Address Translation and Storage Management for Persistent Object Stores

by

Sheetal Vinod Kakkad, B.E., M.S.C.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 1997

Address Translation and Storage Management for Persistent Object Stores

Approved by
Dissertation Committee:

____

____

____

____

____

To my wife Raji,
for always being there
when it mattered the most

Acknowledgments

Getting a Ph.D. is incredibly hard, and I could not have done it without the help and support of many people. First and foremost, I would like to thank my advisor, Paul Wilson, who originally encouraged me to transfer from the master's program to the doctoral program. Over the last six years, I have learned many lessons—both academic and otherwise—from Paul as he helped me steer towards the right problems and issues. I would also like to thank all the members of my dissertation committee, particularly Don Batory, who has always been a source of constant encouragement and has helped with remmoving many administrative obstacles in my life as a graduate student.

I would like to thank the past and present members of OOPS research group, in particular Mark Johnstone and Scott Kaplan, who have directly or indirectly contributed to the work presented in this dissertation. I have also been fortunate to have Vivek Singhal and Jeff Thomas as close friends who have helped me in many different ways, including providing moral support when everything seemed too hard.

Over the years, several staff members in the Department of Computer Sciences have made life easier for me. In particular, Fletcher Mattox, the best systems administrator I have known, who has more than once gone out of his way to help me resolve the problem at hand so that I could continue making progress, and Gloria Ramirez, the most helpful and friendly graduate secretary that any lost student could ask for—thank you both for everything that you have done.

A special acknowledgment goes to Motorola for the financial support over the last year, and especially to my colleagues at the Somerset Design Center in Austin for being very understanding and patient with me as dissertation requirements conflicted with my work duties at times.

To my parents and in-laws, to my brother Darpan, and to Mona, Neela, Lakshmi and Bala—thank you for the trust and faith in my abilities, and for not giving up on me. I am deeply grateful to Geri for the love and positive energy that she showed, even when everything seemed so hard—thanks from the bottom of my heart! Thanks also to all my friends, particularly Sanjay and Rema, Aarthi, and Dave and Maria, for supporting my ambitions and always believing that I could achieve my goals.

Finally, I owe an immense debt of gratitude to Raji, my wife and my best friend, for her unfledgling love and support over the last few years. She has put up with all my erratic and seemingly-crazy behavior, and has always been there when I needed a friendly shoulder or a warm hug. This dissertation would not have been possible without her efforts, and I dedicate it to her for the courage and calmness that she has shown through the years.

SHEETAL VINOD KAKKAD

The University of Texas at Austin
December 1997

Address Translation and Storage Management for Persistent Object Stores

Publication No. ______________________

Sheetal Vinod Kakkad, Ph.D.
The University of Texas at Austin, 1997

Supervisor: Paul R. Wilson

A common problem in software engineering is efficiently saving the state of application data structures to non-volatile storage between program executions. If this is accomplished using normal file systems, the programmer is forced to explicitly save the data to files as a stream of uninterpreted bytes, thereby losing both pointer semantics and object identity. A better approach is to use *persistent object storage*, a natural extension to virtual memory that allows heap data to be saved automatically to disk while maintaining the topology of data structures without any explicit programmer intervention.

If persistent object stores are to replace the functionality of normal file systems, they must be able to address large volumes of data efficiently on standard hardware. High-performance address translation techniques are necessary and important for supporting large address spaces on stock hardware. We present *pointer swizzling at page fault time* (PS@PFT), a coarse-grained address translation scheme suitable for this purpose, and demonstrate it by building a persistent storage system for C++ called the Texas Persistent Store. We also discuss alternative approaches for portably incorporating fine-grained address translation in Texas for situations where coarse-grained swizzling alone is insufficient. As part of the performance results, we present a detailed analysis of various components of a coarse-grained address translation technique, including a comparison with overall I/O costs.

Pointer swizzling requires run-time knowledge of in-memory object layouts to locate pointers in objects. We have developed and implemented *Run-Time Type Description* (RTTD) for this purpose; our implementation strategy is portable because it is based on a novel use of compiler-generated debugging information for extracting the necessary type description. RTTD is also useful for other applications such as data structure browsing, and advanced profiling and tracing.

Another part of this research is a study of the interaction between systems similar to PS@PFT and operating systems, particularly regarding virtual memory management issues. We suggest areas where operating system implementations can be made more open to improve their performance and extensibility. Finally, we briefly discuss storage management issues, specifically *log-structured storage*, *disk prefetching*, and *compressed in-memory storage*, and provide directions for future research in this area.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

It is often desirable to support a virtual address space that is larger than what can be specified directly by the word size of the available hardware. Applications such as persistent object stores (e.g., [ABC^{+}83a, SKW92, DSZ90, AM92]), operating systems with a single shared address space (e.g., [CLLBH92]), distributed shared memories (e.g., [Li86]), etc. can benefit from large address spaces. For example, persistent object stores provide sharable, recoverable heap storage to eliminate the use of files for most purposes, operating systems with single shared address spaces provide a common addressing model for all processes on one or more machines, and distributed shared memory models provide a single address space for applications that span multiple machines.

All these systems typically emphasize simplified programming by preserving pointer semantics in data structures. In other words, they inherently support the notion of *object identity* by maintaining the programmer's default view of data, which is a heap of objects interconnected by pointers. Object identity is defined as the property by virtue of which an object can be *uniquely* identified among a collection of objects, such that any two object references can be compared to determine whether they identify the same object. By this definition, an object identifier is *never* reused, even after the object is deleted. A detailed study of various forms of object identity is available in [KC86].

It is often necessary to send data from one host to another, or to save application data structures to stable storage so that they can be either operated on later (possibly by other applications) or used for recovering the application state in case of a crash. In systems that do not support large shared address spaces, it is usually necessary to write routines that flatten data structures into a low-level linear bytestream and manually reconstruct them later. In general, this procedure tends to be tedious and error-prone because programmer intervention and coding is required for the appropriate data structure conversions. In addition, the linear representations bypass type systems, lose all pointer semantics and object identity, and leave the burden of maintaining data structure consistency up to the programmer.

Persistent objects have the ability to outlive the execution of the program that creates them; in contrast, transient objects disappear with the termination of the program in which they were created. A persistent object store is a repository for persistent objects; it is typically used to allow programmers to save complex pointer-linked data structures directly and automatically to non-volatile storage, without requiring additional code or further intervention to convert between representations. A persistent object store strongly supports the notion of object identity because all type information, as well as the topology of the data structures, is preserved when objects are saved to stable storage, allowing each object to be uniquely identified.

In this dissertation, we propose and implement a persistent storage mechanism based on a coarse-grained address translation technique that exploits existing virtual memory hardware and operating system facilities to achieve high performance and increased portability. The basic idea is to load (one or more) pages from the persistent storage into memory *on demand*, and "fix up" (i.e., translate) all persistent pointers into local hardware-supported virtual memory pointers, guaranteeing that a running program will *never* see any untranslated pointer values. Once the data has been loaded into memory and various pointers have been translated, there is absolutely no overhead for future accesses to that data. Any referents corresponding to newly-translated pointers that are not in memory are marked as such using the operating system's virtual memory protection facilities; any attempt to access protected data will raise an exception that is handled by loading the data from the persistent store and translating pointers as necessary.

An address translation mechanism such as ours has several advantages over other traditional approaches, in terms of both performance and portability. As we will show in later chapters of the dissertation, the performance overhead of our system is *zero* when the application is operating on data that has already been loaded into memory and there is no faulting. For other situations, that is, when new data is being loaded into memory from the persistent store, the overhead of our system is very low, usually between *1 and 5 percent* of the overall run time. We will show that this is much smaller than the I/O costs incurred for loading the data into memory. There are also some indirect costs of our approach because of an inadvertent interaction with the underlying virtual memory system; these costs, however, can be avoided by improving the operating system implementations to provide a better memory management interface. Regarding improved portability, our persistent storage system is compatible with standard off-the-shelf compilers and has been ported to a variety of modern operating systems such as SunOS, Solaris, Linux, Mach, Ultrix, OS/2, etc. It should also be relatively easy to port to other modern operating systems such as Windows NT.

1.1 Scope of the Dissertation

This dissertation is about high-performance address translation techniques for implementing orthogonal persistence. *Orthogonal persistent systems* require that any arbitrary object can be made persistent without regard to its type. That is, persistence is a storage class of an object, and is orthogonal to its type. Our basic approach relies on the use of coarse-grained address translation for performance and portability reasons. In order to support this claim, we analyze the various costs of both coarse-grained and fine-grained techniques and present a cost model which shows that page-wise address translation can be implemented efficiently and can achieve good performance on standard hardware.

1.1.1 Our Thesis

Address translation is the most important issue that must be considered when implementing orthogonal persistence. Once this has been resolved effectively, all other related issues can be resolved independently without affecting address translation. Our thesis can be stated as follows:

> *High-performance address translation for orthogonal persistence can be effectively realized through coarse-grained translation schemes. Pointer swizzling at page fault time is one such coarse-grained scheme that can be implemented efficiently on stock hardware by exploiting the existing virtual memory hardware and protection facilities offered by most modern operating systems, without requiring special system privileges.*

Since pointer swizzling at page fault time uses only standard capabilities of an operating system, it is easily ported to other modern operating systems which also support the same functionality. The basic approach exploits user-level virtual memory protection facilities to avoid (more expensive) software checks for pointer formats, and works with standard off-the-shelf compilers.

1.1.2 Motivation

There are a variety of factors that motivate the need for a flexible and efficient persistence mechanism. Many applications operate on large amounts of data represented using complex data structures. Before such an application terminates execution, the data in volatile memory must be saved to stable storage for future use. A persistent storage system is designed to save and restore data reliably, efficiently and automatically, and is therefore preferable to an *ad hoc* mechanism implemented by the application itself. The underlying persistent object store must be able to support large volumes of data, essentially acting as an eventual replacement for the normal file system.

However, it is important to realize that for most applications, a persistent programming language is still a programming language, and raw speed of computation is usually very important. In fact, orthogonally persistent programming languages (and object-oriented database systems) are largely motivated by a combination of performance and expressiveness considerations relative to traditional database systems. They are intended for use in applications with *rich heap-allocated data structures* and *efficient algorithms* to manipulate those data structures. Where traditional database systems are designed largely to optimize I/O for I/O-intensive applications, persistent programming languages allow programmers to optimize computation for CPU-intensive applications such as CAD tools and simulation programs. Although it is desirable for such programs to be able to transparently traverse pointers through large amounts of disk-resident data, the majority of their execution time is usually spent operating on persistent data previously loaded into memory.

Furthermore, many such applications usually also operate extensively on transient data. Typically, a large majority of these transient objects constitute temporary data that die fairly "young" [Wil97, WJNB95]. Thus the total execution costs in a CPU-intensive application are dominated by operations on transient objects and in-memory persistent objects. A high-performance persistent system should allow these operations to be executed as fast as possible, while imposing minimal overheads on the overall performance.

Thus there is a need for an address translation mechanism that incurs extremely low overheads during CPU-bound operations, low overheads during I/O-bound operations, and no overhead for operations on transient data. Ideally, the mechanism should also work with the best available high-performance compilers without requiring any significant changes or special support from these compilers.

1.1.3 Cost of Orthogonal Persistence

The cost of using orthogonal persistence in an application can be divided into two major components: the cost incurred when accessing data on disk, and the cost incurred when *not* referencing any data on disk (i.e., when not using persistence). The former is the normally-expected cost associated with loading persistent data from secondary storage into memory. The latter, however, represents costs incurred while performing normal CPU-bound operations on data that is already in memory. This is important because although reducing I/O cost is beneficial for most CPU-intensive applications, maintaining computation performance almost always has higher priority. In general, we define the *cost of orthogonal persistence* as the "cost" of making the distinction between transient and persistent data access transparent to the programmer, that is, it is the cost incurred when *not using* the persistence mechanism.

Traditional fine-grained address translation mechanisms implemented by persistent programming languages incur significant overhead and have fundamental performance limitations for normal CPU-intensive applications. Such schemes typically incur *continual overhead* for checking pointer formats—even if a pointer references in-memory data, a validity check is still necessary before it can be dereferenced because compiled code does not "know" that the pointer is already in an appropriate format.[1] Furthermore, these techniques require sophisticated custom compilers to generate additional code for checking and translating pointer values as necessary. This can be a major downside of such approaches because there are few resources to extensively develop, distribute, and support custom compilers. In fact, the cost of using compilers with poor code generation is typically higher than the cost of address translation itself, defeating the original purpose of building a high-performance implementation. Even if resources were available for compiler development, the continual costs of validity checks make the approach less attractive.

1.1.4 Overview

Pointer swizzling at page fault time satisfies all of the requirements outlined earlier for a high-performance address translation scheme. It incurs *zero* overhead during normal CPU-bound operations on data that has already been loaded into memory, and a *small* overhead—roughly *1 to 5 percent* depending on the underlying hardware and operating system—when data is

[1] It is possible to use compiler optimizations (similar to the Self system [Cha92]) to infer information about the data and reduce the excessive checking overhead. However, such optimizations are fairly hard to implement because of inherent distinctions between object types and object residency.

being faulted into virtual memory from persistent storage on disk. This technique does incur a small space overhead for storing meta-data that is necessary for facilitating address translation, but this is very small compared to the amount of persistent data that can be supported. We expect that the overheads incurred when loading data from disk will be reduced further as CPU speeds improve faster relative to disk speeds.

Pointer swizzling at page fault time also works with existing high-performance off-the-shelf compilers without requiring any additional support from these compilers. This is possible because the approach does not require extending the language syntax or relying on the run-time system for implementing necessary checks and translation. In addition, the basic approach is portable to different operating systems because it requires only minimal support from the underlying virtual memory system.

We can also support a larger (e.g., 64-bit or more) address space on a 32-bit machine using pointer swizzling at page fault time as a general-purpose address reconciliation layer. At the same time, it still has advantages on hardware where address reconciliation may not be necessary. In such cases, it can be used for sharing data across multiple machines with different native formats. Persistent data can be maintained in a common data format that is independent of the hardware word size of the different machines operating on that data, and can be translated into appropriate local addresses as necessary.

The key idea behind our approach is a novel layering of mechanisms. We rely on the operating system and the compiler to do their "jobs" mostly as usual, but strategically intervene at appropriate points for mapping one level of abstraction onto another with techniques such as non-traditional use of virtual memory hardware (in particular, the translation lookaside buffer, or the TLB) and extraction of object layout information from compiler-generated debugging information. We believe that approaching the problem at the right level of abstraction helps in resolving various issues independent of each other.

As part of this dissertation, we have implemented coarse-grained address translation using pointer swizzling at page fault time to provide an efficient persistence mechanism for C++. In doing so, we have essentially implemented a form of *reflection* [KdRB91][2] for C++ via a "back door" because the language itself does not provide builtin support for it.[3] Although our approach is simple and elegant, there are still some complexities—albeit hidden from the average user—in the implementation, due to a lack of language features. The mechanism can be made more general, and easier to implement, with improved language support for reflection.

1.1.5 Contributions

This dissertation makes several useful contributions:

- a novel address translation technique that is mostly independent of the underlying operating system implementation, and can be implemented efficiently on stock hardware;

[2] Reflection can be loosely defined as the ability to manipulate or change the internal behavior of a system without actually modifying its implementation (i.e., from the "outside").

[3] C++ does provide some support for reflection, most notably via the operator overloading capability for normal classes. However, it falls short of complete support because builtin types (including pointers) are treated differently than user-defined classes.

- a new classification scheme based on granularity of several important design choices for implementing orthogonal persistence;
- a detailed performance analysis of various components of a coarse-grained address translation mechanism;
- notion of run-time type description for providing implementation-level information about object layouts at run time;
- a persistent storage system for C++;
- a technique for dynamically resolving C++ method dispatch tables (virtual function tables) in applications against those in persistent storage; and finally,
- an analysis of interactions with operating system implementations and recommendations for improving these implementations to provide better support for system extensions such as persistence, garbage collection, etc.

Novel Address Translation Technique

Pointer swizzling at page fault time is a novel address translation mechanism that exploits existing virtual memory hardware and operating system features to efficiently implement orthogonal persistence. The approach is highly portable because it uses only standard features provided by modern operating system, and is also compatible with existing high-performance compilers for languages such as C and C++. Pointer swizzling at page fault time is classified as a coarse-grained address translation technique because the granularity of translation is a virtual memory page.

New Classification Scheme

Various researchers have put forth different taxonomies for address translation approaches based on differences in the pointer swizzling techniques used [Mos92, KK95, MS95, Whi94]. Unfortunately, some of these classifications are unclear, and sometimes even contradictory to each other. Instead of attempting to clarify these taxonomies, we present a new classification scheme using several design choices that we consider important for implementing orthogonal persistence. The classification is presented in terms of the granularity of design choices because we believe that granularity selection is the fundamental issue for implementing persistence.

Performance Analysis

We present detailed performance analysis for various components of a coarse-grained address translation technique, and evaluate the overhead of page-wise address translation against the I/O costs incurred during benchmark operation. As part of the performance results, we also describe our benchmarking philosophy which contends that standard database benchmarks do not accurately model real-world applications, and are not very flexible or configurable. As such, these benchmarks are appropriate only for controlled use in measuring performance of individual components of a persistence mechanism and deriving qualitative conclusions about the system rather than for comparative analysis across multiple systems.

Run-Time Type Description

All address translation techniques require knowledge about the structural layouts of data objects in-memory at run time. This is necessary in order to locate and translate all addresses (pointer fields) in each object that is loaded into memory. We introduce the term *Run-Time Type Description (RTTD)* to describe such implementation-level type information about data objects that is made available to the address translation mechanism at run time. Since C++ is not sophisticated enough to provide builtin support for RTTD, we have implemented our own RTTD mechanism for C++ using compiler-generated debugging information.

A Persistent Storage System for C++

We have implemented the pointer swizzling at page fault time scheme in the Texas persistent storage system to provide persistence for C++. Texas has been ported to several modern operating systems and is highly suitable as a prototype framework for further research. The system comprises of less than 10,000 lines of C++ source, and the design is modularized such that the code for distinct functionalities (for example, address translation mechanism or operating system interaction) has been separated into individual modules. The system is available via anonymous ftp in source form under the GNU Library General Public License.

A Technique for Dynamically Resolving C++ Method Dispatch Tables

C++ implements dynamic binding by using virtual functions [Lip91], and pointers to these functions are stored in virtual function tables (VFTs). When an object of a particular class is instantiated, a pointer to the corresponding VFT is (automatically) inserted in that object—dynamic method dispatch is implemented by indexing into the virtual function table of the object on which the method is originally invoked. Unfortunately, unlike data pointers, the VFT pointer in the object points into the *code segment*, and is therefore tightly coupled with the application. Further, the actual value of the pointer usually varies across applications (or even different versions of the same application). This is obviously a problem for persistent objects which are not related to any specific application. Therefore we dynamically resolve VFT pointers specially by "unswizzling" them into special token values that can later be identified and "swizzled" into actual values valid in the *current* application. In effect, this is equivalent to an *extremely simplified* dynamic linker that resolves VFT pointers in persistent objects against the appropriate values in the current application. Further details about the exact mechanism are described in Chapter 4.

Analysis of Operating System Interactions

Finally, we describe various issues that are related to the interaction of low-level systems (e.g., persistent stores and garbage collectors) with the underlying operating system implementations. We present an analysis of different aspects of virtual memory management and provide recommendations for changes in operating system implementations to improve their coupling with low-level system extensions, and contribute towards making them more portable. We also discuss a few other relevant operating system features such as virtual memory protection, fault handling, etc.

1.2 Advanced Issues

In addition to the various contributions described above, there are many other advanced issues that are beyond the scope of this dissertation and are therefore not addressed here. Some of these issues are:

- *schema evolution*: Currently, there is no support for schema evolution in Texas, although this is independent of address translation and can be implemented on top if necessary. Of course, language support for reflective techniques would be very helpful in such an implementation;

- *security*: We also do not address security issues for access to data in the persistent object store, but it is easy to imagine an implementation along the lines of protection domains in Opal [CLLBH92] or "areas" as in ObjectStore [LLOW91], or just Unix-style owner and group privileges (also supported by Opal); and

- *distribution and fault tolerance*: These issues need to be carefully designed and architected, and must be implemented to interface well with the basic address translation mechanism. However, it is not an impossible task—we are aware of at least one project where the Texas persistent store has been ported to a Fujitsu AP1000 multicomputer [BS96].

1.3 Organization of the Dissertation

The rest of this dissertation is organized as follows.

Chapter 2 describes several important design issues for implementing an orthogonal persistence mechanism. We present a new classification scheme for persistence mechanisms that is based on the granularity choices for different issues, namely, *address translation*, *address mapping*, *data fetching*, *data caching*, and *checkpointing*. For any persistent system, each issue can be resolved at a granularity that is independent of the granularity choice for any other issue. In addition, we also discuss the granularity choices that we have made for each design issue in our implementation of the pointer swizzling at page fault time mechanism in the Texas persistent store.

Chapter 3 contains a detailed description of the pointer swizzling at page fault mechanism. We describe the basic algorithm as well as discuss various related issues such as address space management, and sharing and compatibility with existing systems and code. Although our coarse-grained approach works well for most cases, there are situations where the lack of locality of reference in an application's data structures requires a less coarse-grained approach for address translation. To this end, we discuss fine-grained and mixed-granularity address translation techniques that can also be portably implemented along with the basic coarse-grained technique. We present a competitive argument that pointer swizzling at page fault time incurs zero overhead when the data is already loaded into memory (CPU-bound operations) and a very small overhead during the loading of data from the persistent storage (I/O-bound operations).

Chapter 4 describes the design and implementation of the Texas persistent store, an efficient persistent storage system for C++ that uses the pointer swizzling at page fault time mechanism as a key component for high-performance address translation. We describe the basic design and implementation of Texas; the implementation details also include information about virtual memory and file system abstraction layers designed to make interactions with the underlying operating system easy to implement. It should be noted that although Texas relies on virtual memory caching, it is simply an implementation choice that is completely orthogonal to the address translation mechanism. We intend for Texas to be used as a research platform for further studies in issues related to efficient orthogonal persistence implementation. In the current context, we use it for gathering detailed performance results of the pointer swizzling at page fault time technique.

Chapter 5 presents detailed results for the performance of Texas and pointer swizzling at page fault time using the OO1 database benchmark traversal operations. The performance results presented in this chapter empirically validate our basic competitive argument for pointer swizzling at page fault time. We have measured the performance on Linux and Solaris, two of the most popular operating systems, and show that the total overhead of Texas is usually between 1 and 5 percent for most situations on both platforms. In addition to the empirical results, we also describe our philosophy for benchmarking; specifically, we argue that some of the widely-used standard benchmarks are inappropriate for quantitative performance comparison across different persistent systems, and are acceptable only for controlled measurements and qualitative analysis of a single system. These benchmarks do not represent real applications which are most likely to take advantage of a persistent storage system. We believe that such applications typically have sophisticated data structures and perform significant CPU-bound computations over these data structures, unlike the benchmarks which do not offer rich data structures and include minimal computation behavior.

Chapter 6 tackles the issue of providing implementation-level information for types at run time. We call this Run-Time Type Description (RTTD) to distinguish it from the recently-introduced Run-Time Type Identification (RTTI) feature for C++. RTTD constitutes information about types and in-memory layouts of data objects, and is necessary for the correct operation of pointer swizzling at page fault time. In contrast, RTTI supports only language-level information such as run-time type equivalence checks, which is obviously insufficient for object layout information. RTTD may also be useful for other systems such as garbage collectors, advanced profiling and tracing tools, etc. which also benefit from detailed object layout information. We describe our basic strategy which uses compiler-generated debugging information instead of special-purpose preprocessors, and present arguments about why our approach is preferable. The chapter also presents details about our RTTD implementation for C++ which is currently used for Texas and a real-time garbage collector. We also describe expected performance characteristics of our approach based on some preliminary measurements and show that the additional overhead is negligible compared to the typical compilation and linkage costs.

Chapter 7 is devoted to a discussion of various issues that are important for portability and interaction with various operating system implementations. It also suggests directions for improving operating system implementations to make it easy for integrating low-level system extensions such as persistence mechanisms, garbage collectors, etc. We are mainly interested in interaction with the virtual memory system because pointer swizzling at page fault time is primarily dependent on existing virtual memory hardware and protection facilities supported by the operating system. An important point that is highlighted during the discussion is that unlike some other systems, pointer swizzling at page fault time does *not* require advanced capabilities from the operating system although it is designed to exploit such capabilities, if they exist, for improved performance. We also briefly describe other operating system features, particularly virtual memory protection violation handling, that can be improved for overall performance gains.

Chapter 8 contains a brief sketch of some future research directions that appear promising for further study of high-performance address translation techniques and other extensions to the Texas persistent storage system. Many of these issues are related to the development of a competitive storage management technique for efficient checkpointing and stable storage capabilities. We also reiterate the advanced issues (mentioned above) that are beyond the scope of the current discussion. Finally, we summarize our findings and conclude in Chapter 9.

Chapter 2

Design Issues for Persistence

2.1 Introduction

In this chapter, we provide a general background for persistence, including a description of different types of persistence that are commonly implemented by various persistent systems. Our approach is designed to implement *orthogonal persistence* which provides the cleanest implementation model by separating the persistence property of an object from its type.

We also briefly describe some existing taxonomies for address translation mechanisms, and show that they are unclear for general usage. Instead, we choose to use the *granularity* of an operation as the metric for classifying different persistence mechanisms. We identify a set of basic design issues that must be considered when implementing a persistent system, and define the classification in terms of granularity choices for these design issues. We argue that this classification is better than the existing taxonomies which are primarily concerned only with the issue of address translation.

Another interesting part of this chapter is a brief discussion on fine-grained address translation and how it measures up to a coarse-grained mechanism such as ours. As we will show, fine-grained translation schemes incur some basic costs that are inherent to their general implementations. We believe that fine-grained approaches should be avoided except in those cases where other features such as locking, consistency, etc. are also being implemented at a finer granularity. Finally, we provide a background survey of some other research in persistence that is most relevant to the current discussion.

2.2 Background

File systems have traditionally been used to save data both for temporary storage between program executions and for general long-term storage. Unfortunately, file systems have efficiency drawbacks, because there are two parallel memory hierarchies (disk and RAM[1] in both), and data must be moved between them. Often there are situations where the same data exists in both memory caches, and in both disk areas; this is a poor use of resources. Also, a file system's normal view of data is a "stream of bytes" with no associated structure or type information,

[1] In this dissertation, we use the terms *RAM* and *main memory* interchangeably for referring to the physical memory in a computer system.

while the in-memory representation of data structures is in terms of pointer-linked objects. This creates a fundamental *impedance mismatch* [CM84] between the two representations.

Persistent systems are designed to solve this impedance mismatch between volatile and non-volatile storage, and to alleviate the efficiency problems associated with file systems. In this section, we provide a general background on persistence, including descriptions of various types of persistence, before briefly discussing our approach which is designed to implement persistence for C++ and other high-level languages.

2.2.1 Persistence

All data created and manipulated by normal applications are usually *transient* in nature because their lifetime is bounded by the execution of the process in which they were created. In contrast, *persistent* data can outlive the execution of the process that creates them. Persistent object stores are repositories used for storing arbitrarily complex persistent data structures while maintaining pointer semantics just as in virtual memory. In essence, a persistent object store can be viewed as a long-lived virtual memory that persists after applications complete execution, and which can be accessed by the same (or different) applications when they are run again in the future.

2.2.2 Types of Persistence

We classify persistence implementation mechanisms into different kinds based on the type of persistence supported by each specific approach. In general, persistence can be broadly divided into three kinds:

- class-based persistence,
- orthogonal persistence, and
- reachability-based persistence.

The simplest persistence mechanism incorporated in many applications relies on *class-based persistence*. The basic idea requires that any type or class which may be instantiated to create persistent objects *must* inherit from a top-level abstract "persistence" class. This special class defines the *interface* for saving and restoring data from a persistent object store. Each derived class that inherits from the top-level class is required to *implement* the specified interface (possibly via serialization methods) to save and restore objects of that particular derived class. This is obviously cumbersome for the programmer who must carry the burden of implementing the persistence mechanism, making the whole process extremely tedious and highly error-prone. Another problem with this approach is that it promotes code duplication in the usual case. Any type that may potentially be used to create persistent objects requires two definitions—one for normal transient objects and the other (derived from the special abstract class) for persistent objects. As a result, transient and persistent objects of the same "logical" (application) type are now *not equivalent* in terms of the "physical" (actual) type, and code that operates on one kind of object cannot operate on the other. One obvious solution is to make the derived class multiply inherit from both the actual type and the abstract class but this is likely to add a slew of other problems related to the use of multiple inheritance. Also, this approach does not work for builtin types because their definitions cannot be changed easily in most languages.

Unlike class-based persistence, *orthogonal persistence* [ABC$^+$83a, AM95] decouples the lifetime of an object from its type. In other words, persistence is viewed as a *storage class*[2] rather than as a property of the object type. The name derives from the requirement that the type of an object must be independent of (that is, orthogonal to) its storage class. In other words, persistence is a property of individual objects, not of their classes (or types), and any object can be made persistent regardless of its type. Since persistence is decoupled from the type system, this approach supports a clean implementation model that is transparent to the application programmer who does not need to make any major modifications to the application code to use the persistence mechanism.

Finally, *reachability-based persistence* [ABC$^+$83a, ACCM83] is a general form of orthogonal persistence. The basic principle of this approach requires that all objects reachable from a well-defined persistent root (or roots) automatically become persistent. As with orthogonal persistence, the type of an object is not relevant when making it persistent based on the reachability property. The implementation ease for this approach depends on the support available from the programming language. In general, we believe that orthogonal persistence (and its derivatives such as reachability-based persistence) are preferable to class-based persistence or other *ad hoc* mechanisms.

2.2.3 Our Approach

Historically, implementations of persistence mechanisms have been slow due to at least two different cost factors. One of them is a direct (and fairly significant) cost of actions such as checking pointer formats, maintaining bookkeeping information, etc. in software. The other is an indirect cost related to the use of specialized compilers for implementing persistence through language extensions. Typically, there has been a lack of resources for extensive development of these specialized compilers and as a result, code generated by such compilers is often several times slower than that generated by most widely-available, high-performance optimizing compilers. In addition, we believe that fundamentally slow approaches used for implementing traditional address translation techniques are another potential source of performance problems. For example, some pointer-wise translation techniques require that the format of a pointer be checked every time it is dereferenced, *even if it is a transient pointer.*

We solve both these problems by designing a novel implementation strategy that is compatible with code generated by existing off-the-shelf compilers without requiring any special modifications or sacrificing optimization opportunities. We also reduce our overheads to a minimum by effectively using existing hardware to check for pointer formats, thereby avoiding software checks which are usually more expensive. This has an overall effect of removing major obstacles in the acceptance of general-purpose languages for persistent applications because of its performance and compatibility with both stock hardware and existing compilers.

[2] A storage class describes how an object is stored. For example, the storage class of an automatic variable in C or C++ corresponds to the stack because the space for the object is typically allocated on the data stack, and its lifetime is bounded by the scope in which it was allocated.

Although our approach is focused on implementing mainly orthogonal persistence (for languages such as C and C++), it is designed to be compatible with reachability-based persistence. However, the current implementation of Texas does not support reachability-based persistence. The primary obstacle in implementing this is the lack of language support for identifying type information for arbitrary data on both the stack and the transient heap. This is essentially the same problem as the one faced by garbage collectors for languages such as C or C++. We believe that it is straightforward to use solutions that are similar to those applicable in the other domain[3] but we have not yet done so. We are aware of at least one project that is using pointer swizzling at page fault time techniques and extending Texas to implement reachability-based persistence for C++ and Modula-3 [HN97].

2.3 Address Translation Taxonomies

Persistence has been an active research area for over a decade and several researchers have put forth taxonomies for pointer swizzling techniques [Mos92, KK95, MS95, Whi94]. In this section, we describe important details about each of these taxonomies and highlights various similarities and differences among them. In addition, we also provide motivation for a general classification of persistent systems based on granularity issues.

2.3.1 Eager vs. Lazy Swizzling

Moss [Mos92] describes one of the first studies of different address translation approaches and the associated terminology developed for classifying these techniques. The primary classification is in terms of "eager" and "lazy" swizzling based on *when* the address translation is performed. Typically, eager swizzling schemes swizzle an entire collection of objects together, where the size of the collection is somehow bounded. In other words, the need for checking pointer formats, and the associated overhead, is avoided by performing aggressive swizzling. In contrast, lazy swizzling schemes follow an incremental approach by using dynamic checks for unswizzled objects. That is, there is no predetermined or bounded collection of objects that must be swizzled together. Instead, the execution dynamically locates and swizzles new objects depending on the access patterns of the application.

Other researchers (Kemper and Kossman [KK95], and McAuliffe and Solomon [MS95]) have also used classifications along similar lines in their own studies. However, we consider this classification to be ambiguous for general use because it does not clearly identify the fundamental issue—*the granularity of address translation*—that is important in this context. For example, consider pointer swizzling at page fault time using this classification. By definition, we swizzle all pointers in a virtual memory page as it is loaded into memory and applications are never allowed to "see" any untranslated pointers. As such, there is no need to explicitly check the format of a pointer before using it[4] and therefore, pointer swizzling at page fault time is an eager swizzling scheme. On the other hand, the basic approach is incremental in

[3] Typically, conservative garbage collectors operate by scanning the stack and treating any value that *appears to be a pointer* as a pointer.

[4] The format checking is actually done implicitly by hardware based on the use of virtual memory access-protections.

nature because swizzling is performed one page at a time and only on demand, making it a lazy swizzling scheme as per the original definition.

In general, a scheme that is "lazy" at one granularity is likely to be "eager" at another granularity. For example, a page-wise swizzling mechanism is lazy at the granularity of pages because it only swizzles one page at a time. However, the same scheme would be considered an eager swizzling scheme at the granularity of objects because it swizzles multiple objects—an entire page's worth—at one time. Therefore, the fundamental issue is the granularity at which address translation is performed.

2.3.2 Node Marking vs. Edge Marking Schemes

In addition to eager and lazy swizzling, Moss also describes another classification based on the strategy used for distinguishing between resident and non-resident data in the case of "lazy" swizzling (i.e., the incremental approach). The persistent heap and various data structures are viewed as a directed graph, where data objects represent *nodes* and pointers between objects represent *edges* that connect the nodes. Given this view, the address translation mechanisms are then classified as either *node marking* or *edge marking* schemes.

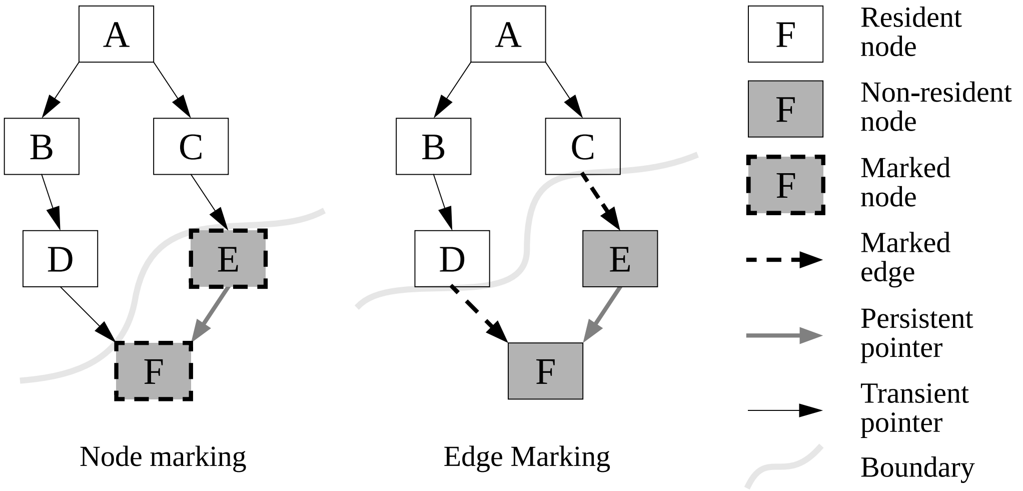


Figure 2.1: Node marking and edge marking schemes

Figure 2.1 pictorially shows the basic technique for both node marking and edge marking schemes. As the name suggests, *edge marking* schemes mark edges of the graph—the pointers between objects—to indicate whether they have been translated into local format and reference resident objects, or not. In contrast, *node marking* schemes guarantee that all references in resident objects are always translated and the graph nodes are marked as resident or non-resident. In other words, edges are guaranteed to be valid local references but the referents may be non-resident.

Figure 2.2 shows a classic implementation of a node marking scheme; non-resident nodes are "marked" as such by using *proxy* objects, that is, pseudo-objects that stand in for non-resident persistent objects and contain their corresponding persistent identifiers. When an object is loaded from the database, all references contained in that object must be swizzled as per the definition of node marking—pointers to resident objects are swizzled normally

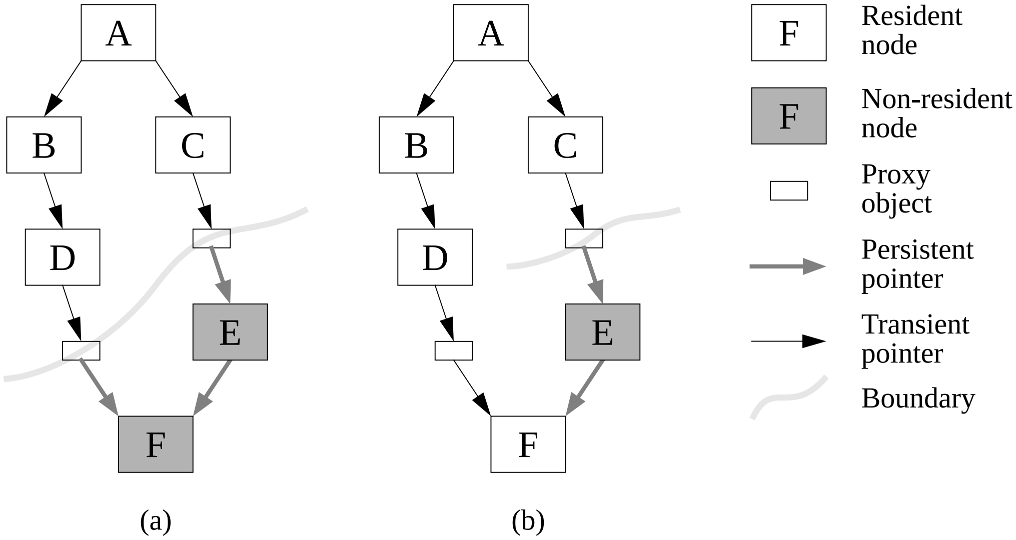


Figure 2.2: Node marking using proxy objects

while pointers to non-resident objects are swizzled into references to proxy objects. When the application follows a reference to a proxy, the system loads the referent (F in the figure) from the database and updates the proxy to reference the newly-resident object (Figure 2.2b). Alternatively, the proxy may be bypassed by overwriting the (old) reference to it with a pointer to the newly-resident object; if there are no other references to it, the proxy may (eventually) be reclaimed by the system. Note, however, that the compiled code must still check for presence of the proxy object on *every* pointer dereference because any general pointer *may* reference a proxy object. This adds continual checking overhead, even if all pointers directly reference data objects without any intervening proxy objects.

Pointer swizzling at page fault time is essentially a node marking scheme because swizzled pointers always correspond to valid virtual memory addresses while the referents are distinguished on the basis of residency. However, it differs in an important way from the classic scheme—we do not use *explicit* pseudo-objects for non-resident nodes. Instead, access-protected virtual address space pages act as proxy objects. The biggest advantage of this approach is that there is no need to reclaim proxy objects (because none exist) as the application progresses and more data is loaded from the database; consequently, there are no indirections that must be dealt with by compiled code, and *continual checks are not necessary.*

2.3.3 General Classification for Persistence

We have shown that existing classifications describe address translation techniques which constitute only one of several design issues that must be considered when implementing persistence. We have identified a set of five design issues that we believe are fundamental to efficient implementation of any persistence mechanism. We contend that a specific combination of these issues can be used to characterize a particular persistence implementation. In effect, we are proposing a classification scheme based on granularity of fundamental design issues.

As described earlier, a classification based on eager and lazy swizzling is likely to be ambiguous because it does not attack the problem at the right level of abstraction. Instead, we notice that the real issue in the lazy vs. eager swizzling distinction is the size of the unit of storage for which address translation is performed. This can range from as small as a single reference (as in Moss' so-called "pure lazy swizzling" approach) to a virtual memory page (as in pointer swizzling at page fault time), or even as large as an entire database (as in the so-called "pure eager swizzling" approach in Moss' terminology).

Based on this observation, we believe that it is better to consider address translation (and other design issues) from the perspective of a *granularity choice* rather than as an *ad hoc* classification based on confusing translation semantics. In fact, the ambiguity described above arises because the classifications either do not clearly identify the granularity choices or unnecessarily adhere to a single predetermined granularity.

We believe that addressing all design issues in terms of granularity choices enables a uniform process for identifying the consequence of each design issue on the performance and flexibility of the resulting persistence mechanism. This is preferable to ambiguous classifications such as eager and lazy swizzling because any scheme is both "eager" and "lazy" at different granularities.

2.4 Granularity Choices for Persistence

Address translation is only one of several issues that must be resolved when implementing an orthogonally persistent system. We have identified a set of five design issues that are relevant to the implementation of any persistence mechanism. Each of these issues can be resolved by making a specific granularity choice that is independent of the choice for any other issue. The combination of granularity choices for all issues can be used to characterize a persistent system.

The specific design issues that we describe in this section are the granularities of *address translation*, *address mapping*, *data fetching*, *data caching* and *checkpointing*. We define and discuss each issue in detail and also present the rationale behind the granularity choices for these issues in our implementation of orthogonal persistence in the Texas persistent storage system.

To a first approximation, the basic unit for all granularity choices in Texas is a virtual memory page because pointer swizzling at page fault time relies heavily on virtual memory facilities, especially to trigger data transfer and address translation. The choice of a virtual memory page as the basic granularity unit allows us to exploit conventional virtual memories, and avoid expensive run-time software checks in compiled code, by taking advantage of user-level memory protection facilities of most modern operating systems. However, sometimes it is necessary to change the granularity choice for a particular issue to accommodate the special needs of unusual situations. As we will explain below, these issues can be addressed at a different granularity in a way that integrates gracefully into the general framework of Texas.

2.4.1 Address Translation

The granularity of *address translation* is the smallest unit of storage within which all pointers are translated from persistent (long) format to virtual memory (short) format. In general, the spectrum of possible values can range from a single pointer to an entire page or more.

The granularity of address translation in Texas is typically a virtual memory page for coarse-grained translation implemented via pointer swizzling at page fault time. The rationale for this choice is the advantages offered by the use of virtual memory pages in terms of overall efficiency because we can use the virtual memory hardware to check residency of the referents. In addition, we also rely on the application's spatial locality of reference to amortize the costs of handling protection faults and swizzling entire pages.

As we will describe in Chapter 3, it is also possible to implement a fine-grained address translation mechanism for special situations where the coarse-grained approaches are unsuitable because of poor locality of reference in the application. Since Texas is designed to perform fine-grained translation on individual pointers, the granularity of address translation in those cases would be a single pointer.

