operating system; the single underlying environment across all of our systems isolates from other factors, the performance differences

caused by memory management interactions. Since UMS is structured much as a kernel extension, the design, implementation and evaluation of UMS, presented in this dissertation, provides some advance experience with transaction buffer management in extensible systems.

7.2.2 Single-level Storage Systems

Atlas and Multics were the genesis and subsequent refinement of single-level storage systems. The convenience and simplicity of their uniform interface to both volatile and persistent data has inspired many projects that followed. The Opal project [Chase et al. 92] explored many of the opportunities provided by modern 64-bit address architectures, including broadening the scope of the single-level store to span a local area network. A key attribute of Opal is its extension of virtual memory to provide a single address-space shared by all applications in a network. The challenge of supporting transactions in this unique environment was a motivation for the work described in this dissertation. Thus UMS can be viewed as the recovery component of a transaction system for a distributed single address-space operating system such as Opal.

Another category of single-level stores, object-oriented database systems, provide transactions in a single-level storage environment. These systems' lack of integration with the underlying operating system was another motivation for our work. The recovery mechanisms of these systems have great diversity, and influenced the design of UMS's recovery policy. Our recovery mechanism was initially inspired by the shadow-page technique used by POMS [Cockshot et al. 84], System R [Gray et al. 81], and GemStone [Maier & Stein 86]. Complaints about the shadow techniques' poor clustering [Gray et al. 81] lead us to use write-ahead logging, also used by QuickStore [White & DeWitt 94], Thor [Liskov et al. 94] and ObjectStore [Lamb et al. 91]. Another system uses page-grain undo logs [Hulse & Dearle 96], but no implementation of this system nor measurements of its performance have been presented, which are two of the key contributions of this dissertation.

7.3 Future Directions

This section describes a number of avenues of further work suggested by this dissertation. First, Section 7.3.1 describes how the isolation transaction property can be provided in a virtual memory-integrated single-level store. Next, Section 7.3.2 describes a number of ways to use extensible operating systems to further improve the performance of transactions in single-level stores. Finally, Section 7.3.3 lists several adjacent research areas in which further work is warranted.

7.3.1 Integrating Support for Transaction Isolation

The isolation property is needed to provide correct behavior in the presence of simultaneous transactions on the same data. In a system that provides isolation, the resulting state of data after any simultaneous transactions must be the same as some sequential ordering of the same transactions. Atomicity and durability ensure that transactions are atomic with respect to their durable state (i.e. on disk). Isolation ensures that transactions are atomic with respect to each other. In Chapter 3 we described how atomicity and durability are implemented via control over buffering and logging. Isolation is implemented via *synchronization*. Synchronization techniques coordinate simultaneous actions (e.g., multiple updates to the same bank account).

The most common form of synchronization is *mutual exclusion* via software *locks*. Applications must *acquire* the lock associated with a unit of data before accessing it, and *release* the lock when finished with the data (often as a part of transaction commit). A common protocol for synchronizing a shared data is the reader-writer lock protocol. In this protocol, access to data is protected by separate read and write locks. Any number of simultaneous transactions may acquire a *read lock*, which enables them to access, but not modify the data. This provides the isolation property because simultaneous read accesses to unmodified data produces the same effect as sequential reads. Modifications to data must be preceded by acquiring that data's *write lock*. In the reader-writer protocol, a write lock may be acquired only when no other transactions hold either read or write locks.

This section discusses the two issues that most affect the efficiency of synchronization in single-level stores: first, the mechanisms for acquiring and releasing locks, and second the granularity of locks (how much data is associated with each lock). The solutions to other issues related to locking have been dealt with elsewhere, including policies to optimize throughput for multiple readers and writers, mechanisms to guarantee progress in spite of arbitrary delays of transactions that hold locks [Herlihy 91], as well as avoiding or recovering from deadlock [Gray & Reuter 93]. In this section, we concentrate on the design decisions directly affected by locking in a single-level storage environment: locking mechanism and granularity.

Lock Mechanisms in a Single-level Store

The possible mechanisms for implementing locks in a single-level store include application-level programmed locks and system-level automatic locks. Application-level locks allow programmers to optimize lock granularity, and can simplify the system by reducing the responsibilities of the lock manager. However, application-level locks place the burden of correctness on the programmer, which can make applications both more tedious to write, and harder to verify. System-managed locks can relieve the programmer of the burden of implementing correct locking, but usually can not adapt the locking granularity according to applications' needs. Page-grain locking has been found to be sufficient for many applications [Maier & Zdonik 90, Lamb et al. 91], because in many cases write-sharing is uncommon, or can made sequential outside the system-level.

We created a modified version of UMS to simulate the overheads of supporting page-grain locking. In the modified system, when an application attaches to a segment, by calling `rms_attach`, the data file is mapped with *neither* read nor write permissions. The virtual memory protection violation signal handler thus receives signals for both read and write accesses (UMS previously only received write-fault signals in order to drive log writes). The read fault handler first determines whether the access is valid, then checks for outstanding write locks on the page. If the read fault handler determines the access can be granted, it acquires the page's read lock, and finally grants the request by enabling read

permission (by calling `mprotect`). We repeated the OO7 T1 traversal experiments with a version of UMS that includes the locking operations, but without contention for the locks, to determine whether the performance results presented in Chapter 6 could be achieved in a system that provides isolation. The results of this experiment are presented in Figure 7.1. This experiment demonstrates that the added signal handling overheads associated with lock management do not significantly modify the system's overall performance.