2.4.2 Address Mapping

A related choice is the granularity of *address mapping*, which is defined as the smallest unit of addressed data (from the persistent store) that can be mapped independently to an area of virtual address space.

To a first approximation, this is a virtual memory page in Texas because any page of persistent data can be mapped into an arbitrary page of the virtual address space of a process. A major benefit of page-wise mapping is the savings in table sizes; we only need to maintain tables that contain mappings from persistent to virtual addresses and vice versa on a page-wise basis, rather than (much larger) tables for recording the locations of individual objects. This reduces both the space and time costs of maintaining the address translation information.

The granularity of address mapping is bigger than a page in the case of large (multi-page) objects. When a pointer to (or into) a large object is swizzled, virtual address space must be reserved for all pages that the large object overlaps. This reservation of multiple pages is necessary to ensure that normal indexing and pointer arithmetic works as expected within objects that cross page boundaries. The granularity of address mapping is then equivalent to the number of pages occupied by the large object.

2.4.3 Data Fetching

As the name suggests, the granularity of *data fetching* is the smallest unit of storage that is loaded from the persistent store into virtual memory. As with the other two granularities presented above, we use a virtual memory page for this purpose in the current implementation of Texas. However, the primary motivation for making this choice was simplicity and ease of implementation, and the fact that this correlated well with the default granularity choices for some of the other design issues in our implementation.

It is possible to change the granularity of fetching without affecting any of the other granularity choices. In essence, we can implement our own prefetching to preload data from the persistent store. As we will discuss in Chapter 7, this may actually be desirable for some applications when using raw unbuffered I/O instead of normal file I/O. Raw I/O is typically used to bypass the file system cache in order to avoid the *double caching* problem (see Chapter 7) but in doing so, we also lose the benefit of file system *readahead* (prefetching) mechanism. Depending on the access characteristics of the application and the dataset size, the overall cost of I/O can be reduced by prefetching several (consecutive) pages instead of a single faulted-on page. Thus the granularity of data fetching is closely tied to the exact I/O strategy selected in the implementation.

2.4.4 Data Caching

The granularity of *data caching* is defined as the smallest unit of storage that is cached in virtual memory. For Texas, the granularity of caching is a single virtual memory page because it relies exclusively on the virtual memory system for caching persistent data.

As we will describe in Chapter 4, a persistent page is usually cached in a *virtual memory* page as far as Texas is concerned. The underlying virtual memory system determines whether the page actually resides in main memory or on disk (i.e., swap space) without any intervention from Texas. This is quite different from some other persistent storage systems which directly manage physical memory and control the mapping of persistent data into main memory. In general, Texas moves data between a persistent store and the virtual memory *without regard to the distinction between virtual pages in main memory and on disk.*

It is, of course, possible to change this behavior such that Texas directly manages RAM (i.e., physical memory). However, we believe that this is unnecessary (and may even be undesirable) for most applications—the fact that Texas behaves like any normal application with respect to virtual memory replacement may be beneficial for most purposes because it prevents any particular application from monopolizing system resources (physical memory in this case). As we will discuss in Chapter 7, additional control over memory management is possible depending on the support available from the underlying operating system.

2.4.5 Checkpointing

Finally, we consider the granularity of *checkpointing* which is defined as the smallest unit of storage that is written to non-volatile media for the purpose of saving recovery information to protect against failures and crashes.

Similar to the address translation strategy, Texas uses virtual memory protections to detect pages that are modified by the application between checkpoints. Therefore, the default unit of checkpointing in the usual case is a virtual memory page. Texas employs a simple logging scheme to support checkpointing and recovery. At checkpoint time, modified pages are written to a log on stable storage before the actual database is updated.[5]

The granularity of checkpointing can be refined by the use of sub-page logging. The approach relies on a page "diffing" technique that we originally proposed in [SKW92], and also briefly describe it again in Chapter 4 of this dissertation. The basic idea is to save clean versions of pages before they are modified by the application; the original (clean) and modified (dirty) versions of a page can then be compared to detect the exact sub-page areas that are actually updated by the application and only those "diffs" are logged to stable storage. This technique can be used to reduce the amount of I/O at checkpoint time, subject to the application's

[5] Chapter 4 provides further details for checkpointing and recovery support in Texas.

locality characteristics. The granularity of checkpointing in this case is equivalent to the size of the "diffs" which are the units of storage saved to stable storage.[6]

Another enhancement to the checkpointing mechanism is to maintain the log in a compressed format. As the checkpoint-related data is streamed to disk, we intervene to perform some inline compression using specialized algorithms tuned to heap data. Further research has been initiated in this area [WKB97a, WKB97b] and preliminary results indicate that the I/O cost can be reduced by at least a factor of two (based on a 2-to-1 compression ratio). Further reduction in costs is possible with improved compression algorithms and adaptive techniques.

2.5 Fine-grained Address Translation

It is obvious from the foregoing discussion on granularity choices that pointer swizzling at page fault time is inherently a coarse-grained address translation mechanism. There are several factors that motivated us to develop and implement a coarse-grained mechanism over a fine-grained approach. Obviously, the primary motivation is related to the fact that we wanted to exploit existing hardware to avoid expensive software checks. However, we believe that there are also some other factors against fine-grained address translation. In this section, we present a discussion on fine-grained address translation techniques and why we believe that they are not practical for high-performance implementations in terms of efficiency and complexity.

Overall, fine-grained address translation techniques are likely to incur various hidden costs that have not been measured and quantified in previous research. In general, we have found that most current fine-grained schemes appear to be slower than pointer swizzling at page fault time in terms of the basic address translation performance.

2.5.1 Basic Costs

Fine-grained address translation techniques usually incur some inherent costs due to their basic implementation strategy. These costs can be divided into the usual time and space components, as well as the less tangible components related to implementation complexity. We believe that these costs are likely to be on the order of tens of percent, even in well-engineered systems with custom compilers and fine-tuned run-time systems. Some of the typical costs incurred in a fine-grained approach are as follows:

- A major component of the total cost can be attributed to *pointer validity* checks. These checks can include both *swizzling* checks and *residency* checks. A swizzling check is used to verify whether a reference is swizzled (i.e., translated into valid local format or not[7] while a residency check verifies whether the referent is resident and accessible. These two checks, while conceptually independent of each other, are typically combined in implementations of fine-grained schemes.

- Another important component of the overall cost is related to the implementation of a custom object replacement policy, which is typically required because physical memory

[6] The basic "diffing" technique has been implemented in the context of QuickStore [Whi94]; preliminary results are encouraging, although more investigation is required.

[7] For example, all swizzled pointers in Texas *must* contain valid virtual memory address values.

is directly managed by the mechanism that implements persistence. This cost is usually directly proportional to the rate of execution since it requires a read barrier[8] implementation approach. This cost component is discussed further in the next subsection.

- As resident objects are evicted from memory (during the course of replacement), a proportional cost is usually incurred in invalidating references to the evicted objects; this is necessary for maintaining *referential integrity* by avoiding "dangling pointers." This cost is also directly proportional to both the rate of execution and the locality characteristics of the application.

- By definition, fine-grained translation techniques permit references to be in different formats during application execution. This requires that pointers be checked to ensure that they are in the right format before they can be used, even for simple equality checks. It may also be necessary to check transient pointers depending on the underlying implementation strategy. As such, there is a continual pointer format checking cost that is also dependent on the rate of execution and pointer use.

- Finally, it is possible to incur other costs that exist mainly because of unusually constrained object and/or pointer representations used by the system. For example, accessing an object through an indirection via a proxy object is likely to require additional instructions. Another example is the increased complexity required for handling languages features such as interior pointers.[9]

Note that all cost factors described above do not necessarily contribute to the overall performance penalty in every fine-grained address translation mechanism. However, the basic costs are usually present in some form in most systems.

2.5.2 Object Replacement

Fine-grained address translation schemes typically require that the persistence mechanism directly manage physical memory because persistent data are usually loaded into memory on a per-object basis.[10] Therefore, it is usually necessary to implement a custom object replacement policy as part of the persistence mechanism. This affects not only the overall cost but also the implementation complexity.

As part of the replacement policy, a read barrier is typically implemented for every object that resides in memory. The usual action for a read barrier is to set one bit per object for maintaining recency information about object references to aid the object replacement policy. The read barrier may be implemented in software by preceding each object read with a call to the routine that sets the special bit for that object. Compiled code then contains extra instructions—usually inserted by the compiler—to implement the read barrier. (Alternatively,

[8] The term *read barrier* is borrowed from garbage collection research [Wil97], and is used to denote a trigger that is activated on every read operation. A corresponding term, *write barrier*, is used to denote triggers that are activated for every write operation.

[9] *Interior pointers* point inside the bodies (i.e., middle parts) of objects rather than at their heads.

[10] The data are usually read from the persistent store into a buffer in terms of pages for minimizing I/O overhead. However, only the objects required are copied from the buffer into memory.

it may be implemented with specialized hardware checks and/or microcoded routines.) The read barrier is typically expensive on stock hardware because, in the usual case, *all* read requests must be intercepted and recorded. It is known that one in about ten instructions is a *pointer store* (i.e., a write into a pointer) in Lisp systems that support compilation. Since read actions are more common than write actions, we estimate that between 5 and 20 percent of total instructions in an application usually correspond to a read from a pointer. The exact number obviously varies by application, and more importantly, by the source language; for example, it is likely to be higher in heap-oriented languages such as Java. It may be possible to use data flow analysis during compilation such that the read barrier can be optimized away for some object references; such analysis is, however, hard to implement.

The object replacement policy also interferes with general swizzling, especially if an edge marking technique is being used. In such cases, the object cannot be evicted from memory without first invalidating all edges that reference it. This obviously requires knowledge about all references to the object being evicted. Kemper and Kossman [KK95] solved this by using a per-object data structure known as a *Reverse Reference List (RRL)* to maintain a set of back-pointers to all objects that reference a given object. McAuliffe and Solomon [MS95] use a different data structure, called the *swizzle table*, which is a fixed-size hash table that maintains a list of all swizzled pointers in the system. Both these approaches are generally not favorable because they increase the storage requirements (essentially doubling the number of pointers at the minimum) and the implementation complexity.

2.5.3 Discussion

One of the problems in evaluating different fine-grained translation mechanisms is the lack of good measurements of system costs and other related costs in these implementations. The few measurements that do exist correspond to interpreted systems (except the E system) and usually underestimate the costs for a high-performance language implementation. For example, a 30% overhead in a slow (interpreted) implementation may be acceptable for that system, but will certainly be unacceptable as a 300% overhead when the implementation is sped up by a factor of ten.

Another cost factor for fine-grained techniques that has generally been overlooked is the cost of maintaining mapping tables for translating between the persistent and local pointer formats. Since fine-grained schemes typically translate one pointer at a time, the mapping tables must contain one entry per pointer. This is likely to significantly increase the size of the mapping table, making it harder to manipulate efficiently.

We believe that the E system [RC89, SCD90] is probably the fastest fine-grained scheme that is comparable to a coarse-grained address translation scheme; however, it still falls short in terms of performance. Based on the results presented in [Whi94], E is about 48% slower than transient C/C++ for the hot traversals of the OO1 database benchmark [Cat91, CS92].[11] This is a fairly significant overhead considering that the overhead of our system is *zero* for hot traversals and much smaller (less than 5%) otherwise. Even with generous estimates for performance improvements (say, double the performance), the costs might be reduced to only

[11] The hot traversals are ideal for this purpose because they represent operations on data that have already been faulted into memory, avoiding performance impacts related to differences in loading patterns, etc.

20% of total costs incurred for a transient application. This number is still quite high for general acceptance in mainstream applications.

We believe that there are several reasons why it is likely to be quite difficult to drastically reduce the overheads of fine-grained techniques. Some of these are:

- Several of the basic costs (outlined above) for a fine-grained translation scheme cannot be changed or reduced easily. For example, the pointer validity and format checks, which are an integral part of fine-grained address translation, cannot be optimized away.

- There is a general performance penalty (maintaining and searching large hash tables for mapping information, etc.) that is typically independent of the checking cost itself. As mapping tables get larger, it will be more expensive to probe and update them, especially because locality effects enter the overall picture.[12]

- Fancy data flow analysis and code generation techniques are required from the compiler to optimize some of the costs associated with the read barrier used in the implementation. Furthermore, such extra optimizations will probably cause unwanted code bloat (e.g., excessive loop unrolling).

- Although the residency property can be treated as a type so that Self-style optimizations [Cha92] can be applied to eliminate residency checking, it is not quite easy to do so because unlike types, residency can changes across procedure calls depending on the dynamic run-time state. In effect, residency check elimination is fundamentally a non-local problem that depends on complex analysis of control flow and data flow.

- It is necessary to check residency of an object *at least once* before it is used. Thus if many different unique objects are referenced by the application, the cost of initial residency checking must still be incurred. For example, an application may traverse a list containing several thousand objects. Unless the compiler can abstract the access over the entire collection, full residency checking cost will be incurred for every list element.

Based on these arguments, we believe that fine-grained translation techniques are not as attractive for high-performance implementations of persistence mechanisms.

Taking the other side of the argument, however, it can certainly be said that fine-grained mechanisms have their advantages. A primary one is the potential savings in I/O because most traditional fine-grained schemes fetch data only as necessary. There are at least two other benefits over coarse-grained approaches:

- fine-grained schemes can support reclustering of objects within pages, and

- the checks required for fine-grained address translation may also be able to support other fine-grained features (such as locking, transactions, etc.) at little extra cost.

In principle, fine-grained schemes can recluster data over short intervals of time compared to coarse-grained schemes. However, clustering algorithms are themselves an interesting topic

[12] Hash tables are known to have extremely poor locality because, by their very nature, they “scatter” related data in different buckets.

for research, and further studies are necessary for conclusive proof. We also make another observation that fine-grained techniques are attractive for unusually-sophisticated systems, e.g., those supporting fine-grained concurrent transactions. This becomes further attractive if the fine-grained checking is supported in hardware (as in the early Lisp machines).

2.6 Survey of Related Work

Persistence has been an active research area since the early eighties and various approaches have been proposed and developed for implementing different types of persistence. These approaches range from special languages (or language extensions) with builtin support for persistence to general-purpose languages that support persistence with the help of some kind of external mechanism. In this section, we survey several representative persistence mechanisms with respect to our proposed classification based on different granularity choices.

2.6.1 Persistent Programming Languages

We start by looking at approaches that incorporate persistence as part of the programming language implementation. Some of the important ones that we describe below are PS-algol, Napier88, LOOM, E and PS-Smalltalk. PS-algol and Napier88 are among the first persistent languages that implemented fine-grained orthogonal persistence. While E and PS-Smalltalk are implemented by extending existing popular non-persistent languages (C++ and Smalltalk respectively), LOOM is actually a virtual memory system for Smalltalk that implements persistence as a side effect of providing large virtual memory. We briefly describe the salient features of each language implementation and how it relates to our own approach. We focus exclusively on advanced languages that support object-oriented data models, and do not discuss database programming languages (e.g., Pascal/R, RIGEL, etc.) that extend relational programming techniques because the former are more relevant to our research.

PS-algol

PS-algol [ACC82, ABC$^+$83a, ABC$^+$83b] was the first truly persistent programming language, and has contributed much to the study and development of efficient persistent systems. The notion of orthogonal persistence was pioneered in the implementation of PS-algol, which supports full reachability-based orthogonal persistence.

PS-algol is built by adding functional extensions on top of S-algol, a high-level algol used for teaching at the University of St. Andrews. The basic goal was to implement persistence in a transparent manner, such that there is a minimal impact on existing code and programming styles. Therefore, the language syntax was left unchanged and the run-time system was modified to recognize and support a collection of procedures (e.g., opening or closing databases, committing or aborting transactions, etc.) that embodied the persistence facilities.

Pointers and references in PS-algol are dynamically typed, and therefore full type checking is usually necessary at run time. An explicit residency check is piggy-backed onto the type check made by the run-time system to ensure that the referent is either resident, or can be located and loaded from the database. Persistent references are implemented as object identifiers

(OIDs), and the granularity of address translation is individual pointers. The granularities of address mapping and data fetching is individual objects which are loaded from the database into the heap by the run-time system.

Napier88

Napier88 [MBC+89] is the successor to PS-algol; while PS-algol uses dynamic typing, Napier88 attempts to use strong typing in most cases. There are some special situations where run-time dynamic type checking is necessary because the type cannot be determined statically.

The basic implementation strategy is in terms of *environments* which are treated as first-class objects. An environment is an encapsulation of variables and their storage bindings. All expressions are evaluated in the context of some specific environment, using the information encapsulated in that environment. Persistence is implemented via a top-level environment that associates persistent objects to their corresponding values in the persistent store.

As in PS-algol, Napier88 implements reachability-based persistence, using the run-time type information that is available for all variables in the corresponding environment. The various granularity choices for persistence are also same as before, that is, individual objects are loaded from the database on demand when OIDs that reference non-resident objects are traversed by the application.

LOOM

LOOM [KK83, Kae86], or Large Object-Oriented Memory, is a virtual memory system that was designed to support large address spaces for early Smalltalk-80 implementations on machines with 16-bit hardware word size. Persistent references are stored as 32-bit OIDs and translated into 16-bit OIDs when persistent objects are fetched from disk; these 16-bit OIDs are used to index into a resident object table that holds references to actual object locations in memory.

LOOM attempts to provide transparent persistence; once objects are loaded into memory, they contain only 16-bit fields, just as in a non-persistent Smalltalk-80 implementation.[13] When all objects of the working set have been loaded into memory, LOOM behaves similarly to a normal Smalltalk-80 implementation because there is no distinction between transient objects and resident persistent objects.

An obvious question is how the system maintains references to non-resident objects in 16-bit fields of resident objects. LOOM accomplishes this by using one of two mechanisms, namely *leaves* and *lambdas*. A leaf is a proxy for a persistent object on disk; it is, however, resident and occupies an entry in the object table. In other words, leaf objects are used as proxy objects to implement node marking. Each leaf contains the 32-bit OID of the corresponding non-resident object so that it can be found easily when the reference to the leaf is traversed. On the other hand, a lambda is a 16-bit OID with a special value `0` (zero) that is distinct from all other values for 16-bit OIDs. This is equivalent to an edge marking approach because the references are tagged as invalid. As such, there is no need for a proxy object or an entry in the object table corresponding to a lambda. The lambda mechanism does not maintain the 32-bit

[13] This is very similar to the invariant maintained by pointer swizzling at page fault time such that all objects loaded into memory contain only hardware-supported virtual addresses.

OID of the original object—in order to load the object corresponding to a lambda value in the current object, the system must first read the current object from disk, locate the 32-bit value for the non-resident object corresponding to the lambda and then load that object from disk, overwriting the lambda with the newly-generated 16-bit OID into the object table.

The interpreter implements explicit checks for references to non-resident objects via either leaves or lambdas, and triggers loading as necessary using different mechanisms depending on whether a leaf or a lambda is being processed. This is termed as *object faulting*, drawing a similarity to paged virtual memory systems. The granularity of address translation is individual object references and the granularities of address mapping and data caching is individual objects. Finally, it should be noted that LOOM does not provide support for transactions or for saving recovery data in case of a crash/failure because it was primarily designed to implement a virtual memory system, not a persistence storage system.

The E Programming Language

The E programming language [RC89, SCD90] was developed at University of Wisconsin as a persistent extension to C++. Persistence is implemented by adding special *database* types such that only objects of these types can persist. This is a necessary but not sufficient condition for persistence. In order to persist, an object must be of one of the database types, *and* must be allocated specially in persistent heap. This approach breaks the orthogonal persistence model because persistent objects are tied to a special type (and associated type hierarchy).

The basic strategy in E is implemented by extending `gcc`, the GNU C compiler, to support database types and to generate special residency checks for triggering address translation and object faulting as necessary. In addition, the E Persistent Virtual Machine (EPVM), an interpreter, is used for accessing persistent objects. The code generated by the compiler invokes the interpreter to load persistent objects and translate pointers as necessary.

The Exodus Storage Manager (ESM) [Car89] is used as the object storage management layer in the language implementation. Each persistent pointer is represented by a 12-byte OID in the storage manager, and is swizzled to the word size of the local hardware. The first implementation of the interpreter, EPVM 1.0, interfaced with ESM to explicitly manage client buffer pool by pinning and unpinning physical memory pages as necessary. A new architecture, EPVM 2.0, has since been implemented to improve performance by copying data into virtual memory while translating OIDs into virtual memory addresses, and then unpinning the original page from the memory [WD92]. Copying into virtual memory can be done in terms of either individual objects or entire pages containing those objects. Applications manipulate data directly in virtual memory and no further overhead is incurred in OID translation, while pinning is only required during copying.

The granularity choice for address translation is in terms of individual pointers, based on the compiler-generated residency checking code that is inserted at appropriate points in the code. Address mapping and data caching is handled by the storage manager which explicitly manages the client buffer pool. In the case of EPVM 2.0, the granularity of caching varies depending on whether individual objects or entire pages were copied from the buffer pool into virtual memory. The granularity of checkpointing is in terms of individual objects because the system updates them in the database from their corresponding copies in virtual memory.

PS-Smalltalk

PS-Smalltalk [Hos95] is also designed to implement persistence for Smalltalk. The basic architecture of the system is similar to the EPVM 2.0 object caching architecture [WD92] but the underlying storage manager is Mneme [Mos92]. Objects are copied from the Mneme buffer pool into virtual memory on demand, translating Mneme OIDs into virtual memory address values. Mneme implements reachability-based persistence, as well as garbage collection for all objects reachable from a persistent root. The basic mechanism relies on user-supplied callback routines to find object references contained in other objects. In essence, this pushes off the type description responsibility on the programmer who must supply the callback routines.

The system uses node marking for supporting object faulting. The implementation uses *fault blocks* which are proxy objects for non-resident objects that hold the corresponding persistent addresses.[14] All object references corresponding to non-resident objects are converted into valid virtual memory addresses of corresponding fault blocks, while maintaining the invariant required by node marking. Note that this is very similar to pointer swizzling at page fault time which effectively uses protected virtual memory pages as fault blocks.

Address translation is done only when object faults are generated, that is, the application attempts to dereference a pointer to a fault block. The actual object is loaded into memory from the buffer pool, swizzled as necessary (including generation of references to other fault blocks), and is then made available to the application. The fault block, now called an *indirect block*, contains the actual memory reference to the object. The extra level of indirection through indirect blocks is periodically cleaned up by a garbage collector.

Method invocation in Smalltalk is usually done by *sending a message* whose receiver is the object on which the method must be invoked. Given this approach, the residency checks can usually be piggy-backed onto normal message sends which can check whether the intended receiver is resident and load it if necessary. Recall that data are copied into virtual memory on demand and address translation is done automatically as objects are loaded into memory. The granularity of address translation, therefore, is an individual object. The granularity of caching is a single object as far as the data copied from the buffer pool into virtual memory are concerned. Mneme itself uses segments (or collections of objects) for fetching data from stable storage into the buffer pool.

2.6.2 External Libraries

Apart from implementing persistence as part of the run-time system of a persistent programming language, it is also possible to support persistence through external mechanisms such as class libraries or shared object modules, without actually modifying the language implementation. These approaches are typically designed and implemented to exploit existing features of a general-purpose language (and operating systems) to achieve their goals.

In this section, we describe three such approaches, each of which implements persistence for C++. Although it is not an absolute requirement to use virtual memory protection facilities to implement persistence outside the language's run-time system, all three approaches described here are similar to Texas in that respect.

[14] Note the similarity between PS-Smalltalk's fault blocks and LOOM's leaf objects.

Vaughan and Dearle's Hybrid Approach

Vaughan and Dearle [VD92] have developed a scheme that is similar to ours because it also utilizes virtual memory protections for residency checking, but differs in the way the actual swizzling is performed. Unlike our approach where the pointers are always swizzled directly into valid virtual memory addresses, their scheme performs swizzling in an "incremental" fashion as described below.

When a newly-loaded pointer is being considered for swizzling, the first step is to check the residency of the referent. If the referent has already been loaded into memory, the pointer is translated into the corresponding virtual address. However, if the referent is not resident, the pointer is translated into a reference to an object table entry that contains the 32-bit OID of the non-resident object. This is referred to as "partial" swizzling because more actions are required before the referent can be accessed via that pointer. The object table entry is access-protected so that any attempt to dereference the pointer will trigger another protection fault. This fault is handled by locating the object on disk using the information from the object table entry,[15] loading it into memory and translating the partially-swizzled pointer into a valid virtual memory address corresponding to the new object location.

The primary goal of this approach is to reduce consumption of virtual memory (i.e., swap space) by avoiding allocation of backing storage for objects that are never accessed by the application. This is accomplished by only partially swizzling pointers until they are actually used, at which time full swizzling is completed by allocating memory for the referent and translating the pointer appropriately. Although this is the right idea in principle, we believe that the implementation strategy has been adversely affected because the authors misunderstood our approach. As we will discuss in Chapter 3, we reserve only virtual *address space*, not actual memory or swap space, for pages that are not yet accessed by the application. Unfortunately, modern operating systems do not always provide such flexible control over memory management. We defer further discussion on this issue until Chapter 7 where we describe interactions with operating systems.

There are also several other problems associated with an incremental swizzling approach. Since pointers are swizzled in two steps, the system must handle pointers in two different formats, partially-swizzled and fully-swizzled. This is similar to the pointer format checking problem in fine-grained address translation mechanisms. The basic scheme is also unlikely to work with standard operating systems and off-the-shelf compilers—when a partially-swizzled pointer is dereferenced and is fully swizzled into the final virtual memory address of the actual object, it is necessary to overwrite the (old) partially-swizzled pointer value with the (new) fully-swizzled pointer value. The fault handler must therefore actually update the saved machine state, which may not be allowed without special system-level privileges.[16] Overall, we believe that there is no extra benefit to this scheme, while there is an added overhead of handling partially-swizzled pointers.

[15] Once again, note the similarity between protected object table entries and fault blocks in PS-Smalltalk or leaves in LOOM.

[16] Pointer swizzling at page fault time does not have this problem because we alway translate pointers into valid virtual addresses, which do not need to be changed.

ObjectStore

ObjectStore [LLOW91] is a commercial system that also uses pointer swizzling at page fault time as the primary address translation mechanism to implement persistence for C++. Although both Texas and ObjectStore use the same underlying address translation mechanisms, each was developed independently without any influence from the other, and the two systems differ in several ways.

While Texas is designed to provide simple persistence for C++ with a small run-time code footprint, ObjectStore is a full-fledged object-oriented database system. Texas is designed to be compatible with existing off-the-shelf compilers and uses compiler-generated debugging information from object files to extract type and object layout information (Chapter 6). In contrast, ObjectStore uses a special preprocessor to extract the same information from the application source code. Chapter 6 also enumerates several reasons why we chose to use debugging information over a special-purpose preprocessor.

The most significant difference between the two systems is the strategy for performing pointer swizzling. ObjectStore is designed to *avoid* swizzling as far as possible by attempting to map a page to the same virtual address as it was the last time it was mapped into memory. If this can be done successfully for referents of all pointers on the page, no swizzling is necessary. In order to implement this strategy, however, additional information regarding previous mappings must be maintained for each persistent page so that it can be consulted before mapping the page into memory. Furthermore, after a page has been mapped into memory, it *must* be scanned to ensure that all pointers are either swizzled correctly, or that their referents can be mapped into the same location as the last time. In addition, since the application can modify arbitrary data on a page, a similar scanning is necessary (only for dirty pages) at checkpoint time to regenerate the mapping information for referents of all pointers on that page.

We believe that avoiding swizzling by mapping pages at the same locations is simply an "optimization" to the basic pointer swizzling mechanism. It is not clear whether this optimization provides any significant performance improvement because our measurements (presented in Chapter 5) indicate that the major component of the overall cost is usually the I/O cost, while the actual cost of translating the pointer is much smaller in comparison. Any small advantages are usually negated by the cost of maintaining additional mapping information for each page of the persistent store. It would be relatively easy to incorporate a similar optimization in Texas; however, we have not done it so far because it does not appear to provide any major benefits.

QuickStore

QuickStore [WD94, Whi94] is a research system that is very similar to ObjectStore in terms of its implementation strategy. QuickStore also uses pointer swizzling at page fault time approach for address translation, and like ObjectStore, it maintains additional mapping information to avoid swizzling as far as possible. An interesting difference, however, is the use of Exodus as the storage manager. As in the implementation of E, this allows the system to directly manage physical memory and the client buffer pool by explicitly moving data between the two hierarchies.

In the area of checkpointing for saving recovery information, QuickStore uses a page "diffing" technique similar to the idea that we first proposed in [SKW92]. The results presented in [Whi94] indicate that such "diffing" should perform well depending on the locality characteristics of the application. This observation is in line with our original projections of expected performance characteristics. However, further study is necessary in this area, along with research on storage management techniques for persistent object stores.

2.6.3 Other Approaches

In addition to the various systems described above, many other approaches for supporting persistence have been designed and implemented by various researchers. Below, we provide a brief overview of some of the more interesting systems.

Recoverable-Persistent Virtual Memory

The Recoverable-Persistent Virtual Memory (RPVM) system [CS93, CRRS93] promotes the extension of virtual memory to support a Recoverable-Persistent Update (RPU) model for realizing a wide range of recovery services. Although the model essentially supports a mechanism for implementing persistence, the primary focus is on ensuring that the state of virtual memory can be recovered. As such, RPVM is not a classic persistence mechanism and there is no explicit pointer swizzling or address translation in the system. Instead, the database is memory-mapped into the virtual address space using a Mach-equivalent of `mmap`.

The RPU model defines mechanisms that control the propagation of virtual memory pages to stable storage. In particular, the system allows *flush locks* to be placed on virtual memory pages. A flush-locked page is pinned in virtual memory and cannot be propagated to the database.[17] The model also defines *page-flush before* rules that specify a partial order between two pages for propagation to the database. Page-flush policies are defined per database to guarantee that each database is in a recoverable-persistent state.

RPVM is implemented by modifying the Mach 3.0 kernel to incorporate support for page-flush policies. Specifically, user processes can add and remove both flush locks and page-flush before rules to specify the appropriate policies for a given database. The granularity of address translation is not applicable because no swizzling is performed, while granularities of address mapping and data fetching correspond to the entire database which is mapped into memory in a single step.

Cricket

Cricket [SZ90] uses the memory management primitives supported by Mach to implement a single-level persistent object store. The primary strategy relies on Mach external pager facilities [You89] to locate and load the persistent data from the disk into memory. Cricket also supports transparent concurrency control and recovery facilities.

The basic architecture is distributed, with a centralized server that is the primary interface of clients for accessing the persistent object store on disk. The clients communicate

[17] Note that pages are pinned in virtual memory, *not* physical memory; a flush-locked page can be paged out to swap space if necessary.

with the Cricket server via an RPC interface. The server, which acts as an external pager, treats the persistent store as a *memory object*[18] and maps it directly into the client's virtual address space. The client can then access persistent data as if it were in virtual memory.

Cricket does not implement any kind of pointer swizzling mechanism. Instead, it always maps the database at the same range of virtual addresses such that all references are automatically validated. This also means that the size of the database is restricted by the maximum address space supported by the operating system. The granularity of address translation is obviously not applicable in this case, and the granularities of address mapping and data fetching correspond to the entire database.

Dali

Dali [JLR$^+$94] is a storage manager optimized for main memory databases, that is, situations where the persistent store resides in memory. It does not perform any address translation, and uses memory mapping techniques to map *database files* into the virtual address space of a user process. A collection of database files forms a single database (i.e., a persistent object store).

Persistent references are maintained through the use of *database pointers*. These are typically represented by using a file identifier (for example, the full path to a database file) and an offset into that file. The system also supports indirect references through the use of object identifiers, also known as `ItemIDs`. Address translation is typically done in a fine-grained manner as each database pointer is dereferenced. The virtual address is calculated by adding the offset to the base address where the corresponding database file has been mapped,

The granularity of address translation is typically a single pointer because of the use of special database pointers. The granularity of address mapping is on the order of a database file, since the entire file is usually mapped into memory. Checkpointing and crash recovery is implemented using either physical (data) logging or operation logging. The granularity of checkpointing in the usual case is in terms of predetermined checkpointing units (or *chunks*) as defined by the system.

P3L

P3L, developed by Suzuki *et al.* [SKT94], is a persistent variant of the C language. The system introduces and describes the notion of *reservation* and *residency* in the context of object faulting. The term *reservation* refers to the action of reserving a local identifier corresponding to a persistent identifier in preparation for an upcoming load of the referent into memory while the term *residency* refers to the state of the referent, including information on whether the data has been loaded from the persistent store. The latter is similar to residency checks that are part of the pointer validity tests for most fine-grained address translation mechanisms.

P3L has been implemented by modifying the GNU C compiler to generate additional code at appropriate points in code. Address translation is performed at the granularity of objects—references to non-resident objects are translated into special *surrogate* values (similar to LOOM's lambda). The compiler automatically inserts extra instructions for reservation

[18] Mach supports the abstraction of memory objects to allow external memory management in user-level processes.

checking and incremental translation. The Exodus Storage Manager is used for underlying persistent storage, although a specific storage manager is not dictated by any design choice of the system.

[SKT94] presents a performance comparison between the software-only approach of P3L and several other systems, including pointer swizzling at page fault time, using the OO1 database benchmark. The results show that coarse-grained address translation is generally superior to other approaches in almost all situations. The only variation where P3L outperforms pointer swizzling at page fault time is a non-standard benchmark traversal where all locality in the data structures has been eliminated artificially. This result, however, is not surprising because coarse-grained address translation techniques implicitly rely on locality of reference to amortize the higher costs of faulting and swizzling entire pages.

2.7 Conclusions

The primary goal of this chapter was to identify the basic design issues that are important when implementing a persistence mechanism. As part of this exercise, we provided a basic definition of persistence and described the different types of persistence that are popularly used and implemented in current systems. We believe that orthogonal persistence is the right approach, and our system is designed to be compatible with this approach.

We have found that existing classifications are primarily concerned with address translation mechanisms only, and do not approach the problem at the right level of abstraction. We believe that the fundamental issue is the granularity at which address translation is performed. As such, we have developed a new classification scheme for persistent systems that is based on granularity choices for the basic design issues, and described where our approach fits into the overall hierarchy. Identifying the fundamental design issues for implementing persistence is important and useful for understanding their impact on overall performance and flexibility of persistent systems. By using granularity as the main factor, we have provided a general classification mechanism that is not constrained or unclear. We believe that the five issues identified in this chapter form a core set of fundamental design issues for implementing persistence.

Our scheme is primarily a coarse-grained address translation mechanism, with few special situations where it can be changed to finer granularity. The coarse granularity allows us to exploit existing hardware and reduce total overheads while maintaining compatibility with existing compilers and operating systems. On the other hand, fine-grained schemes incur some basic costs that make them inherently slower overall, except in a few cases that also require fine-grained control over other mechanisms such as transactions, locking, etc.

Finally, we have provided a survey of related work in the area of persistence implementation in a variety of systems. We broadly divided this into two groups based on whether the persistence facilities are provided as part of the language implementation. Persistent programming languages such as PS-algol, E, etc. fall into the first category that contains special persistent languages. The other category includes mechanisms that are implemented outside the language (usually as a object code library), and take advantage of existing features of the language, compiler and operating system to do their job.

Chapter 3

Pointer Swizzling at Page Fault Time

3.1 Introduction

Persistent object stores are designed to manipulate large volumes of data by implementing virtual address spaces that are larger than hardware-supported address spaces. Early schemes for supporting large virtual addresses on normal hardware (e.g., LOOM [KK83, Kae86], E [WD92], etc.) have typically incurred significant overhead due to their use of traditional fine-grained address translation techniques.

There are at least two basic approaches that are commonly used for implementing large address spaces in software. One is to use an *object table* and indirect all object references through the object table by translating object identifiers into table offsets as objects are loaded into memory. Untranslated object identifiers can be marked as such and translated lazily (as necessary). The second approach is called *pointer swizzling*—rather than using indirect references through an object table, object identifiers are converted into actual hardware-supported addresses (that is, virtual memory pointers) in an incremental fashion.

In this chapter, we describe our approach which is a variation on the basic pointer swizzling mechanism. Conventional pointer swizzling schemes perform the translation only when the running program tries to use a particular persistent pointer. As part of the translation process, the object is loaded into virtual memory if it is not already present. Unfortunately, translating individual pointers may involve checking each pointer at each use to determine if it is a valid address, thereby increasing the overhead. Alternatively, it is possible to swizzle pointers in an object the first time the object is referenced [Mos92]. However, this approach also requires that pointers are checked before each use to ensure that they are swizzled as necessary.

We would like to avoid these extra costs, so that programs that do not access persistent data do not pay the cost of checking, and programs that do access persistent objects multiple times do not incur additional costs at *every* access. Ideally, we would like this scheme to operate efficiently on standard hardware without requiring any special-purpose hardware such as that of the MUSHROOM project [WWH87].

Our approach, called *pointer swizzling at page fault time* (PS@PFT), is to load pages into virtual memory *on demand*, swizzling persistent pointers into normal hardware-supported virtual memory addresses at page fault time. Pointer swizzling at page fault time is a novel address translation mechanism that relies on standard virtual memory hardware to check whether referents are already in memory and to trigger swizzling as necessary. The scheme swizzles entire pages at a time, translating all pointers into corresponding virtual addresses; no extra hardware is required and there is no continual checking overhead because swizzled pointers can be dereferenced at normal memory speeds. Our strategy is based on exploiting locality of access by amortizing the cost of swizzling over multiple accesses to the same data. In addition, we also take advantage of the fact that I/O costs are typically much higher than swizzling costs so that swizzling an entire page as it is faulted in does not add significant overhead compared to the I/O costs for loading the page from disk.

The remainder of this chapter is organized as follows. We start by discussing the motivation behind designing a page-wise address translation scheme (Section 3.2). Next, we describe the basic algorithm (Section 3.3) and briefly discuss indirect costs of pointer swizzling (Section 3.4). Other related details such as handling of large objects (Section 3.5), address space exhaustion (Section 3.6), and issues regarding sharing and compatibility of pointer swizzling at page fault time (Section 3.7) are also described. Finally, we discuss fine-grained and mixed-granularity address translation schemes in Section 3.8 before concluding in Section 3.9.

3.2 Motivation

Pointer swizzling at page fault time is inherently a page-wise address translation scheme. Our decision to implement a coarse-grained scheme was motivated by several factors.

A potential advantage of fine-grained (pointer-wise or object-wise) address translation is the savings in I/O costs because typically only the data required by the application is loaded in. Since today's disks have high latencies, the argument about savings in I/O is truly applicable only when data is fetched from a remote host via a fast network instead of a local disk. However, if the application has good locality and accesses most objects on a page, the advantage of fine-grained loading and swizzling is quickly lost. Furthermore, although experimental network protocols have achieved surprisingly good performance [TL93, vEBB95], most widely available current networks are still an order of magnitude (or more) slower, further reducing potential benefits of fine-grained schemes.

In addition to the I/O cost, the cost of maintaining meta-data for address translation is also likely to affect the performance of a fine-grained scheme. Specifically, the table used to hold the mappings between persistent and virtual addresses is likely to get significantly large. While page-wise swizzling requires only one entry *per page* in the mapping table, fine-grained swizzling will require one entry *per object* (or more likely, *per pointer*) thereby increasing the table size.

A larger table size affects the cost of the actual address translation; each time a persistent pointer is swizzled, the mapping table must be probed to check if a mapping already exists, and a new one is created if necessary. Research in garbage collection techniques has shown that typically 75-80% of the pointers in an application are likely to be unique [Wil97]. This means that the mapping table lookup will also fail as often. As table sizes increase, the cost of probing (and inserting new mappings) also tends to increase, adding to the overall costs of translation. In general, the table size and access characteristics coupled with additional overhead associated with the translation itself can impact the overall performance of fine-grained schemes.

If most objects on a page are referenced by the application, fine-grained schemes end up mapping and swizzling all those objects eventually, creating several entries in the table. In contrast, pointer swizzling at page fault time will create a single entry for the entire page and swizzle all objects as the page is loaded into memory. Fine-grained schemes are therefore preferable if the application references less than $n\%$ of the objects on the page, where n is some threshold whose exact value depends on factors such as object size, average number of pointers in objects (and pages), mapping table implementation, etc.

Compared to object-wise (or pointer-wise) address translation schemes, a page-wise approach allows additional flexibility in the usual case. We can exploit the existing virtual memory hardware and memory protection mechanism offered by most modern operating systems. This makes our scheme compatible with stock hardware and off-the-shelf compilers, without requiring any special features or support from the operating system.

We believe swizzling at page fault time to be especially attractive because it scales well to systems with large main memories. As memories get larger, the average number of instructions executed between page faults goes up; this should make the cost of pointer swizzling proportionally smaller. Conventional pointer swizzling schemes usually do not have this property, because their checking and translation overheads are tied directly to *the rate of program execution.*

3.3 Algorithm Description

Pointer swizzling at page fault time is a coarse-grained address translation scheme because we load and translate entire pages at a time. This is different from other schemes with finer granularity that load and swizzle either entire objects or individual pointers only. The basic approach is incremental because pages are faulted into virtual memory only as required by the application and address translation is done for an entire page when it is loaded from the persistent store (on disk). Address translation involves translating pointers from persistent (long) format into actual hardware-supported virtual memory address (short) format.

Any incremental faulting scheme must somehow detect references to objects in persistent memory,[1] so that they can be loaded into virtual memory before being operated on. We choose to use existing virtual memory hardware and page-wise access-protection capability of the operating system for this purpose. This avoids continual overhead in software and works well with modern operating systems on standard hardware.

Our scheme relocates objects into virtual memory somewhat sooner than a straightforward (software-only) pointer swizzling scheme because we load an entire page when *any* object on that page is accessed. This allows us to preserve one essential invariant—the application is *never* allowed to "see" any pointers into the persistent address space. Pages containing persistent pointers are access-protected so that when the application attempts to access such a page, a trap handler is invoked to relocate the whole page from the persistent store into virtual memory. The trap handler also translates all persistent pointers in the page into transient

[1] Recall that these residency checks are part of the general pointer validity checks described in Chapter 2.

pointers, reserving address space for their referents as needed. The page is then unprotected and the application may continue without further interruptions for accesses to objects on that same page.

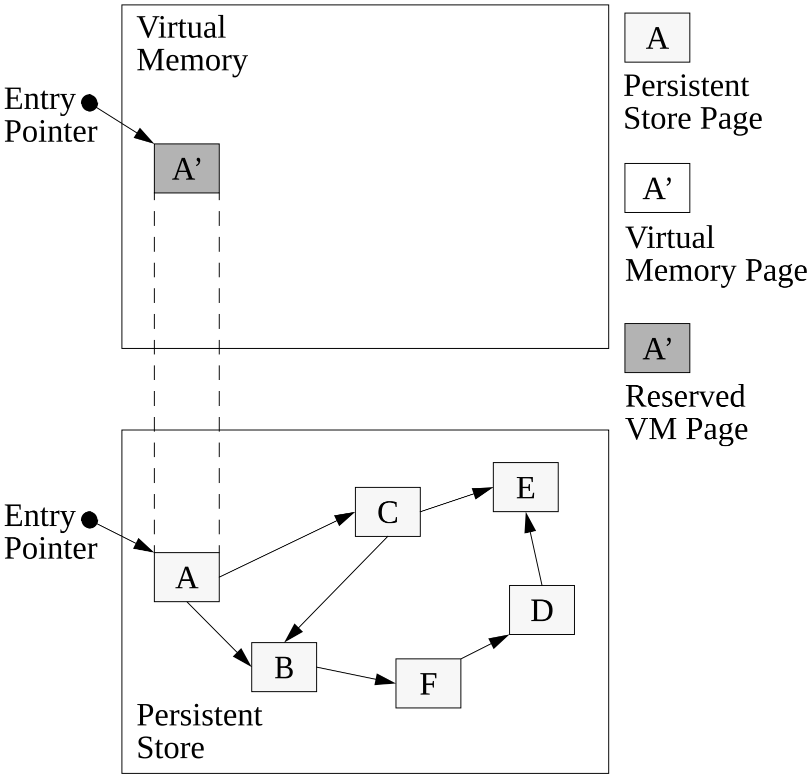


Figure 3.1: Bootstrap state for swizzling

Figures 3.1 through 3.3 illustrate the pointer swizzling at page fault time mechanism. When an application is given access to the persistent store, it can request pointers to one or a few special *persistent roots* that can be retrieved by name. These roots act as *entry pointers* into the data stored in the persistent store. When a rooted object is requested by name, the first step is to relocate the page(s) that are referenced by the entry pointers. This allows the entry pointers to be translated into hardware-supported virtual address format so that the application can begin execution. Figure 3.1 shows this *bootstrap* state of our system. Note, however, that we do not actually relocate the page (page A in the figure); instead, we simply reserve and access-protect a page of virtual address space (page A' in the figure) without loading the data from the persistent store. The virtual memory address of the object is then returned as the entry pointer to the application.

As the application attempts to dereference a pointer into an access-protected page (page A' in our example), an access-protection violation is generated. We provide a handler that intercepts this violation, locates the corresponding page (page A in our example) in the persistent store, and loads it into memory at the predetermined (reserved) location A'. Next, we convert all pointers from persistent address format into virtual address format to maintain the aforementioned invariant. However, for all pointers to be swizzled correctly, their corresponding virtual memory address values must be known. Since all referents may

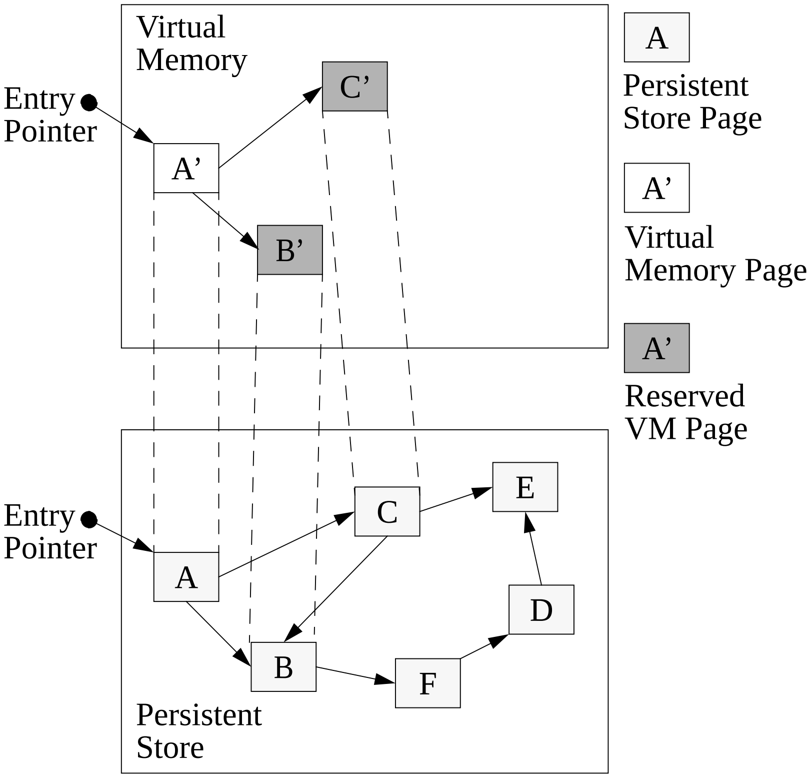


Figure 3.2: Incremental faulting and swizzling

not already be in memory, the system may need to reserve and access-protect other pages of the virtual address space (pages B' and C' in Figure 3.2). After this has been completed successfully, we are done with actions for swizzling and control is returned to the application.

As shown in Figure 3.3, the same process repeats whenever a pointer into an access-protected page is traversed by the application. The swizzling mechanism ensures that if there exists a mapping for a given page, a new one is not created. In our example, as the application traverses a pointer in page C' causing it to be loaded from the persistent store and swizzled, the address space for page B does not need to reserved again. Instead, pointers in page C' are swizzled appropriately to point into page B'. Of course, address space for any other page(s) not already reserved (page E in our example) must be allocated and access-protected as usual.

Our approach is analogous to that of Appel, Ellis and Li's incremental copying garbage collection scheme [AEL88] which, in turn, is a variation incremental copying scheme described by Baker [Bak78, Bak91]. However, unlike the Appel-Ellis-Li model which incrementally relocates *objects* from *from-space* to *to-space*, our scheme relocates *pages* of objects from persistent memory to transient memory. This reduces the size of tables required to hold mappings between persistent and transient addresses—only the page numbers (addresses) must be recorded, not individual objects. It also meshes well with page faulting mechanisms; caching pages is more attractive than faulting objects when memories are not very small [Sta82, Wil91, WD92].

The technique reserves virtual address space "one step ahead" of the access pattern of the application, essentially forming a read barrier in a page-wise "wavefront" that is extended just past the pages that are already referenced by the application. This is shown pictorially

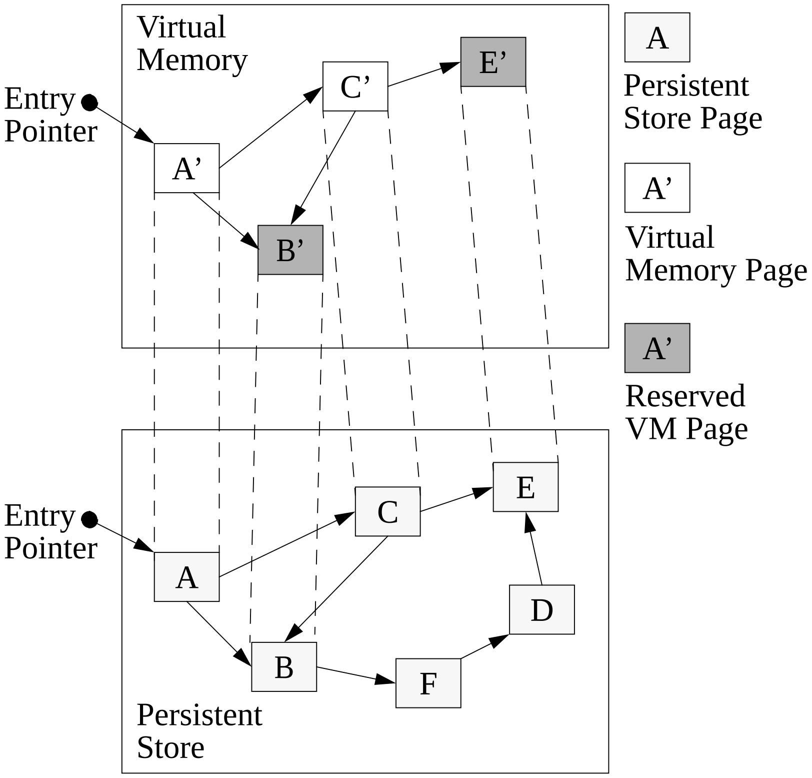


Figure 3.3: Incremental faulting and swizzling (cont'd.)

in Figure 3.4. The wavefront formed by the access-protected pages allows us to maintain the invariant that the application can never reference any persistent (unswizzled) pointers. Pages that are already referenced by the application only contain hardware-supported virtual addresses, although some of these may point into protected pages. However, we are guaranteed that if any pointer into a protected page is dereferenced, the resulting fault will be intercepted and handled by our fault handler. The page then becomes part of the group of pages that have been referenced by the application and the wavefront is extended to include newly-protected pages created as a result of the swizzling. This mechanism ensures that the application never references pages with unswizzled pointers.

Since the application only encounters pointers in virtual address format, pointers can be dereferenced in the normal way, that is, in a single machine instruction with no extra overhead except at page faults. We exploit locality of reference in an application similar to how normal virtual memory systems operate. We also take advantage of the fact that there is a huge disparity between CPU speeds and disk speeds—by swizzling pages when they are faulted in from disk, we do not add significant overhead to the faulting cost. In addition, if an application exhibits the usual locality of reference, we do not incur any additional cost for further accesses to a swizzled page, thus avoiding a lot of the overhead typically encountered by pointer-wise or object-wise swizzling.

It should be emphasized that we only consume virtual address space, not virtual memory (that is, swap space) for untranslated (reserved and access-protected) pages. However,

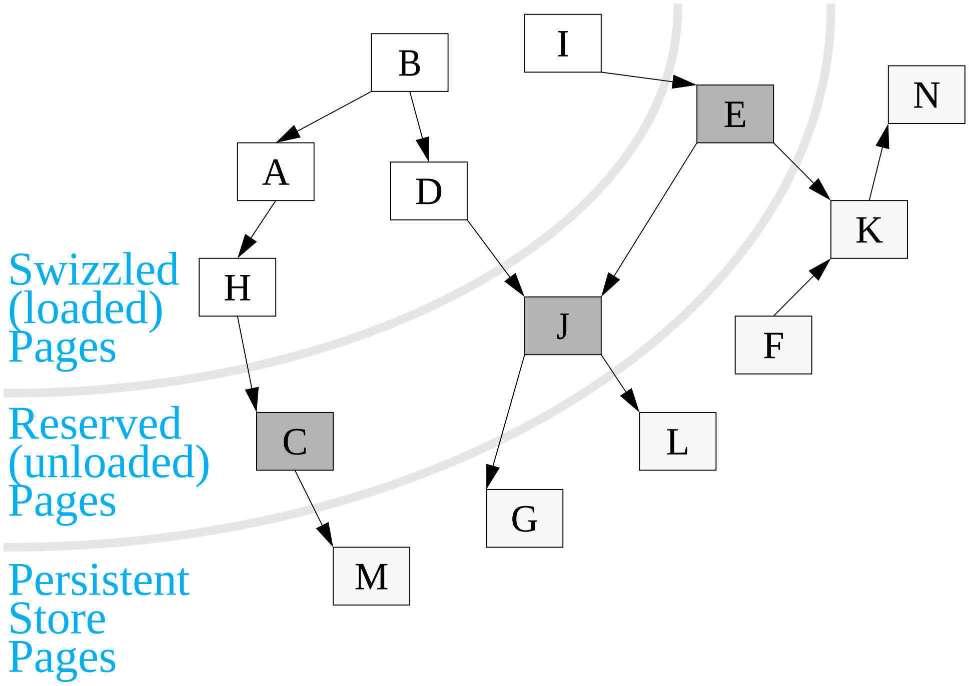


Figure 3.4: “Wavefront” of address space reservation

for some pathological cases which have data structures with high fanout,[2] this approach may still consume address space quickly. In Section 3.6, we discuss the problem of address space exhaustion and describe several different ways to solve it while still maintaining compatibility with our basic swizzling mechanism.

3.4 The Mistaken-Dirty-Pages “Problem”

Pointer swizzling at page fault time effectively uses the virtual memory hardware, instead of the more expensive software-only approaches, for residency checking. Chapter 5 presents performance results which show that the basic costs of this approach are very small compared to the I/O costs. Unfortunately, there is an inadvertent interaction with the virtual memory system on operating systems that do not support an advanced memory management interface, leading to some *indirect costs* related to pointer swizzling. In this section, we describe the basic issue, and make some important observations which indicate that these costs are not a fundamental problem with our approach to address translation. We also briefly describe possible solutions, deferring detailed discussion until Chapter 7.

3.4.1 What it is

When a page is loaded from the persistent store into virtual memory, all pointer fields in that page are swizzled into corresponding virtual memory addresses. Typically, these actions occur *after* the page has been loaded into memory but before the application has actually used any

[2] We define *fanout* as the number of pointer fields (i.e., “outgoing” pointers) in a given object.

data from that page. Thus the page is "clean" from the application's perspective because it has not actually modified any data on that page. However, the virtual memory system considers the page to be "dirty" because the act of swizzling has modified the page, even if the application itself may never do so. Unfortunately, in the usual case, the virtual memory system cannot automatically distinguish between modifications done by our system for pointer swizzling and those done by the application. We call this the *mistaken-dirty-pages problem* because pages that are clean from the application's point of view are "mistakenly" marked as dirty by the virtual memory system.

Note that the mistaken-dirty pages are not really an issue unless the application exhibits paging behavior. When a page must be evicted from main memory, the virtual memory system has two choices. If the page is clean (i.e., there exists an unmodified copy on disk), then it can simply be discarded. On the other hand, if the page is marked dirty, it must first be *paged out* (i.e., written to the backing store) before it can be discarded. In the current scenario, it is obvious that the mistaken-dirty pages must be paged out by the virtual memory system before they can be evicted from memory.

Under certain conditions, the page-outs for mistaken-dirty pages may be considered unnecessary because the corresponding data already exists in the persistent store, albeit in an unswizzled form. Instead, the memory can be reclaimed simply by discarding the data and removing the virtual-to-physical mapping for the page. The virtual address space for the page must still be retained (and reprotected) so that future accesses to the page will be intercepted by the normal swizzling mechanism and will cause data to be reloaded from the persistent store. In essence, we are "paging" from the persistent store rather than from local swap space.

Narasayya *et al.* [NNM+96] originally raised the issue about swizzled pages that are erroneously marked as dirty by the virtual memory system and the corresponding page-outs that are unnecessary if paging from the persistent store. They classified it as the *virtual memory overhead* of pointer swizzling resulting from the cost of additional actions necessary for pages that are mistakenly considered dirty by the virtual memory system.

3.4.2 Is it a Bug or a Feature?

Although the issue with mistaken-dirty pages appears to be a major problem at first glance, we argue that there are other factors related to the general configuration that contribute to the ultimate classification of this issue as a problem or as expected behavior. We consider three different configurations in this context:

1. In a traditional relational database-style setup, there are one or more dedicated database servers, typically configured with large amounts of main memory. If the persistent store is maintained on such well-equipped servers, the mistaken-dirty pages are indeed a problem because the server-side caching mechanism—designed explicitly for such use—is not being exploited.

2. Another configuration is where the persistent store is on local storage, that is, there are no servers or networks involved. In this case too, the mistaken-dirty pages are undesirable because it is unnecessary to have two copies—one in the persistent store and the other in the backing store—of a clean page.

3. Finally, the third configuration is where the persistent store is maintained on a remote, centralized file server that is explicitly designated for file service, and normal applications are *not allowed to page off that server.* This is an important kind of situation for which the current implementation of our persistent object store is designed as described below in our observations on client caching.

3.4.3 Observations

We have argued that the "problem" with mistaken-dirty pages is not always a problem depending on the configuration as well as application usage patterns. We now make some important observations about the basic issues; these observations indicate that, (1) the problem is not as bad as it seems at first glance, (2) it is not a fundamental shortcoming of the pointer swizzling approach, and (3) the default behavior is often the intended behavior for most normal applications that use our mechanism to incorporate basic persistence.

Onetime Costs. The first observation is to note that the costs related to unnecessary page-outs and mistaken-dirty pages are *not* continual costs; rather, they are only *onetime* costs for every such page. That is, a swizzled page is considered dirty by the virtual memory *only the first time it is loaded from the persistent store and swizzled.* If it is referenced again after having been paged out once locally, the virtual memory system will load it in from the swap space and no swizzling would be necessary. As such, the page is now marked clean and will not need to be paged out before it can be evicted. In other words, the page has effectively been "cleaned" by the virtual memory system because it was paged out once.

Effect of Locality and Paging. The mistaken-dirty-pages problem is strongly tied to the locality characteristics and paging behavior of the application. If there is no paging during application execution, there is obviously no additional overhead because the virtual memory system does not need to evict any pages. The mistaken-dirty pages are harmless in this case, and will be reclaimed by the operating system at the end of program execution. The other end of the spectrum is when the working set of an application is much larger than the available main memory. In this case, we expect the overall performance to be dominated by heavy paging behavior, and the onetime cost of page-outs due to mistaken-dirty pages will only be a small fraction of the overall paging costs. Finally, the middle of the spectrum is characterized by light to moderate paging; this is the least favorable for pointer swizzling because the unnecessary page-outs are likely to be a larger fraction of the overall paging costs depending on the access characteristics of the application and the working set size. Further research, including detailed studies using actual applications, is necessary for quantifying these costs in general.

Client Caching. The final observation is about the importance of *client caching* in the basic run-time environment. One can imagine a situation where it is cheaper to page off the local swap space instead of using the persistent store. For example, if the persistent store resides on a centralized file server across a (slow) network, the costs of local paging may offset the costs of loading a page from the persistent store and swizzling it. Furthermore, it is possible that centralized file servers are configured to prohibit general paging. Thus in situations where local paging is preferred over paging from the persistent store, page-outs due to the mistaken-dirty pages are typically not an issue because those pages would have been written to local swap space regardless of whether they were swizzled or not.

We have designed a persistent object store, the Texas Persistent Store (Chapter 4), to incorporate persistence in normal applications, not just database-style applications. As stated earlier, in a traditional database context, there are dedicated servers with large amounts of main memory for improved performance from server-side caching. However, in a normal computing environment, the configurations are often very different. Texas is implemented as a library archive; it provides an interface to allow applications to manipulate persistent object stores that may be stored remotely (e.g., on an NFS file server).

File servers are commonly centrally administered to provide *reliable file service*, not to act as servers that provide backing storage for virtual memory on clients. In fact, in many environments, *paging off a central file server is not allowed.*[3] If Texas were to avoid the mistaken-dirty-pages problem by evicting pages to the persistent store, and the persistent store were NFS-mounted from a central server, this would amount to paging off the server. Thus the current approach of client-side local caching is the right one in many situations. If Texas were to do otherwise by default, applications would unintentionally violate server usage policies (and possibly affect general network performance) simply by linking against the Texas library.[4] Of course, if the persistent store is stored locally or if the user has a right to use a dedicated server, client-side caching is not the best approach.

3.4.4 Discussion

It is obvious that the mistaken-dirty-pages problem and the associated page-outs are definitely a source of additional overhead for some configurations, affecting overall performance depending on factors such as memory size, locality characteristics, etc. It must be emphasized, however, that these are *not fundamental costs* of the pointer swizzling at page fault time technique. Instead, we classify these costs as *indirect costs* because pointer swizzling is indirectly responsible for them due to its interactions with the virtual memory system. Depending on operating system features that are available, the mistaken-dirty-pages problem can be solved in several ways. We briefly sketch some of these here; Chapter 7 contains further discussion about interactions between pointer swizzling and virtual memory management. Of these solutions, the most desirable ones usually require features that are currently not available on most production operating systems. However, the most portable solutions require only the ability to mount a new file system that is designed to manage the paging for the persistent store.

An obvious solution is to not do pointer swizzling at all, following an approach similar to the one used by systems that directly map the entire persistent store in memory [SZ90]. However, this usually limits the amount of persistent data that can be accessed at one time. More importantly, it affects portability because the same mappings and virtual address ranges cannot be used across different operating systems. A slightly better alternative is to use an approach similar to the one used in ObjectStore [LLOW91] and QuickStore [WD94]; specifi-

[3] Our own environment is an example of this situation.

[4] Systems administrators typically tend to frown upon software that arbitrarily changes the application paging behavior and adversely affects general performance in networked environments.

cally, this approach swizzles a page only if it cannot be mapped at the old address. However, this approach still has limitations in the general case, and does not resolve the basic problem.

Narasayya *et al.* [NNM+96] suggest a special system call to clear the dirty status bit of a page. While this is a good idea, it can be generalized to implement an extended primitive that can communicate a variety of information from an application to the virtual memory system. An even better approach is to modify operating systems to support external memory management mechanisms. Operating systems that support such models already exist (e.g., Mach [BKLL93], and L3/L4 microkernels [Lie95]) and can be exploited. For example, Mach supports external pagers which can be used as *pointer swizzling servers* to swizzle pages *before* loading them into memory so that they appear clean to the virtual memory system.

Another solution that does not require external memory management support or other kernel modifications is based on exploiting the *virtual file system* (VFS) and *vnode* interface provided by most operating systems [Vah96]. Using this interface, a special "file system" can be implemented to handle the paging for a given persistent store; this file system can be designed to handle the pointer swizzling mechanism such that the virtual memory system only receives clean, swizzled pages thereby avoiding the mistaken-dirty-pages problem. We elaborate on this solution further in Chapter 7.

Ultimately, the righteous solution is to improve operating system implementations to provide a better *separation of concerns* between components such as address mapping and virtual memory management. Further discussion about this and other related issues is deferred until Chapter 7.

3.5 Handling Large Objects

The description of the basic algorithm for pointer swizzling at page fault time implicitly assumed that objects were smaller than a virtual memory page, and one or more objects fit on a single page. A potential problem in our scheme is the need to ensure that if an object crosses page boundaries in the persistent store, the corresponding pages must be adjacent in the transient virtual memory as well. If an object straddling a page boundary is not relocated as a contiguous object, indexing to access its fields will not work properly.[5] We resolve the problem of large objects by handling them slightly differently; as many pages as necessary to fit the large object are reserved when *any* page of the object is faulted on the first time, but only the page that was accessed is loaded into memory. In other words, address space is reserved for the entire object, but only parts of the object that are referenced by the application are faulted in. Lazy copying of data is particularly helpful here—although address space must be reserved for the whole object, there need not actually be *any* physical memory (RAM or disk) used for unreferenced pages.

To support incremental copying/faulting of large objects, the language implementation must support operations for locating object boundaries and maintaining mapping tables to track pages that belong to large objects. These requirements are similar to those of garbage collected systems that must perform page-wise (or "card-wise") operations [AEL88, WM89]

[5] An example of such an object would be a large array which spans multiple pages, even though the size of each individual element may be smaller than a page.

within the heap. There is no major difficulty supporting such operations efficiently for languages like Lisp or ML; slightly conservative versions of these schemes will work well for languages with derived pointers and (limited) pointer arithmetic, in much the same way that conservative garbage collectors operate with languages like C or C++ [BW88]. The main modifications are to the allocation and deallocation routines, which must provide headers and/or groupings and/or alignment restrictions to allow objects to be identified.

Large objects still pose a potential problem for our system in terms of exhaustion of virtual address space. If a page is touched and it holds pointers to several large (multi-page) objects, address space must be reserved for all of those objects' pages, even if they are never touched. Programs that deal with many large objects may therefore benefit from a larger hardware address space by decreasing the frequency of address space reuse. While we think that this is unlikely to be a serious problem for most applications on most machines, it is still worth considering. As discussed in Section 3.7, it is possible to integrate machines that require large hardware addresses with those that do not, for sharing of most data between them.

3.6 Avoiding Address Space Exhaustion

A potential problem with the basic scheme is that the transient memory could fill up with relocated pages that are used for a while, and then not used again for a long time. These pages could fill up the virtual memory, causing excessive paging. This is actually not much of a problem because the process of swizzling is nearly orthogonal to issues of levels in the storage hierarchy—an inactive page can still be paged out to backing store as in normal virtual memory. It may be paged to swap space temporarily, or it may be unswizzled and evicted back to the persistent store.

The real problem, then, is not the exhaustion of hardware memory, but the *exhaustion of the hardware-supported virtual address space*. As mentioned in previous sections, this is not just a problem for programs that actually reference millions of pages, because touching one page may cause reservation of several pages of the address space. In the worst case, a page contains nothing but pointers may be referenced, causing reservation of as many pages as there are pointers—in our current system, about 500 times as many pages may be reserved as are actually touched. While this is unlikely for most programs, it is conceivable and in fact rather near-fetched—pages holding multi-way index tree nodes may approximate the worst case.

To avoid exhausting the virtual address space, we have three strategies. The first is to have smaller pages or to reduce the effective page size, and slow the rate of address space use. Since this strategy may not be entirely effective, we have also devised an algorithm for reclaiming virtual address space incrementally and reusing it. Finally, we also describe ways to implement fine-grained and mixed-granularity schemes which may be useful in situations where the programmer has more control over the data structures.

3.6.1 Smaller Page Sizes

Since rate of address space consumption is directly proportional to number of objects and pointers in a page, it is obvious that smaller page sizes would favor our scheme by reducing the number of pointer that are swizzled. There are many ways in which page sizes can be reduced; we discuss some of these below.

We can reduce the effective page size by only using part of each virtual page when allocating objects with large numbers of pointers. For example, if we only use one fourth of each 4KB page, we reduce the fanout by a factor of four. A naive implementation of this strategy would be very wasteful, however, so it is desirable to avoid using actual RAM and disk storage sparsely. A better strategy is to only use a fraction of each *virtual* page, but arrange the fragments in a complementary pattern so that several virtual pages can share a *physical* (RAM or disk) page. For example, suppose we wanted to implement 1KB fragments of 4KB pages; we could map four virtual pages to a single physical page, and use a different quarter of each virtual page for data. While the physical page as a whole would have four aliases (virtual page numbers), the non-overlapping pattern of allocation would ensure that no object (or cache block) was actually aliased.

This solution is not entirely satisfactory for two reasons. First, it does not deal with large objects very well. Second, it wastes little or no physical storage, but decreases the effectiveness of translation lookaside buffers—each partially-used virtual page requires its own virtual-to-physical page mapping in the virtual memory system. While it defers the exhaustion of the address space in the sense of delaying the discovery and swizzling of pointers, it actually increases total address space usage (in the long run) by decreasing the usable size of each virtual page.

If the hardware and/or operating system provide a facility for sub-page protections, it would be possible to fault in full pages but only swizzle partial pages, thus reducing the amount of new address space reserved. Using sub-page protection produces the same effect as smaller page sizes in terms of address translation; the whole page must still be loaded in the first time it is faulted on. We defer further discussion of this issue until Chapter 7.

3.6.2 Address Space Reuse

An easy approach for dealing with the exhaustion of the address space is simply to occasionally evict all pages from virtual memory, throw away the existing mappings, and then begin faulting pages in again.[6] Pages that are no longer in use will not be faulted in again, but the current working set will be restored quickly. Once the new mappings have been built, pages (of address space) from the old mappings that are not present in the new mappings can be reused.

Unfortunately, this method incurs unnecessary and bursty traffic between the transient memory and the persistent store when mappings are rebuilt because the working set is faulted out and immediately faulted back in. To avoid this, we take advantage of the fact that address translation and data caching are essentially orthogonal. We do not really have to write data out to reclaim the corresponding pages of the virtual address space. Evicting pages from virtual memory is easy; clean pages can simply be discarded, and dirty pages can be unswizzled and written back to the persistent store.

[6] Note that we cannot just evict a page from the virtual address space, because we do not keep track of pointer assignments—any page that is in the virtual address space must be assumed to have pointers into it from other pages in the address space. Therefore, we cannot reuse that page until we rebuild the mappings—we have to first traverse the graph of pointers and rebuild the mappings to find out which pages are reclaimable.

Rather than actually writing everything out, we can simply invalidate and incrementally rebuild the virtual memory mappings. That is, we "pretend" to write out all of the data, but we leave it cached locally, and just access-protect the pages. We can then incrementally fault on them to build a new set of mappings. If a page is faulted on and it is still in local storage (RAM or disk), so much the better—its pointers can simply be reswizzled according to the current mappings, in much the same way as when the page was originally faulted in. The "obsolete" mappings could be consulted, and could even be re-used in many cases. (If a page is not dirty since it was faulted into transient memory, then it can contains no pointers into pages that it did not previously contain pointers into.)

Reclamation of pages can begin after the application has run a while, to recreate or revalidate all the mappings of its current working set. Candidates for reclamation are pages that have not been referenced since the mass invalidation, and which are not directly reachable from pages that have been. The reclamation policy should probably favor evicting pages that are only directly reachable from pages that have not been touched for a long time. To increase efficiency in systems where page faults are expensive, the virtual memory system's recency information might be consulted (if accessible via the operating system), and the most recently-touched pages could be assumed to be part of the current working set. These pages would have their addresses recomputed or revalidated immediately (rather than being access-protected) to avoid most of the flurry of access-protection faults immediately after the mass invalidation.

Note that address space reuse is not implemented in our system yet—we currently have no applications requiring it, but expect to in the future.

3.6.3 Fine-grained and Mixed-granularity Translation

The last strategy for dealing with address space exhaustion is to use fine-grained (pointer-wise) address translation mechanism for specific data structures that have high fanout. A mixed-granularity scheme (that is, coarse-grained address translation by default, and fine-grained address translation for selected data structures) should provide most of the benefit without much additional overhead.

We have implemented fine-grained address translation by using the C++ *smart pointer* idiom [Str87, Ede92b]. Smart pointers allow implementation of pointer-wise address translation that behaves well with the standard page-wise pointer swizzling scheme without requiring additional hardware support. Further details about mixed-granularity address translation are discussed in Section 3.8.

3.7 Sharing and Compatibility

Pointer swizzling at page fault time can be used as a general purpose reconciliation layer between distinct systems at little performance cost. For example, it can be used to support data formats that allow sharing of data between machines with 32-bit and 64-bit addressing. Of course, applications that truly require a huge flat address space (for example, applications that need flat array indexing into multi-gigabyte arrays) cannot be executed on 32-bit machines.

Sharing pages across nodes in a distributed system would not be costly; in a straightforward scheme, pointers could be unswizzled on transmission and re-swizzled according to the prevailing mappings on the receiving machine. This cost would probably be small relative to the basic trapping and messaging costs in a shared virtual memory. Also, the costs of pointer swizzling could be optimized away in those cases where it is not needed. In a network of 64-bit machines where a larger address space is unnecessary, pages could be permanently assigned to the same virtual addresses on all nodes. Data could then be shared in a “preswizzled” format, with no translation costs whatsoever.

Pointer swizzling at page fault time has a significant advantage in that it can serve as a reconciliation layer to resolve conflicts between different address spaces in a heterogeneous network containing machines with different hardware word sizes. Even in a world of purely 64-bit hardware, this is very desirable. For example, consider the case of merging two local-area networks, each with its own flat shared address space (*a la* [CLLBH92]). Pointer swizzling can be used to resolve conflicts between address spaces without an agonizing renaming process—by its very nature, pointer swizzling at page fault time allows different machines (or sub-nets) to map the same data to different local virtual addresses. Therefore, it requires no clairvoyance on the part of system administrators to ensure that conflicts do not arise between systems that might eventually be merged, e.g., when an organization is restructured or one company acquires another.

Finally, another remaining concern is the complexity added by having the memory system rely on ability to locate pointer fields within heap data. We believe this to be a very small cost; as discussed in Chapter 4, our interface to C++ does not require modifying the compiler at all. True “higher-level” languages (e.g., Smalltalk, Eiffel, Modula-3) would be even easier to interface with the memory system.

3.7.1 Data Formats for Sharing across Machines

For compatibility across different machines, it may be desirable to have a single data format that can be used, irrespective of the address word size of the machine operating on the data. This is particularly attractive for a shared persistent store or a distributed virtual memory. It is easy to accomplish this by using pointer swizzling to adjust pointer sizes. When pages are transferred from one machine to another, it is only necessary to translate the pointers in a page into the native format of the receiving machine.

Pointer swizzling only requires that it be easy to find the pointers in a page, and that it be easy to convert a large persistent pointer into the hardware-supported format. This is done by translating the high order bits (page number) to the shorter bit pattern of the virtual page number, and adjusting the low-order bits that represent the offset within the page. The simplest way to ensure this is to have the persistent data format be the same as the transient format, so that the offset part of a pointer does not change at all. This can be done for multiple pointer sizes by simply leaving enough room for the largest hardware-supported pointer size, whether it is needed on all machines or not. So a 64-bit pointer field can be used on 64-bit machines, and also on 32-bit machines—but only half of the field is used for transient pointers on 32-bit machines. The other half of the field goes to waste, but this space cost is relatively small, especially for languages such as C/C++ because most fields are not pointers.

This is similar to the approach used in the Commandos [MG89] operating system, where object identifiers are used on disk, but are swizzled to actual pointers when data is loaded into memory. However, Commandos does not use page-wise swizzling and incurs high overhead in checking for unswizzled pointers. Using object identifiers rather than persistent addresses also makes translations more expensive.

3.7.2 Linking to Existing Code

Because pointer swizzling at page fault time requires no changes to objects' data formats or the code that manipulates them, it allows swizzled and unswizzled objects to be used freely in the same programs, with only a few restrictions on how they may interact. As discussed in Chapter 2, we use pointer swizzling at page fault time to implement *orthogonal persistence* [AM95].

The orthogonal persistence model allows both transient and persistent objects to be treated in exactly the same way. This allows existing code, typically object code libraries, to be linked with an application without requiring any recompilation, *as long as* these libraries do not need *to create* persistent objects. Also, transient objects may hold pointers to persistent objects, and vice versa, as long as they follow a few simple rules. As persistent objects are saved to the store, all references to transient objects from those persistent object become "stale," and the system must ensure that such references are cleared before the persistent object can be accessed again.

3.7.3 Interfacing with Languages and Compilers

While pointer swizzling at page fault time is obviously applicable to languages like Lisp and Smalltalk that use tagged pointers, it can also be used for strongly-typed languages such as Modula-3, ML, and (with slight restrictions) C or C++. The main restriction for C and C++ is the avoidance of untagged unions with pointers in the variant part. Untagged unions anyway are not very attractive in C++ because of its object-oriented features. The success of conservative garbage collectors shows that most C/C++ programs require little or no modification to meet the necessary constraints.

Unfortunately, conservative pointer identification—as done for conservative garbage collectors [BW88]—is not sufficient for pointer swizzling. Instead, precise information about object layouts is necessary at run time to accurately locate and swizzle pointers. Thus there is a need for a mechanism that facilitates run-time type description (RTTD) rather than simple run-time type identification (RTTI), the latter being designed more towards supporting queries about language-level information. Chapter 6 describes the design of our RTTD mechanism in further detail, along with a description of our implementation for C++. We use the debugging information in object files to extract the necessary object layouts. This allows us to interface with existing compilers since the format of debugging information is typically independent of both the source language and the compiler.

We have implemented pointer swizzling at page fault time for C++ in the Texas Persistent Store (Chapter 4) using existing off-the-shelf compilers. We use RTTD in conjunction with allocator modifications to maintain information about object layouts and swizzle pointers in heap-allocated persistent data structures.

3.8 Fine-grained and Mixed-granularity Translation

Pointer swizzling at page fault time usually provides good performance for most applications with good locality of reference. However, certain applications that exhibit poor locality of reference, especially those with large sparsely-accessed index data structures, may not produce best results with such coarse-grained translation mechanisms. Applications that access big multi-way index trees are a good example; usually, such applications sparsely access the index tree, that is, only a few paths are followed down the tree from the root. If the tree nodes are large in size and have a high fanout, the first access to a node will cause all those pointers to be swizzled, and possibly reserve several pages of virtual address space—most of this swizzling is probably unnecessary since only a few pointers will be dereferenced.

The solution is to provide a fine-grained address translation mechanism which translates pointers individually, instead of doing it a page at a time. Unlike the coarse-grained mechanism where the swizzling was triggered by an access-protection violation, the translation of a pointer may be triggered by one of two events—either when it is "found"[7] or when it is dereferenced.

There are many ways of implementing a fine-grained (pointer-wise) address translation mechanism. We have selected an implementation strategy that remains consistent with our goals of maintaining portability and compatibility with existing off-the-shelf compilers, by using the C++ *smart pointer* abstraction [Str87, Ede92b]. In this section, we first briefly explain the smart pointers abstraction and then describe how we use smart pointers for implementing fine-grained translation in Texas. Finally, we discuss how both fine-grained and coarse-grained schemes can coexist in a single application to create a mixed-granularity environment.

3.8.1 Smart Pointers

A smart pointer is a special C++ parameterized class such that instances of this class behave like regular pointers. Smart pointers support all standard pointer operations such as dereference, cast, indexing etc. However, since they are implemented as C++ classes with overloaded operators to support these pointer operations, it is possible to execute arbitrary code as part of any such operation. A smart pointer class declaration is typically of the following form:

```
template <class T> class Ptr
{
  public:
    Ptr (T *p = NULL);          // constructor
    ~Ptr ();                    // destructor
    T& operator * ();           // dereference via ‘*’
    T *operator -> ();          // dereference via ‘->’
    operator T * ();            // cast operator (cast to ‘T *’)
    ...
};
```

Given the above declaration of a smart pointer class, we can then use it as follows:

[7] A pointer is "found" when its location becomes known. This is similar to the notion of "swizzling upon discovery" as described in [WD92].

```
class Node;                    // assume previously defined
Node *node_p;                  // regular pointer to Node object
Ptr<Node> node_sp;             // smart pointer to Node object
...
node_p->some_method();         // invoke method via regular pointer
node_sp->some_method();        // invoke method via smart pointer
```

It is obvious from the above code fragment that the declaration of a smart pointer is different from that of a regular pointer, but the usage is identical.

Note that we have only shown some of the operators in the declaration of the smart pointer above. Also, we avoid describing the private data members of the smart pointer because the interface is much more important than the internal representation. In other words, it is only necessary to ensure that a smart pointer instance will support all standard pointer operations; it does not matter *how* the class is structured as long as the interface is implemented correctly. In fact, as will be clear from our discussion about variations in fine-grained address translation mechanisms, the smart pointer will need to be implemented differently for different situations and implementation choices.

Smart pointers were originally used in garbage collectors to implement write barriers [Wil92, Wil97] so that pointer updates by the application (also called the mutator) can be tracked easily, allowing the garbage collector to do its job. However, smart pointers are also suitable for implementing address translation for persistence; the overloaded pointer dereference operations (via the "*" and "->" operators) can be implemented to translate persistent pointers into transient pointers as necessary.

Smart pointers were designed with the goal of transparently replacing regular pointers (except for declarations), and providing additional flexibility because arbitrary code can be executed for every pointer operation. In essence, it is an attempt to introduce reflection [KdRB91] into C++ for builtin data types (i.e., pointers).[8] However, as described in [Ede92b], it is impossible to truly replace the functionality of regular pointers in a completely transparent fashion. Part of the problem stems from some of the inconsistencies in the language definition and the implementation dependence. Thus we do not advocate smart pointers for arbitrary usage across the board, but they are useful in certain situations.

3.8.2 Fine-grained Address Translation

We are interested in building a fine-grained address translation mechanism using smart pointers. The idea is to swizzle individual pointers, instead of entire pages at a time, to reduce the consumption of virtual address space for sparsely-accessed data structures with high fanout. By using smart pointers, the programmer can easily choose the data structures that are swizzled on a per-pointer basis, without requiring any inherent changes in the implementation of the basic swizzling mechanism.

Note that although the pointers are swizzled individually, the granularity of data transfer is still units of pages, not individual objects, to avoid excessive I/O costs. Below we describe

[8] C++ already provides limited reflective capabilities in the form of operator overloading for user-defined types and classes.

at least two possible ways to handle fine-grained address translation, and discuss why we choose one over the other.

Fine-grained Swizzling

A straightforward way of implementing fine-grained address translation is to cache the translated address value in the pointer field itself; we call this *fine-grained swizzling*, because the pointer value is cached after being translated.[9] We chose not to follow this approach because of a few problems with the basic technique.

First, fine-grained swizzling incurs checking overhead for every pointer dereference; the first dereference will check and swizzle the pointer, while future dereferences will only check (and find) that the swizzled virtual address is already available and can be used directly. A more significant problem is presented by equality checks—when two smart pointers are compared, the comparison can only be made after ensuring that both the pointers are in the same representation, that is, either both are persistent addresses or both are virtual addresses. In the worst-case scenario, the pointers will be in different representations, and one of them will have to be swizzled before the equality check can complete. Thus a simple equality check, on average, can become more expensive than desired.

One obvious solution is to make the pointer field large enough to store both the persistent and virtual address values as implemented in E [RC89, SCD90]. In the current context, the smart pointer internal representation could be extended such that it can hold both the pointer fields. This technique avoids the overhead on equality checks which can be carried out by simply comparing persistent addresses, without regard to swizzling or existence of the corresponding virtual address.

Unfortunately, a more serious problem with fine-grained swizzling is presented by its peculiar interaction with checkpointing. When a persistent pointer is swizzled, the virtual address must be cached in the pointer field (either E-style or otherwise), that is, we must *modify* the pointer. However, since virtual memory protections are also used to detect updates initiated by the application for checkpointing purposes, updating a smart pointer to cache the swizzled address will clash with this approach, generating "false positives" for updates and causing unnecessary checkpointing. We could, of course, work around this problem by first resetting the permission (i.e., the virtual memory protection) on the page, swizzling (and caching) the pointer, and then restoring the protection on the page. However, this solution is very slow on average, since it requires kernel intervention to change page protections and most modern operating systems are not optimized for such actions.

Translations at Each Use

As described above, a simple fine-grained swizzling mechanism is likely to have some unusual interactions with the operating system and the underlying virtual memory system, thus reducing its attractiveness. However, we can slightly modify the basic technique and overcome most of the disadvantages without losing any of the benefits.

[9] The term "swizzling" implies that the translated address is cached—as opposed to discarded—after use.

The idea is to implement smart pointers that are translated on *every* use and avoid any caching of the translated value. In other words, these smart pointers hold only the persistent addresses, and must be translated every time they are dereferenced because the virtual addresses are not cached. Equality checks do not incur any additional overhead because the pointer fields are always in the same representation, that is, they hold only persistent addresses, which can be compared directly.

Pointer dereferences also do not incur any additional checking overhead. The cost of translating at each use does not add a very large overhead to the overall cost, and is usually amortized over other "work" done by the application, that is, the application may dereference a smart pointer and then do some computations with the resulting target object before dereferencing another smart pointer.

The advantage of this approach is that since the translated address values are never cached, the pointer fields do not need to be modified, and all unwanted interactions with checkpointing and the virtual memory system are avoided. However, this approach is still unsuitable as a general-purpose swizzling mechanism compared to the costs incurred by pointer swizzling at page fault time.

3.8.3 Mixed-granularity Address Translation

It is possible to implement a mixed-granularity address translation scheme that consists of both coarse-grained pointer swizzling and fine-grained address translation. The interaction of swizzling with data structures such as B-trees can be handled without compiler intervention through the use of smart pointer abstraction. The details of a fine-grained address translation scheme are hidden by the abstraction, thus making the approach partially reflective.

In a system configured with only fine-grained address translation, we would not need to examine (and swizzle) any objects at page fault time, because we know that all data pointers are smart pointers that will be translated at each use. In other words, virtual memory access protections are not required to trigger the transfer of data (and pointer swizzling), since all pointer operations will be through user-defined code (via smart pointers). If access protections are never used, no access-protection violations will be generated by the application.