Page-grain locking is not efficient for workloads with frequent, fine-grained sharing. In these cases, finer granularity locking is more appropriate. Finer than page-grain locking can be accomplished with support from the language to implement object-grain locking, or without such support to implement fixed-size fine grain locking. In some systems, locking granularity is the same as logging (and buffering) granularity. The ARIES system [Mohan et al. 92] demonstrated that logging, locking and buffering granularities can be decoupled, which can significantly improve transaction performance. Extending upon this work, researchers in the SHORE project demonstrated that locking granularity can be dynamically adapted to usage patterns [Carey et al. 94]. A similar technique is used in VMS/Rdb [Joshi 91]. Transactions in these systems initially acquire coarse-grain locks and *de-escalate* them to multiple finer grain locks as additional transactions access other data covered by the coarse-grain lock. The AIRES and SHORE approaches could be used by UMS because their lock managers are implemented above the buffer management

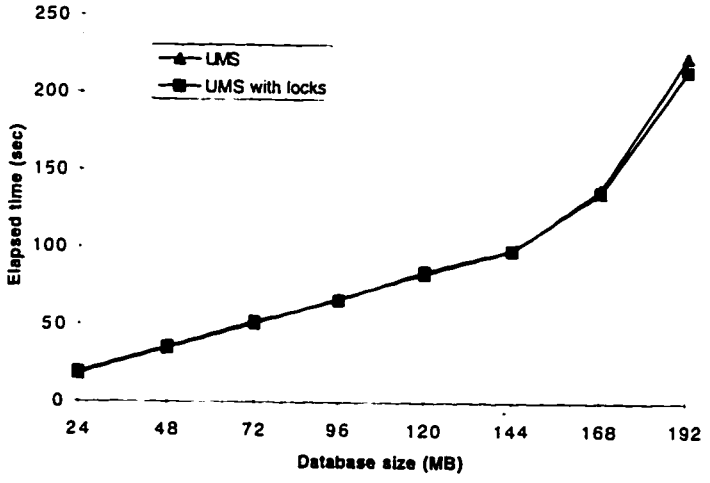


Figure 7.1: OO7 T1 traversal on UMS with and without lock overheads, with no contention

layer, and (as ARIES demonstrated) the two layers can be considered largely independently.

Our measurements of signal handling overheads associated with lock management demonstrate that communication costs associated with lock management would not significantly affect the performance of UMS, even though it uses signals and system calls for communication between the operating system and the transaction buffer manager. One of the primary benefits of using extensible operating systems to customize the implementation of a system service is the reduced communication costs associated with resource management. Another benefit of implementing a system like UMS as a kernel extension is that, as an extension, it has access to operating system-level resource usage information that it can use to adapt log management to system load. In the next section we investigate both benefits of implementing UMS as an operating system extension.

7.3.2 Improving Performance via Extensible Operating System Support

Extensible operating systems, such as SPIN, will enable systems like UMS to achieve better performance. This section describes three ways an extensible system can improve the performance of UMS: first, by reducing the overhead of protection-boundary crossings between the operating system and UMS' logging mechanism, second by allowing logging to adapt to system workload, and third, by enabling application-controlled prefetching in a single-level store.

Using an Extensible Operating System to Reduce the Overhead of Protection-boundary Crossing

The actions performed while processing a write fault in UMS, and the resulting protection-boundary crossings are shown in Figure 7.2. The total cost of the protection-boundary crosings consists of the time required for signal delivery combined with the overhead of re-entering the kernel for the system calls to write the log, update the metadata and to call