A fully fine-grained approach may, however, introduce a strange interaction with virtual function table (VFT) pointers in C++. Virtual functions are used for dynamic dispatch in C++; they are implemented by incorporating a VFT pointer field, that points to a table of virtual functions, in the object. A VFT pointer is inherently a pointer field, and thus it needs to be swizzled like other pointer fields in the object. However, the difference is that it is a pointer into the code segment (instead of the data segment) and is also compiler-defined, which means that its representation cannot be changed by the user; in other words, we cannot implement it via a smart pointer. Without using virtual memory access protections, it would be impossible to detect use of the VFT pointer without special compiler-generated code. We now have conflict with the default behavior of fine-grained address translation that needs to be resolved somehow. In general, using a mixed-granularity approach, with fine-grained address translation used only for specific data structures will solve the problem with VFT pointers.

3.9 Conclusions

We have presented a novel address translation technique for supporting large address spaces on stock hardware, using only standard compilers and operating system features. Our pagewise approach is designed to take advantage of the fact that most applications exhibit spatial and temporal locality; we exploit this locality of reference in much the same way as a normal virtual memory, gaining many desirable performance characteristics, especially given the trend toward larger main memories. Swizzling at page fault time does not add significant overhead because CPU speeds are much higher compared to disk speeds.

The implementation uses conventional virtual memory hardware and the operating system's memory protection facilities to check residency of persistent data and trigger address translations as necessary. This avoids the need for any software checks which are likely to be more expensive in general. There is, however, a one-time indirect cost associated with pointer swizzling due to its interaction with the underlying virtual memory system. Fortunately, this is not a fundamental limitation of the technique itself, but rather an external overhead due to lack of interaction with the operating system, and can be resolved with operating system support.

Pointer swizzling at page fault time is highly portable because it uses only standard features supported by most modern operating systems. Furthermore, continual checking of pointer format is unnecessary once a persistent address has been translated into a hardware-supported address format because the translated value is cached locally. Thus data access proceeds at full memory speeds after the initial faulting and swizzling is completed. The approach is also compatible with existing off-the-shelf compilers because no special code generation is necessary to incorporate the address translation mechanism. The basic technique is a coarse-grained scheme because the default unit of translation is a virtual memory page. However, for situations where coarse-grained address translation is not appropriate (e.g., data structures with poor locality characteristics), we have developed portable fine-grained and mixed-granularity address translation schemes.

We believe that pointer swizzling at page fault time has a wide variety of applications. As described in the next chapter, it can be used to provide a portable and efficient persistence mechanism for mainstream languages such as C and C++, in essence providing support for 64-bit (or larger) address spaces on standard 32-bit hardware. In general, it is also suitable as a reconciliation layer between otherwise-incompatible system components and abstractions.

Chapter 4

Design and Implementation of the Texas Persistent Store

4.1 Introduction

Texas is a persistent storage system for C++, providing high performance while emphasizing simplicity, modularity and portability. A key component of the design is the use of pointer swizzling at page fault time as the default address translation technique for implementing persistence and large address spaces on standard hardware. In this chapter, we describe the basic design and complete implementation details of the Texas persistent store.

Texas is designed to support orthogonal persistence as its underlying persistence model. Our scheme is also compatible with reachability-based persistence, which can be implemented on top of our approach. Orthogonal persistence allows Texas to support standard off-the-shelf C++ compilers which emit code in the usual way, without having to distinguish between transient and persistent objects. Using standard compilers also provides the added benefit of efficiency and compatibility.

The current implementation offers simple checkpointing capabilities and basic logging mechanisms for storage management and data recovery. As described later in the chapter, these modules are independent of other parts of the system and can be replaced with better algorithms as necessary. Currently, the implementation allows a persistent store to be saved either as a regular file in the file system or directly to a raw disk partition. A file system abstraction layer has been designed to allow advanced storage management and logging strategies to change the underlying implementation transparently. Similarly, we have also implemented a virtual memory abstraction layer to simplify portability to different operating systems.

As discussed earlier, pointer swizzling at page fault time can be used to efficiently support very large address spaces on standard hardware. We intend for Texas' addressing scheme to be extensible and scalable to networked systems where a single address space is used across many machines with large amounts of data apiece.[1]

[1] Despite the fact that we actually live in a hilly area, the name "Texas" is intended to suggest *a large, flat space.*

4.2 Goals and Features

Texas has been designed with several specific goals and features in mind:

- *Portability* Texas is compatible with off-the-shelf C++ compilers and standard operating systems. It requires only a minimal support from the operating system for virtual memory protection and access-protection violation handling capabilities. Most modern operating systems provide these features. In addition, Texas does not require any special system privileges; any user can link Texas with their application without superuser intervention.

- *Transparency* Texas allows an application to access both transient and persistent objects in the same way without distinguishing between them. In other words, persistent objects can be manipulated by the same code that manipulates transient objects because persistent objects "seem" to reside in virtual memory. Thus if the client code does not need to distinguish between transient and persistent objects, it is not forced to do so. The types of persistent and corresponding transient objects are "same," unlike other systems where persistence is implemented by deriving from a top-level "persistence" class and adhering to a specific interface for reading and writing objects.

- *Efficiency* In most cases, access to persistent objects is as fast as access to transient objects. The only overhead associated with persistent object access is the initial cost of translating persistent pointers into swizzled pointers when a page is brought into virtual memory. All future accesses to a persistent object in memory occur at full memory speed without any additional checks. In addition, no overhead is imposed on access to transient objects.

- *Robustness* Texas uses simple logging techniques to provide checkpointing and crash recovery facilities. The system is also designed to be compatible with advanced storage management and logging facilities for achieving improved performance and flexibility.

- *Scalability* Repeated touches to a page incur no extra overhead in address translation, that is, once a page has been swizzled, it is unprotected so that all future accesses cause no further access-protection violations. In addition, the costs incurred at page fault time should decrease as memory sizes increase and thus the number of instructions between faults increases.

- *Compatibility* The implementation is designed to be compatible with existing code libraries which can manipulate both persistent and transient objects alike. Recompilation is necessary only if the library needs to *create* persistent objects. Moreover, the user interface is simple so that minimal source code modifications are necessary for an application to take full advantage of Texas' persistent storage and recovery facilities. The address translation scheme can also reconcile data formats for sharing data between heterogeneous machines and/or merging distinct address spaces.

- *Modularity* Texas is composed of a set of largely orthogonal modules, with address translation, caching and checkpointing handled in nearly disjoint code. The system also contains abstractions for operating system interaction with respect to virtual memory and file system facilities. This has made development simpler, and facilitates easier experimentation and enhancements.

- *Pay-as-you-go Costs* Pointer swizzling costs are incurred only by programs that use the feature, rather than by all programs. This parallels the usual policy of C++ language implementations—you only pay for features that you use.

4.3 Basic Design

The driving requirement for the basic design is to maintain simplicity and modularity in the system. To achieve this, Texas is divided into several modules, each designated with a specific responsibility. However, note that several of the algorithms and their implementations are straightforward, and could easily be replaced with modules that are better suited to specific applications. For example, although Texas currently supports only C++, it could easily be adapted for use with other languages by replacing some of the language-specific modules.

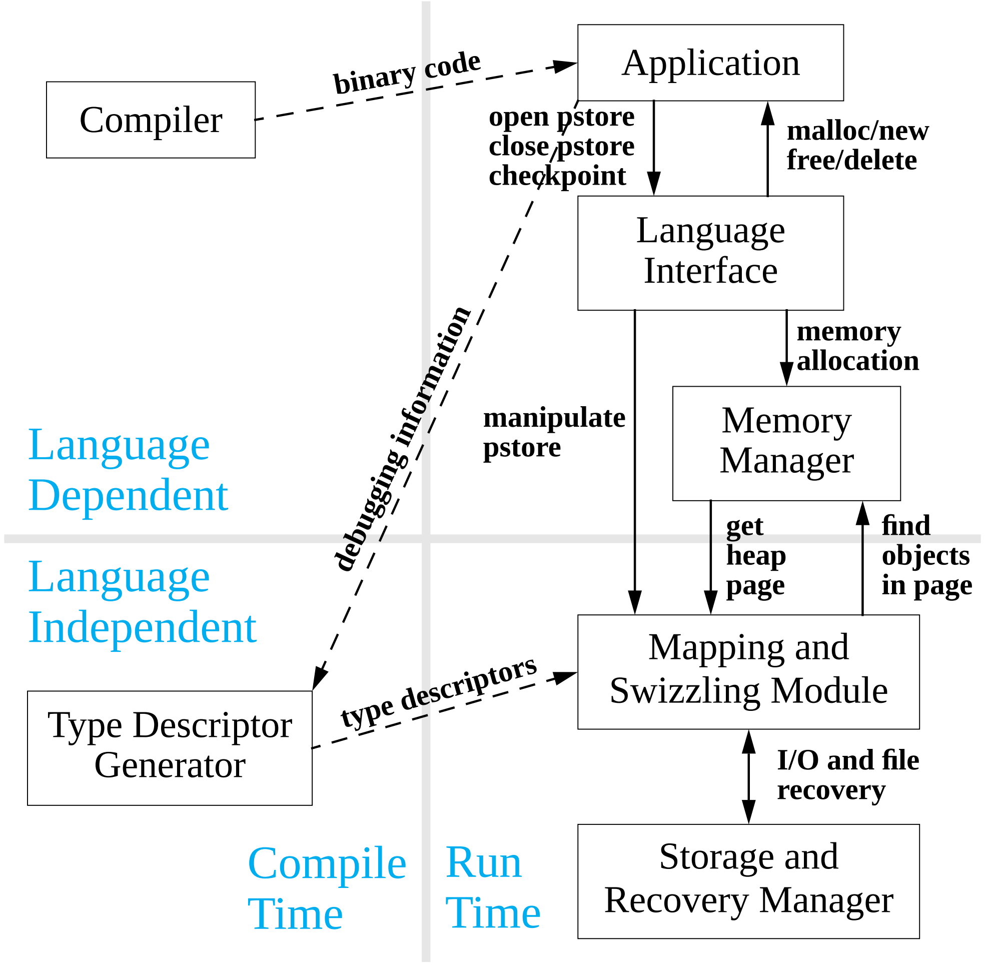


Figure 4.1: Basic design of Texas

Figure 4.1 shows the main modules of the system and their interactions. It is obvious that only the language interface and the memory manager are language-dependent because they need to interact with the actual data objects in the application. In contrast, the caching and storage/recovery management are done in terms of uninterpreted blocks of data. Currently, we have implemented a language interface only for C++, but it would be trivial to extend it to C by making minimal changes, especially because the memory manager would not require any further modifications.

Although the mapping and pointer swizzling module is independent of any specific language, it still needs to interact with the memory manager to locate data objects in memory. In addition, it also needs information about layouts of these objects at *run time* to locate and swizzle pointers. This kind of run-time type description (RTTD) is captured at compile time and provided to the swizzling mechanism at run time via *type descriptor records*. Our implementation of the type descriptor generator is based on the use of debugging information, which is typically language-independent. Complete details about RTTD and our case study implementation for C++ are discussed in Chapter 6.

It should be noted that the various modules shown in Figure 4.1 are designed to be orthogonal to each other. For example, the swizzling and mapping manager does not interact with the storage and recovery manager except as specified by the latter's published interface. This orthogonality allows easy replacement of specific modules without affecting other modules or the rest of the system.

4.4 Implementation Details

We have implemented Texas as a C++ library; client applications can be linked with this library so that they can create and manipulate persistent objects using Texas' persistence mechanism. Texas is relatively small in terms of implementation—the code size is less than 10,000 lines of C++ and the run-time footprint is only about 100KB, making it suitable for applications that must operate with small memory constraints. The code has been ported to a variety of operating systems (SunOS, Solaris, Linux, Mach, Ultrix, and OS/2), and ports should be possible to other modern operating systems such as Windows NT. Texas also works with the GNU g++ compiler (for all Unix-like platforms), Sun C++ compiler (for SunOS/Solaris only) and the IBM VisualAge compiler (for OS/2).

In the remainder of this section, we describe implementation details about various components of the system and how they implement the features mentioned in Section 4.2. Note, however, that several of the algorithms are straightforward, and could easily be replaced with other similar algorithms that are better suited to specific applications. Although Texas is currently implemented for C++, it can be modified to adapt to other languages by replacing the user interface (for languages such as C) and other modules such as the heap management and run-time type description (for languages that are not similar to C/C++).

4.4.1 Heap Management

Texas allows applications to access multiple persistent stores, each with its own heap; in addition, applications may also create transient objects on a normal transient heap. A naive memory manager would create separate heap areas for persistent and transient objects. Each persistent heap would start at some (different) arbitrary address, and each heap would grow and shrink independently. Obviously, this design requires an *ad hoc* static partitioning of a process' virtual address space, which may not be possible (or desirable) on different platforms. Our memory manager avoids statically partitioning the address space and the unnecessary restrictions on the number of pages used for any particular heap.

To avoid static partitioning limitations, our memory manager manages heap space as non-contiguous sets of pages. A given page holds objects belonging to exactly one heap, but pages belonging to several heaps may be interleaved in any order in memory. Large objects that do not fit on a single page are allocated on contiguous pages to allow normal indexing and pointer arithmetic to work as expected; each of such pages is flagged as being part of a large object to ensure correct swizzling behavior.

Like any heap allocation system, the Texas memory manager maintains data structures that record free heap space. Because transient and persistent objects cannot reside on the same page, separate *free lists* are maintained for each heap. The free lists for persistent heaps are themselves stored as data structures in the appropriate persistent store, so that free space within partially-filled persistent pages can be (re)allocated during subsequent program runs.

Currently, Texas uses a segregated storage allocation policy for memory allocation of both transient and persistent objects. We describe the basic algorithm for this memory manager and discuss abstractions that can be used to plug in an arbitrary memory manager instead of the segregated storage allocation model.

Algorithm Description

As the name implies, a segregated storage allocator segregates the allocation of objects based on some specific criteria. In our current implementation, objects are segregated on the basis of their size. A given virtual memory page is split into uniformly-sized chunks, each of which holds a single object. Since it is possible to have an arbitrarily large number of unique object sizes, the allocator maps different sizes into a limited number of *size classes* to allow easy memory management. A size class is defined simply as a representation of a small range of object sizes; objects are allocated in free chunks of memory large enough to hold the actual objects, but possibly with some wasted space if there is not an exact fit. Using this approach, an object of a given size is allocated in the page containing chunks that correspond to that object's size class.

A typical scheme is to use size classes that are powers of 2 (for example, 2, 4, 8, and so on). We compute the size class for a given object size by finding a number n such that the value 2^n is the *closest* power of 2 that is higher than the object size. In other words, the size class is derived by rounding *up* the object size to the closest value that is a power of 2 and then computing the log (base 2) of that value. It is obvious that this approach will generate chunk sizes of 1 byte, 2 bytes, 4 bytes, 8 bytes, 16 bytes, 32 bytes, and so on[2] corresponding to a *powers-of-2* (2^n) series starting at $n = 0$.

[2] Note that very small chunk sizes (for example, 1 to 2 bytes) may not be suitable for actual allocation due to memory alignment constraints and storage requirements for allocator meta-data.

The above scheme works fairly well and is quite easy to implement in practice. Unfortunately, it is also subject to potentially severe *external fragmentation* [RK68] because no attempt is made to split or coalesce blocks in order to satisfy requests for other size classes. However, there is a tradeoff between expected internal fragmentation and external fragmentation. As the spacing between chunk sizes gets large, a larger number of different object sizes fall into each size class, allowing space for some sizes to be reused for others. On the other hand, using a powers-of-2 series is also likely to generate larger internal fragmentation as size classes get bigger because more space is potentially wasted. To reduce some of the fragmentation effects, we use two interspersed series for chunk sizes—the normal powers-of-2 (2^n) series starting at $n = 0$ and a *powers-of-2-times-3* ($2^{n-1} * 3$) series starting at $n = 1$. The resulting chunk sizes (in bytes) would then be 1, 2, 3, 4, 6, 8, 12, 16, 24, 32, and so on, producing twice as many chunk sizes compared to the typical approach, allowing some finer granularity control over object allocation.

The free list for each heap is actually structured as a vector of separate free lists, one for each different size class. When an object must be allocated and the free list for the appropriate size class is empty, new storage is allocated from the operating system to that heap, and is immediately divided into uniform-sized chunks (corresponding to the required size class) that are linked onto the free list. No special handling is required to allocate large objects because the size classes for the larger object sizes will automatically ensure that multiple pages are allocated as necessary.

Splitting each page into uniform-sized chunks makes object identification extremely easy. We only need to examine the header of the first object in the page; its size class—and thus the size class of all the objects on the page—can be determined from the header. The alignment of objects' headers follows trivially. A special exception is the case of objects that are too large to fit on a single page. Instead of recording the actual size class for such pages, each page is marked as being part of a large object and object boundaries are stored in a special table so that their starting addresses (and type descriptors) can be found.

Allocator Abstraction Layer

Our allocator expends little effort attempting to coalesce (or split) free blocks into larger (or smaller) blocks although such coalescing and splitting could potentially affect locality and fragmentation. Simple segregated storage allocator can be made quite fast in the usual case, especially when objects of a given size are repeatedly freed and reallocated over short periods of time. Since the policy does not coalesce or split free blocks, almost no work is done when an object is freed, and subsequent allocations of the same size can be quickly satisfied by removing that block from its free list.

In general, segregated storage policies are known to be prone to higher fragmentation [Joh97]. However, we still chose to implement a memory manager based on this policy primarily because study of memory allocation policies is not our research focus, and because we only needed a fast implementation of a simple allocation policy. A desirable approach would be to define an allocator abstraction layer that specifies the basic functionality required by Texas from a memory allocation policy. Given such a layer, it would be possible to replace the underlying allocator simply by "plugging in" implementation of a different policy that adheres to the interface of the abstraction layer.

Since Texas uses a virtual memory page for most of the granularities of persistence (Chapter 2), the allocator must ensure that a given page belongs to a single heap only. In addition, the allocator must provide a mechanism to store the type information (Chapter 6) for every persistent object allocated.[3] The memory manager requires that the allocator be able to locate this type information for an object using a pointer to (or to interior of) the object. Finally, given some virtual memory page, it must be possible to find all objects that exist on that page.

Given the above abstractions, it would be easy to use an arbitrary policy (such as first-fit or best-fit) for memory allocation in Texas as long as the implementation of policy provides the appropriate functionality dictated by the abstraction layer.

4.4.2 Caching

Pointer swizzling at page fault time implements address translation on top of an abstraction of a conventional virtual memory; we exploit this fact to use the underlying virtual memory as a caching mechanism. Once a page has been loaded into virtual memory from the persistent store, it may be paged out and paged back in again as necessary without intervention from the swizzling mechanism. That is, pages containing swizzled pointers may be paged in and out independently of the transfer of pages between virtual memory and persistent store.

Texas does not explicitly manage physical memory; instead it relies on virtual memory to do the caching in the usual way. While Texas does take advantage of the protection features provided by a modern virtual memory system, it does not look beneath the virtual memory abstraction *per se*—that is, it does not distinguish between pages cached in main memory (RAM) and in the backing store (on disk).

This approach is appropriate for many applications such as typical CAD databases, or for simply replacing conventional files in normal applications. Paging swizzled data locally avoids unnecessary communication with the persistent store, and also avoids the (much smaller) cost of unswizzling and reswizzling pages. In other applications, it might be preferable to page directly from the persistent store. This would avoid redundant storage of pages in both the persistent store and the local disk's swap space. It would also reduce the possibility of dirty pages being paged out to backing store, only to be subsequently paged back in so that they can be written to the persistent store.[4] Naturally, evicting pages directly back to the persistent store would be especially appropriate for diskless clients.

Texas could easily be modified to evict pages back to the persistent store, given control over page-outs. One approach for achieving this is to use something similar to Mach's external pager facility [BKLL93], but it would make the system somewhat more complex and less portable. We will discuss this issue in detail in Chapter 7.

It is also possible to implement the persistent store as a normal file in the file system (see Section 4.4.6). When using the virtual memory system for caching, it is important to avoid caching the persistent storage file in the file system cache. Once a page has been loaded

[3] The current implementation uses a hidden header field in the allocator meta-data.

[4] We believe this problem can also be addressed satisfactorily by writing dirty pages back to the persistent store early, in effect "cleaning" them, as we will describe in Section 4.4.6.

from the persistent store into the virtual memory, it is implicitly cached, and caching the same page in a file system buffer is a waste of resources. Thus the persistent storage file should be stored in an uncached area of disk to avoid this *double caching*.

4.4.3 Virtual Memory Abstraction Layer

We have implemented all virtual memory interactions through an intermediate abstraction layer, allowing us to abstract away the virtual memory facilities required by the system without getting involved in the low-level details of how these facilities are implemented on different operating systems. This has greatly helped with the effort required to port Texas to multiple operating systems, as well as to experiment with different virtual memory primitives on a single operating system.

Texas requires minimal interactions with the underlying virtual memory system. The basic operations required by Texas are supported by most modern operating systems and are as follows:

- ability to allocate a page of virtual address space,
- ability to set and unset memory access protections (no access, read-only or read-write) on a virtual memory page,
- generation of a predefined signal for an application's access-protection violations, and
- ability to specify user-defined signal handlers to catch signals generated due to access-protection violations.

The abstraction layer defines interfaces that allow Texas to communicate with the underlying virtual memory system without becoming involved with the actual implementation details. Chapter 7 contains a detailed discussion on various interactions of Texas and pointer swizzling at page fault time with the underlying operating system.

4.4.4 Run-Time Type Description

By definition, pointer swizzling needs to know the exact locations of pointer fields within various persistent objects that are being swizzled. Traditionally, other run-time support systems such as garbage collectors tend to use a conservative approach such that any value that appears to be a pointer is assumed to be a pointer. Unfortunately, this is not sufficient for pointer swizzling techniques; precise information about pointer locations is required in order to function (i.e., swizzle pointers) correctly.

To solve this problem, it is necessary to have access to implementation-level information, such as object layouts, at run time so that pointer fields can be identified accurately. The recently-introduced C++ Run-Time Type Identification (RTTI) facility is not sufficient for this because it only provides language-level information. Thus we have introduced a notion of Run-Time Type Description (RTTD) facility which is designed with specific goals for providing implementation-level information at run time. Chapter 6 provides the description of our RTTD approach, including details about the C++ implementation used for Texas.

4.4.5 Handling Virtual Function Table Pointers

The most common implementation used for dynamic binding in C++ is via virtual functions [Lip91]. To minimize performance impacts of dynamic binding, virtual functions are implemented via virtual function tables (VFTs). There is one VFT for every class that has at least one virtual function, and a pointer to the appropriate VFT is stored in every object that is instantiated for a class with virtual functions.

This implementation only adds a few instructions of overhead (typically, an index into the table and a load) for every virtual function invocation. However, it poses a challenge for pointer swizzling schemes, because VFT pointers usually reference executable code[5] unlike normal data pointers and therefore cannot be swizzled (or unswizzled) as usual.

We have modified our normal swizzling algorithm to adapt to virtual function table pointers. The basic idea is to convert a VFT pointer into a pointer to a string representing the "name" of the corresponding table in the executable. Conceptually, this string is implemented via a persistent object such that the converted VFT pointer will be handled automatically by the normal swizzling mechanism. In practice, we implement this by converting the VFT pointer into a index into a persistent table that contains names of all VFTs in the executable. The actual conversion is done by performing a lookup in a table that maps VFT addresses to names and vice-versa. Swizzling a VFT pointer is the reverse process; we use the VFT name to look up the corresponding address in the current executable and replace the index with the actual address.

4.4.6 Disk Storage Management

Texas allows a persistent store to be implemented either as a normal file in the standard file system or as a raw disk partition that is not explicitly managed by the file system. We have implemented an abstraction layer that contains standard file operations such as *open*, *read*, *write*, etc. for interacting with the underlying storage management module. This abstraction layer makes porting to different file systems (with different interfaces) relatively easy. For example, an interface to a raw disk partition can easily be implemented through the abstraction layer such that the higher level code is completely unaware of the actual details.

In the remainder of this section, we briefly discuss some issues regarding storage management in Texas. Chapter 8 provides further details into important issues related to storage management for persistent object stores in general.

Checkpointing and Recovery

With a persistent store, the distinction between a conventional heap and files is lost, and explicit checkpointing must take the place of "saving changes to a file." Texas, like any other useful persistent store, supports checkpointing and recovery. As with pointer swizzling, we use virtual memory access protections to determine which pages are modified by the application and save those pages to the persistent store; the actual checkpointing is triggered by a programmer-controlled language-level interface.

[5] Typically, VFT pointers reference virtual function tables corresponding to a specific executable.

Texas is most conducive to both *no-undo/redo* and *undo/no-redo* logging strategies as described in [HR83]. Our current implementation uses a two-phase, write-ahead logging mechanism to provide atomic checkpoints. We implement the no-undo/redo strategy as shown in Figure 4.2.

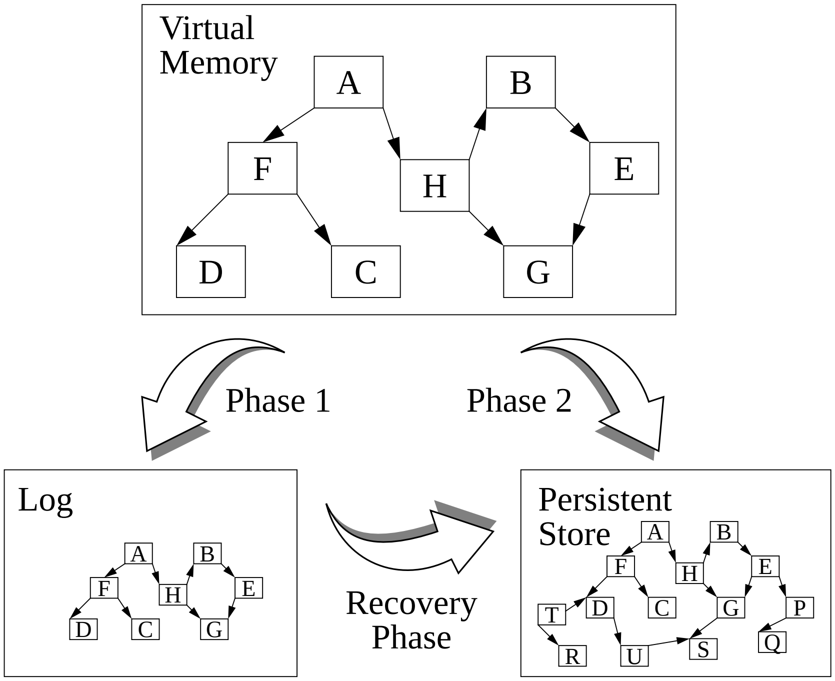


Figure 4.2: Logging mechanism

The basic idea is to ensure that "dirty" (modified) copies of pages are safely stored in the log before the persistent store is updated. When the application requests a checkpoint action, all modified pages since the last checkpoint are first saved to the log (Phase 1) before the persistent store is modified (Phase 2); a crash during the second phase would leave the store in an inconsistent state but we have sufficient information in the log to *redo* the updates to the persistent store (Recovery Phase) to a consistent state by *undoing* the (partial) changes. A crash during the recovery phase only requires repeating that phase until it succeeds. Thus the recovery process is *idempotent*, that is, repeated writes due to retries will produce the same end result.

The same write-ahead logging mechanism can also be used to implement a undo/no-redo strategy. Unlike no-undo/redo, the basic idea is to retain "clean" (unmodified) copies of pages modified by the application. These copies are stored in a log "off to the side" every time the application attempts to update pages that have never been modified before. In this strategy, Phase 1 ensures that all unmodified pages are saved to to the log before Phase 2 is started; a crash during the second phase would leave the persistent store in an inconsistent state, but we have sufficient information in the log to *undo* the updates to the persistent store. As before, the recovery phase is guaranteed to be idempotent, ensuring against any further crashes during recovery actions.

Sub-page Logging

Coarse-grained page-wise pointer swizzling is attractive in most cases because it is designed to exploit spatial locality. However, for very short transactions or transactions with poor locality characteristics, page-wise checkpointing is likely to be inefficient because it will save too much unmodified data—for example, a single write to a page during a transaction will cause the entire page to be written to disk with page-wise checkpointing.

While our system is designed primarily for applications with relatively long transactions, such as typical CAD applications, we would like to provide support for small transactions as well. *Sub-page logging* (as we originally proposed in [SKW92]) is attractive for short transactions because we can checkpoint areas of memory that are smaller than pages. Rather than writing out entire dirty pages, we write out only those parts of a page that have actually changed by "diffing" against a clean copy of the page. In effect, we are trading CPU cycles against disk I/O costs by expending CPU cycles to reduce the amount of data to be written. This is advantageous because of the huge disparity between CPU and disk speeds.

Log-structured Storage System

Instead of refining our simple write-ahead logging scheme, we can replace both the log and the persistent storage file with a log-structured storage system (LSS) that supports checkpointing and recovery both directly and efficiently. An LSS is essentially the lower levels of a *log-structured file system* [RO91]; and manipulates a single large uncached file (typically, a raw Unix disk partition).

In a log-structured store, the entire disk (or file) is used as a log, and the log itself acts as the final repository of data pages. Blocks do not have a single "home" location on disk. Instead, logical blocks can migrate such that the "current" version of a block is simply the last one written to the log. The blocks used for indexing information are also treated similarly. All changes to a file are committed when the top-level indexing information is updated to point to new versions of modified blocks.

4.5 Conclusions

We have presented the design and implementation of Texas, our persistent storage system for C++ that uses pointer swizzling at page fault time as the primary address translation mechanism to support large address spaces on standard hardware. We enumerated some of the main goals and features of Texas that we strived for while designing and implementing the system. Our basic design philosophy was to ensure that the system was divided into independent modules which interacted with each other using only the published interfaces.

We have described some of the important details corresponding to our implementation of Texas. Among these, we discussed issues in heap management and caching, and the abstraction layers that we implemented for interacting with the memory allocator and the underlying virtual memory system. The implementation of such abstraction layers is in line with the general design philosophy which dictates use of orthogonal modules that are easily replaceable with other similar modules that implement the same published interface. It should be noted that these issues, particularly the virtual memory caching, are simply implementation choices and, as such, are independent of the address translation strategy.

We also discussed various factors related to the file system interaction and permanent storage management. Specifically, we described logging techniques for implementing simple checkpointing and recovery facilities in Texas. Although the current implementation incorporates only simple write-ahead logging, advanced mechanisms such as sub-page logging are also feasible and certainly not impossible to implement. However, we have not implemented advanced storage management techniques in detail because our focus is on high-performance address translation schemes. We briefly discuss issues in storage management and also present some future research directions in Chapter 8.

Although the implementation of Texas is only about 10,000 lines of C++ code, the system is fairly robust and has been used in real applications—both commercial and otherwise—for providing fast and inexpensive persistence for C++. We have ported the basic system to several different flavors of Unix, as well as to a non-Unix system (OS/2), and believe that it can be easily ported to other modern operating systems with few technical obstacles.

Chapter 5

Performance of the Texas Persistent Store

5.1 Introduction

The previous chapters have presented the entire theory behind pointer swizzling at page fault time and address translation in the Texas persistent store. We have noted that the overhead of Texas is likely to be very small compared to the I/O costs when data is being loaded into memory from the persistent store, and zero when the data has already been loaded into memory and there is no further faulting. In this chapter, we discuss various issues regarding the performance of Texas and pointer swizzling at page fault time, and present results that support our original assertions about the performance. We use the standard OO1 database benchmark [Cat91] with some minor variations as the workload for most of our performance measurements.

Note that the OO1 benchmark is a synthetic benchmark designed specifically with the purpose of measuring the performance of object-oriented database systems and persistence facilities. However, OO1 and other similar benchmarks are not necessarily suitable for quantitative comparisons across different systems because they have not been validated against real applications in the domain represented by the benchmark. Instead, the results should be interpreted as qualitative results about quantitative performance of real applications. Specifically, it is important to always remember that although the results are obtained empirically, they are ultimately derived from a synthetic benchmark and are only as good as the mapping of benchmark behavior onto real applications. Further discussion on benchmarking limitations is available in Section 5.8.

Although OO1 is a crude benchmark and does not strongly correspond to a real application, we use it for most of our performance measurements for several reasons. First, OO1 is simple for measuring raw performance of pointer traversals (which is what we are interested in) and is fairly amenable to modifications for different address translation granularities. Use of a synthetic benchmark (as opposed to a real application) is appropriate in this situation because our performance is very good in some cases (i.e., zero overhead when there is no faulting) and dependent on the rate of faulting (usually minimal overhead compared to I/O costs) for other cases. As such, crude benchmarking is the most practical way to measure performance of different components of our system because it is easy to separate our costs from those of the underlying benchmark; this is usually more difficult with a real application. Of course, as cautioned earlier, we must be careful to interpret the results in qualitative terms only.

The rest of the chapter is organized as follows. Section 5.2 presents the experimental design used for gathering the actual results, which are presented in Sections 5.3 through 5.6 followed by a discussion in Section 5.7. As part of the overall performance results, we present data for benchmark runs on two popular operating systems—Linux (Section 5.4) and Solaris (Section 5.5)—to highlight the impact of operating system implementation on the overall performance. In Section 5.8, we discuss some issues related to limitations in benchmarking, focusing mainly on the OO1 and OO7 database benchmarks. Finally, we present some concluding remarks in Section 5.9.

5.2 Experimental Design

We are most interested in measuring the overall performance of pointer swizzling at page fault time as implemented in Texas and the specific overheads of the various subcomponents of the system. In addition, we are also interested in studying the impact of variations in the basic scheme (for example, changing the address translation granularity) on the general performance. In this section, we describe the experimental design and methodology followed for gathering our experimental results.

We first briefly examine different benchmarks that are available and discuss the reasons that motivated our choice of the OO1 benchmark, which is then described in further detail. We also describe the experimental methodology that we used for the performance analysis and measurements, including issues about I/O strategies (raw I/O vs. file system I/O) and precise timing requirements. Finally, we describe the hardware and operating systems used for gathering our results.

5.2.1 Benchmarks

One of the most popular object database benchmarks that has become the *de facto* standard is the OO1 (Object Operations One) benchmark [CS92]. OO1 was also one of the first widely-used database benchmarks, followed by others such the HyperModel [ABM$^+$90] and OO7 [CDN93] benchmarks.

The OO7 benchmark is designed as a successor to the OO1 benchmark, and supports some advanced data structures and complex operations over these structures to represent a hypothetical CAD application. We used OO1 for all our performance measurements because of several fundamental reasons. OO1 is simple for what we are most interested in measuring—the raw performance of pointer traversals—to calculate the basic overhead of a coarse-grained address translation mechanism. However, the results obtained from this benchmark should be interpreted carefully. For the pointer traversal performance, we use the traversal results as qualitative indicators of quantitative performance of real applications.

We believe that the OO1 and OO7 benchmarks are unsuitable for performance measurements of orthogonally persistent systems in general; this is further discussed in Section 5.8.

The rest of this section describes the OO1 benchmark, the primary workload used to measure the performance of various subcomponents of pointer swizzling at page fault time and Texas.

OO1 Benchmark Database

The OO1 benchmark database is made up of a set of *part objects* (representing parts in a hypothetical engineering database application) interconnected to each other. The benchmark specifies two database sizes based on the number of parts stored in the database—a *small database* containing 20,000 parts and a *large database* containing 200,000 parts. The rationale behind specifying two database sizes is to allow performance measurements of a system when the entire database is small enough to fit into main memory and compare it with situations where the database is larger than the available memory.

The parts are indexed by unique *part numbers* associated with each part.[1] Each part is "connected" via a direct link to exactly three other parts, chosen partially randomly to produce some locality of reference. In particular, 90% of the connections are to "nearby" 1% of parts where "nearness" is defined in terms of part numbers, that is, a given part is considered to be "near" other parts if those parts have part numbers that are numerically close to the number of this part. The remaining 10% of the connections are to (uniformly) randomly-chosen parts.

The direct connections are also referred to as *forward connections*. In addition to these, each part also maintains a set of *reverse connections* containing pointers to other parts that have forward connections to this part. The forward connections are implemented through direct pointers to part objects. Each part has a fixed-size array of pointers that represent forward connections because the number of forward connections is fixed (i.e., three) by the benchmark specification. In addition, each part also has a few other data fields (integers and strings) that are used during the benchmark operations.

OO1 Benchmark Operations

The OO1 benchmark suite comprises of several different types of operations, rather than just a single test. These operations are broadly classified into three types:

- *Lookup* Locate a predetermined number of randomly-chosen parts by using the parts index and invoke an empty procedure on each part;

- *Traversal* Perform a depth-first traversal of all connected parts starting from a randomly-chosen part and traversing up to seven levels deep for a total of 3280 parts (including possible duplicates), and invoke an empty procedure on each visited part; and

- *Insertion* Allocate and insert new parts into the database using the same criteria that were used for making the forward connections.

The OO1 operations are designed to represent various phases of an engineering database application. For example, the Lookup operation can be used to measure the performance of indexed object retrieval in a database system. In contrast, the Traversal operation concentrates

[1] The benchmark specification does not define a data structure that must be used for the index; we used a B+ tree for all our experiments.

on raw performance of pointer traversals, which is what we are interested in for measuring the performance of pointer swizzling at page fault time. The Insertion operation is suitable for measuring performance of checkpointing and updates because it actually modifies the database on disk. Another approach for this is a variation on the traversal operation described in [WD92]; the basic idea is similar to the traversal except that, in addition to invoking an empty procedure on visited parts, it also allows for updates with some predetermined probability. However, it is not clear whether this variation makes for a good benchmark because it is tightly coupled to the (randomized) interconnections and is likely to cause scattered updates across the entire database with poor locality of reference. We believe that this approach should be used with caution because it makes page-wise checkpointing look unnecessarily bad, while making page-wise "diffing" [SKW92, Whi94] look unrealistically attractive.

Although the randomized interconnection scheme exhibits some locality (that is, 90% of connections are close by), it has disastrous effects on locality of simple algorithms operating over the data because, on average, *every tenth pointer traversal accesses a randomly-chosen part that is not close.* The OO1 designers were aware of this, at least to some degree; they specify that traversals must be executed ten times, each starting at a different "root" part. The first traversal is for a "cold cache" when none of the database is cached in memory, and the subsequent traversals are for a cache that is getting "warmer." In addition, ten traversals must also be executed for a "hot cache" which already contains the data to be traversed. (This is accomplished by starting the hot traversals at the same root as the last warm traversal, essentially repeating the last warm traversal exactly, and guaranteeing that all visited parts are already in memory.)

5.2.2 Methodology

We use the OO1 benchmark traversal operation for all our performance measurements. Each *traversal set* contains a total of 45 traversals split as follows: the first traversal is the *cold* traversal (when *no data* is cached in memory), the next 34 are *warm* traversals (as *more and more data* is cached in memory) and finally the last 10 are *hot* traversals (when *all data* is cached in memory). Note that this is different from the standard benchmark specification which contains only 20 traversals (split as 1 cold, 9 warm, and 10 hot traversals). However, we chose to run more warm traversals because we believe that 9 traversals are not sufficient to provide meaningful results, especially for the large database case. As we will describe in Section 5.8, OO1 (and other benchmarks) are not necessarily good indicators of general application behavior, and are unsuitable for quantitative comparisons between different systems.

We use a random number generator to ensure that each warm traversal selects a new "root" part as the initial starting point, thus visiting a mostly-different set of parts in each traversal. (Of course, some warm traversals are likely to visit parts that have already been visited in a previous traversal because of the randomized interconnections in the data structures.) We run the entire traversal set (45 traversals) multiple times interspersed with a "chill" program that "cools" the memory between runs to ensure that cold traversals are truly cold.[2] Then, we average over all runs after discarding outliers to obtain the final performance results.

[2] The program allocates as much data as the available memory size, writes everything to a file on disk and then reads it back in, clearing both memory and file system buffers in the process.

The remainder of this section describes the methodology used to measure the basic performance of our system, and to study various granularities of address translation and their impact on the overall performance of a pointer swizzling system. We also discuss some issues related to precise timing before presenting the empirical results starting with Section 5.6.

Basic Performance Measurements

For basic performance analysis, we are primarily interested in measuring the overhead of pointer swizzling at page fault time during various phases of the benchmark execution. We can accomplish this by placing timers at various strategic points in the code and using the measurements from these timers to accurately identify the costs of different components of the system. The basic timers setup is pictorially depicted in Figure 5.1.

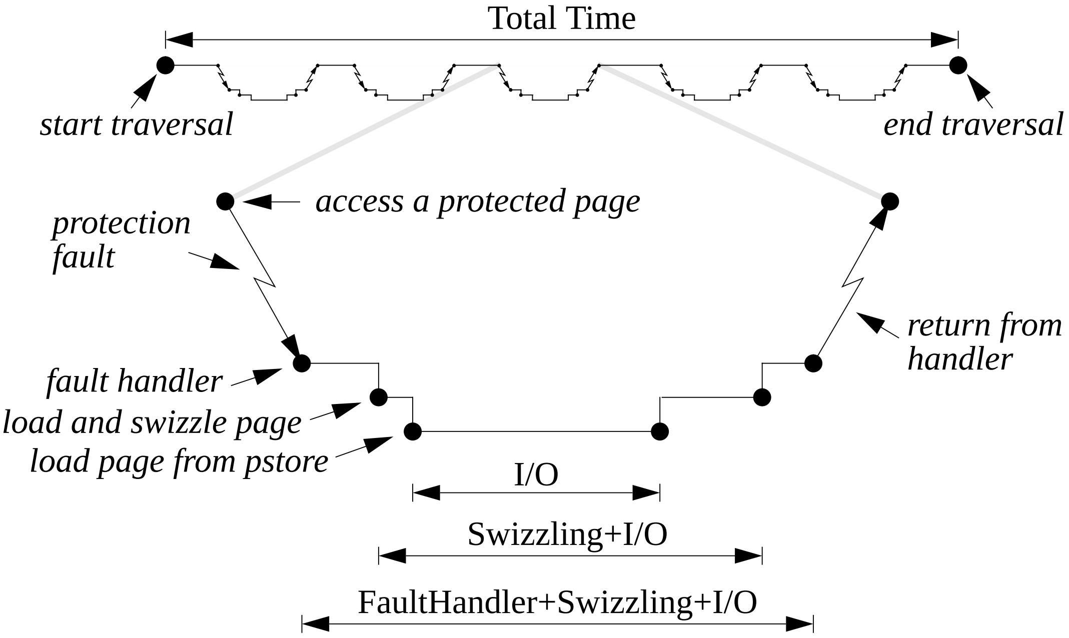


Figure 5.1: Timer placements for run-time measurements

The figure shows an imaginary time line for a single traversal. The total time for a traversal, including any faulting and swizzling that may have been necessary, is measured by starting a timer at the beginning of the traversal, and stopping it at the end. It is possible that the benchmark will access objects on protected pages during the traversal, and an access-protection violation (also called a protection fault) will be generated for the first access to every protected page. The pointer swizzling module of Texas services each fault by loading and swizzling the corresponding page from the database. For calculating the time spent in different parts of the system during one traversal, we record the time for various events that occur during the faulting and swizzling of each page. As shown in the figure, we use individual timers to measure the time for reading a page from disk (“I/O”), the total time for reading a page and swizzling it (“Swizzling+I/O”) and the total time for handling a single protection fault at the user-level including swizzling and I/O (“FaultHandler+Swizzling+I/O”). Summing the timer values over the entire traversal gives us the total time for each component of the system during that traversal. Using simple arithmetic subtraction, we can then calculate the time spent only in pointer swizzling and the corresponding overhead compared to I/O and benchmark costs.

The only time component that we cannot accurately measure using this approach is the time taken by the kernel itself to service a fault, that is, the time from the point when the fault is generated until our fault handler gains control (shown as a jagged edge in Figure 5.1). Similarly, the return from the fault handler also cannot be timed, although it is likely to be minor (equivalent to a function call return). We estimate these values by measuring them using a stand-alone test program that generates a few thousand protection faults in a tight loop. We measure the total time for the entire loop and divide by the number of faults to get the average time required by the kernel to service a fault, transfer control to a user-level fault handler, and return from the handler. We believe that this estimation is acceptable because we are likely to get an *underestimate* (a lower bound) on the cost due to caching effects.[3] Further details about general exception handling performance of modern operating systems are discussed in Chapter 7.

OO1 Benchmark Traversal Characteristics

The basic OO1 traversal set (comprised of 45 traversals described earlier) can be broadly divided into three phases. The overhead of our system typically varies for each phase depending on the behavior and access patterns of the benchmark during that phase. Each phase may also be thought of as representing applications that exhibit behavior similar to that particular phase. The three phases can be characterized as follows:

- The hot traversals correspond to a situation where there is no I/O activity and the benchmark is operating on data that has already been faulted into memory. This is similar to CPU-intensive applications that first load a fixed amount of data into memory and then operate exclusively on that data throughout their execution; such applications typically want a simple persistence mechanism for their data without being forced to "roll their own" through *ad hoc* techniques. Pointer swizzling at page fault time imposes absolutely *no overhead* for such applications because no new data is faulted in and all existing pointers have already been swizzled for the major part of their execution.

- The cold traversal and the first few warm traversals typically represent the other end of the spectrum. Since most of the data accessed by the benchmark during these traversals has to be fetched from disk, this phase is characterized by a lot of faulting and I/O requests as the traversal references parts that have never been seen before. This phase corresponds to applications that are largely I/O-intensive, and do not have an equivalent amount of computation. For such applications, our overheads are much smaller than the cost of I/O, which usually dominates the overall performance.

- Finally, the third phase is characterized by moderate I/O and faulting behavior interspersed with general computation, representing the middle ground between the other

[3] Since faults are generated in a tight loop in the test program, the kernel code and data structures for fault handling are likely to be cached (possibly in the second-level cache) after the first iteration. However, an actual application is unlikely to benefit from such caching with normal faulting behavior.

two phases. Most applications generally end up in this phase only after going through an I/O-intensive phase for loading large amounts of data. The overhead of our system for this phase is likely to vary significantly depending on the application behavior and faulting patterns.

As we present the performance results in the following sections, we will show that the empirical data supports this overall classification of the benchmark traversal set.

File I/O vs. Raw I/O

On most operating systems, under normal circumstances, reads and writes to a regular file go through the kernel (for example, using the `seg_map` driver on SVR4 systems [Vah96]). When a `read` system call is invoked, data is first read from the file into kernel space and then *copied* into user space (into a user-specified buffer). In addition, operating systems also implement a sequential *readahead* mechanism (akin to simple lookahead prefetching) to read more data than requested, at every disk seek, for minimizing the overall I/O costs. The prefetched data is stored in a file system cache and transferred to user space if the application requests that data before it is evicted from memory.

The file system readahead and caching works favorably for most normal applications that read and write data from the disk. For our purposes, however, it is obviously inefficient because our access patterns are unsuitable for such caching. Specifically, we know that the data will be cached in virtual memory, causing unnecessary *double caching* (that is, caching in both user space and kernel space). This is particularly undesirable because the file system cache also competes for the available physical memory, reducing the effective RAM available for virtual memory caching.

One solution for avoiding file system caching is to use a *raw device* which provides a direct interface to a raw partition on disk without involving the file system. Each read or write from the raw device causes an actual I/O operation, and data is copied directly into the user space. This avoids double caching and the file system reads only as much data as requested (i.e., no prefetching). Such I/O is usually known as *raw I/O* to distinguish it from *file I/O* done via a normal file system.

Currently, Linux does not provide any user-level support for raw I/O,[4] and hence we use a normal file (on a local disk attached to the test machine) for storing the benchmark databases. The effects of file system caching and readahead are typically more evident in the small database results because the database fits easily in the available RAM, and the readahead pays off because almost the entire database is accessed during the traversal set. For large database results, we found that Linux is aggressive with file system buffer management, reducing any adverse effects on virtual memory caching and overall performance.

The situation is quite different for the experiments on Solaris. Unlike Linux, Solaris does support I/O to a raw device, allowing us to avoid the unwanted caching and readahead. In Section 5.5, we present results on Solaris corresponding to the use of both file I/O and raw I/O for loading data from the database, and highlight some important differences between the two strategies.

[4] We believe that this feature is under development as of this writing.

Address Translation Granularities

In addition to the performance overheads, we are also interested in comparing various address translation granularities described in Chapter 3. The standard pointer swizzling at page fault time strategy corresponds to a coarse-grained address translation approach where *all* pointers in a page are swizzled when the page is loaded into memory regardless of whether they will be accessed by the application. A pure fine-grained scheme falls at the other end of the spectrum, and can be realized by using smart pointers instead of normal (language-defined) pointers for all persistent data structures. This scheme performs pointer-wise address translation, translating persistent pointers into virtual memory addresses every time the persistent pointers are dereferenced. Between the two extremes is a mixed-granularity address translation approach which uses a combination of smart pointers and normal pointers in the persistent data structures to have a mix of both coarse-grained and fine-grained address translation granularities.

We modified the data structures used in the basic traversal of OO1 benchmark to implement the above three different address translation granularities. As might be expected, the pure coarse-grained and fine-grained approaches were implemented by using all normal pointers and all smart pointers respectively in the benchmark data structures. We implemented the mixed-granularity approach by modifying only the parts index structure to use smart pointers while maintaining normal pointers for the rest of the data. We believe that a B+ tree (the data structure used to implement the parts index) is appropriate for such conversion because it is a sparsely-accessed data structure with high fanout, typical access characteristics and topological properties of data structures suitable for fine-grained address translation.

Precise Timing

For all experiments described so far, we need a highly *precise* timing mechanism to accurately measure the overheads of our system in different situations. Furthermore, as we will show through the empirical results, some of the overheads in our system are extremely small, thus placing an additional requirement for high *resolution* on the timers. Unfortunately, commonly available timers on most modern operating systems have a poor resolution, sometimes on the order of several milliseconds.

Most modern operating systems provide various system calls that can be used to measure either *CPU time* or *real ("wall-clock") time* for some event in the application. CPU time is the time actually spent in processor execution, while real time is the total wall-clock time for the event, including time spent in I/O waits, paging, context switches, etc. The CPU time is further split into *user* and *system* components corresponding to the time spent executing on the processor in user mode and kernel mode respectively. Typically, the best resolution of real-time timers on most modern operating systems, without any special hardware support, is on the order of microseconds. The resolution tends to be even lower for CPU-time timers because of the added overhead in maintaining appropriate data structures and tracking kernel boundary crossings during execution. For example, the resolution of a CPU timer on a 200MHz Intel Pentium Pro processor running Solaris 2.5 is approximately ten milliseconds; this is very coarse considering the fact that the processor can execute about two million instructions in that time frame. Even a one millisecond resolution is equivalent to the time required to execute approximately 200,000 instructions.

Obviously, these resolutions are too coarse for our purposes given the low overheads of our system (especially when compared to I/O costs). Fortunately, platforms that are compatible with the Intel Pentium architecture contain special hardware that allows high resolution timing measurements at the granularity of processor *clock cycles*. The basic idea exploits a 64-bit register in the Pentium architecture; this register counts the number of processor clock cycles since the last reboot, thus providing a true fine-grained mechanism for precise timing. There is an instruction (`rdtsc`, mnemonic for "read time stamp counter") that can be used to read the current value stored in the register. Using this counter register and the associated instruction, it is relatively easy to build a timer that can be used for precise measurements of various low-overhead events in terms of clock cycles.[5] A (relatively minor) downside of using such cycle timers is that they can be used to measure only real time without additional support from the operating system for measuring CPU time at the same granularity. In order to minimize the effects of arbitrary swings in real-time measurements due to transient events, we run the benchmark suite multiple times on unloaded machines and average the results after discarding significant outliers.

We also experimented with Solaris high-resolution time via the `gethrtime` system call. This timer also measures real time with typical resolution in microseconds; the actual resolution, however, depends on the underlying hardware. For the platforms that we used (described next), this resolution was between one and two microseconds. Since the `gethrtime` call is officially supported for arbitrary hardware, it is obviously more portable than the clock-cycle timer described above, which is usable only on Pentium-compatible processors. However, since the cycle timer provides a much better granularity, we use it for all our experiments that were run on Pentium-based platforms, and only use the CPU time measurements for address translation granularity comparisons that were run on SPARC-based platforms.

5.2.3 Hardware and Operating Systems

We ran the various benchmark operations on both Linux and Solaris platforms to gain insight in the behavior of two of the most popular and widely-used operating systems. Although we were able to derive the same qualitative conclusions for both operating systems, we found some interesting differences between them, as well as potential for both to be improved.

For both operating systems, we used identical hardware setup (except for differences in component manufacturers, which are not relevant for our study). Each test machine was equipped with a 200MHz Intel Pentium Pro processor (with a 256KB L2 cache) and 32MB of EDO RAM. The operating system versions used were Linux 2.0.0, (the current major stable release of Linux)[6] and Solaris 2.5 (the current major release of Solaris). (During the course of gathering our results, we also used some SPARC-based platforms for performance measurements on Solaris. We found that the results (not presented here) correlated well with those on the Pentium-based platform, and as such were useful as a good sanity check.)

[5] We use a modified version of a timer originally developed by Mark Johnstone. Other approaches, such as preprocessor macros and inline procedures containing assembly code to access the special register, have also been suggested on various Usenet newsgroups devoted to Linux.

[6] As of this writing, the current minor stable release is version 2.0.30.

In addition to the standard Solaris measurements, we also needed access to a Solaris platform with main memory that was big enough to completely fit the large database in RAM. As we will describe later, this setup was needed to validate a theory about specific behavior of Solaris for workloads involving databases bigger than available memory size. For this purpose, we used the same test machine after upgrading its memory to 64MB, which was sufficiently large for our purposes.

Finally, for the comparison between different address translation granularities, it is necessary to measure CPU time for each variation because the costs typically differ only in terms of CPU execution. However, since our overheads are relatively small, and because most CPU-time timers have a coarse granularity, it is difficult to *accurately* measure CPU time on most modern processors. Instead, we used an older 33MHz SPARCstation ELC for these experiments; the processor on this machine is slow enough to offset the coarse granularity of the CPU-time timer. We believe that such controlled use of an older (and slower) processor is acceptable here because the results are reported relative to each other, that is, the performance is compared to other variations that are also executed on the *same* processor.

5.3 Instruction-Count Profiling Results

As part of the overall results, we first present an analysis of instruction-count profiling for various key components of the pointer swizzling at page fault time mechanism as implemented in Texas. We used QPT [BL92]—an instruction-count profiling tool—for this purpose and measured the costs of swizzling a single pointer and swizzling an entire page. QPT is similar to the Unix profiling tool `gprof` with one important distinction. Unlike `gprof`, which reports results in terms of absolute time per procedure, QPT is capable of calculating the number of instructions for each procedure by analyzing and instrumenting basic blocks in executable code. This is very useful for pure overhead measurements because the output of QPT is in terms of number of instructions for each procedure, an absolute result that is independent of other procedures in the application. In contrast, `gprof` is more suitable for general profiling and comparative analysis because it highlights "problem areas" that are most likely to benefit from optimization compared to other parts of the application.

feature	instructions
translate a single pointer	40
decode a type descriptor record	130
swizzle a page (with normal decoding)	12,000
swizzle a page (with optimized decoding)	8,000

Table 5.1: Estimated instruction counts

Table 5.1 shows the cost of several important components of our system. The most basic result is the cost of translating the value of a single pointer which is approximately 40 instructions. The bulk of this cost is attributed to the hash table lookup based on the address value in the pointer field being translated. Note that this estimate is for an untuned hash table implementation that does not use highly-optimized data structures and algorithms.

With further optimizations, it should be possible to reduce the translation cost by half the current amount.

Note that the cost of translating a single pointer is measured in isolation, and excludes costs for all other actions that may be necessary during swizzling but are not *directly* related to the actual translation itself. For example, if the swizzled value references a new page that has not been seen before, we must reserve a page by allocating virtual address space from the operating system and access-protecting the new space. Although this is necessary for ensuring that swizzling works correctly over the course of the application, it is not directly involved with the translation of a single pointer, and is therefore excluded from our measurements.

Another important cost to be accounted for during pointer swizzling is the cost of decoding a single type descriptor record[7] for locating various pointer fields in the object described by the record. On average, for the OO1 benchmark, each type descriptor record contains information about four data pointers and one virtual function table (VFT) pointer, and the cost of decoding such a type descriptor record is approximately 130 instructions.

Apart from measuring costs of specific routines in isolation, it is equally important to measure the costs at a higher level of abstraction so that we can study the effect of swizzling on the overall performance. For this purpose, we measured the number of instructions required for swizzling an entire page, *including* the costs of all supporting actions such as reserving new pages of address space, decoding type descriptor records, etc. As shown in the table, approximately 12,000 instructions are necessary on average to swizzle a page during the OO1 traversal operation.

This cost can be reduced by optimizing the decoding of type descriptor records. Using the size of type descriptor record objects, we calculate that there are about 30 such objects on any given virtual memory page. Thus the cost of decoding type descriptor records on one page is approximately 3,900 instructions, or about one-third the total cost of swizzling an entire page. In Chapter 6, we describe an optimization that can reduce the cost of decoding a type descriptor record to a single procedure call, thereby reducing the per-page swizzling cost to approximately 8,000 instructions. On today's commonly available processors rated at 200 million instructions per second (or more), this is equivalent to one-twentieth of a millisecond (or less). Obviously, this is very insignificant compared to typical I/O costs incurred by an application for fetching data from disk.

5.4 Performance on Linux

The hot traversals for both small and large database experiments on Linux correspond to the first (CPU-intensive) phase of the traversal set described earlier. In general, the large database is better suited for the second (I/O-intensive) phase; since the database is relatively big, even later warm traversals reference at least one or more pages that have not been seen before, thus keeping up the I/O activity. On the other hand, the small database results typically highlight the third phase, because most of the database is loaded into memory within the first few traversals due to both benchmark locality characteristics and file system readahead.

[7] Type descriptor records are objects used to maintain run-time type information (Chapter 6).

We first present detailed results for the large database, followed by a corresponding set of results for the small database. As mentioned above, each of these highlights different characteristics of the access patterns and corresponding behavior from Texas. The set of results for each database size are further split into three parts comprising the raw performance data, a measure of real I/O activity[8] during the traversal set, and lastly the overhead of Texas and pointer swizzling as a percentage of both I/O time and total benchmark time. We end the section with a brief analysis of the basic results for the two database sizes.

5.4.1 Large Database Results

We start with the raw performance numbers of OO1 forward traversals over the large database. The database, which contains 200,000 parts, is approximately 43MB in size, roughly one-third more than the available RAM on the test machine.

Basic Performance

Figure 5.2 shows the overall run time of the entire traversal set (45 traversals) on the large database. The same data is also plotted on a log scale in Figure 5.3 for additional detail during later traversals. These figures (and all other figures for performance results) contain multiple plots, each corresponding to a different component of the system. The total time per traversal (labeled as "Total" in the figures) is the most obvious measure. In addition to this, the figures also include the costs for different components plotted in a cumulative manner starting with the I/O cost. In particular, the plots labeled as "I/O," "S+I/O" (short for "Swizzling+I/O") and "FH+S+I/O" (short for "FaultHandler+Swizzling+I/O") correspond to measurements from the timers placed at various strategic points in the code (see Figure 5.1). We also generate a plot that includes the estimated time required by the operating system for trapping protection faults, labeled as "F+FH+S+I/O" (short for "Fault+FaultHandler+Swizzling+I/O"). Alternatively, this plot can also be labeled as "Texas+I/O" because the sum of the three components other than I/O ("Fault+FaultHandler+Swizzling") is effectively the total overhead of Texas and pointer swizzling at page fault time.

[8] We use the term *real I/O* to indicate that actual disk I/O (including a disk seek) was performed for an I/O request that could not be satisfied from a cache.

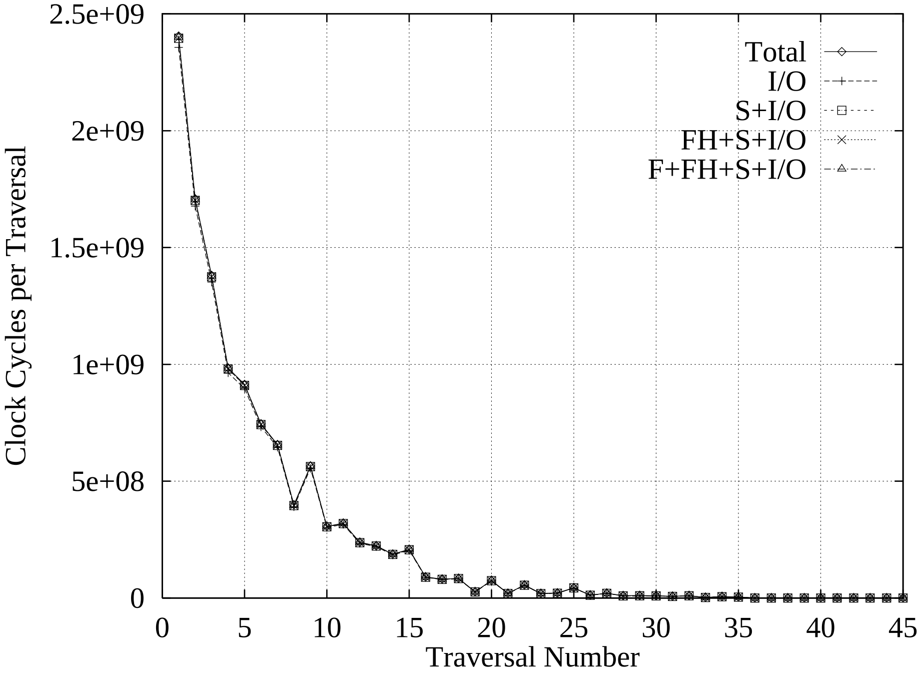


Figure 5.2: Times for all traversals, large database (Linux)

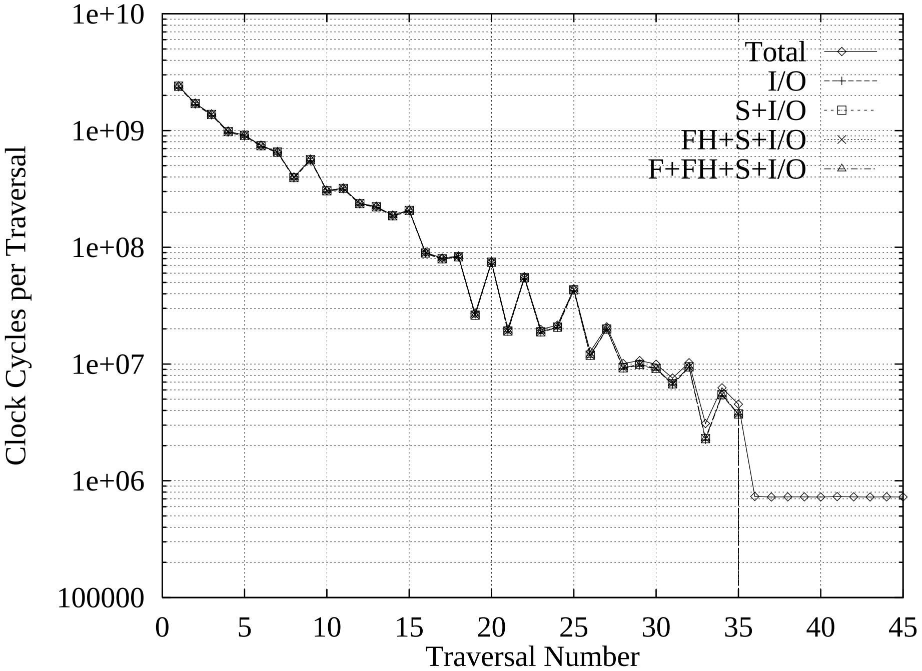


Figure 5.3: Times for all traversals, large database, log scale (Linux)

The individual plots for the different components are not discernible from Figure 5.2, but I/O time is the largest component for the cold and warm traversals, while the Texas overhead for the same is comparatively very small. This observation is supported by the fact that the lines for various plots are very close to each other even on a log scale (see Figure 5.3).

The low overhead of pointer swizzling at page fault time is further evident from results presented in Figures 5.4–5.8 which show the closeups of all cold and warm traversals (i.e., traversals 1 through 35). Finally, Figure 5.9 shows the overall performance for the ten hot traversals (i.e., traversals 36 through 45); as expected, there are absolutely no I/O costs and the Texas overhead for these traversals is *zero*.

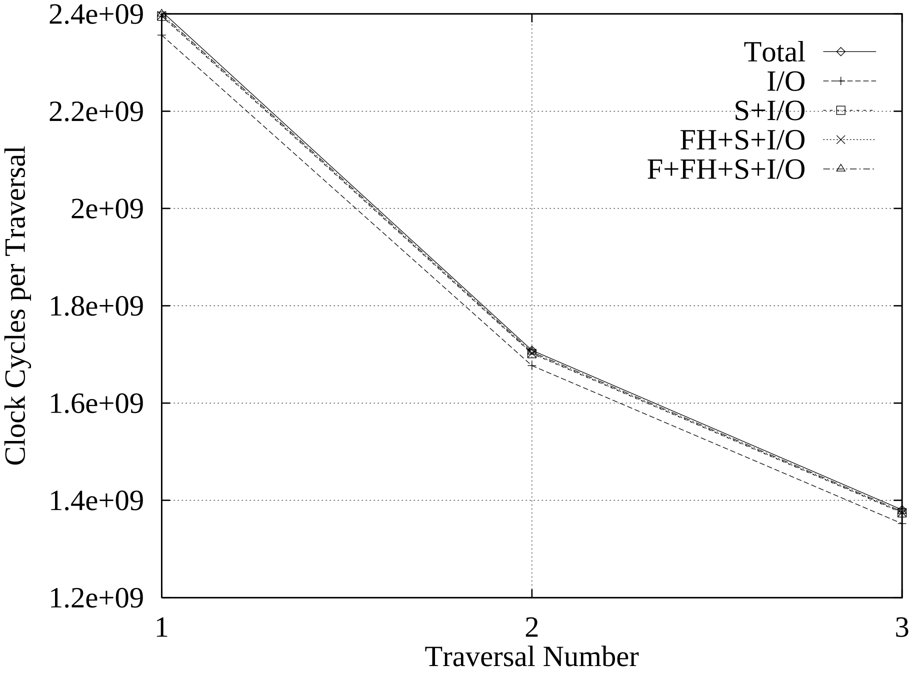


Figure 5.4: Times for traversals 1 through 3, large database (Linux)

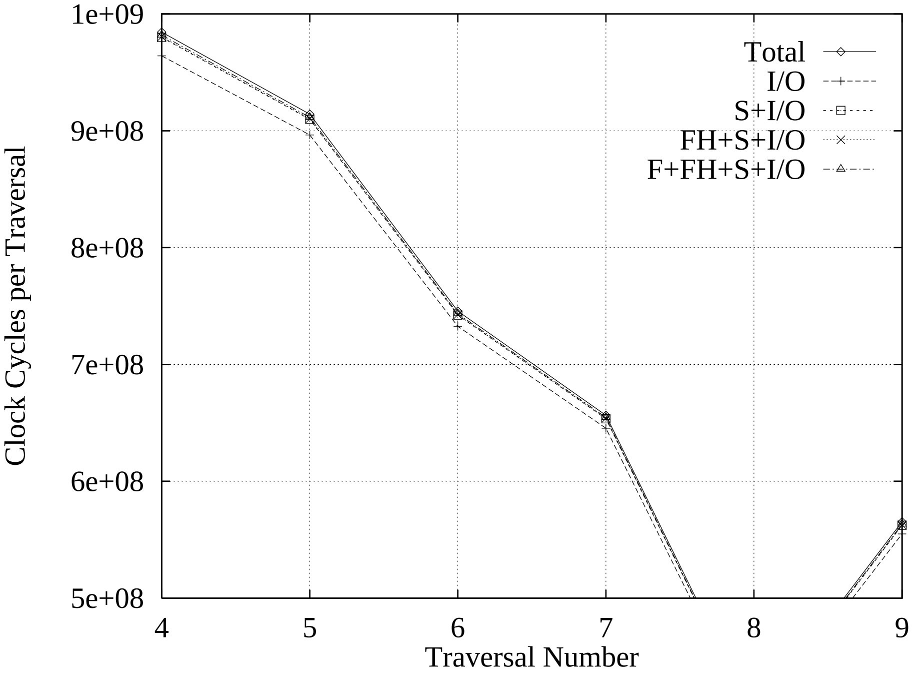


Figure 5.5: Times for traversals 4 through 9, large database (Linux)

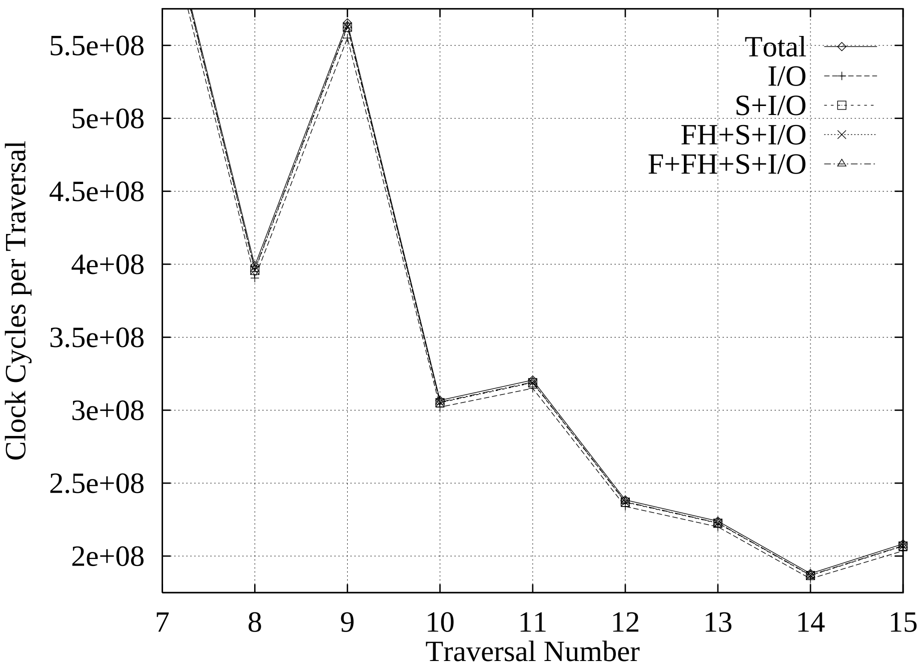


Figure 5.6: Times for traversals 7 through 15, large database (Linux)

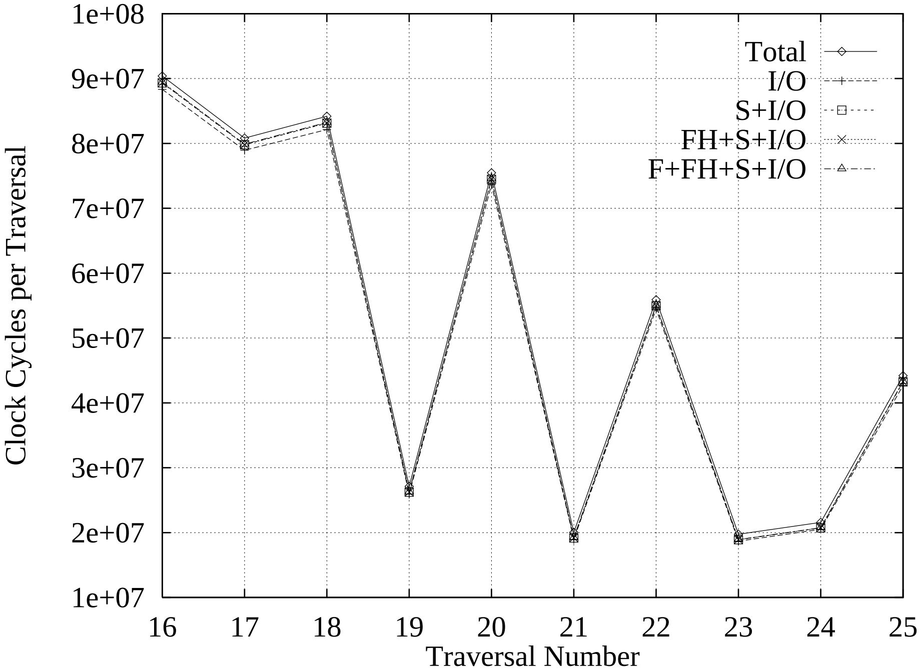


Figure 5.7: Times for traversals 16 through 25, large database (Linux)

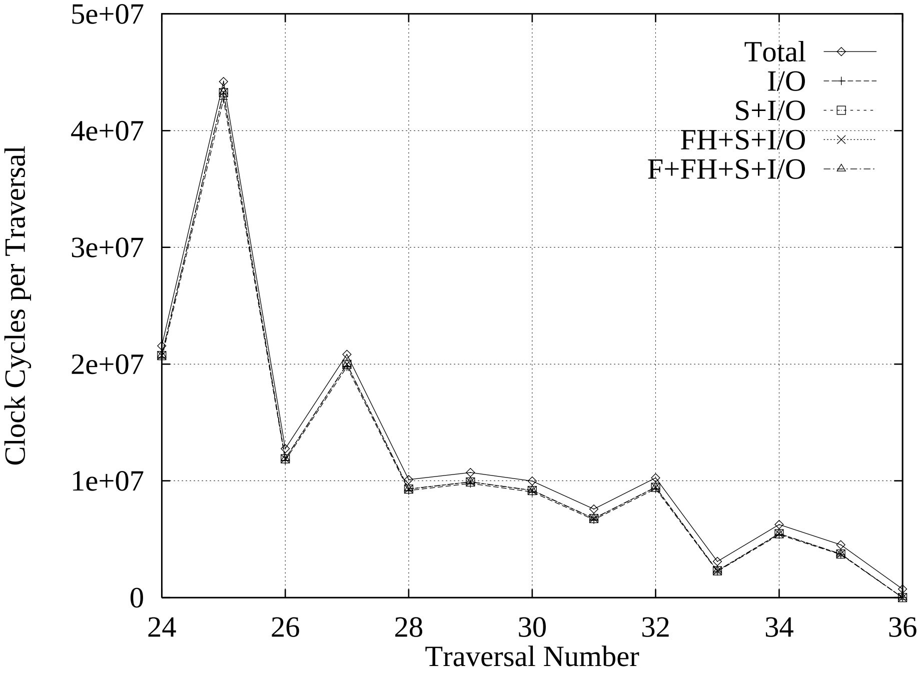


Figure 5.8: Times for traversals 24 through 36, large database (Linux)

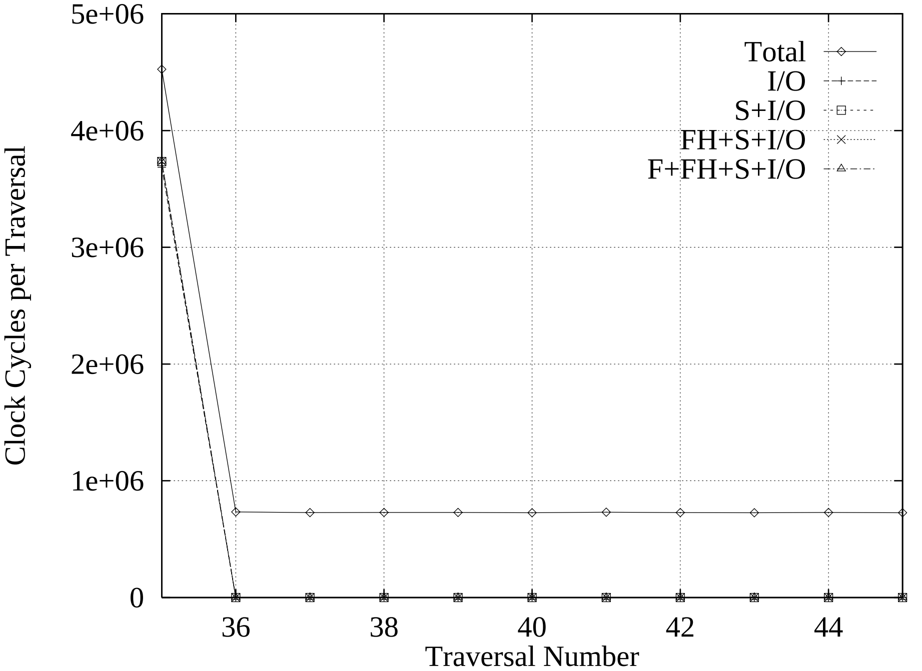


Figure 5.9: Times for traversals 35 through 45, large database (Linux)

From Figures 5.4–5.8 above, we note that the I/O cost dominates for all traversals while the overhead of Texas is comparatively minimal. (Recall that the Texas overhead is essentially the difference between plots labeled "F+FH+S+I/O" and "I/O.") In fact, for most of the later traversals, it is hard to distinguish individual plots corresponding to I/O and other components of Texas from the various figures.

Measuring Real I/O Activity

Based on the above results, it is obvious that the OO1 benchmark traversals on the large database exhibit the characteristics of an I/O-intensive application during the first 35 traversals. We speculated earlier that this behavior occurs because the database is large enough such that the randomized interconnections cause new pages to be referenced (and faulted upon) even during the later warm traversals. An obvious way to confirm this hypothesis is by measuring the number of real I/O requests during each traversal for the entire traversal set. Note that this is different from the number of reads issued by Texas (for loading pages from the database into memory) since not all those read requests translate into real I/O due to file system caching and readahead.

Unfortunately, most operating systems do not provide a convenient way to precisely measure real I/O activity. However, we can count the number of *major page faults* incurred during each traversal to get a good approximation. This is reasonable because a major page fault is any kernel fault that requires a real disk I/O to be serviced. In other words, the kernel has to wait for an I/O request to be satisfied by actually reading from (or writing to) disk, rather than via a file system cache. Major page faults are a good indicator of real I/O activity because all reads and writes through the file system are implemented via internal kernel page faults for most modern Unix variants [Vah96].

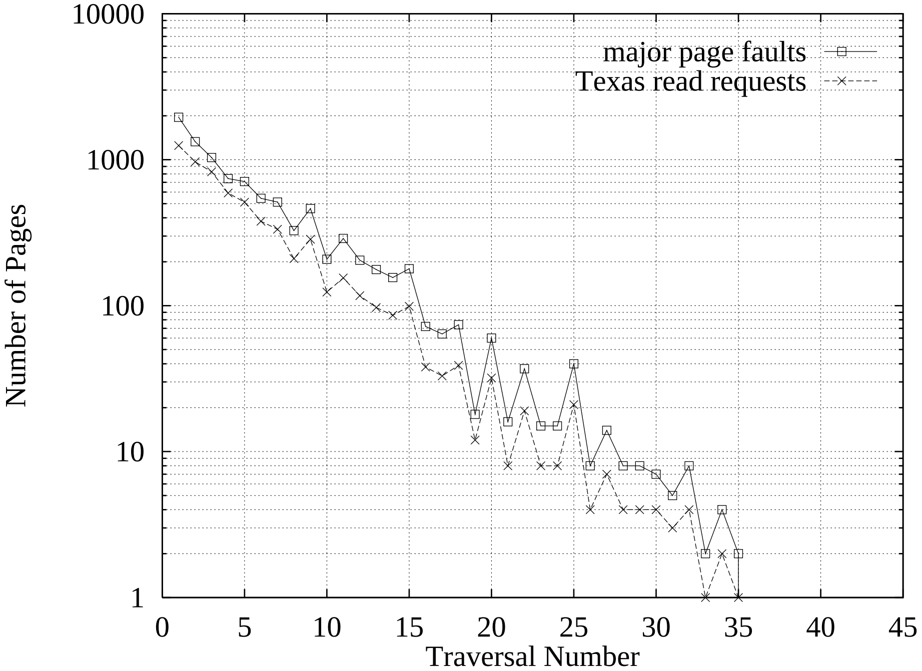


Figure 5.10: Page faults for all traversals, large database (Linux)

Figure 5.10 presents this measure along with the number of reads issued by Texas for all traversals. Note that there is a one-to-one correspondence between the number of reads issued by Texas, the number of protection faults and the number of pages swizzled; this is because, for each fault on a protected page, Texas first issues a read request to load that page from the persistent store and then swizzles it.[9] It is obvious that the number of new pages read into memory gradually decreases as the cache gets warmer. The real I/O activity also decreases proportionately, although it never reaches zero during the warm traversals.[10]

Percentage Overheads

Based on the results presented so far, we can conclude that the direct overhead of pointer swizzling at page fault time (and Texas) is minimal in the presence of I/O, and zero when there is no I/O. Figure 5.11 shows the empirical data that supports this conclusion for large database traversals.

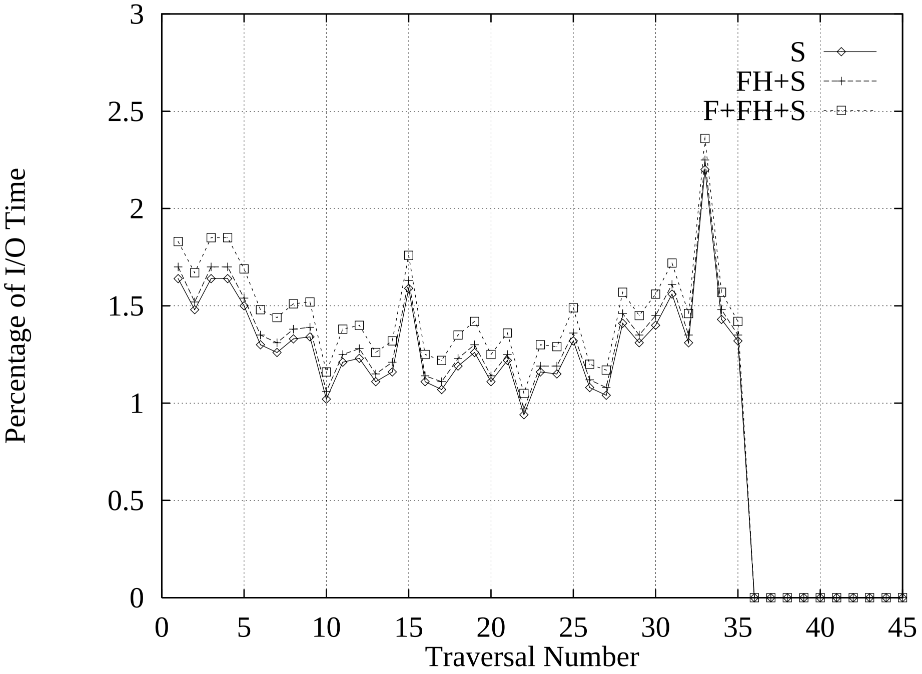


Figure 5.11: Overhead as percentage of I/O time, large database (Linux)

We plot the costs for various components of the pointer swizzling mechanism as a percentage of I/O time for each traversal. The plot labeled “F+FH+S” represents the total overhead of pointer swizzling at page fault time (as implemented in Texas). It is clear from the figure that the average overhead is only around 1.5% of I/O cost while the maximum is just under 2.5%. This is obviously very small compared to the overall I/O costs incurred when running the application.[11]

[9] As such, we use the three phrases, that is, *number of protection faults*, *number of reads issued*, and *number of pages swizzled*, interchangeably depending on the context of the usage.

[10] The number of major faults is always higher than the number of reads issued by Texas in the figure. This may seem unusual, but it is actually okay because major page faults also include I/O for other faulting behavior, such as that required by the memory replacement policy.

[11] Note that this does not include the indirect cost of pointer swizzling related to unnecessary page-outs of mistaken-dirty pages, although it is not an issue in the current experiment because there is no paging, even though we are using the large database.

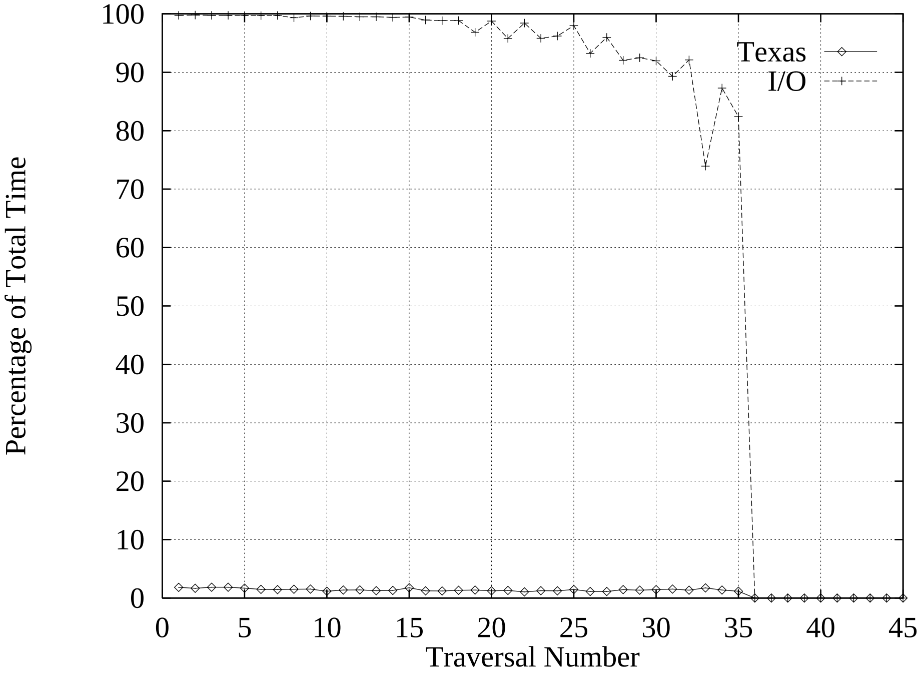


Figure 5.12: Overhead as percentage of total time, large database (Linux)

Another interesting metric is the overhead of Texas as a percentage of the total run time of the actual benchmark traversal, *including* any I/O that may have been necessary for that traversal but *excluding* the costs of Texas itself. This measure gives us an approximation of the overhead that would be imposed on an ordinarily non-persistent application that is modified to use Texas as a persistence layer. Figure 5.12 plots this metric for all traversals on the large database. In addition, the figure also plots I/O time as a percentage of total time for each traversal to determine the fraction of benchmark time typically spent in I/O (i.e., the I/O "overhead"). Once again, we note that the Texas overhead is very low (around 2%) for all cold and warm traversals, reinforcing our earlier conclusions about the performance of pointer swizzling at page fault time. In comparison, I/O cost makes up the majority of the benchmark run time (e.g., 90% or more for 31 out of 35 traversals). As expected, both overheads drop to zero for the hot traversals which, by definition, do not cause any faulting or swizzling.