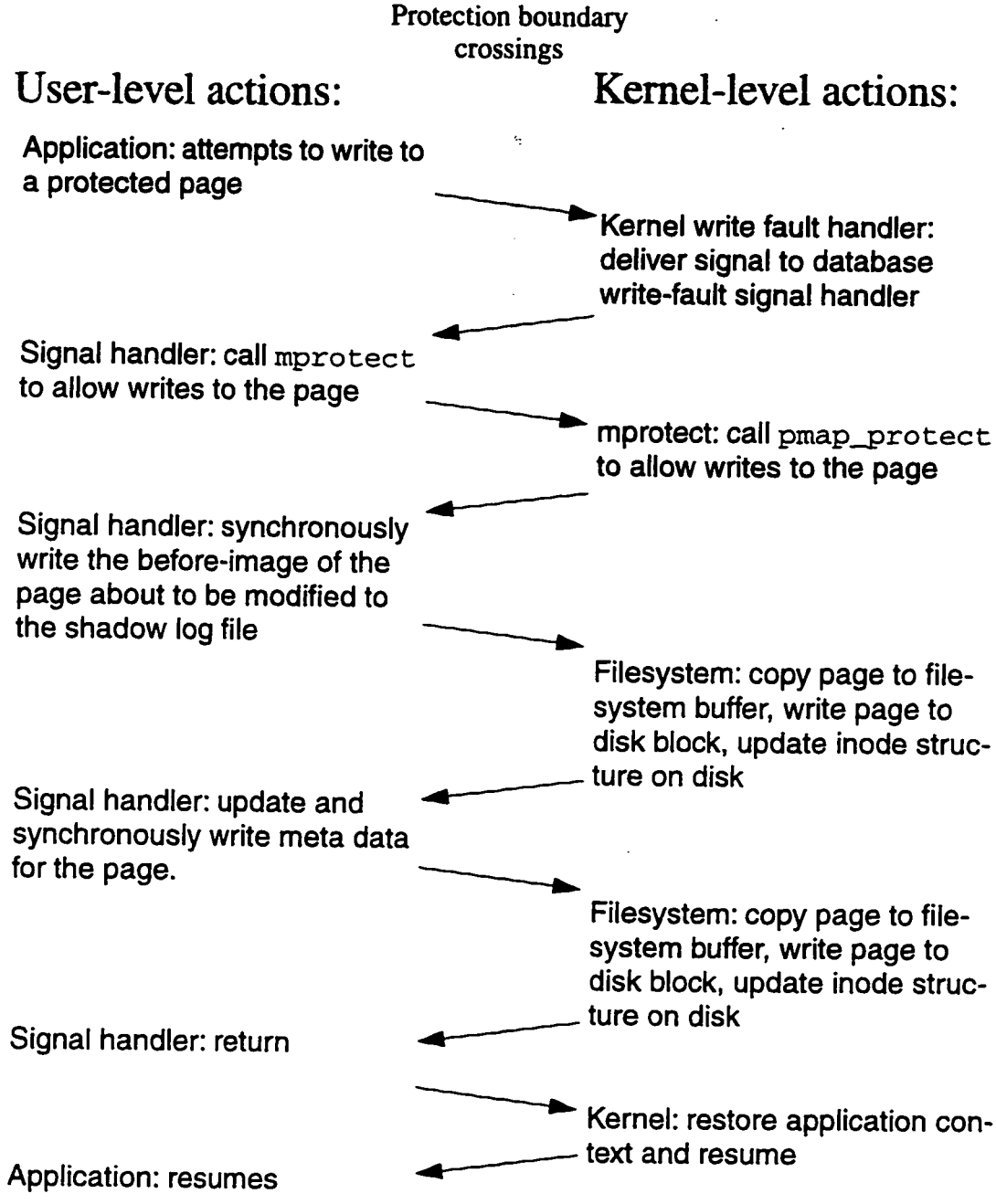


Figure 7.2: Communication and actions to process a write fault in UMS.

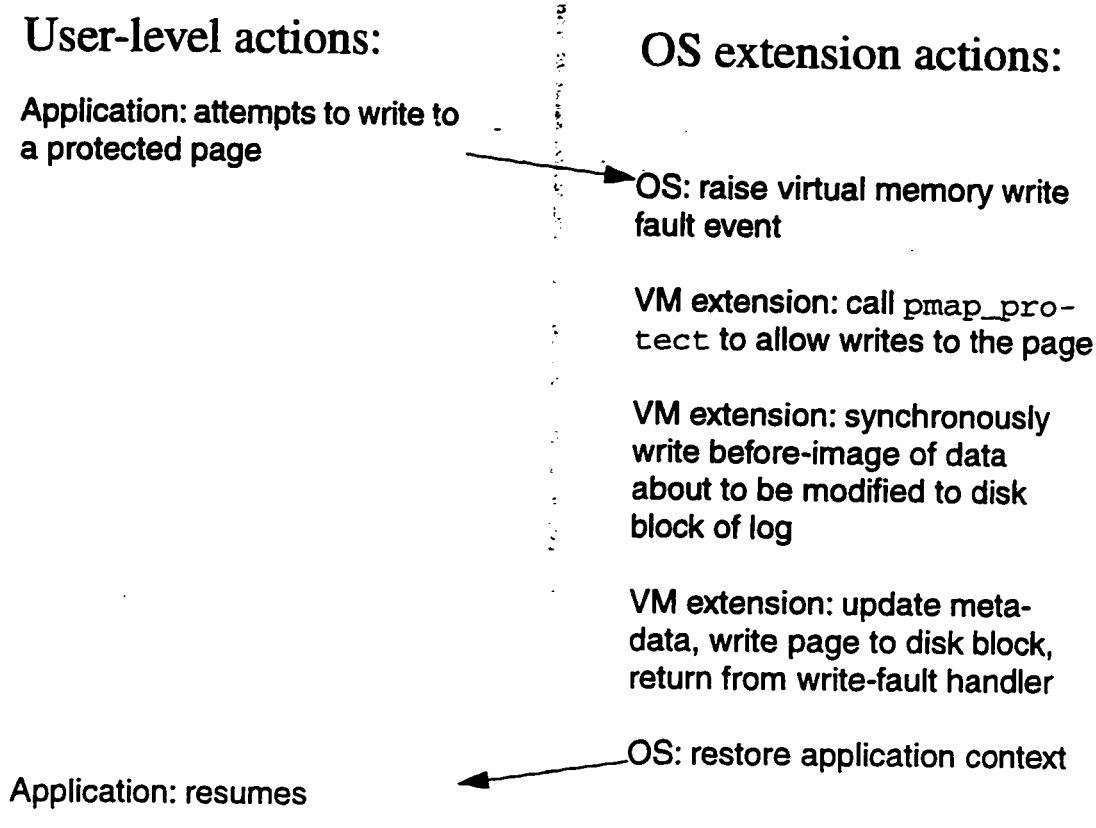


Figure 7.3: Communication and actions performed to process a write fault within an OS extension