5.4.2 Small Database Results

The results for the large database have unequivocally shown that pointer swizzling at page fault time techniques do not impose a major overhead on the run time of an application in the presence of I/O activity. We now present results for OO1 benchmark traversals on the small database which highlights some important situations. The database contains 20,000 objects and is small enough (one-tenth the size of the large database, or about 4MB) to easily fit into the main memory. The basic conclusions about performance of Texas are still valid, but there are a few quantitative variations related to interactions between locality characteristics and the operating system.

Basic Performance

Figures 5.13–5.17 present the performance of the entire traversal set on the small database. Figure 5.13 shows overall run time of the entire traversal set (45 traversals) on a *log scale*, and the rest are closeups (on a linear scale) of different traversals.

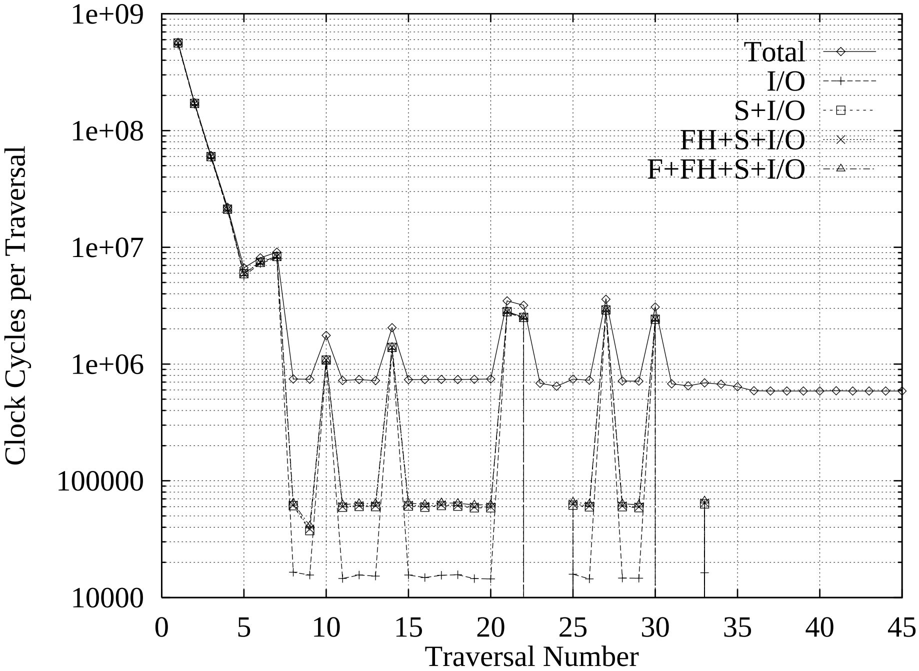


Figure 5.13: Times for all traversals, small database, log scale (Linux)

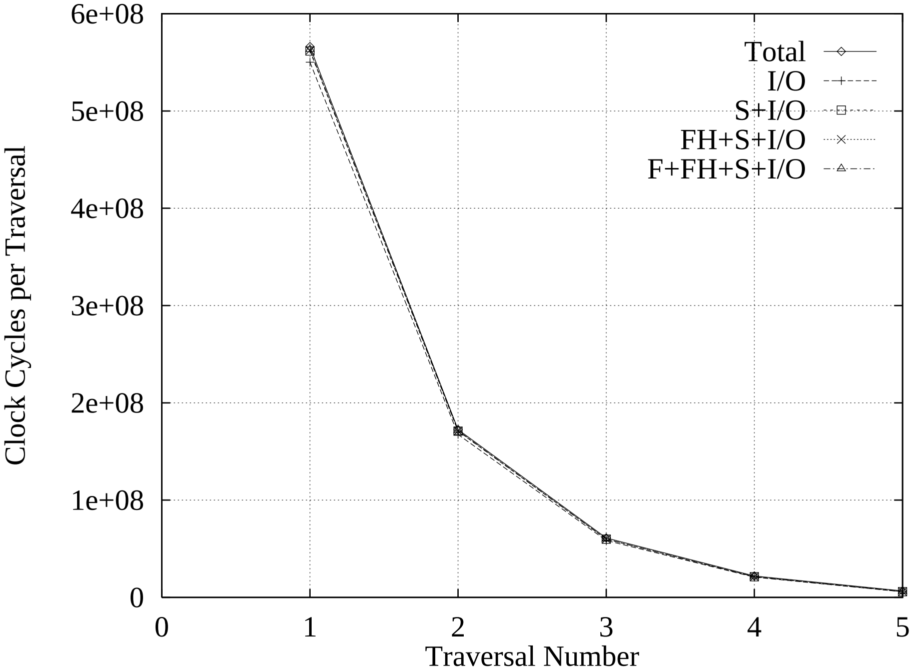


Figure 5.14: Times for traversals 1 through 5, small database (Linux)

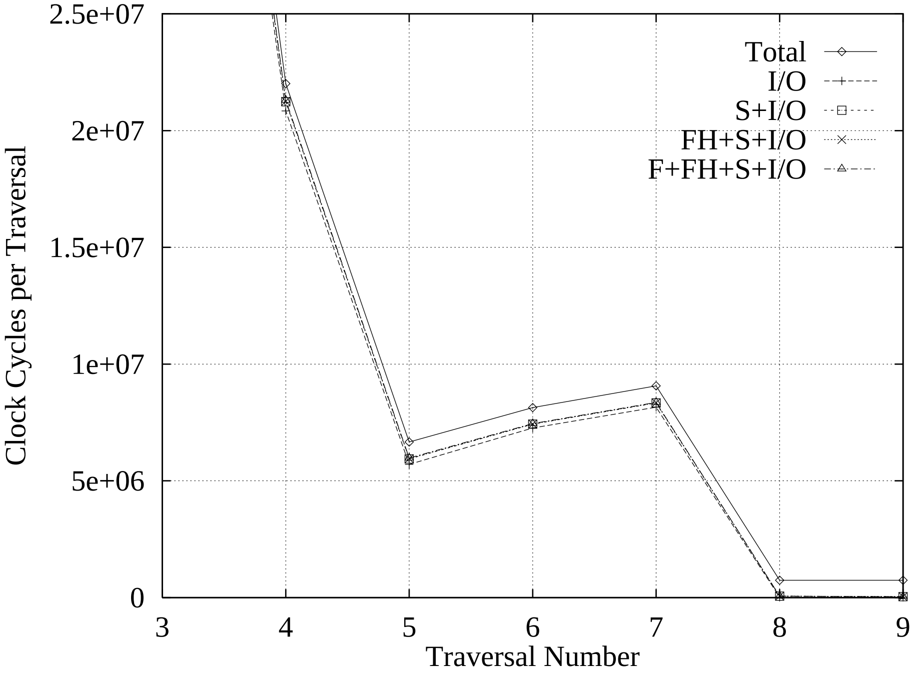


Figure 5.15: Times for traversals 3 through 9, small database (Linux)

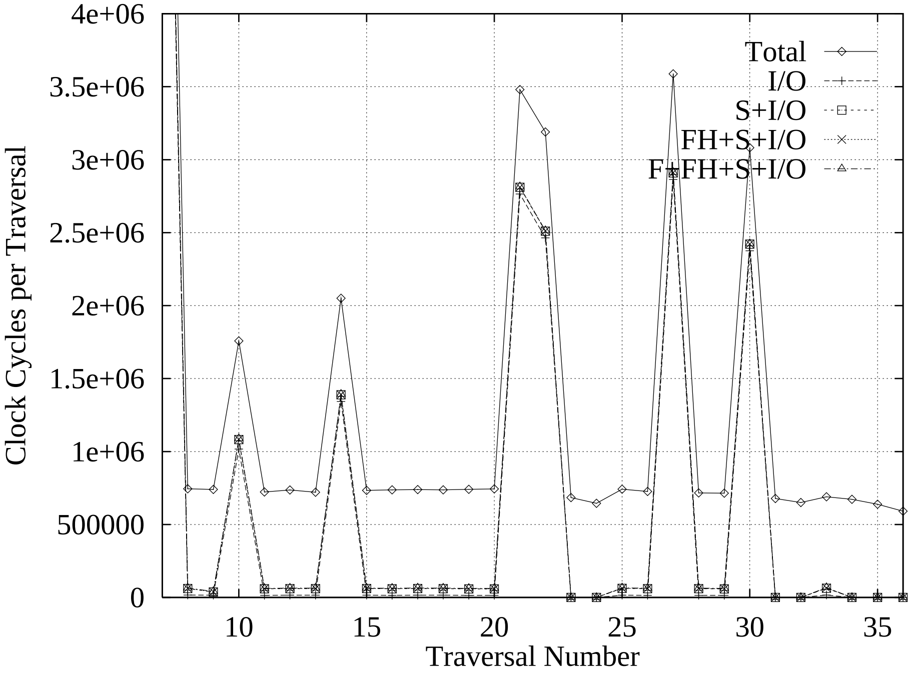


Figure 5.16: Times for traversals 8 through 36, small database (Linux)

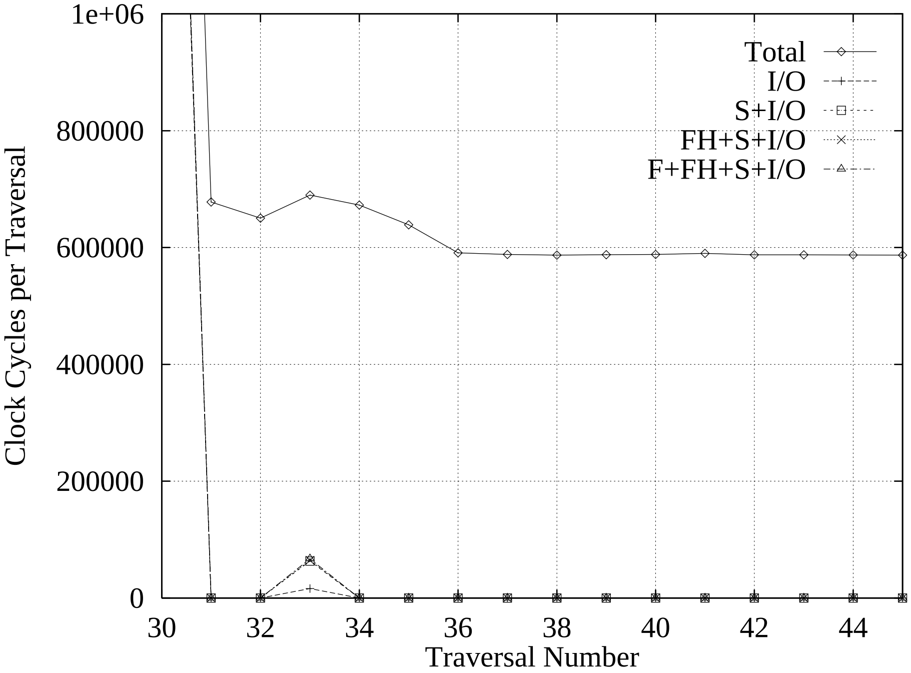


Figure 5.17: Times for traversals 30 through 45, small database (Linux)

Consider Figure 5.13, which plots the performance results of various components for all traversals, and compare with the corresponding plot for the large database (Figure 5.3). The major differences between the two sets of plots are partly due to the size of the database and poor locality in the benchmark traversals, as well as their interaction with file system caching and readahead mechanism of the operating system. However, although the overall plots look very different for traversals on the small database, the overall structure still conforms to the three phases described earlier, and can be divided it into three qualitative regions as follows:

- *I/O-intensive region* consisting of the cold traversal and first few warm traversals (i.e., left end of Figure 5.13 up to and including traversal 7, as well as Figures 5.14 and 5.15), where the Texas overhead is obviously small compared to the I/O cost, which typically dominates the overall run time;

- *CPU-intensive region* consisting of the hot traversals (i.e., right end of Figure 5.13, traversals 36 through 45, as well as Figure 5.17), where Texas overheads are zero and there is no I/O; and

- *mixed-behavior region* consisting of rest of the warm traversals (i.e., middle part of Figure 5.13, traversals 8 through 35, and Figure 5.16), which is a little complicated because the overheads vary significantly with number of faults and I/O requests per traversal.

The first two regions obviously support the conclusions drawn from the large database results, and are not discussed further here. Instead, we focus on the third (mixed-behavior) region which corresponds to the phase that exhibits moderate I/O and faulting activity interspersed with computation. The unusual behavior in this region is related to the fact that the entire database is small enough to fit into memory and most of it is loaded into memory (or prefetched into a file system cache) within the first few traversals because of poor locality in the randomized interconnections. As a result, most warm traversals do not pay the cost of real I/O (i.e., disk seeks) since their requests are likely to be satisfied from the file system cache.

From Figure 5.13 (and also Figure 5.16), we note that for six traversals (specifically traversals 10, 14, 21, 22, 27 and 30) in the mixed-behavior region, the I/O cost is fairly high (between one and three million clock cycles) while the Texas overhead is very small in comparison. This is because the I/O requests for these traversals cannot be satisfied from the file system cache and require real I/O activity. In contrast, sixteen other traversals (specifically traversals 8, 9, 11 through 13, 15 through 20, 25, 26, 28, 29 and 33) intermixed with the six mentioned above incur I/O cost of only about 15,000 cycles, which is equivalent to 75 microseconds on the test machine with a 200MHz clock rate. Given the fact that typical disk latencies are on the order of milliseconds, it is impossible that this number represents the cost of a real I/O request. Instead, it is likely that the requests were satisfied from a (file system) cache without ever involving any moving parts while paying only for software costs. For these traversals, the Texas overhead appears to be significant (about 45,000 clock cycles) since it is not masked by I/O costs. This is not a problem during actual execution because overall performance is typically swamped by the cost of real I/O requests for all other traversals.

Measuring Real I/O Activity

In the foregoing discussion, we have argued that seven traversals in the I/O-intensive region and only six traversals (out of twenty-two that have non-zero I/O costs) in the mixed-behavior region have some real I/O activity. We can confirm this hypothesis by measuring the number of real I/O requests using the major page faults described earlier. Figure 5.18 presents this measure along with the number of reads issued by Texas for the entire traversal set.

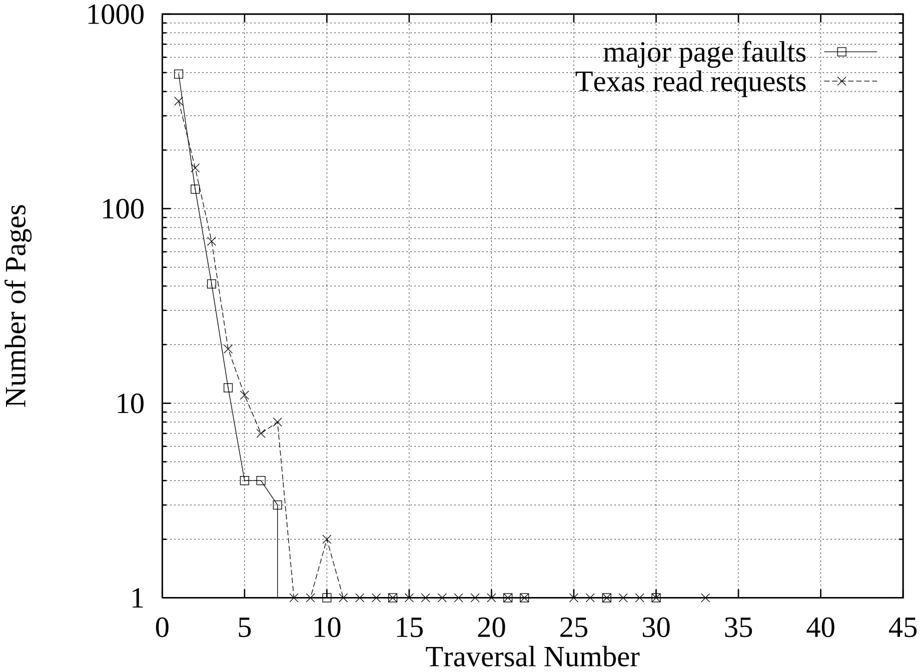


Figure 5.18: Page faults for all traversals, small database (Linux)

Note that while the two plots are visually similar, they do not coincide with each other. That is, the number of real I/O requests is not exactly the same as the number of pages read by Texas due to the operating system readahead mechanism. By correlating the number of real I/O requests (from the figure) with the overhead results in Figure 5.13, we confirm that real I/O activity indeed corresponds to higher I/O costs in overall performance. Conversely, there are no major page faults (i.e., no real I/O) for traversals that exhibit low I/O costs.

Percentage Overheads

Finally, we plot the Texas overhead both as a percentage of I/O time and as a percentage of total benchmark time for each traversal in the traversal set. These plots are shown in Figures 5.19 and 5.20 respectively. The latter also plots the fraction of total benchmark time spent in I/O for each traversal. (The corresponding plots for large database were shown in Figures 5.11 and 5.12 respectively.)

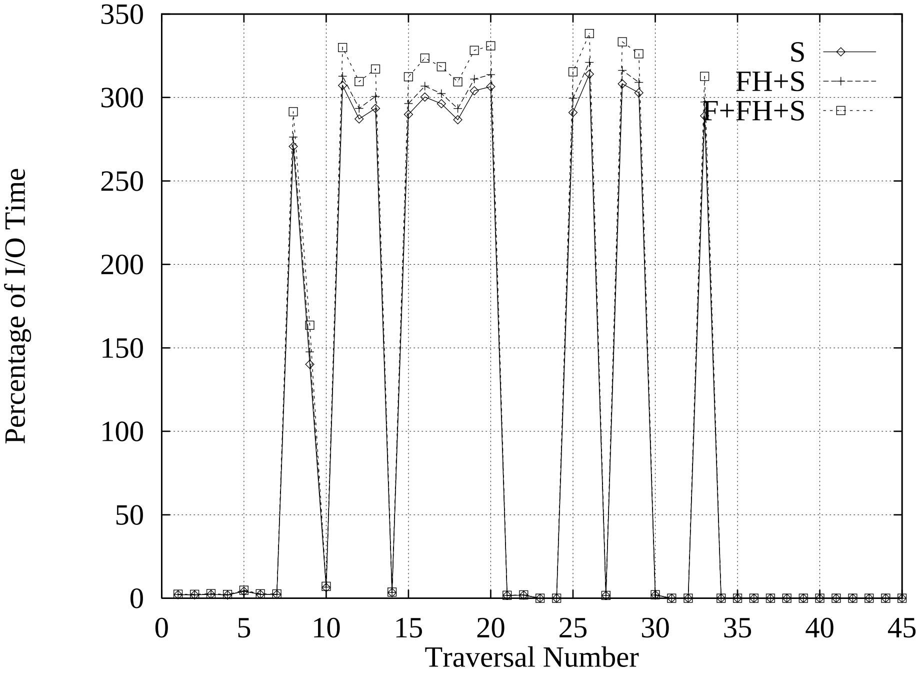


Figure 5.19: Overhead as percentage of I/O time, small database (Linux)

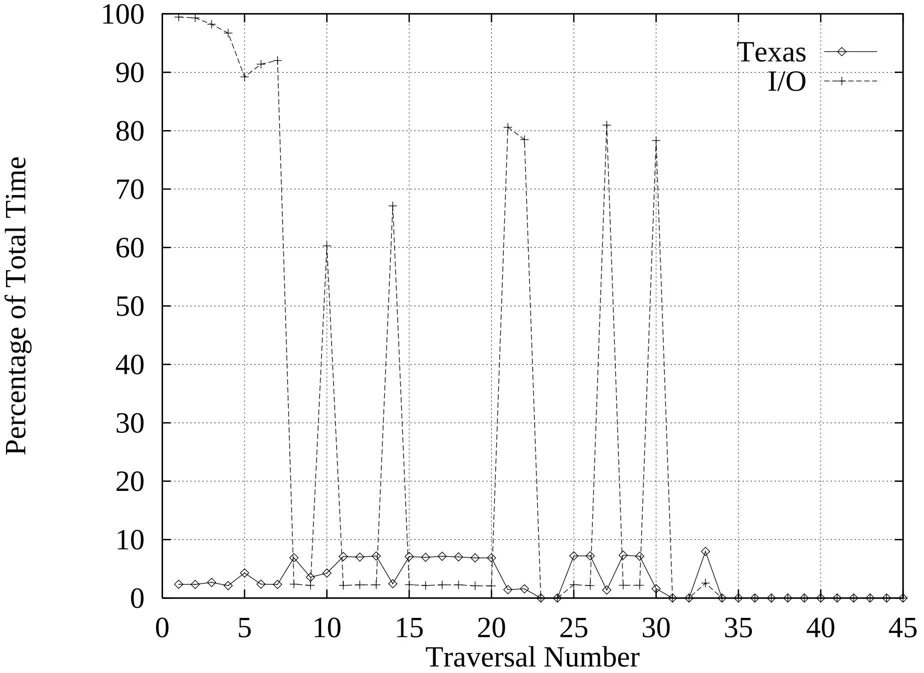


Figure 5.20: Overhead as percentage of total time, small database (Linux)

It may be startling at first to see that the overhead of Texas is around 300% of I/O cost for several warm traversals in Figure 5.19. However, on closer inspection, we note that these traversals are the same as the sixteen traversals for which there is no real I/O activity as per Figure 5.18. As before, correlating the two figures clearly shows that the Texas overhead is small in the presence of real I/O activity, exactly the same conclusion that was derived from the large database results. In terms of actual numbers, from Figure 5.20, we conclude that Texas overhead is around 2-4% of real I/O costs. Finally, we note that although the small database traversals do not always have real I/O activity because of file system caching and readahead, when present, real I/O tends to dominate the overall run time of the benchmark, usually by accounting for 60-80% (or more) of the total time.

5.4.3 Analysis

We have broadly divided the OO1 benchmark traversal results into three qualitative regions, each representing applications with a different mix of I/O and computation phases, and discussed our overhead under various situations. The results for both small and large databases have shown that the overhead of Texas and pointer swizzling at page fault time is minimal in the presence of real I/O, and zero when there is no I/O. Furthermore, although the Texas overhead seems outrageously high in the absence of real I/O (especially for the small database), the overall performance during actual execution is not affected much because the I/O activity, when it does occur, tends to swamp the rest of the costs. In fact, we calculated the cumulative overhead of Texas as a percentage of total benchmark time (including I/O) over *all* cold and warm traversals, and found it to be only about 2% for the small database and slightly more than 1.5% for the large database. Both these numbers are obviously very small compared to the other costs incurred by the benchmark.

The performance results presented for Linux correspond to the use of normal file I/O for loading data from the benchmark database. As described earlier, this means that the operating system usually prefetches more data than requested during a read to minimize overall I/O costs; the prefetched data is stored in a file system cache and is used to satisfy future I/O requests wherever possible. We have clearly seen the effects of this action in the small database results, where the mixed-behavior region contains traversals with I/O costs that are too small to be real I/O.

It is possible to avoid the operating system readahead and caching by using a raw device, instead of a normal file in the file system, for storing the database. Although Linux currently does not support this feature, the heuristics used by the operating system to balance the file system and virtual memory caches seem favorable for our usage patterns because the performance results do not appear to be adversely affected. (As we will see next, the Solaris buffer management policies do not work as favorably for our usage patterns.)

5.5 Performance on Solaris

We ran the same performance experiments using the OO1 benchmark on Solaris to compare and contrast the results with those obtained on Linux. We have found that the basic conclusions derived from the Linux results are still valid for Solaris, although there are a few quantitative variations in the raw data for the latter. In this section, we present the performance results obtained on Solaris, and briefly discuss the factors responsible for the various differences and their impact on the benchmark measurement.

An important difference here is that, unlike Linux, Solaris supports a raw I/O mechanism allowing us to measure the performance of the system in the absence of file system caching and readahead. We present results corresponding to the use of both file I/O and raw I/O,[12] and highlight important differences between the two strategies.

5.5.1 Large Database Results

Following the earlier format, we first present the results obtained for benchmark traversals on the large database. As before, we split the results into three parts: the raw performance data, a measure of real I/O activity, and the overhead of Texas as a percentage of both I/O time and total benchmark time.

Basic Performance

Figure 5.21 shows the overall run time of the entire traversal set (45 traversals) on Solaris. As before, we plot the cumulative costs of different components, starting with the I/O time. Comparing the results to the corresponding plots for Linux (Figure 5.2), we note that the overall cost of various components is qualitatively similar to that on Linux, that is, both sets of plots have similar features.

[12] Unless otherwise specified, all results presented are for normal file I/O.

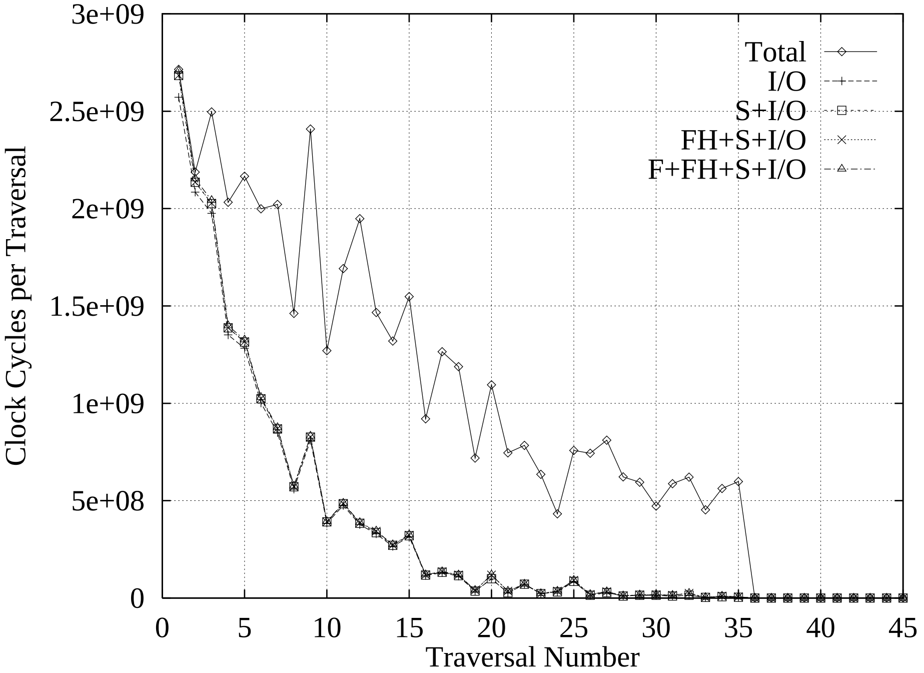


Figure 5.21: Times for all traversals, large database (Solaris)

One obvious difference between Solaris and Linux results is that the total benchmark time on Solaris exhibits unusual spiky behavior for most of the warm traversals. This particular plot represents the total time measured for the entire traversal, including costs of I/O and all pointer swizzling components (see Figure 5.1). It is obvious from the figure that the time for the traversal component of the benchmark (measured by subtracting cost of all other components from the total time) is quite high, varying from two to four times larger than the rest of the costs. This is unusual because the traversal component itself is not CPU-intensive and is not expected to add a big overhead to the overall execution time.

We believe that the unusually high total time for each traversal is actually due to excessive virtual memory paging because the database is much larger than the available main memory and is accessed with poor locality. Based on the randomized interconnections, on average, every tenth pointer visits a randomly-chosen part that is not nearby (and has not been referenced before), causing more data to be faulted in and swizzled. The interaction between these access characteristics and the operating system's buffer management policies indirectly leads to paging because there is insufficient memory available for the virtual memory system; as a result, the benchmark execution spends much time waiting during paging. Since our cycle timer can measure only real (wall-clock) time, the time spent in waiting for I/O during paging is "mistakenly" billed to the traversal component.

Measuring Real I/O Activity

We can confirm our observation about paging by measuring the real I/O requests for each traversal and comparing with the number of read requests issued by Texas. As with Linux results, the number of major page faults is ideal for getting an approximation of the real I/O activity. Figure 5.22 shows this result, along with the number of reads issued by Texas, for all

45 traversals. Also, in Figure 5.23, we plot the time billed *only* to the traversal component, that is, the total time for the entire traversal less the cumulative time for all other components including I/O.[13]

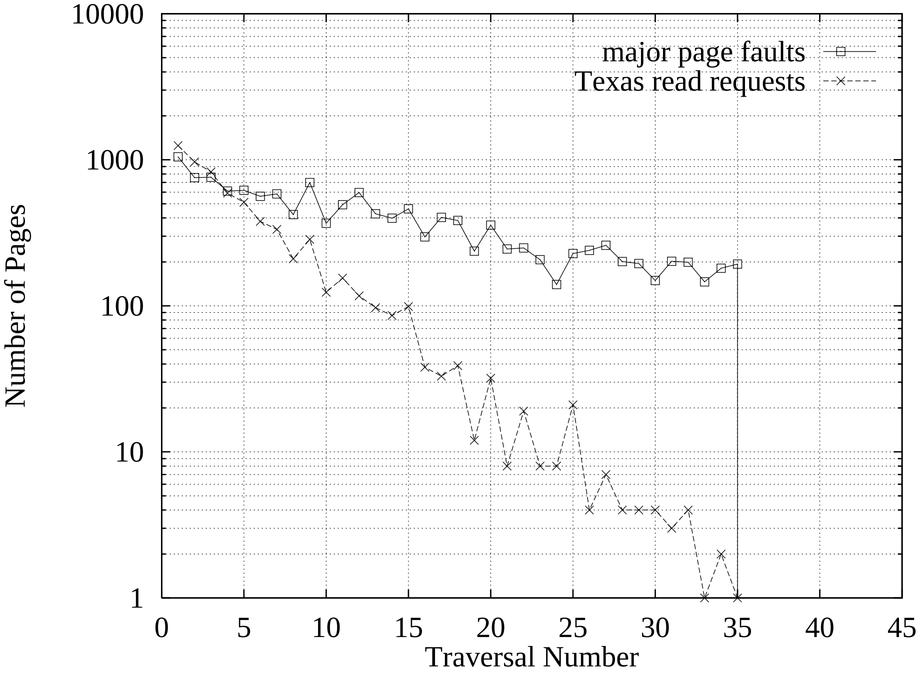


Figure 5.22: Page faults for all traversals, large database (Solaris)

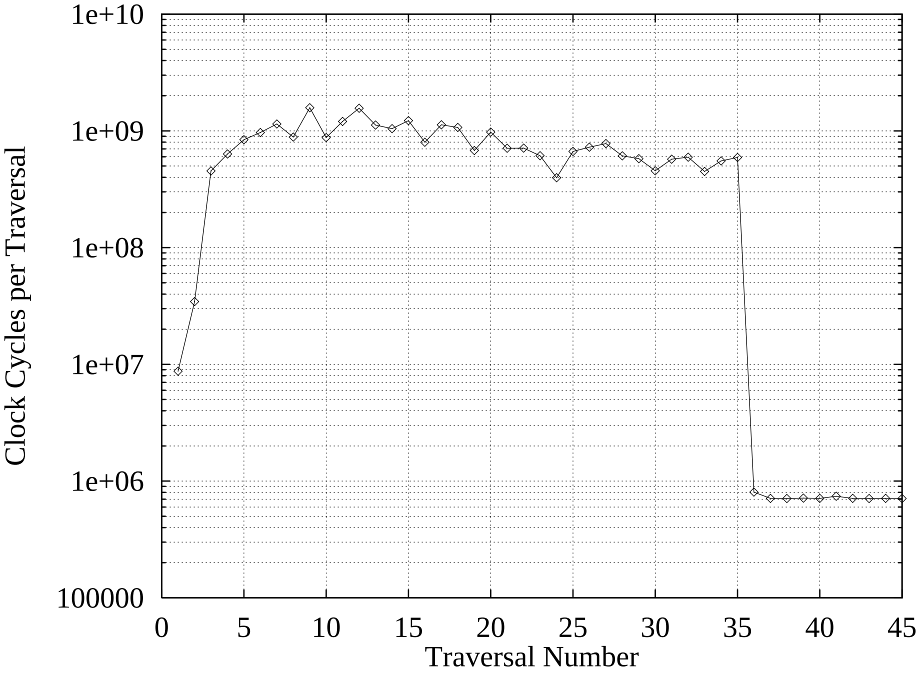


Figure 5.23: Benchmark-only time for all traversals, large database (Solaris)

[13] Note that both figures plot their respective data on a log scale.

The first thing to notice from Figure 5.22 is that the number of page faults remains relatively high throughout the cold and warm traversals rather than gradually decreasing as the cache gets warmer. This is quite different from the downward trend of the number of pages read (and swizzled) by Texas as the cache gets warmer. We also contrast this with the corresponding plots for Linux in Figure 5.10. However, an even more interesting observation is that the plots corresponding to major page faults and to the benchmark-only time exhibit identical visual characteristics for all warm traversals starting after traversal 5. This observation strongly supports our theory about the high traversal time (in Figure 5.21) being due to excessive paging. The number of page faults indicates heavy paging behavior over the benchmark interval, and the visual "tracking" of features between the benchmark-only time and the page faults confirms that overall run time is directly affected by paging.

Percentage Overheads

We also plot Texas overhead as a percentage of I/O cost for each traversal; this is shown in Figure 5.24. As expected, the overhead of different components is very small compared to the I/O cost. There are, however, a few unusual features in the plot. In particular, the overhead of user-level and kernel fault handling (plots labeled "FH+S" and "F+FH+S" respectively) varies significantly for later warm traversals (traversals 26 through 35, except 33) although the overhead of swizzling itself (plot labeled "S") remains low and stable. The reasons for this behavior are unclear, especially because the user-level fault handler has only a few actions and cannot account for such large fraction of I/O. As described in detail later (Section 5.5.6), we believe that this is likely due to interaction between paging and our approach of measuring the overheads with different timers.

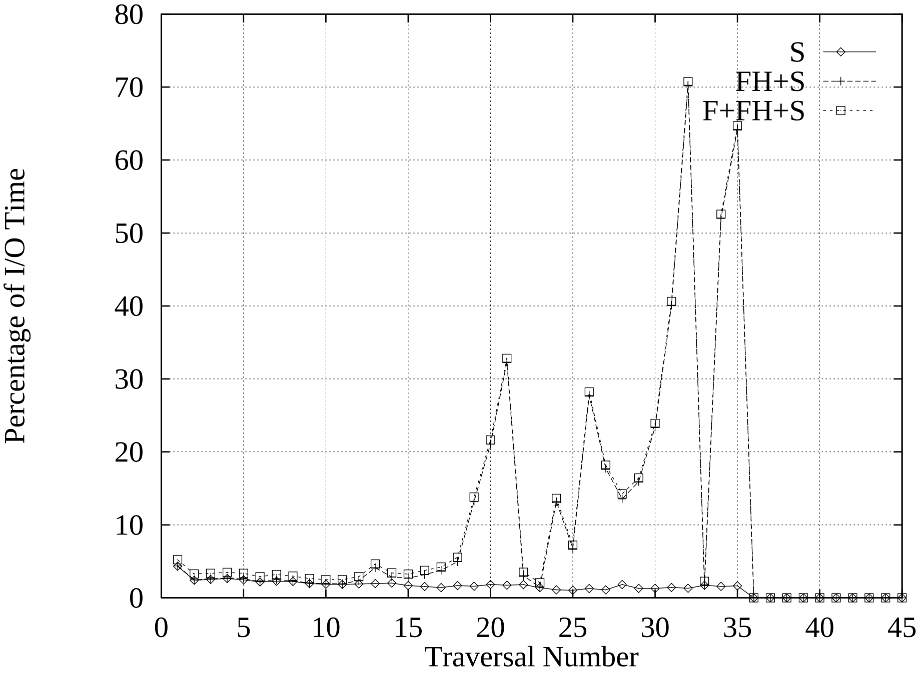


Figure 5.24: Overhead as percentage of I/O time, large database (Solaris)

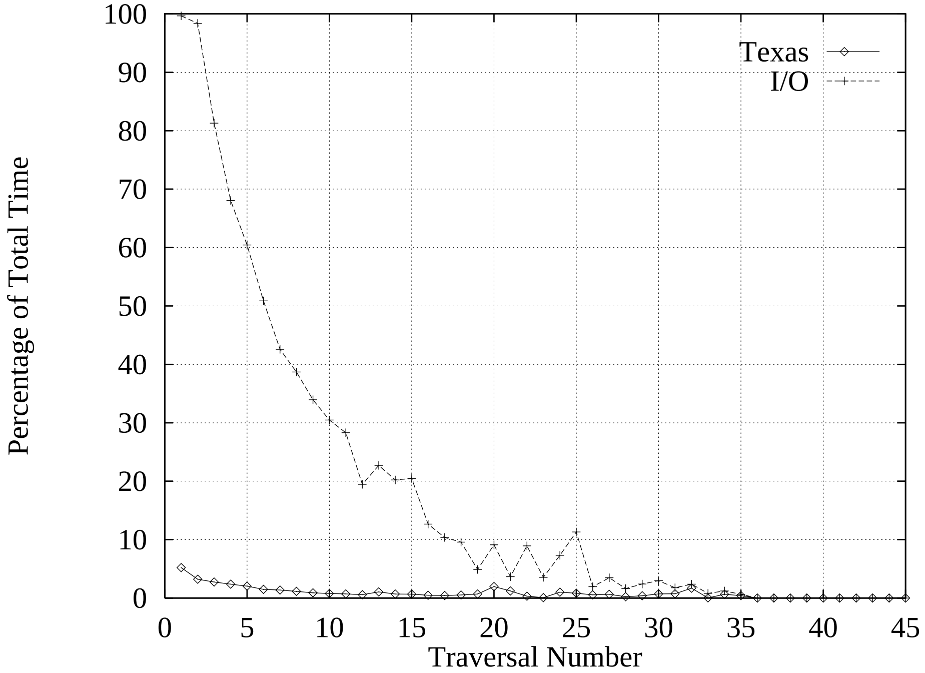


Figure 5.25: Overhead as percentage of total time, large database (Solaris)

Finally, Figure 5.25 shows the overhead of Texas overhead as a percentage of the total benchmark time for all traversals. In the same figure, we also plot the I/O costs as a percentage of the total benchmark time. It is obvious that the I/O time typically makes up a significant portion of the total benchmark run time (especially for the early traversals), and using pointer swizzling at page fault time is relatively inexpensive. Note that I/O costs percentage in this case reduces faster compared to corresponding results on Linux (Figure 5.25) because of the much higher total benchmark time due to paging.

5.5.2 Small Database Results

We have seen that the large database results on Solaris are qualitatively similar to the corresponding results on Linux, although there are a few unusual quantitative variations in the raw data. Nevertheless, the overall performance results support the conclusions derived so far about the low overhead of Texas and pointer swizzling at page fault time. We now present the small database traversal results on Solaris, and show that the basic conclusions are valid even for the small database.

Basic Performance

We start by presenting the performance of the entire traversal set on the small database; this is shown on a log scale in Figure 5.26. Not surprisingly, this looks very similar to the small database results on Linux (presented in Figure 5.13), modulo a few minor details. These variations are due to the differences in the operating system readahead policy and its implementation on both operating systems.

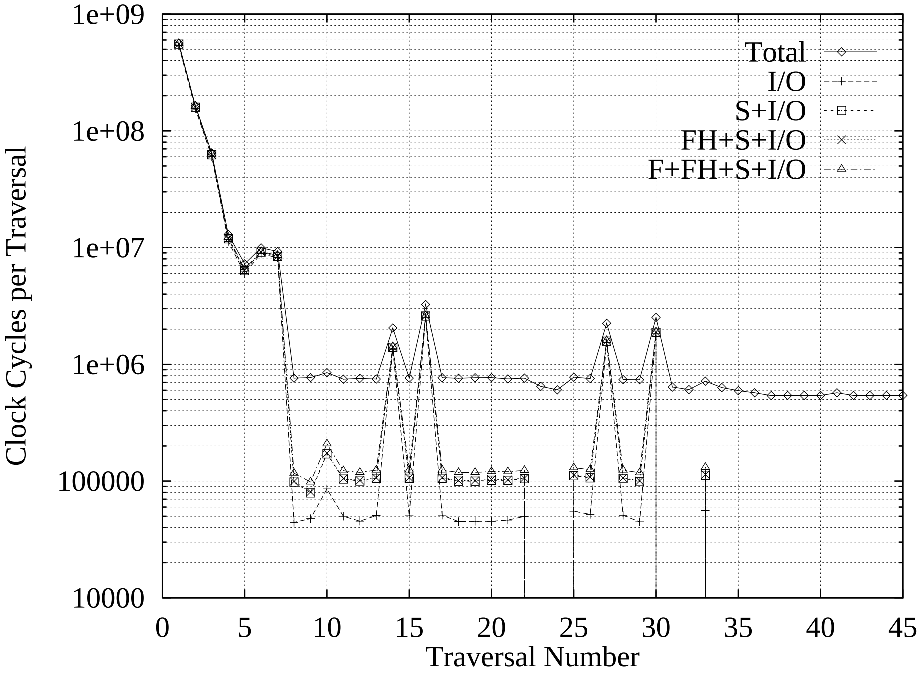


Figure 5.26: Times for all traversals, small database, log scale (Solaris)

Most of the arguments presented for the corresponding results on Linux are also applicable here. As before, we can divide the plot into three qualitative regions, each with different I/O and faulting characteristics. Unlike the Linux results, the mixed-behavior region in the current data contains only four traversals (specifically traversals 14, 16, 27, and 30) that appear to exhibit real I/O activity.

Measuring Real I/O Activity

Figure 5.27 shows the number of reads issued by Texas and the corresponding number of major page faults (i.e., real I/O requests) for all traversals. Compare this with the corresponding plot for Linux (Figure 5.18), and we can draw the same conclusions as before regarding performance of Texas in the presence of real I/O. That is, we can confirm that higher I/O costs (Figure 5.26) are directly related to real I/O activity (as represented by major page faults). Similarly, traversals with low I/O costs correspond to those with no real I/O activity.