`mprotect`. Systems like UMS will soon be able to be implemented as a kernel extension in an extensible operating system such as SPIN. This approach will greatly reduce the cost of protection boundary crossings because the write-fault handling is done entirely within the operating system. The resulting actions performed in response to a write fault in an implementation of UMS as an operating system extension are depicted in Figure 7.3. We can project the performance of UMS implemented in SPIN by measuring the costs of protection-boundary crossings incurred by UMS's log management, and substituting for them the costs of the equivalent operations in SPIN. The two kinds of protection-boundary

crossings in UMS are *signals* and *system calls*. The corresponding mechanisms in SPIN are *event dispatch* and *procedure calls* [Pardyak & Bershad 96]. We timed the kernel communication events on our Digital AlphaStation 250 and compared them to measurements of the corresponding SPIN events on the same class of machine. The results of this comparison are summarized in Table 7.1. The overall costs of protection-boundary crossings for UMS and UMS in SPIN for the 192 MB T2b traversal, and the resulting performance, are presented in Table 7.2. The results show that although communication overheads are reduced by 99.7 %, this reduction does not result in significant performance improvement, because protection-boundary crossing costs account for only 0.3 % of overall runtime.

An application of Amdahl's law suggests improving the portions of the system responsible for the most elapsed time. For OO7, this means reducing the time spent doing I/O. The next section discusses how extensible systems could be used to improve I/O performance by either adapting logging behavior to system workload, or by prefetching transaction data.

Table 7.1: Overheads of UMS Operating System Interaction in Digital Unix and SPIN

Action	Digital Unix	SPIN
Enter/exit Kernel	OS Trap/return 45,000 ns	OS Trap/return 45,000 ns
Enter/exit UMS	Signal handler 50,000 ns	Event dispatch 380 ns
Call system	System call 45,000 ns	Procedure call ~100 ns

Table 7.2: Total UMS Communication Overhead for 192 MB OO7 T2b traversal in Digital Unix and SPIN

Action	Digital Unix	SPIN
Enter/exit Kernel	267 msec	267 msec
Enter/exit UMS	297 msec	2.3 msec
Call system	1.1 sec	2.4 msec
Total Overhead	1.66 sec	272 msec
T2b Elapsed Time	494.6 sec	493.2 sec

A Load-Adaptive Write-Ahead Log Manager

The performance results in Chapter 6 demonstrate that UMS and RPVM have performance advantages over one other, depending on the size of the database. For databases that can be accommodated in memory along with the log, RPVM performs significantly better than UMS, because its log writes are performed all at once, at commit time. In contrast, for databases that do not fit in memory along with the log, UMS provides better performance because it provides higher buffer hit rates, since log pages are not buffered. This performance trade-off suggests a hybrid mechanism that switches between RPVM's "lazy" log writes and UMS's "eager" log writes, according to the load on the machine. Such an adaptive system would be able to combine the best performance of both UMS and RPVM. We were not able to implement this hybrid scheme in the Digital Unix environment because the two systems, UMS and RPVM, were implemented on opposite sides of the kernel boundary, and we were not able to switch between them for two reasons. First, the user-level signal handler that controlled both UMS and RPVM does not have access to the system paging-load information required to know when to switch between modes, and second, because UMS and RPVM write their log data to raw disk and to the filesystem respectively, and switching between them requires very long-duration filesystem operations.

An extensible system provides exactly the support required to overcome the obstacles for implementing an adaptive logging mechanism. First, by locating both mechanisms within the operating system, UMS and RPVM can both perform logging to the filesystem, obviating switches between raw disk- and file-based logging. Second, the paging-load information required to know when to switch between modes is available within the operating system. The resulting predicted performance of a hybrid logging system, implemented in an extensible operating system, is presented in Figure 7.4. The performance of the hybrid system includes both the benefits of adapting logging to system load as well as the reduced protection-boundary crossing overheads.

Application-controlled Prefetching

The last optimization opportunity we consider is the ability to prefetch data in a single-level store. User-directed prefetching policies and mechanisms for the filesystem interface have recently been developed [Cao et al. 94, Gibson et al. 92]. Further, efficient policies for combining prefetching and page replacement are an area of ongoing research [Kimbrel et al. 96]. Chapter 2 described how Digital Unix prefetches (and eagerly-discards) sequentially accessed pages of both files and mapped memory, however user-directed prefetching in a single-level store has not yet been investigated.

We investigated the potential benefits of prefetching in UMS by incorporating a separate prefetching process that is activated at application start-up when it is determined that the database can be accommodated in available memory. The traditional benefits of prefetching (e.g., in filesystems and relational databases) are accrued from overlapping computation with I/O, since in these systems I/O has otherwise been fully optimized by transferring large amounts of data per request. In a single-level store, however, data is often not transferred in large contiguous amounts, because it is fetched in response to application requests, and written out during page replacement. Current virtual memory systems attempt to optimize this behavior by *clustering* disk reads and writes. Figure 7.5 compares the performance of the OO7 T1 traversal using both the default virtual memory

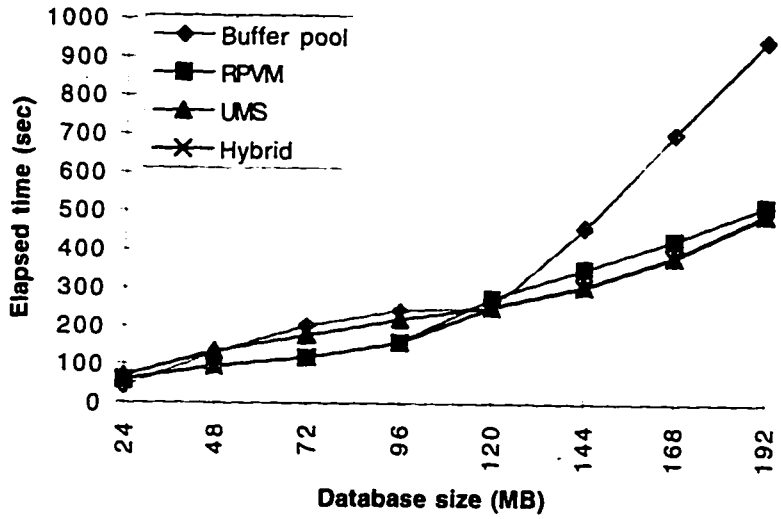


Figure 7.4: OO7 T2b Traversal Performance UMS, RPVM, Buffer pool and predicted performance of Hybrid

fetching and replacement, as well as with user-directed prefetching. Because of the potential interaction between prefetching and competing applications, further research will be required before this technique can be efficiently incorporated into desktop applications. These results demonstrate that allowing applications to control prefetching of data in a single-level store can provide significant performance improvements.

7.3.3 Work in Adjacent Areas

Further work in three adjacent areas of research was suggested by the work in this dissertation. First, since we demonstrated the benefits of integrating buffer management with virtual memory by using only one recovery mechanism, there is promise in work involving alternate recovery mechanisms, their efficient implementation, and their relative performance for a range of workloads. Second, there is a great need for empirically significant benchmarks for distributed single-level storage systems. Third, in order for prefetching to be viable in memory-competitive environments, further research of memory allocation policies between competing applications will be required, because current replacement policies (such as LRU) do not behave well in the presence of prefetching applications.

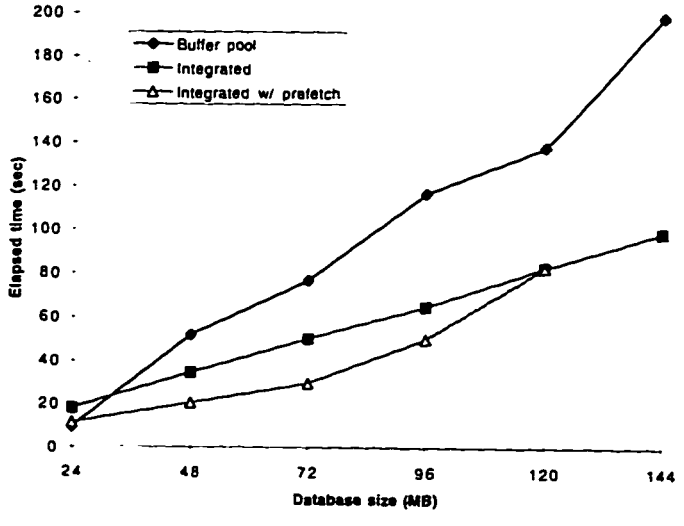


Figure 7.5: OO7 T1 Traversal with Buffer Pool, Integrated and Prefetching buffer management

7.4 Conclusion

In this dissertation we demonstrated that current practice of transaction buffer management performs poorly in the increasingly common memory-competitive environments. We argued that this poor performance is the result of fundamental structure decisions that result in redundant memory management between the transaction buffer manager and the operating system. We lent evidence to this argument by demonstrating significant performance improvements in a restructured system that uses existing operating systems' facilities to eliminate the memory management redundancy. Finally, we demonstrated prototype implementations of transaction support that will be available in future extensible operating systems that provide further performance improvements.

Since extensible operating systems *could* be used to marginally improve the performance of existing transaction system structures, the eventual contribution of this dissertation will be determined by whether future transaction systems retain private memory management, or are instead restructured to better coexist in memory-competitive environments.

Bibliography

[Anderson 92] Thomas E. Anderson. The Case for Application-Specific Operating Systems. In *Proceedings of 3rd IEEE Workshop on Workstation Operating Systems*, pages 92–94, Key Biscayne, Florida, April 1992.

[Anderson et al. 92] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992. Also appeared in *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, October 1991.

[Bensoussan et al. 69] A. Bensoussan, C. T. Clingen, and R. C. Daley. The Multics Virtual Memory. In *Proceedings of the 2nd ACM Symposium on Operating Systems Principles*, Princeton, New Jersey, October 1969.

[Bershad et al. 95] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain Resort, Colorado, December 1995.

[Boral et al. 90] Haran Boral, William Alexander, Larry Clay, George Copeland, Scott Danforth, Michael Franklin, Brian Hart, Marc Smith, and Patrick Valduriez. Prototyping Bubba, A Highly Parallel Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):4–24, 1990.

[Buhr et al. 92] Peter A. Buhr, Anil K. Goel, and Anderson Wai. *μDatabase: A Toolkit for Constructing Memory Mapped Databases*, pages 166–185. Springer-Verlag, September 1992.

[Carey et al. 86] Michael J. Carey, David J. DeWitt, Joel E. Richardson, and Eugene J. Shekita. Object and File Management in the EXODUS Extensible Database System. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 91–100, Kyoto, Japan, August 1986.

[Carey et al. 93] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 Benchmark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Washington D. C., 1993.