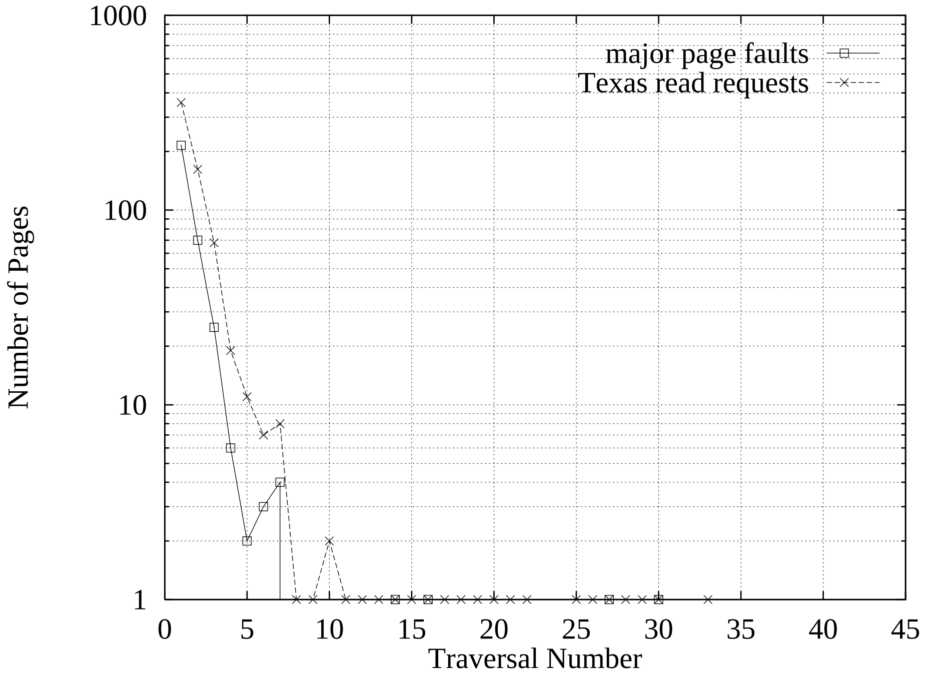


Figure 5.27: Page faults for all traversals, small database (Solaris)

Percentage Overheads

Finally, for completeness, we present results for the Texas overhead plotted both as a percentage of I/O costs and as a percentage of the total benchmark time for the whole traversal set. These plots are shown in Figures 5.28 and 5.29 respectively. The latter also includes a plot of I/O time as a percentage of the total benchmark time for all traversals. The corresponding results for Linux were presented in Figures 5.19 and 5.20 respectively.

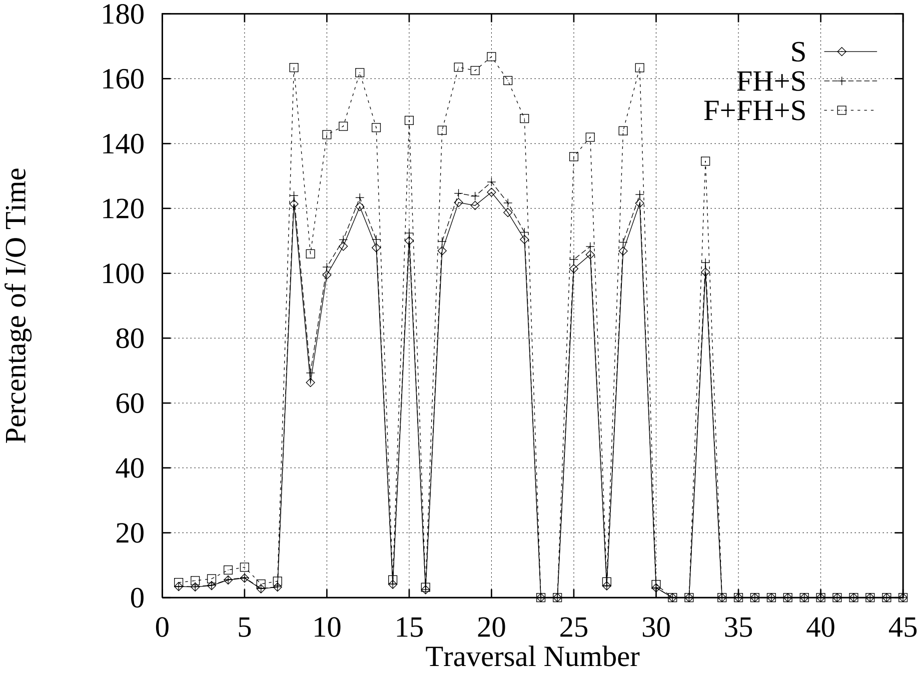


Figure 5.28: Overhead as percentage of I/O time, small database (Solaris)

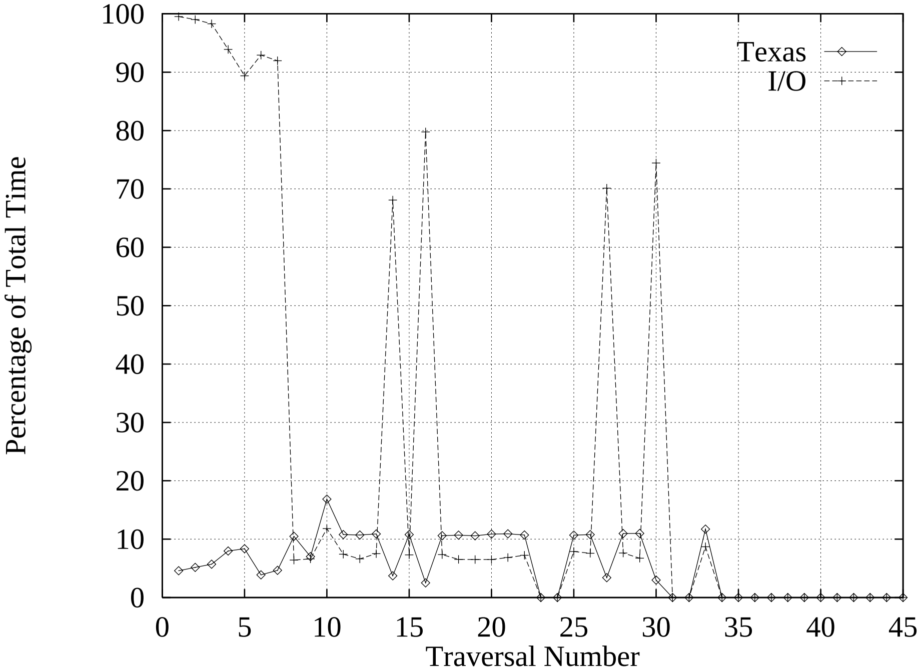


Figure 5.29: Overhead as percentage of total time, small database (Solaris)

Once again, we note that the Texas overhead is consistently more than 100% for all traversals that do not have real I/O associated with them, very small for traversals that have real I/O, and zero for all others when there is no I/O. Although the pointer swizzling overhead seems too high for specific traversals, the overall execution is not affected proportionately because it is usually swamped by real I/O activity for all other traversals. As before, we calculated the cumulative overhead of Texas as a percentage of total benchmark time (including I/O) over all cold and warm traversals, and found it to be only about 5%. Although this is higher than the corresponding overhead measured for Linux (around 2%), it is still very reasonable compared to overheads for some individual traversals (measured at 100% or more).

5.5.3 Large Database Results Using Raw I/O

The large database results presented earlier correspond to the use of file I/O for loading data from the database. Obviously, this includes the effects of the file system caching and readahead mechanism of the operating system. We also ran the OO1 benchmark traversal using raw I/O for loading data into memory. This was achieved by storing the database on a raw disk partition that did not have any associated file system. We present a subset of the results for these experiments and briefly discuss some differences compared to the results for file I/O.

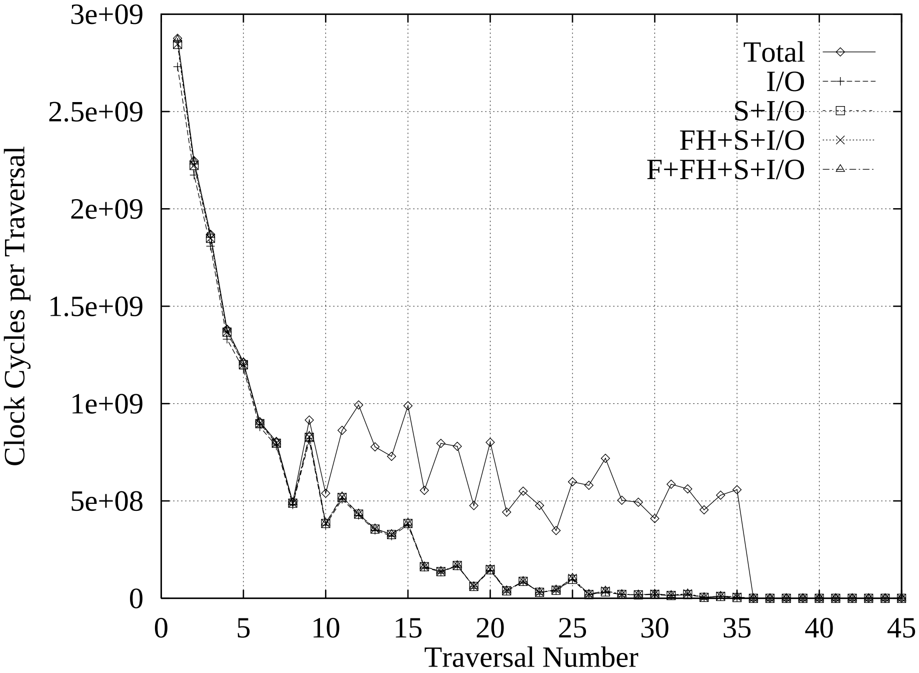


Figure 5.30: Times for all traversals, large database, raw I/O (Solaris)

Figure 5.30 shows the overall run time for the entire traversal set on the large database using raw I/O; the corresponding plot for file I/O was shown in Figure 5.21. From the figure, we see that the overhead of various components of the system is still small compared to I/O costs. Note that the total traversal time for most warm traversals is unusually high and, as before, this can be attributed to paging behavior. However, unlike the earlier results (when using file I/O), paging behavior does not start until traversal 9 when using raw I/O.

Figure 5.31 plots the number of major page faults for the entire traversal set. It is obvious from the figure that there are no major page faults before traversal 9, indicating that paging does not start until then. We also plot the time billed only to the benchmark traversal component in Figure 5.32.

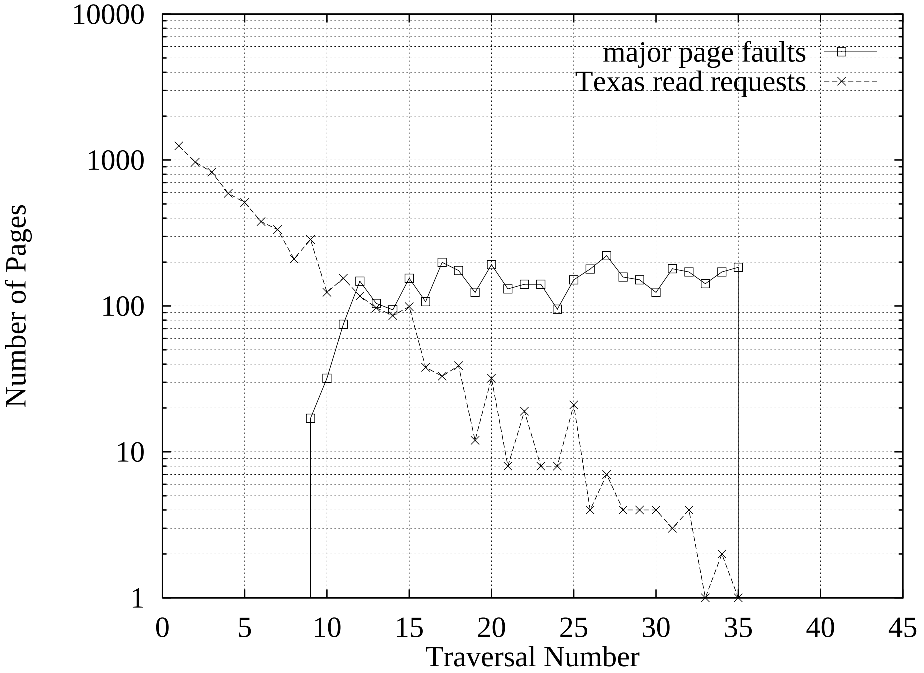


Figure 5.31: Page faults for all traversals, large database, raw I/O (Solaris)

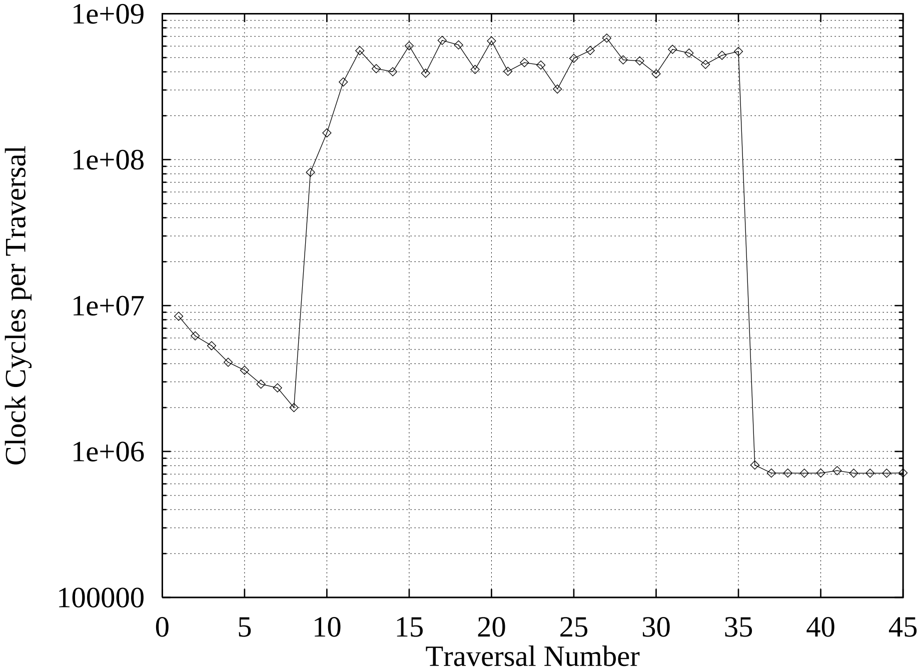


Figure 5.32: Benchmark-only time for all traversals, large database, raw I/O (Solaris)

As before, we match the number of page faults from Figure 5.31 with the benchmark-only time in Figure 5.32 for all traversals. As might be expected, we find a strong correlation between the two such that the run time is directly proportional to the number of page faults incurred starting at traversal 9.

Comparing File I/O and Raw I/O Results

In general, the performance results when using raw I/O correspond to the basic results when using normal file I/O, with some interesting differences in the paging behavior. However, apart from this obvious difference, there are a couple of other issues that should also be highlighted.

First, if we compare the total run-time when using file I/O (Figure 5.21) and when using raw I/O (Figure 5.30), we notice that it is a little lower in the latter case for the first few (about 8) traversals. This is because the former involves additional I/O attributed to file system readahead, thus increasing the overall run time. Another interesting difference is the number of pages involved in the paging behavior for each I/O strategy. When using file I/O, the page faults plot (Figure 5.22) indicates that about 600 pages were involved in paging (at traversal 5) gradually reducing to about 200 by the last warm traversal. In contrast, when using raw I/O, this number always remains between 100 and 200 for all warm traversals. This is because the readahead mechanism in the file I/O case prefetches many additional pages during earlier traversals and stabilizes as the cache gets warmer. The extra pages involved in I/O also increase the total benchmark run time for file I/O.

5.5.4 Small Database Results Using Raw I/O

Since the benchmark traversals have a poor locality of reference, the file system caching and readahead mechanism of the operating system cause most of the small database to be either loaded (or prefetched into file system buffers) from disk within the first few traversals. As described earlier, file system caching can be avoided by using raw I/O for loading data from the database. We now present results for the OO1 benchmark traversals on the small database using raw I/O for all database access.

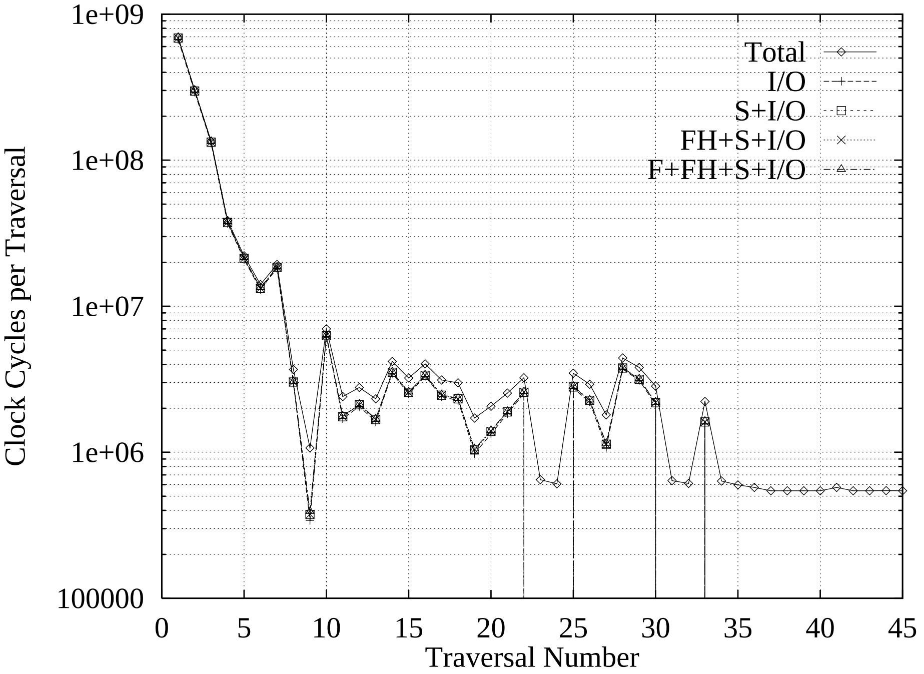


Figure 5.33: Times for all traversals, small database, raw I/O, log scale (Solaris)

Following the format used so far, we present the overall performance results for the entire traversal set; this is shown on a log scale in Figure 5.33. We notice several differences between this figure and the corresponding results for normal file I/O in Figure 5.26. The part of the results that is interesting for this comparison is the mixed-behavior region (among the three qualitative regions described earlier) that comprises of traversals 8 through 35. Most notably, the I/O time for all traversals (with the exception of traversal 9) in this region is between one and six million clock cycles, that is, between 5 and 30 milliseconds,[14] compared to a very low number of 45,000 clock cycles observed earlier. The higher I/O time for the raw I/O configuration is more in line with the expectation of real I/O activity for each traversal.

By enforcing real I/O for each traversal, we have essentially transformed the small database traversal benchmark into an I/O-intensive benchmark, much like the large database traversals. Therefore, the performance results presented in Figure 5.33 are *qualitatively* similar to the large database results; the Texas overhead in both situations is minor compared to the I/O costs incurred during each traversal in the traversal set. Of course, in the current configuration, we do not see the paging behavior that normally occurs during traversals on the large database because the small database easily fits into memory.

In order to get a qualitative approximation of the various overheads of pointer swizzling at page fault time, we plot the overhead of each individual component as a percentage of I/O time for all traversals in the entire traversal set; the corresponding results are plotted in a cumulative fashion in Figure 5.34. We also plot the overhead of Texas as a percentage of total benchmark time in Figure 5.35. As before, this figure includes the fraction of total benchmark time spent in I/O, that is, the I/O "overhead," for the purpose of comparison.

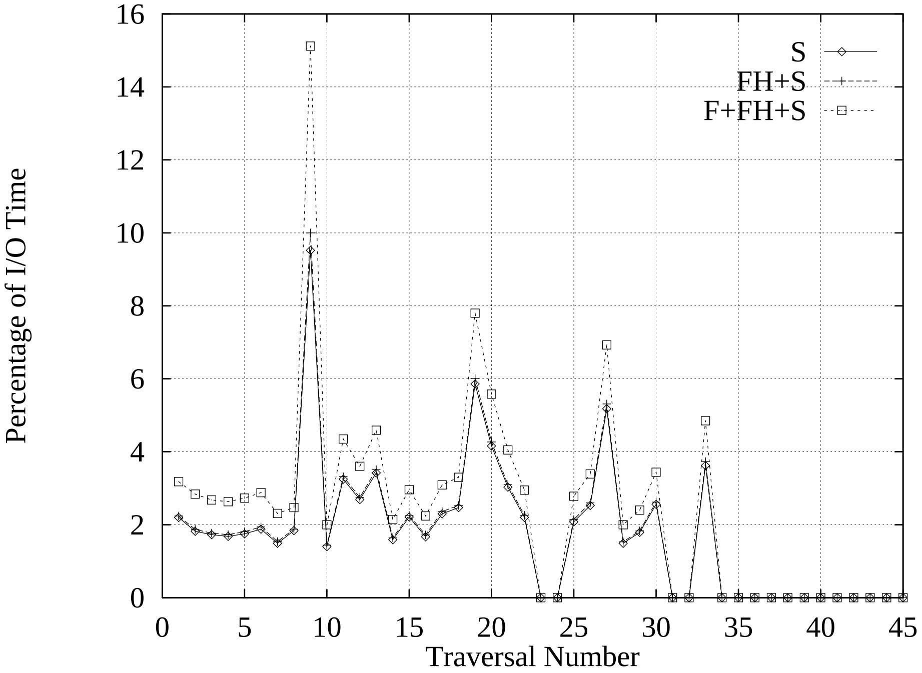


Figure 5.34: Overhead as percentage of I/O time, small database, raw I/O (Solaris)

[14] The I/O time for traversal 9 is about half as much as the lowest I/O time for any other traversal, although it is still on the order of milliseconds. It is possible that this is because the request is satisfied from a "cache" (e.g., track buffer) on the disk itself.

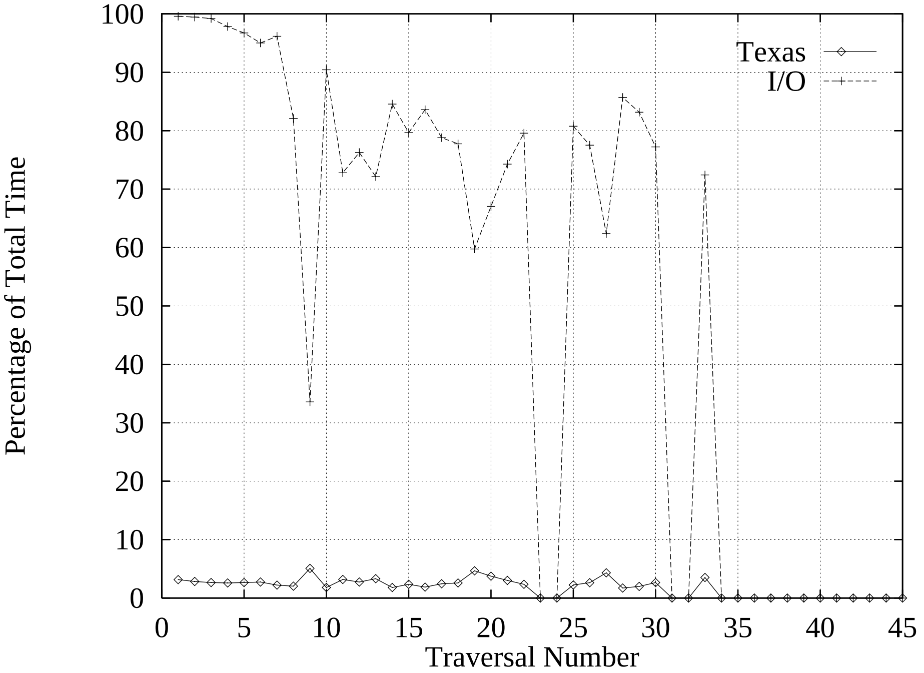


Figure 5.35: Overhead as percentage of total time, small database, raw I/O (Solaris)

From the above figures, we can conclude that the Texas overhead is a very small fraction of the I/O cost, usually about 3-5% on average while the maximum is about 8% (excluding traversal 9). Although the average overhead is more than that on Linux, it is not outrageously high enough to significantly affect the overall performance. In comparison, the I/O cost is a major fraction of the benchmark run time and more or less dominates all other costs. The cumulative overhead of Texas (calculated separately) as a percentage of total benchmark time (including I/O) over all cold and warm traversals is about 3%. This is slightly less than the 5% overhead reported for normal file I/O-based traversals because raw I/O is a little more expensive due to the cost of extra disk seeks.

5.5.5 Large Database Results with Bigger Memory Size

In both file I/O-based and raw I/O-based results for the large database presented above, the total benchmark run time has shown some unusual behavior (marked by spikes in the plots). Even after avoiding the effects of caching and readahead by using raw I/O, we were able to delay the spiky behavior by only a few traversals. We have speculated that this is due to excessive paging that occurs because the database is larger than the memory size and is typically accessed with poor locality of reference. To verify this hypothesis, we upgraded the memory in the test machine from 32MB to 64MB, and reran the benchmark traversal with the new configuration.[15] Since the new memory size is big enough to easily fit the entire large database, we do not expect any paging; instead, the general behavior should be very similar to the small database results.

[15] Although we used raw I/O for the new configuration, using file I/O should be acceptable because 64MB is large enough to avoid unwanted contention between file system and virtual memory caches.

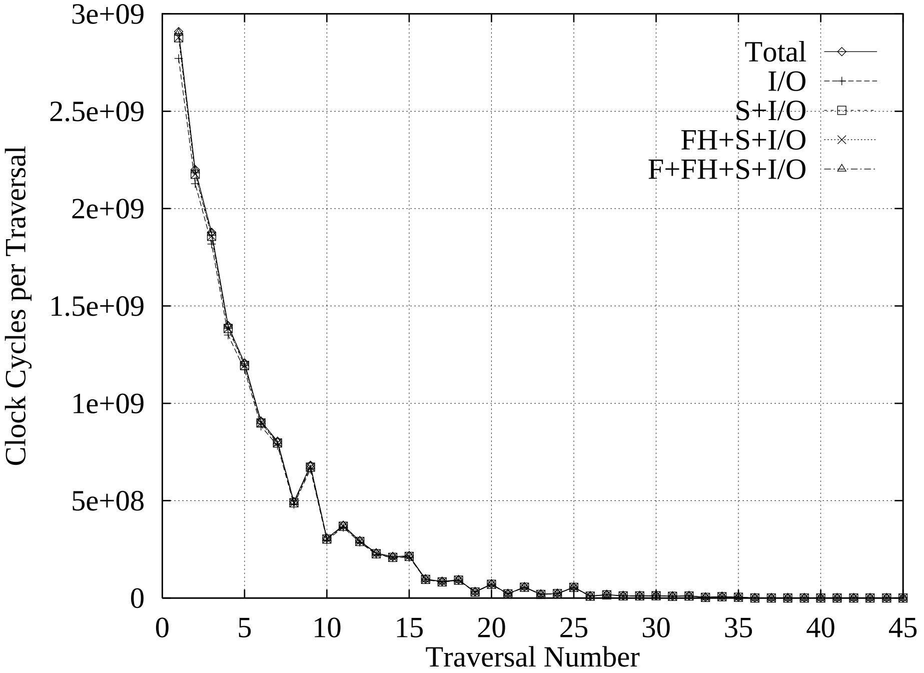


Figure 5.36: Times for all traversals, large database (Solaris, large memory)

Figure 5.36 presents the results for large database traversals on the new (large memory) configuration. There are at least two important features that should be noted here. First, as expected, the unusually high run time is no longer present and we can safely conclude that there is indeed no paging. An even more interesting observation, however, is the strong similarity between this figure and the large database results for Linux presented in Figure 5.2. The similarity is especially interesting because the Linux results correspond to the use of file I/O for database access while the Solaris results are for raw I/O.

We also plot the overhead of Texas both as a percentage of I/O time and as a percentage of total benchmark time. Figure 5.37 shows the overheads of different components of Texas as percentage of I/O time for each traversal. It is obvious that the overhead of Texas (the plot labeled “F+FH+S”) is a small percentage of I/O time, about 2.5% on average, for most warm traversals on the large database when there is little or no paging because the database fits into memory.

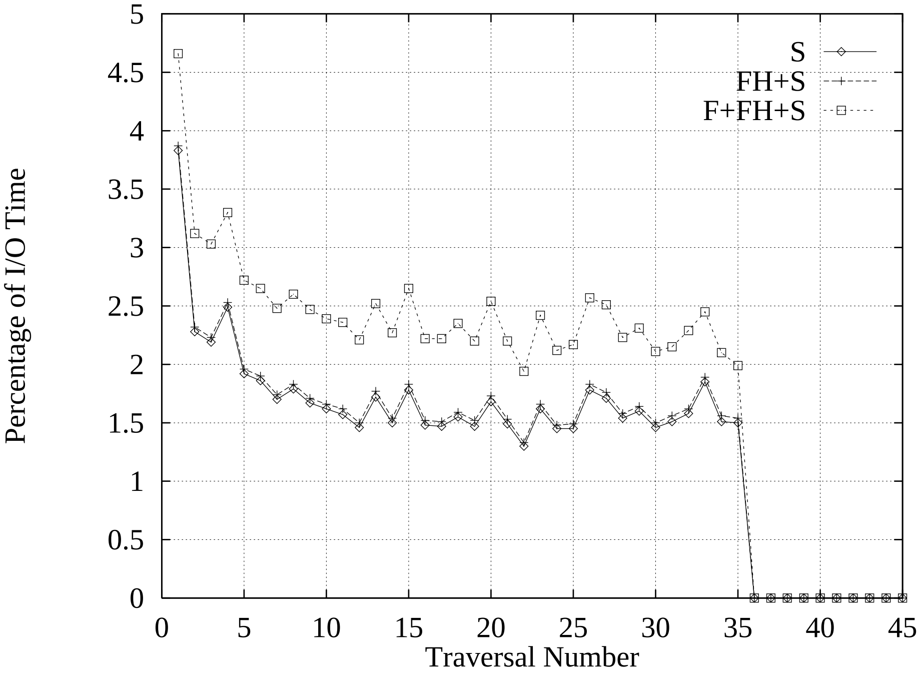


Figure 5.37: Overhead as percentage of I/O time, large database (Solaris, large memory)

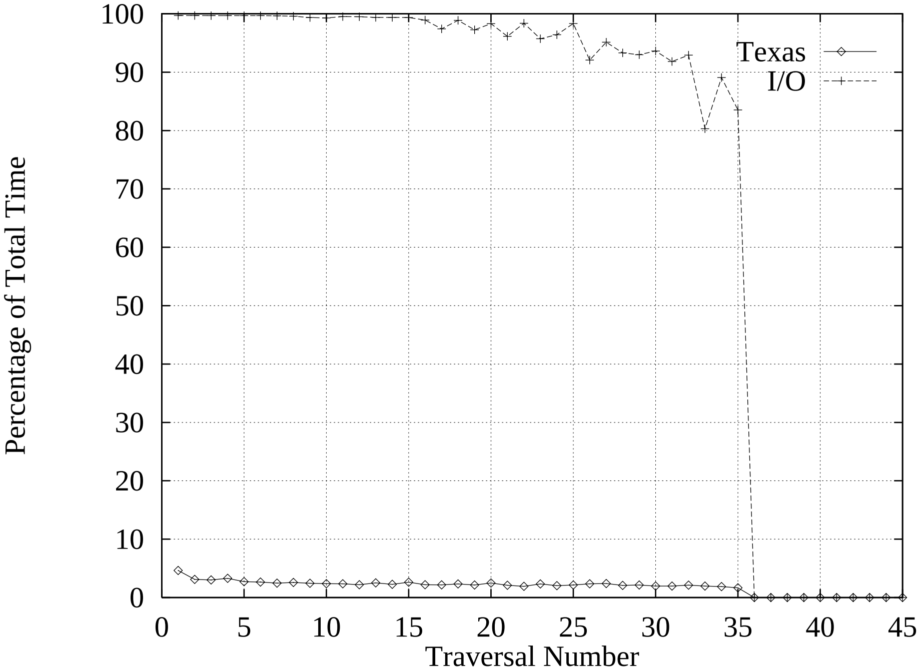


Figure 5.38: Overhead as percentage of total time, large database (Solaris, large memory)

Finally, Figure 5.38 shows the overhead of Texas as a percentage of total benchmark time. We note that the overhead is much smaller compared to the fraction of total time spent in I/O. Once again, note the remarkable similarity between these results and the corresponding Linux results (in Figure 5.12). In terms of raw numbers, the Texas overhead is slightly higher on Solaris (around 2.5%) than on Linux (around 1.5%).

5.5.6 Analysis

For the performance measurements on Solaris, we ran the benchmark for two database sizes using both file I/O and raw I/O, and presented the corresponding results for each database size. In general, the results confirm our basic assertion that the overhead added by Texas is relatively small compared to typical I/O costs incurred for loading the data into memory.

Analysis of Large Database Results

The results for both I/O strategies confirm our assertion that Texas overhead is relatively small compared to I/O costs. There are, however, a few quantitative variations that seem unusual. For example, the plot in Figure 5.24 shows that the actual swizzling overhead itself is relatively small for all warm traversals, but there are a few unexpected spikes in the fault handling (both user-level and kernel-level) costs. We believe that these are actually transient effects, possibly due to paging, rather than a reflection on the faulting overhead.

There are several reasons that led us to this conclusion. First, we have observed that these spikes "shift" to other traversals for different runs of the same experiment with no discernible pattern in the variation, except that they mostly occur in the later warm traversals. In addition, the height of the spikes also varied for different runs, further indicating that the overhead is transient in nature. Finally, based on the number of pages swizzled for each traversal (Figure 5.22), we know that there are only a few (usually less than ten) protection faults in the later warm traversals and it is not possible to have such large overheads for handling just a few faults. Instead, we believe that the measurements are affected by spurious effects of paging behavior and therefore incorrectly billed to faulting.

We have also shown that the effects of paging behavior disappear as we move to a bigger memory size and the database fits into memory. In general, the performance of Texas on Solaris (using raw I/O with larger memory to avoid paging) is very similar to that on Linux (using file I/O with smaller memory). This is further indication that buffer management on Linux is more aggressive than on Solaris.

Raw I/O vs. File I/O

The main difference between raw I/O and file I/O strategies is the file system caching and readahead that is implicitly performed by the underlying operating system in the latter case. We can bypass this behavior simply by storing the data in a raw disk partition and using raw I/O for all data access. Turning off file system caching is usually a good idea for most applications running under Texas because the problem of double caching is avoided.[16] However, as illustrated below, it may or may not be beneficial to turn off caching depending on the application characteristics.

For example, consider the small database results for both file I/O and raw I/O cases (Figures 5.26 and 5.33 respectively), and compare the absolute I/O costs for the initial I/O-intensive region (traversals 1 through 7) for both sets of results. Specifically, I/O times for traversals that use file I/O are uniformly lower than those that use raw I/O. This is because

[16] Of course, if client-side (local) caching is important (e.g., multiple distinct runs of one or more applications on the same input data), then file system caching may be beneficial.

every I/O request in the latter case results in an actual disk I/O request and therefore incurs the cost of extra disk seeks. In contrast, since normal file I/O prefetches data during real disk I/O, it avoids the cost of extra disk seeks corresponding to prefetched pages for which the readahead pays off sooner.

Now consider the absolute I/O times for large database results using the two different I/O strategies (Figures 5.21 and 5.30). In this case, the I/O times for raw I/O-based traversals are lower than those for file I/O after the first two traversals. This is because the file system prefetches are not paying off soon enough and the competition between file system cache and virtual memory cache reduces the effective memory size causing paging behavior which, in turn, increases the number of I/O requests, quickly wiping out any advantages gained due to prefetching.

As such, turning off the readahead mechanism may have unexpected effects on the performance of various applications depending on their access characteristics and the amount of data accessed. It may be beneficial to perform readahead for many applications because their access characteristics are favorable to prefetching and can effectively avoid the extra disk seeks. However, others may do well to not pay the cost of readahead because they are unlikely to benefit from it. Unfortunately, most operating systems currently do not provide independent user-level control over readahead and file system caching other than support for raw I/O. Of course, we can implement our own prefetching, and possibly "preswizzling," using only raw I/O. This avoids unnecessary interference between file system and virtual memory caches, and affords more control over the readahead mechanism.

5.6 Comparison of Address Translation Granularities

The results presented so far have concentrated on the overall performance of pointer swizzling at page fault time, a coarse-grained address translation mechanism. We now present results of the OO1 benchmark traversals corresponding to different address translation granularities for the data structures used during the traversals. Recall that we use the smart pointer idiom (Chapter 3) for implementing fine-grained and mixed-granularity address translation without requiring any additional support from the compiler or the operating system.

In particular, we are interested in the three different address translation granularities, namely *coarse-grained*, *mixed-granularity* and *fine-grained* strategies, that were described in Chapter 3. The following table describes the types of pointers used for each granularity and the corresponding key for the performance results plots.

Granularity	Type(s) of pointers	Key
coarse-grained	All "raw" (language-supported) pointers	all-raw
mixed-granularity	"Smart" pointers for index, raw otherwise	smart-index
fine-grained	All smart pointers	all-smart

Unlike the performance results presented in Sections 5.4 and 5.5, when presenting results for address translation granularities, it is important to use CPU time rather than real time because the difference in performance is primarily due to differences in faulting and swizzling, and allocating address space for reserved pages. Unfortunately, as discussed in

Section 5.2.2, CPU-time timers on most operating systems have a very coarse granularity, and it would be impossible to measure any reasonable differences in the performance due to a change in the address translation granularity because our overheads are very small. Hence we use an older (and slower) SPARCstation ELC, which is slow enough to offset the coarse granularity of the timers.

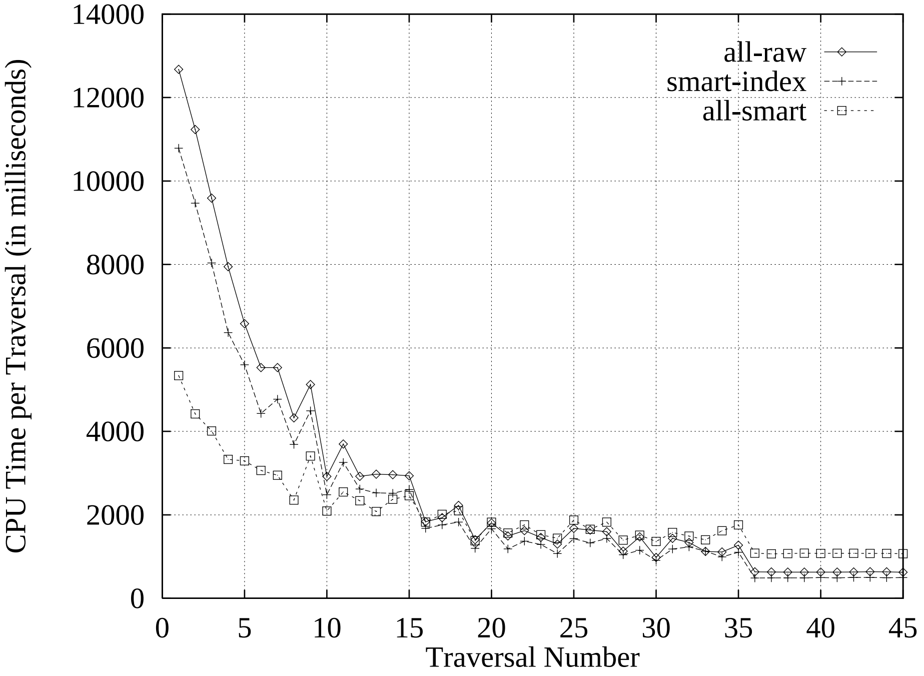


Figure 5.39: CPU time for translation granularities, large database (Solaris, SPARC ELC)

Figure 5.39 presents the CPU time for all traversals in an entire traversal set run on a large database. As expected, the cost for coarse-grained address translation (the "all-raw" case) is the highest for the first 15 or so traversals. This is not unusual because the coarse-grained address translation scheme swizzles all pointers in the faulted-on pages and reserves many pages that may never be used by the application. This is exacerbated by the poor locality of reference in the benchmark traversals because many pages of the database are accessed during the initial traversals, causing a large number of pages to be reserved. The number of new pages swizzled decreases as the cache warms up, and we see the corresponding reduction in the CPU time.

We also note that the cost for fine-grained address translation (the "all-smart" case) is the lowest for the first 15 traversals. Again, this is expected because the address translation scheme does not swizzle any pointers in a page when it is faulted in because they are all smart pointers that must be translated at every use. Finally, the CPU time for mixed-granularity address translation (the "smart-index" case) falls between the other two cases for the first 15 traversals. This is also reasonable because only the parts index structure contains smart pointers, and each traversal uses this index only once (to select the root part for the traversal). This cost is only slightly less than the "all-raw" case because our B+ tree implementation generated a tree that was only three levels deep, reducing the number of smart pointers that had to be translated for each traversal.

Now consider the hot traversals. The first thing to note is that the CPU time for the "all-smart" case is higher than that for the other two cases. This is because smart pointers impose a continual overhead for each pointer dereference, that is, the cost is incurred even if the target object is resident. In contrast, the "all-raw" case has zero overhead for hot traversals. (We expect the "smart-index" results to be identical to the "all-raw" results because there is no index lookup during hot traversals, and no smart pointers need to be translated. We attribute the small difference between the hot results in these two cases to caching effects.)

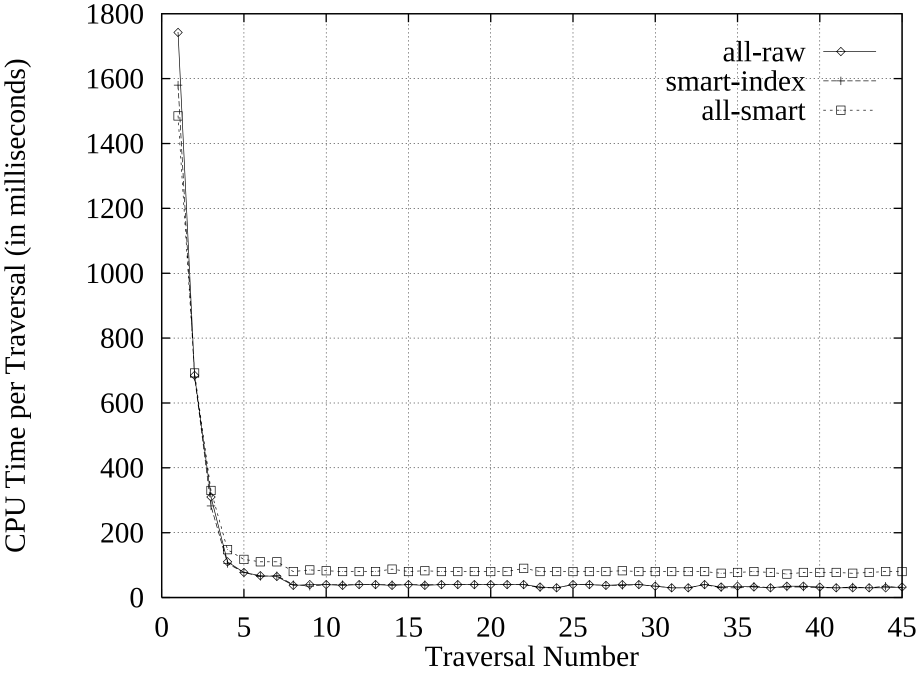


Figure 5.40: CPU time for translation granularities, small database (Solaris, SPARC ELC)

Figure 5.40 shows the corresponding results for the small database. In this case, only the first few (3 or 4) traversals contain faulting and swizzling. Once again, a phenomenon similar to the one in large database results can be seen in the current results. In particular, the CPU time is highest for the first few traversals of the "all-raw" case and lowest for the "all-smart" case. However, for the hot traversals, the two granularities swap their positions; the "all-smart" case is more expensive because of the continual translation overhead, while the "all-raw" and "smart-index" results are identical for hot traversals because no index pointers are dereferenced.