[Carey et al. 94] Michael J. Carey, Michael J. Franklin, and Markos Zaharioudakis. Fine-Grain Sharing in a Page Server OODBMS. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Minneapolis, MN, May 1994. available at ftp://ftp.cs.wisc.edu/tech-reports/reports/94/tr1224.ps.Z.

[Chamberlain et al. 96] Bradford Chamberlain, Tony DeRose, Dani Lischinski, David Salesin, and John Snyder. Fast Rendering of Complex Environments Using a Spatial Hierarchy. In *Proceedings of Graphics Interface 1996*, May 1996.

[Chang & Mergen 88] Albert Chang and Mark F. Mergen. 801 Storage: Architecture and Programming. *ACM Transactions on Computer Systems*, 6(1):28–50, February 1988.

[Chase 95] Jeffrey S. Chase. *An Operating System Structure for Wide-Address Architectures*. PhD dissertation, Department of Computer Science and Engineering, University of Washington, August 1995.

[Chase et al. 94] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and Protection in a Single-Address-Space Operating System. *ACM Transactions on Computer Systems*, 12(4):271–307, November 1994.

[Chen et al. 96] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. The Rio File Cache: Surviving Operating System Crashes. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, Massachusetts, October 1996.

[Cheng et al. 84] J. M. Cheng, C. R. Loosely, A. Shibamiya, and P. S. Worthington. IBM Database 2 performance: Design, implementation and tuning. *IBM Systems Journal*, 23(2):189–210, 1984.

[Cheriton & Duda 94] David R. Cheriton and Kenneth J. Duda. A Caching Model of Operating System Kernel Functionality. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 179–194, Monterey, CA, November 1994.

[Cheriton & Duda 95] David R. Cheriton and Kenneth J. Duda. Logged Virtual Memory. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 26–39, Copper Mountain Resort, Colorado, December 1995.

[Chew & Silberschatz 92] Khien-Mien Chew and Avi Silberschatz. Toward Operating System Support for Recoverable-Persistent Main Memory Database Systems. Technical Report TR-92-05, Department of Computer Sciences, University of Texas at Austin, February 1992.

[Chou & DeWitt 85] Hong-Tai Chou and David J. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *Proceedings of the 11th International Conference on Very Large Data Bases*, pages 127–141, Stockholm, August 1985.

[Cockshot et al. 84] W. P. Cockshot, M. P. Atkinson, K. J. Chisholm, P. J. Bailey, and R. Morrison. Persistent Object Management System. *Software–Practice and Experience*, 14(1), January 1984.

[Corbato 69] F. J. Corbato. A paging experiment with the Multics system. In *In Honor of P. M. Morse*, pages 217–228. M.I.T. Press, 1969.

[Crus 84] R. Crus. Data Recovery in IBM Database 2. *IBM Systems Journal*, 23(2):178–188, 1984.

[Custer 93] Helen Custer. *Inside Windows NT*. Microsoft Press, 1993.

[Diel et al. 84] Hans Diel, Gerald Kreissig, Norbert Lenz, Michael Scheible, and Bernd Schoener. Data Management Facilities of an Operating System Kernel. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 58–69, Boston, MA, June 1984.

[Draves 93] Richard Draves. The Case for Run-Time Replaceable Kernel Modules. In *Proceedings of the 4th IEEE Workshop on Workstation Operating Systems*, pages 160–164, Napa, California, October 1993.

[Engler et al. 94] Dawson Engler, M. Frans Kaashoek, and James O'Toole. The Operating System Kernel as a Secure Programmable Machine. In *Proceedings of the 6th SIGOPS European Workshop*, Germany, September 1994.

[Engler et al. 95] Dawson Engler, Sandeep K. Gupta, and M. Frans Kaashoek. AVM: Application-Level Virtual Memory. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 72–77, Orcas Island, Washington, May 1995.

[Fiuczynski & Bershad 96] Marc Fiuczynski and Brian Bershad. An Extensible Protocol Architecture for Application-Specific Networking. In *Proceedings of the USENIX 1996 Annual Technical Conference*, pages 55–64, San Diego, California, January 1996.

[Garrett et al. 93] W. E. Garrett, M. L. Scott, R. Bianchini, L. I. Kontothanassis, R. A. McCallum, J. A. Thomas, R. Wisniewski, and S. Luk. Linking Shared Segments. In *Proceedings of the 1993 Winter USENIX Conference*, pages 13–27, January 1993.

[Goldberg & Hassinger 74] Robert P. Goldberg and Robert Hassinger. The double paging anomaly. In *AFIPS Conference Proceedings*, pages 195–199, Chicago, Illinois, May 1974.

[Gray 78] Jim Gray. Notes on Database Operating Systems. In *Operating Systems: An Advanced Course*. Springer-Verlag Lecture notes in Computer Science. Vol. 60. New York, 1978.

[Gray & Reuter 93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, California, 1993.

[Gray et al. 81] Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger. The Recovery Manager of the System R Database Manager. *ACM Computing Surveys*, 13(2):223–242, June 1981.

[Haderle & Jackson 84] D. J. Haderle and R. D. Jackson. IBM Database 2 overview. *IBM Systems Journal*, 23(2):112–125, 1984.

[Haerder & Reuter 83] Theo Haerder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[Harty & Cheriton 92] Keiran Harty and David Cheriton. Application-Controlled Physical Memory using External Page-Cache Management. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 187–197, Boston, Massachusetts, October 1992.

[Herlihy & Moss 93] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–299, May 1993.

[Herlihy 91] Maurice Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, January 1991.

[HP 90] Hewlett Packard Company. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, 1990.

[Hulse & Dearle 96] David Hulse and Alan Dearle. A Log-Structured Persistent Store. Technical Report GH-14, University of Stirling and University of Sydney Departments of Computer Science. Available at http://nezz.cs.stir.ac.uk/pub/papers/GH-14.ps.gz, 1992.

[Joshi 91] Ashok M. Joshi. Adaptive Locking Strategies in a Multi-node Data Sharing Environment. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 181–191, Barcelona, Spain, September 1991.

[Kane & Heinrich 92] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[Keppel 93] David Keppel. Managing abstraction-induced complexity. Technical Report UWCSE 93-06-02, Department of Computer Science and Engineering, University of Washington, June 1993.

[Kiczales 92] Gregor Kiczales. Towards a New Model of Abstraction in the Engineering of Software. In *Proceedings of IMSA 1992 Workshop on Reflection and Meta-level Architectures*, 1992.

[Kiczales et al. 91] Gregor Kiczales, Jim Des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[Kiczales et al. 93] Gregor Kiczales, John Lamping, Chris Maeda, David Keppel, and Dylan McNamee. The Need for Customizable Operating Systems. In *Proceedings of the 4th IEEE Workshop on Workstation Operating Systems*, pages 165–169, Napa, California, October 1993.

[Kilburn et al. 62] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. One-level storage system. *IRE Transactions on Electronic Computers*, EC-11(2):223–235, April 1962.

[Kimbrel et al. 96] Tracy Kimbrel, Andrew Tomkins, R. Hugo Patterson, Brian Bershad, Pei Cao, Edward W. Felten, Garth A. Gibson, Anna R. Karlin, and Kai Li. A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Seattle, Washington, October 1996.

[Koldinger et al. 92] Eric J. Koldinger, Jeffrey S. Chase, and Susan J. Eggers. Architectural Support for Single Address Space Operating Systems. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 175–186, Boston, Massachusetts, October 1992.

[LaMarca 96] Anthony LaMarca. *Caches and Algorithms*. PhD dissertation, Department of Computer Science and Engineering, University of Washington, 1996.

[Lamb et al. 91] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore Database System. *Communications of the ACM*, pages 50–63, October 1991.

[Lee et al. 94] Chao Hsien Lee, Meng Chang Chen, and Ruei Chuan Chang. HiPEC: High Performance External Virtual Memory Caching. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, Monterey, California, November 1994.

[Levin et al. 75] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/ Mechanism Separation in Hydra. In *Proceedings of the 5th ACM Symposium on Operating Systems Principles*, pages 132–140, November 1975.

[Liskov et al. 94] Barbara Liskov, Mark Day, Sanjay Ghemawat, Robert Gruber, Umesh Maheshwari, Andrew C. Myers, and Liuba Shrira. The Language-Independent Interface of the Thor Persistent Object Store. Technical Report Programming Methodology Group Memo 80, Massachusetts Institute of Technology, Laboratory for Computer Science, March 1994.

[Maeda & Bershad 93] Chris Maeda and Brian Bershad. Protocol Service Decomposition for High-Performance Networking. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 244–255, Ashevile, North Carolina, December 1993.

[Maier & Stein 86] David Maier and Jacob Stein. Development of an Object-Oriented DBMS. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 472–482, September 1986.

[McNamee & Armstrong 90] Dylan McNamee and Katherine Armstrong. Extending the Mach External Pager Interface to Accommodate User-Level Page Replacement Policies. In *Proceedings of the First USENIX Mach Symposium*, pages 17–29, Burlington, Vermont, October 1990.

[Mogul et al. 87] J. C. Mogul, R. F. Rashid, and M. J. Accetta. The Packet Filter: An Efficient Mechanism for User-level Network Code. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 39–51, Austin, Texas, November 1987.

[Mohan et al. 92] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A transaction recovery method support fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

[Mossenbock 94] H. Mossenbock. Extensibility in the Oberon System. *Nordic Journal of Computing*, 1(1):77–93, February 1994.

[Mot 93] Motorola Incorporated and IBM, Phoenix, Arizona. *PowerPC 601 RISC Microprocessor User's Manual*, 1993.

[Narasayya et al. 96] Vivek Narasayya, Eugene Tze Ng, Dylan McNamee, Ashutosh Tiwary, and Henry Levy. Reducing the Virual Memory Overhead of Swizzling. In *Fifth International Workshop on Object-Orientation in Operating Systems*, October 1996.

[Nelson et al. 88] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[Nelson et al. 93] Michael N. Nelson, Yousef A. Khalidi, and Peter W. Madany. The Spring File System. Technical Report SMLI TR-93-10, Sun Microsystems Laboratories, Inc., February 1993.

[O. Deux et al. 91] O. Deux et al. The O2 System. *Communications of the ACM*, pages 34–48, October 1991.

[Pardyak & Bershad 96] Przemyslaw Pardyak and Brian N. Bershad. Dynamic Binding for Extensible Systems. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Seattle, Washington, October 1996.

[Parker 94] Mike Parker. W. Starling Burgess, Type Designer? *Printing History, The Journal of the American Printing History Association.* 16(1,2): 52–108, 1994.

[Parnas 72] David L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[Rashid & Robertson 81] Richard Rashid and George Robertson. Accent: A Communication Oriented Network Operatingc System Kernel. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pages 64–75, Pacific Grove, California, December 1981.

[Rosenberg 92] John Rosenberg. Architectural and Operating System Support for Orthogonal Persistence. *Computing Systems*, 5(3):305–335, Summer 1992.

[Rosenberg et al. 96] John Rosenberg, Alan Dearle, David Hulse, Anders Lindstrom, and Stephen Norris. Operating System Support For Persistent and Recoverable Computations. *Communications of the ACM*, 39(9):62–69, September 1996.

[Rosenblum & Ousterhout 91] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1991.

[Satya et al. 94] M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. Lightweight Recoverable Virtual Memory. *ACM Transactions on Computer Systems*, 12(1):33–57, February 1994.

[Scott et al. 90] Michael L. Scott, Thomas J. LeBlanc, and Brian D. Marsh. Multi-model parallel programming in Psyche. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 70–78, March 1990.

[Sechrest & Park 91] Stuart Sechrest and Yoonho Park. User-Level Physical Memory Management for Mach. In *Proceedings of the Second USENIX Mach Symposium*, pages 189–199, Monterey, California, November 1991.

[Shaw & Wulf 80] Mary Shaw and William A. Wulf. Toward Relaxing Assumptions in Languages and Their Implementations. *SIGPLAN Notices*, 15(3):45–51, 1980.

[Shekita & Zwilling 92] Eugene Shekita and Michael Zwilling. Cricket: A Mapped, Persistent Object Store. In *Proceedings of the 4th International Workshop on Persistent Object Systems Design, Implementation and Use*, 1992.

[Singhal et al. 92] V. Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: An Efficient, Portable Persistent Store. In *Proceedings of the Fifth International Workshop on Persistent Object Systems*, San Miniato, Italy, September 1992.

[Sites 92] Richard L. Sites. *Alpha Architecture Reference Manual.* Digital Press, Burlington, MA, 1992.

[Small & Seltzer 94] Christopher Small and Margo Seltzer. VINO: An Integrated Platform for Operating System and Database Research. Technical Report TR-30-94, Harvard University, 1994.

[Spector 91] Alfred Z. Spector. *The Design of Camelot.* Morgan Kaufmann, San Mateo, California, 1991.

[Spector et al. 85] Alfred Z. Spector, Jacob Butcher, Dean S. Daniels, Daniel J. Duchamp, Jeffrey L. Eppinger, Charles E. Fineman, Abdelsalam Heddaya, and Peter M. Schwarz. Support for Distributed Transactions in the TABS Prototype. *IEEE Transactions on Software Engineering*, SE-11(6):520–530, 1985.

[Stonebraker 81] Michael Stonebraker. Operating System Support for Database Management. *Communications of the ACM*, 24(7):412–418, 1981.

[Sweeney et al. 96] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *Proceedings of the USENIX 1996 Annual Technical Conference*, pages 1–14, San Diego, California, January 1996.

[Talluri & Khalidi 95] Madhusudhan Talluri and Mark D. Hill Yousef A. Khalidi. A New Page Table for 64-bit Address Spaces. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 184–200, Copper Mountain Resort, Colorado, December 1995.

[Teng & Gumaer 84] J. Z. Teng and R. A. Gumaer. Managing IBM Database 2 buffers to maximize performance. *IBM Systems Journal*, 23(2):211–218, 1984.

[Traiger 82] Irving L. Traiger. Virtual Memory Management for Database Systems. *OS Review*, 16(4):26–48, 1982.

[Turner & Levy 81] Rollins Turner and Henry Levy. Segmented FIFO Page Replacement. In *Proceedings of 1981 SIGMetrics*, pages 48–57, Las Vegas, Nevada, September 1981.

[Vaughan et al. 92] Francis Vaughan, Tracy Lo Basso, Alan Dearle, Chris Marlin, and Chris Barter. Casper: a Cached Architecture Supporting Persistence. *Computing Systems*, 5(3):337–359, Summer 1992.

[Vo 96] Kiem-Phong Vo. Vmalloc: A General and Efficient Memory Allocator. *Software–Practice and Experience*, 1996.

[White & DeWitt 94] Seth J. White and David J. DeWitt. QuickStore: A High Performance Mapped Object Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 395–406, May 1994.

[Yokote et al. 89] Yasuhiko Yokote, Fumio Teraoka, and Mario Tokoro. A Reflective Architecture for an Object-Oriented Distributed Operating System. In S. Cook, editor, *Proceedings of the 1989 European Conference on Object Oriented Programming*, pages 89–106, Nottingham, July 1989.

[Young 89] Michael Young. *Exporting a User Interface to Memory Management from a Communication-Oriented Operating System.* PhD dissertation, Carnegie-Mellon University, 1989.

[Zdonik & Maier 90] Stanley B. Zdonik and David Maier. *Readings in Object-Oriented Database Systems*. Morgan Kaufmann, 1990.

Vita

Dylan James McNamee was born on September 27, 1966, in San Francisco, California. He grew up in Eugene, Oregon, with Summer stints in Elgin, and Enterprise, (also Oregon). In 1989, he received a B.A. with honors in Computer Science from the University of California, Berkeley. He moved to Seattle in 1989 to begin graduate studies at the University of Washington. In 1995, he received the M.S. degree in Computer Science, and in 1996, received the Ph.D. degree in Computer Science. He is currently a visiting Assistant Professor of Computer Science at the Oregon Graduate Institute of Science and Technology in Portland, Oregon.