5.7 Discussion

We have presented results for the performance of pointer swizzling at page fault time as implemented in the Texas persistent store. We used the standard OO1 benchmark traversals for measuring the cost of various components of our system, and compared them with I/O costs. We measured the performance of benchmark traversals for both small and large database sizes. All results were presented for benchmark runs on two popular operating systems, Linux and

Solaris. For Solaris, we also presented results using raw I/O (instead of file I/O) for database access to study the effects of file system caching and readahead on the overall performance.

We have seen that the empirical results are qualitatively similar across Linux and Solaris, but there are some quantitative variations in the raw data depending on the combination of specific experiment and the operating system. However, these variations are not significant overall and do not affect the results adversely. In this section, we present our basic argument for the performance of Texas, and discuss the impact of operating system implementations.

5.7.1 Basic Argument

We have shown that pointer swizzling at page fault time imposes absolutely *no overhead* if the data has already been loaded into memory, and minimal overhead when faulting (i.e., in the presence of I/O activity). The basic idea exploits locality of reference in the application to amortize the cost of swizzling at page fault time. Since processors are much faster than disks, the cost of swizzling entire pages at fault time is very small compared to the I/O costs. By using existing virtual memory hardware, we do not incur any other overheads for compiler or operating system support.

The various results presented earlier correspond to the benchmark database stored on a local disk. However, the database may be stored in the main memory of a remote host, and faulted in over a fast network without involving any disk access. An example of this is a distributed system where the data is stored on a centralized data server and fetched over the network by different clients. Since networks are much faster than disks, network I/O costs are likely to be much smaller than disk I/O costs. As a result, the overheads of pointer swizzling at page fault time cannot "hide" as well in the I/O costs. However, there are at least two counterarguments for this. First, very fast networks are still in the experimental research state, and are not likely to be widely available in the very near future, and second, as fast networks become ubiquitous, we expect corresponding improvement both in hardware and operating systems, which will combine to reduce our overheads as well.

5.7.2 Impact of Operating System Implementations

We have run the same benchmark experiments on both Linux and Solaris, using identical underlying hardware setup. Although the overall results conformed in qualitative terms, there were some unusual quantitative variations, especially for Solaris. In particular, Linux appears to be more aggressive than Solaris in terms of the kernel memory usage. Although both test machines had the same amount of RAM (i.e., 32MB), the amount of main memory available to user applications on Solaris was a few megabytes less than that on Linux (approximately 27MB vs. 30MB). Normally, such a minor difference should not have any significant impact on the overall performance. However, for the large database results, the split happens to be "just right" in terms of total data faulted into memory from the database. As a result, we see some paging behavior on Solaris, while the few extra megabytes of memory (combined with aggressive buffer management) is sufficient to avoid any unnecessary paging on Linux. Of course, depending on its buffer management algorithms, the Solaris performance may be suboptimal even with a slightly larger memory size.

Another interesting difference between Solaris and Linux is the cost of handling a protection fault. Specifically, we focus on the time taken from the point when a protected page is accessed by an application till the point where a user-level fault handler gains control. We found that this number is several times larger on Solaris than on Linux. The higher faulting cost obviously affects the performance of Texas, especially during the cold and early warm traversals where many protection faults are generated (and handled). Further details about exception handling, including some measurements on fault handling costs, are provided in Chapter 7.

5.7.3 Indirect Costs of Pointer Swizzling

We do not separately measure or account for the indirect cost of swizzling in our performance measurements. There are at least two reasons for this. First, benchmark traversals for small and large database sizes correspond to situations with no paging and heavy paging respectively.[17] Recall that the indirect costs of pointer swizzling are not a major issue in either of these two situations. Another important reason is that the indirect costs are affected heavily by the locality characteristics of the application. As is obvious from some of the performance results presented so far, the OO1 benchmark exhibits extremely poor locality characteristics in its data structures, and consequently the traversal operations. As such, any measurements of indirect costs using the OO1 benchmark are likely to be artificially skewed and not very useful for drawing any meaningful conclusions.

5.8 Benchmarking Limitations

Over the last few years, several new systems for implementing persistence and full-fledged object-oriented database capabilities have been proposed, implemented, and studied by academic research groups and commercial database vendors. During this period, several benchmarks have also been developed for measuring the performance of these new systems. Of these, the OO1 [CS92] and OO7 [CDN93] benchmarks have become the most popular, and have been used widely for measuring the performance of various systems in isolation, as well as for comparing two or more systems.

The canonical application domain for object-oriented databases (OODBs) is the Computer Aided Design (CAD) domain. We posit that the OO1 and OO7 benchmarks are *not representative* of typical applications in the CAD domain. As such, the results from these benchmarks are not necessarily indicative of typical CAD application behavior. In this section, we discuss some issues about benchmarking methodology, and present our views on the limitations of synthetic benchmarks, particularly the OO1 and OO7 benchmarks. We are especially interested in their use as object database benchmarks, and as benchmarks for measuring orthogonally persistent systems. As discussed in the rest of this section, we believe that the benchmarks are unsuitable for both these roles.

[17] Actually, large database results for Linux appear to correspond to the no paging situation.

5.8.1 Synthetic Benchmarks

Most performance measurements and analysis of persistent object systems (and OODB systems) have been done using *synthetic benchmarks* in lieu of using real applications. There are two reasons for this: first, there are few large, realistic applications that exercise all persistence mechanisms of the underlying system and of those that exist, few are available for general use; and second, it is typically extremely hard to adapt a large piece of code to any given persistence mechanism without having a detailed understanding of the application.

Synthetic benchmarks, however, provide a useful solution to these problems. Usually, the benchmarks are much smaller than any real applications and are (hopefully) designed to be ported to different systems without requiring large modifications. The underlying assumption is that the benchmarks are designed to model behavior of real applications and as such the results from the benchmark studies can be extrapolated for a wide variety of applications. However, this is not always true, and results from synthetic benchmarks must be interpreted with extreme caution.

Often, the synthetic benchmarks reflect their designers' intuitions about program behavior, and these intuitions may not be exactly right. Worse, the benchmarks may implicitly incorporate unrealistic assumptions about underlying common analytic models. The apparently "empirical" nature of these "experimental" results is likely to lull people into relying on the results more than appropriate. A benchmark may resemble real applications in certain ways that are relevant to certain aspects of system design, but in other ways, synthetic benchmarks indicate very little about the behavior of real applications. Even when a benchmark has been validated with respect to certain issues, it may be quite inappropriate for any other purpose for which it has not been validated.

This is not to say that synthetic benchmarks are never useful. In fact, synthetic benchmarks often have the advantage that they can be varied systematically by using a few parameters, which allows for experimentation with a range of possible behaviors. Of course, the results should be interpreted cautiously to ensure that the conclusions drawn from those results are valid for real application behaviors.

The OO1 and OO7 Benchmarks. The OO1 benchmark [CS92] was one of the first widely-used benchmarks for performance measurements of OODBs, designed to model applications in the engineering CAD domain. The benchmark database schema is very simple and is based on a network of biased random interconnections of *part objects*, which are manipulated using some simple benchmark operations. The OO7 benchmark [CDN93] was developed at the University of Wisconsin as a successor to the OO1 benchmark. While the benchmark retains the CAD application model, the data structures are enhanced to add much more hierarchy and additional complexity is incorporated for tuning various benchmark parameters. OO7 has been widely used by OODB developers to measure the performance of their systems, and by researchers to benchmark and compare performance of various persistence mechanisms. However, to the best of our knowledge, OO7 has not been validated against real applications from the CAD domain to ensure that it indeed represents a realistic workload. We believe that OO7 is not representative of typical CAD applications; other researchers [TNL95] have also reached similar conclusions.

5.8.2 Common Problems with the OO1 and OO7 Benchmarks

We are primarily interested in the behavior of the OO1 and OO7 benchmarks, specifically for performance measurements in an orthogonally persistent systems such as ours. In particular, we believe that these benchmarks typically measure the overall I/O performance instead of measuring the costs of address translation and orthogonal persistence, which is our primary focus.[18] This is obviously a problem because OODBs and persistent programming languages are usually intended to be used for CPU-intensive applications. For applications that exhibit I/O-intensive behavior, it may be preferable to use traditional relational databases which are optimized to improve I/O performance.

For the purpose of this discussion, we define two terms, normal program behavior and database program behavior, to categorize behaviors of different applications. *Normal program behavior* denotes typical CPU-intensive applications that spend most of their execution time performing some computation, and use the persistence mechanism only to save their final results. For such applications, a majority of objects need not to be saved to stable storage because they constitute transient data that does not live for very long [Wil97, WJNB95, Joh97]. In such situations, execution costs are dominated by operations over transient data; the persistence mechanism must not interfere with these operations, making them as fast as possible. In contrast, *database program behavior* denotes applications that are usually I/O-intensive and do not perform significant computation during their execution. Traditional relational database systems are better suited for stable storage management in such applications. Most OODB systems and persistent programming languages are (and should be) targeted towards applications that exhibit normal program behavior, enabling the persistence technology to be incorporated into normal applications that operate primarily on in-memory data.

Below, we discuss several common problems that we have identified with both the benchmarks; although we focus mostly on the OO1 benchmark, many of the same issues are applicable to the OO7 benchmark also. While the following is not meant to be an exhaustive compilation of the problems, we believe these issues are the most important especially in the context of a coarse-grained persistent system.

Separation of I/O Costs

The benchmarks do not specify any way to separate I/O costs from other costs, including those that are necessitated by architectural choices in the implementation of the persistent system. For example, if a particular system aggressively prefetches data, the majority of the persistent store will be loaded into memory during early traversals, and the later warm traversals will be closer to hot traversals. In contrast, if the system prefetches little or not at all, the warm traversals are comparatively "cooler." Thus the results for warm traversals mostly represent the loading and caching costs, rather than the overhead of the persistence mechanism itself. That is, they are a measure of the "warmth" of warm traversals rather than fundamental costs of the architecture.

[18] Recall that the costs of orthogonal persistence include the costs incurred when *not using* the persistence facilities—for example, when accessing transient data or persistent data that has already been loaded into memory.

The designers of OO7 appear to have recognized this issue, and specify only "cold" and "hot" performance measurements. On the other hand, OO1 requires that results be reported only for cold and warm performance, omitting the hot performance measurements. This is a problem because the hot performance essentially represents the baseline and, without this information, it is not possible to judge the performance of a given persistent system, regardless of how well caching is working. This is particularly important for our approach which incurs absolutely no overhead during hot traversals when accessing both in-memory persistent pointers and transient pointers. As a result, our hot performance is equal to the best-case performance in a fully-transient configuration. In other words, the cost of orthogonal persistence is zero in our system.[19] However, this is not evident unless the hot performance is also reported as part of the benchmark results.

Locality of Reference

Another issue with these benchmarks is regarding the locality of reference. Although the randomized interconnection scheme in OO1 exhibits some locality—90% of the connections are local—it actually has disastrous effects on locality of simple algorithms operating over the data. On average, *every tenth pointer references a randomly-chosen object.* Because of this, the OO1 benchmark has extraordinarily poor locality of reference. The cold and hot performance represents two extremes of behavior—very bad locality and very good locality—which can be used to roughly assess the overall performance of a system under two very different kinds of use. Unfortunately, there is no guidance in terms of the "expected" mix of these behaviors.

The unusually poor locality may not matter for some purposes, but may be crucial for other purposes. For example, fine-grained systems incur several instructions of overhead at *every pointer dereference* (and perhaps at every pointer comparison), while the coarse-grained schemes (such as ours) incur thousands of instructions of overhead *only at page faults*, and *zero otherwise.* If the frequency of pointer traversals is several orders of magnitude higher than the frequency of page faults, coarse-grained techniques will obviously be more efficient. This is almost always true for most normal applications, which achieve good CPU utilization, usually greater than 50%. (Recall that, on a modern processor, a program that incurs a fault every million instructions is probably paging heavily.) For object databases, this is less clear. Most relational databases are designed for applications that tend to be I/O-intensive, but object-oriented databases are likely to be used for CPU-intensive tasks such as CAD applications. The lack of locality in OO1 raises questions regarding its appropriateness as a general benchmark for arbitrary systems.

The only other useful information in the graph of part objects is the connectivity, which is based on a biased random distribution of interconnections. This ensures that there will be a strong correlation between the static structure of the graph and the dynamic locality of the benchmark traversals. Furthermore, it means that the locality characteristics are likely to be *consistent*, especially with respect to relative heat of links. If an object or link is hot during one traversal, it is very likely to be hot during any other traversal that encounters it at all.

While the OO7 benchmark was designed to support better locality characteristics, it is

[19] This is unlike other (fine-grained) systems that usually incur costs *at every pointer dereference*, including both in-memory persistent pointers and transient pointers.

not clear whether this goal has been achieved. We are not aware of any studies that measures this factor; the closest appears to be one by Tiwary *et al.* [TNL95], and their conclusions indicate that OO7 is unsuitable as a generic CAD workload. Our own preliminary analysis of OO7 (not presented in this dissertation) has shown that the OO7 database connectivity exhibits poor locality of reference characteristics.

Computation Behavior

Both OO1 and OO7 benchmarks specify minimal computation behavior when a persistent object is visited during a benchmark operations (e.g., a traversal). In OO1, as each new object is visited, an empty procedure is invoked on that object to represent the "computation" performed by a real application. The direct effect of this is that the benchmark operations are data- or I/O-intensive rather than CPU-intensive. This is not representative of most real applications, which usually do much more "work" on their data compared to just invoking an empty procedure. OO7 assumes a uniform workload behavior, which is unrealistic because real application usually exhibits different phases during a single execution.

The I/O-intensive nature of benchmark traversals makes coarse-grained address translation techniques look unnecessarily bad because their basic premise—locality of reference in data access and page faults interspersed with long periods of computation—is violated. In addition, each persistent pointer is dereferenced only once during a traversal. This is unrealistic for CAD applications; for example, Tiwary *et al.* [TNL95] found that OO7 reused pointers[20] about *100 times less frequently* than their CAD visualization application. Furthermore, no transient pointer traversal is included in both benchmarks. This unfairly affects the results for a coarse-grained address translation scheme which relies on high pointer reuse for overall performance benefits over a fine-grained scheme, that adds overhead to *every* pointer dereference, regardless of whether it is a persistent or a transient pointer.

Data Structures and Algorithms

Another point against the benchmarks is their failure to exactly specify data structures and algorithms that must be used for the benchmark schema and during various benchmark operations. For example, the original OO1 specification does not specify the exact structure of the parts index. Similarly, the OO7 benchmark does not specify the kinds of containers (e.g., sets, bags, lists, etc.) that must be used for various collections of objects in the benchmark schema. This is quite undesirable, especially for comparison across different systems, because performance differences between these systems can be significantly affected due to fine-tuning of specific data structures. The benchmarks also do not specify transient data structures, although they are required for the benchmark operations.

In a similar vein, both OO1 and OO7 benchmarks do not specify some important algorithms that may affect the connectivity of the graph and the corresponding traversals. For example, the default implementations use the standard pseudo-random number generator available in the underlying operating system. The algorithm for pseudo-random number

[20] A pointer is said to be *reused* when an application traverses (or dereferences) it more than once.

generation is likely to vary between different operating systems, or even between different versions of the same operating system. The random number generator is an integral piece of the OO1 benchmark—the initial connections when building the original connectivity graph and the root part for each traversal are selected randomly—and differences in its implementation may significantly affect the performance of a system.

5.8.3 Summary

We have presented some common problems with OO1 and OO7, two of the most popular database benchmarks used for studying the performance of various OODB systems and persistent object stores. We believe that these benchmarks are unsuitable for performance measurements of orthogonal persistent systems and as general object database benchmarks. There is also some independent evidence that these benchmarks do not emulate the characteristics and behavior of CAD applications, which typically represent the kind of applications that exploit OODB systems.

With both OO1 and OO7 benchmarks, we have identified several issues— especially poor locality characteristics and lack of computation behavior—that are not representative of normal application behavior. In general, we believe that these benchmarks are not designed to measure address translation costs which is what we are most interested in; instead, they mostly measure the cost of loading and caching persistent data. Furthermore, because of the randomization in the data, it is unclear whether the results are at all meaningful; most real applications exhibit both locality of reference and distinctive phase behavior.

Finally, we believe that OO1 and OO7 are also not suitable for studying other issues such as clustering. Since the characteristics of both benchmarks are fairly random, there is little opportunity for a sophisticated clustering scheme to exploit the same regularities that it might successfully exploit for real applications. Furthermore, the interconnections between database structures (at least in OO7) also discourage static clustering. Of course, the effects of clustering, etc. on the overall performance is beyond the scope of this dissertation.

5.9 Conclusions

The various performance results presented in this chapter have supported our basic argument that pointer swizzling at page fault time has zero costs when there is no faulting from the persistent store. The major cost of the system is incurred when a page is loaded into memory and the pointers are swizzled into virtual memory addresses. Even then, the overhead is small compared to the I/O costs incurred for fetching the page from disk. The basic idea takes advantage of the locality of reference in the application access patterns such that the higher per-page costs of swizzling at page fault time are offset by avoiding unnecessary overhead during all subsequent accesses to the same object.

As networks get faster, it is likely that the persistent stores will be stored in the main memory of a remote data server rather than on a local disk of the client host. This reduces the places where we can "hide" our overheads, and has a potential to make coarse-grained address translation look unattractive. However, we believe that this will not be a major issue because we expect significant improvements in processor speeds, as well as operating systems, that will effectively reduce the swizzling overheads as well.

We also experimented with different granularities of address translation for studying the general applicability of these granularities. We found that mixed-granularity schemes may be appropriate in some cases, specifically for data structures that have a high fanout and are only accessed sparsely. An example of such a data structure is the parts index used in the OO1 benchmark. For such structures, it may be preferable to use pointer-wise address translation to reduce the rate of address space consumption. Depending on the application characteristics, a mixed-granularity approach might work well in such situations.

In the process of measuring performance of the pointer swizzling at page fault time mechanism and its different components, we have also learned some interesting lessons about operating system implementations. For example, the cost of handling protection faults on Solaris is several times higher than that on Linux, even when using identical hardware. The exact reasons for this slowdown are not clear but it does affect the performance of Texas, especially during the first few traversals. In addition, we also found that the buffer managements on both operating systems played an important roles in overall results. The aggressive buffer management under Linux frees up a few extra megabytes which appears to be just sufficient to avoid unnecessary paging activity during the actual traversals.

Chapter 6

Run-Time Type Description

6.1 Introduction

Access to information about data object layouts at run time is necessary for clean and efficient implementation of various families of run-time support software. Texas (Chapter 4) and other persistent object stores [ABC$^+$83a, AM95, LLOW91, WD94] that use pointer swizzling techniques such as pointer swizzling at page fault time need to know the locations of pointers in data objects at run time in order to find and manipulate these pointers correctly. Similarly, precise garbage collectors [WJ93] also use this information to locate pointers in objects for tracing the reachability graph and reclaiming garbage.[1] Other applications that benefit from the knowledge of low-level layout information are:

- data structure browsing,
- data structure pickling,
- data format conversion for sharing between machines with opposite endianness,
- parameter marshaling for distributed communication (including remote procedure calls),
- advanced foreign function call interfaces for efficient cross-language data sharing, and
- advanced profiling, tracing and debugging.

The ability to support run-time type queries about data objects is available for some high-level languages such as Smalltalk [GR89], CLU [LAB$^+$81] and Modula-3 [Nel91]. The term *Run-Time Type Identification (RTTI)* [SL92] has been used to represent language-level semantics and the ability to support operations that allow the application to ask whether a given type is a subtype of some other type. Recently, the C++ standard has added RTTI to support operations such as “downcasts” without circumventing the type system. Unfortunately, the standard RTTI information is insufficient for our purposes since it does not describe

[1] Traditional garbage collectors for languages such as C++ have been conservative [BW88]. That is, any data value which “looks” like a pointer is treated as such while tracing the reachability graph. In contrast, precise garbage collectors, which are typically used for real-time applications, cannot be conservative because they need exact pointer information to honor the necessary correctness and performance guarantees.

the *implementation-level* information necessary for run-time support systems. We introduce the term *Run-Time Type Description (RTTD)* to denote low-level object layout descriptions and other implementation-dependent information made available at run time.

In this chapter, we describe a portable, general purpose, high-performance mechanism for generating and manipulating RTTD; this mechanism is designed to be applicable to various high-level programming languages and is compatible with conventional compilers. The fundamental idea is to use compiler-generated debugging information to extract the implementation-level information necessary for RTTD. We believe that this is the most portable approach for a variety of applications and argue that compiler-generated debugging information is preferable to the use of preprocessors because preprocessors are hard to use, develop, maintain, often incompatible with other preprocessors, and not portable.

A Note on "Portability." We use the term "portable" at two levels: the portability of our approach, and the portability of our implementation. Our approach relies heavily on the compiler providing object layout information in its debugging output. It is reasonable to expect any modern compiler that supports debugging will provide this information. Hence, our approach works with standard compilers and operating systems. The other level of portability is related to the implementation of our system. We do not mean that our system is portable in the "compile out-of-the-box and run" sense, but rather that it is relatively easy to port because the implementation does not rely on any unusual compiler or operating system features. In other words, our system is less portable than an ANSI C program, but much more portable than (say) a "portable" compiler or operating system.

Scope of this Approach. We focus mainly on dynamically-allocated objects because we interact with the allocator to locate and record the type information with the object; this is sufficient for most applications. Extensions to handle statically-allocated instances are possible (using link information from the object files), but are beyond the scope of this dissertation. Stack-allocated objects may pose greater difficulties, and are also a topic for future work. The use of debugging information is sufficient for most purposes, but some may require further enhancements if the compiler-generated debugging information does not contain all the required low-level information.

Current Status and Availability. We have implemented this system for multiple platforms by leveraging code from the GNU debugger, `gdb`, and the source is publicly available under the GNU General Public License (GPL)[2] at `ftp://ftp.cs.utexas.edu/pub/garbage/texas`. We are currently using this system in Texas (Chapter 4) and a real-time garbage collector for C++ [WJ93]. The system has been tested under several flavors of Unix (SunOS, Solaris, Ultrix, Linux, etc.) with the GNU C++ compiler and under OS/2 with IBM VisualAge compiler (previously known as CSet compiler).

[2] Because our code does not link with the application code, the application does not fall under the scope of the GPL. In addition, the output (i.e., the type information embodied by type descriptor records) generated by the system is not covered by GPL either. Thus our system can be (and is) used with commercial applications without royalty or licensing restrictions.

Structure of the Chapter. The remainder of the chapter is organized as follows. Section 6.2 provides an introduction to RTTD and motivation for its need and use. It also discusses other techniques for generating RTTD, including preprocessors, and compares them to our approach of using debugging information. Section 6.3 discusses details of RTTD generation and manipulation, including the overall steps necessary for providing RTTD in a high-level language. We provide a detailed description of a case study implementation for C++ in Section 6.4, followed by a description of our storage model (Section 6.5) and a sketch of expected performance characteristics (Section 6.6). Sections 6.7 and 6.8 describe current status and some related research, and finally we conclude in Section 6.9.

6.2 RTTD Issues

This section discusses some fundamental issues concerning RTTD and provides motivation behind the design of a general-purpose mechanism to provide implementation-level type information. In addition, we introduce our approach of using debugging information for RTTD generation, and compare it with a seemingly obvious (but problematic) scheme of using special-purpose preprocessors.

We believe that preprocessors are not suitable for RTTD generation because of a fundamental "impedance mismatch"—preprocessors operate at the *language-level* (for example, parsing language constructs) whereas we are primarily interested in the *implementation-level* semantics (for example, exact locations of pointers in objects). Because a preprocessor is not an integral part of a compiler, it can only *infer* the compiler's actions based on the information available at the language level (that is, in the source code). In contrast, using debugging information allows us to "ask" the compiler about its exact behavior that is relevant for our purposes.

6.2.1 Motivation

The primary motivation for a portable RTTD mechanism is to support the development of efficient, powerful language extensions for use with off-the-shelf high-performance conventional compilers for languages such as C, C++ and Ada. Off-the-shelf compilers typically do not directly support low-level object layout information required to implement such extensions well. In addition, we want to ensure that RTTD can be used with any language whose compiler provides the necessary debugging information. Other goals of the design are to achieve:

- *efficiency*, by performing all complex steps at compile time and minimizing run-time space and time overheads,
- *ease of use*, by requiring minimal changes to the source programs,
- *elegance*, by providing graceful integration at appropriate steps in the usual compilation and linkage process, and
- *portability*, by relying on standard compilers and debugging information formats.

6.2.2 RTTD vs. RTTI

The current C++ draft standard [WC96] describes a proposal for Run-Time Type Identification (RTTI) as part of the language. Similar features are also available in other languages such as Java. Typically, the information provided by the RTTI mechanism is useful only for *language-level* semantics such as run-time type equivalence checks and "safe downcasts." In contrast, RTTD is designed to provide *implementation-level* information such as the actual in-memory layout of data objects, including sub-objects as well as the layout dictated by the inheritance hierarchy.

As described in the C++ standard, the language implementation must provide a class called `type_info`; objects of this class are used (at run time) to represent type information about application types. An object of this class is returned as a result of applying a `typeid` expression on an application data object. The only operations permitted on an object of type `type_info` are equality checks to compare with other objects of the same type, and a `name` function that returns a null-terminated string containing a unique implementation-defined value which represents the name of the corresponding type.

Compared to the RTTI schemes, the RTTD mechanism described here is significantly more powerful and allows an application to "ask" various questions about the object layout. For example, it is possible to query the types or offsets of all fields, or explicitly determine locations of pointer fields, in an object at run time. In general, RTTD is not equivalent to RTTI as specified for C++ and Java because. unlike the latter, it provides extensive low-level information required for applications such as persistent stores, precise garbage collectors, schema evolution mechanisms, etc.

6.2.3 Type Descriptor Records

Type descriptor records form an integral part of RTTD and are used to represent object layout descriptions at run time. We generate a type descriptor record corresponding to each type for which RTTD is desired; this record contains the low-level layout information for all objects of that type. Section 6.5 provides further details about the different formats used for storing type information in the type descriptor records.

6.2.4 Preprocessors vs. Debugging Information

We consider two main approaches for generating type descriptor records:

1. using special-purpose preprocessors, and

2. using debugging information.

We will argue for the second approach. Other possible techniques include requiring significant programmer intervention,[3] extending language syntax or using custom compilers. We believe that these methods are unsuitable because they are expensive, inflexible and place unnecessary demands on the programmer. For example, building a high-quality compiler is a complex task,

[3] In practice, as described in Section 6.4, we do require a *minor* amount of programmer intervention for our C++ implementation.

and is not always worth the effort for implementing one or a few features. In addition, porting and maintaining such a compiler on various platforms is likely to be prohibitively expensive.

Using Preprocessors

An obvious technique for building type descriptor records is to provide a preprocessor that parses the source and extracts the necessary information. At first glance, allowing portable source-to-source translation using a preprocessor seems to be a simple and effective solution for RTTD. However, we believe that it is not the right approach for several reasons.[4] Preprocessors are typically:

- *hard to use*: Simple preprocessors do not provide a clean syntax or complete syntactic error-checking; errors are reported inconsistently and may confuse the programmer;

- *hard to develop*: Sophisticated preprocessors that gracefully extend the language syntax and do exhaustive syntactic error-checking and reporting must duplicate a significant amount of the work done by the compiler. In effect, they become precompilers rather than just preprocessors. As compilers evolve, it is difficult to keep preprocessors consistent with them;

- *hard to maintain*: Many languages, such as C++, are still undergoing standardization. This makes preprocessors hard to maintain because unrelated changes to the syntax of the language still require modifications of the preprocessor to parse these changes;

- *usually incompatible with other preprocessors*: Relying on preprocessors leads to a trend of providing a specific preprocessor for solving each problem. This eventually results in a sequence or a pipeline of preprocessors that are usually incompatible—each is confused by the constructs understood by later preprocessors in the series. This problem is exacerbated for "nested constructs" which require repeated applications of a preprocessor such that no specific order of preprocessor invocation is acceptable. In general, constructs implemented by different preprocessors in a sequence do not interact properly; and

- *not portable*: Preprocessors are compiler-dependent with respect to several issues:

  - *compiler-specific language extensions.* Some compilers may extend the language with additional keywords and syntactic variations;
  - *structural alignment and padding.* Different compilers and operating systems may impose different alignment and padding restrictions on objects;
  - *component order.* Some languages, such as C++, do not specify the placement order of fields within objects; and
  - *hidden fields.* Some language features may require implementation-defined fields that are generally not exposed at the source code level.[5]

[4] We believe that these arguments against preprocessors are valid not only for RTTD, but also for most other purposes.

[5] One example is the usual implementation of virtual functions in C++ using virtual function table pointers inserted by the compiler.

In general, preprocessors reduce flexibility and we do not advocate their use, both for RTTD generation and other situations. In contrast, our approach is similar to a postprocessor in that we rely on actual information generated by the compiler itself, rather than *inferring* it from an examination of the source code.

Using Debugging Information

Most compilers can emit debugging information that includes a description of the layout of the types used in the application so that a source-level debugger can be used to examine the data structures while running the program. The debugging information is included in the object files when the application source is compiled using the (compiler-specific) debug option. It is possible to extract this information and format it into type descriptor records to provide RTTD. This approach requires minimal compiler cooperation, that is, only to the extent of existing capabilities of most modern compilers, and has major advantages over other solutions:

- it is mostly independent of the source language used, because the format of debugging information for a specific platform does not typically depend on the details of the source language, and
- it is mostly compiler-independent; the only compiler cooperation required is that the debugging information be generated in one of the standard formats.

Finally, it should be noted that our method does not impose a space or time penalty on the production version of the application because the debugging information can be "stripped" from the object files once the type descriptor records have been generated.[6] Alternatively, the application may be recompiled without debugging and with additional optimization.[7] Thus our approach is usable with compilers that prohibit or reduce optimization when producing debugging information. We are unaware of any compilers that change the layout of heap-allocated objects based on the presence or absence of debugging information, or the level of optimization used during compilation. In general, compilers should *not* do this anyway because it complicates normal library linkage; if libraries linked into the applications are compiled with different optimizations, the final result can be disastrous if object layout varied with degree of optimization. This situation is exacerbated by mechanisms such as persistence where the objects stored in a persistent store may suddenly have different layouts than expected because the application that is accessing these objects is compiled with different optimization than the one that created the objects.

6.2.5 Adapting to Future Compiler Support

Some compilers may eventually provide some form of implementation-level type information. Currently there are no standards for the format or methods by which such information will be made available, and we do not expect standardized, full-featured low-level type information to be available soon. However, when this information does become available, code may be

[6] Of course, the type descriptor records need to be stored somewhere, but that cost is negligible compared to the cost of retaining debugging information.

[7] This recompilation can easily be automated using standard *makefiles*.

written to transform such non-standard information into our type descriptor records. Such a technique will require neither a preprocessor nor the debugging information. We hope that standards and programming interfaces for implementation-level type information will emerge, making it trivial to write such “adaptor” code.

6.3 RTTD Generation and Manipulation

The basic goal of RTTD is to provide a mechanism to obtain the low-level object layout information given the address of an object. Before accomplishing this, the following two problems must be solved:

- constructing type descriptor records that describe the layouts of objects of various types, and
- associating these type descriptor records with actual instances of appropriate types at run time.

The type descriptor records are stored in a table indexed by type names. To generate these records from the debugging information, the application source is first compiled with compiler-specific debug flag. Next, the resulting debugging information in the object code is parsed to extract the layout information which is then formatted into type descriptor records. Type descriptor generation can easily be accomplished at compile time as an additional action after each object file has been generated from a corresponding source file.

The second problem is more challenging because we need a mechanism to identify *concrete types*[8] being instantiated at each allocation site, use this information to look up the corresponding type descriptor record and associate it with the newly-allocated instance. Unfortunately, conventional languages do not provide any direct support for associating compile-time information with run-time instances. For languages such as C that do not have parameterized or nested types, concrete types can easily be identified from the source. For others, such as C++ and Ada, that provide more complex type systems, a sophisticated approach is necessary and it may involve some minor system dependencies.

Once these problems have been solved, a mapping from an object to its type descriptor record can be made available at run time. We accomplish this by providing a two-step mapping: from an object to its *type identifier*, and from the type identifier to the type descriptor record. While conceptually a single mapping is sufficient for most uses of RTTD, we chose a two-step mapping for added implementation flexibility and efficiency. (As we explain later, the extra level of indirection is inexpensive, and is useful for linking separately-compiled modules.)

The type identifier (or `typeid`, for short) is a token that uniquely represents a concrete type in the application. Note that any representation may be used for the type identifier as long as it provides a key that uniquely identifies the associated type descriptor record. In our current implementation, a type identifier is simply an integer offset into a table of type descriptor records. A unique type identifier also allows us to eliminate duplicate type descriptor records across separately-compiled modules, as described in Section 6.3.4.

[8] A *concrete type* is any basic type such as `integer` or `character`, an aggregate type or the instantiation of a parameterized type; a concrete type is instantiated to create an actual instance.

6.3.1 Generating Type Descriptor Records

The basic approach for generating type descriptor records is to parse the debugging information from object files and build a table that maps type names of concrete types to corresponding object layout information embodied by the type descriptor records.[9] This table is made available to the running programs via an application program interface (API).

Although the debugging information format varies from platform to platform (i.e., across different operating systems), there are a few representative formats that can be translated into a common type descriptor record format. Our implementation of the type descriptor generator is divided into two parts: a platform-specific part to extract debugging information from object files, and a platform-independent part to build the type descriptor records from the extracted debugging information. To further generalize the implementation of the platform-specific part on different flavors of Unix systems, we have leveraged code from the GNU debugger, `gdb`, to parse and extract debugging information from various object files formats understood by `gdb`. However, it is not difficult to implement this functionality directly; a version for OS/2 and IBM VisualAge compiler has also been implemented using a 2000-line C++ module instead of `gdb` code.[10] Using `gdb` for the platform-specific part of the implementation makes our type descriptor generator highly portable to other systems as `gdb` is enhanced to understand a variety of debugging information formats on different systems.

A note about reliability of debugging information is perhaps in order here. The correctness of code generated by a compiler is obviously much more critical than the correctness of debugging information, and sometimes compilers are released with incomplete or broken debugging support. This is a possible problem with our approach. Fortunately, our system relies on only a subset of the standard debugging information—the layout information—which is much more reliable than other debugging information such as the mapping between line numbers and program counter, or between variables and registers. Occasionally, some version of a compiler may be "broken" in a relevant way, and in such cases, a different version must be used to work around the problem.[11]

6.3.2 Associating Type Descriptor Records with Objects

As mentioned earlier, associating type descriptor records with actual objects is somewhat more difficult than generation them. We divide this problem into two smaller parts, each of which can be solved individually:

- identify concrete types at allocation sites, and
- record the type descriptor record with the instance.

Recall that the table of type descriptor records generated at compile time maps type names to corresponding object layout information. Hence, we can locate type descriptor records

[9] Type descriptor records are actually linked together into a data structure that represents the type graph of the application.

[10] In retrospect, this module is more general than necessary, and could probably be half as long. Future ports should be easier because we can reuse code from this module as well as avoid the excessive generality.

[11] For example, a few of the many releases of the GNU C++ compiler have affected our system, but it has not been a serious problem in general.

corresponding to concrete types by using their (unique) type names. The type information is passed to the allocator, which performs a table lookup and retrieves the corresponding type descriptor record to be recorded with the instance.

Type identification using type names may be tricky for some languages, because the necessary information is not directly available at an allocation site. For a language like C, it is easy to capture the type name at an allocation site by implementing the allocator call as a preprocessor macro that takes the type name as an argument and converts it into a string representation. This is acceptable because the language does not provide an advanced type system and actual type names are directly available at allocation sites. However, for languages such as C++ and Ada, this is not a viable solution because the type name is not directly available at allocation sites such that *macro expansion occurs "too early" in the overall compilation process*—before parameterized and nested types have been resolved to unique concrete types—to capture the type name.

Instead, we determine concrete types *after* the code has been compiled and types at allocation sites have been resolved to concrete types. This approach requires a mechanism for *backpatching* the allocation sites. That is, we must plug in a concrete type identifier at each allocation site *after* it has been compiled and the concrete types have been resolved. As we describe in detail later, this backpatching can easily be implemented by adding an extra level of indirection and leveraging linker name resolution. We introduce special backpatching variables that hold type identifiers for each type, and reference these variables at corresponding allocation sites. After the application has been compiled, we can generate code to initialize these variables appropriately—the linker resolves references to the variables in the normal way, and their initialized values will be available at the allocation sites.

Finally, our allocator simply stores the type identifier in a hidden header for the object. Most allocators already attach a header to every object for bookkeeping data, and we can augment that data to include the type information for RTTD. In addition, the allocator must also support mapping from pointers to (or to the interior of) an object to the header of the object. This is especially necessary for C and C++ because pointers may not point to beginnings of objects, due to pointer arithmetic or the standard implementation of multiple inheritance that implicitly uses pointers to sub-objects.

6.3.3 Compilation and Linkage Model

We gracefully integrate the generation of type descriptor records and their association with actual instances at appropriate steps in the normal compilation and linkage process. This allows us to build type descriptor records when corresponding debugging information is available, generate code to create and initialize backpatching variables to hold type information, and link object modules together along with the initialization code and libraries for manipulating type descriptor records. We describe the overall compilation and linkage process below, specifically noting the steps required for integration of RTTD.

Source code for an application is typically divided across multiple files. The application executable is built using a *compile-and-link* model—source files are usually compiled individually to generate corresponding object files (compile phase), which are then linked together to create a single executable for the application (link phase). We have designed the RTTD generation mechanism to use a similar model; the type descriptor generator extracts type information from individual object files to build corresponding mapping tables ("tdcompile" phase), which are then merged to produce a single table corresponding to the application executable ("tdlink" phase).[12]

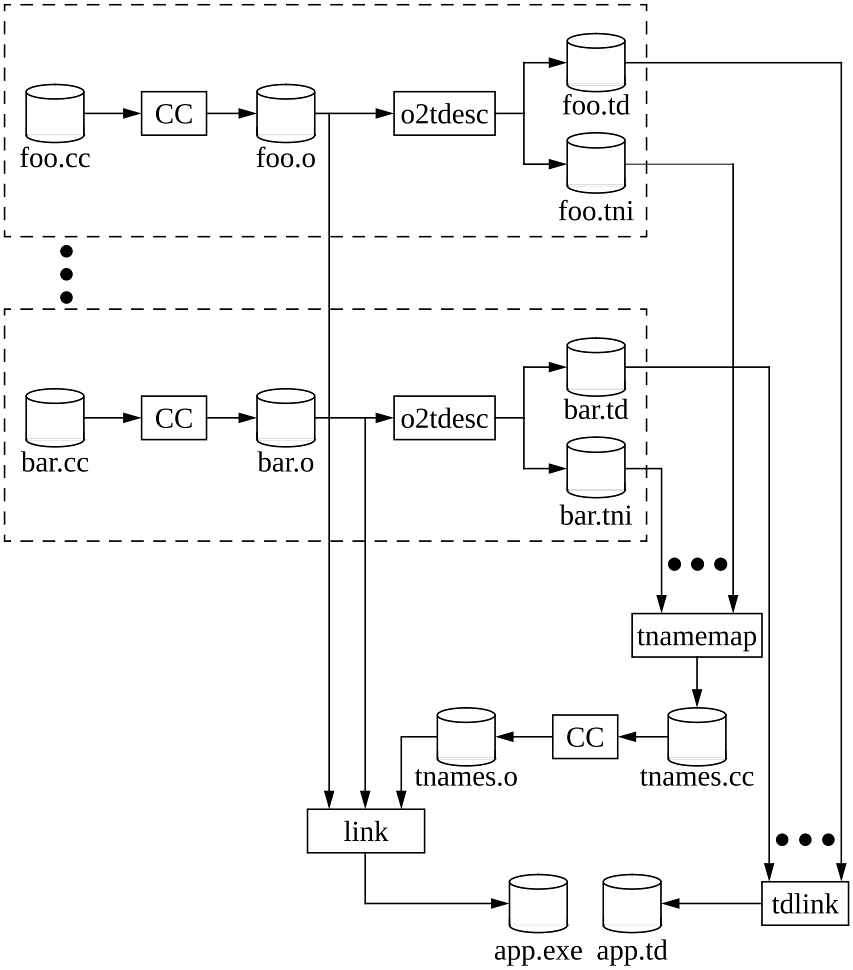


Figure 6.1: Compilation and linkage process

Figure 6.1 shows the overall compilation and linkage process. The basic steps involved in RTTD generation for a particular application are as follows:

1. Compile source files with the debugging option enabled and generate object files (standard compile phase).

2. Generate type descriptor records from debugging information in each object file ("tdcompile" phase).

   This is done by a stand-alone program, `o2tdesc`, supplied with our system.

3. Merge type descriptor records from multiple modules to be linked, into a single table of type descriptor records for the application and eliminate duplicates ("tdlink" phase).

[12] We use the terms *tdcompile* and *tdlink* to denote the two phases of type descriptor records generation, and to distinguish them from the standard compile and link phases.

This is done by another utility, tdlink, also provided with our system.

4. Generate auxiliary object file containing initializations of all backpatching variables that are used to hold information about concrete types.

   First, an auxiliary source file containing the initialization code is generated by a standalone utility, tnamemap. This source file is then compiled with the same compiler that was used to compile the application source files. (Section 6.4 provides full details about the backpatching variables while describing the case study implementation for C++.)

5. Link all object files (including the one from the previous step) with a support library for accessing type descriptor records, to generate the final application executable (standard link phase).

It should be noted that our "tdcompile" phase *follows* the standard compile phase because we use object files to generate the type descriptor records. In contrast, the "tdlink" phase *precedes* the standard link phase because it generates auxiliary source code which must then be compiled and linked into the final executable.

The above steps can be easily automated by using straightforward makefiles. Since most large applications already use makefiles to automate the standard compile and link phases, it is easy to extend this by providing additional targets and actions for RTTD generation.

6.3.4 RTTD Across Multiple Compilation Units

For languages such as C or C++, each compilation unit usually refers to an object file that is typically generated from a distinct source file. Several compilation units are put together to generate a single application executable. As can be seen from Figure 6.1, our compilation and linkage model has been designed to be conducive for supporting RTTD across multiple compilation units. The type descriptor records are generated for each compilation unit, and then linked together into a single set of type descriptor records for the entire application. This allows us to regenerate type descriptor records selectively based on compilation units that are modified. If makefiles are used to regenerate object files when source files change, a simple modification of the makefile will also cause new type descriptor records to be generated when a new object file is created.

The use of backpatching variables for type identifiers is also favorable for providing RTTD across multiple compilation units. Usually, the type descriptor generator only has local knowledge about type information in the object file being processed in the current execution. It is not possible to assign unique type identifiers to types during a particular execution without maintaining some global knowledge about type information from all object files processed earlier. However, since type identifiers are stored in backpatching variables that are initialized separately after the standard compile phase, it is relatively trivial to assign unique type identifiers to types; the tnamemap utility, which generates the initialization code, can ensure that each type is assigned a unique type identifier value.

The compilation and linkage model also facilitates elimination of duplicates. Because the same types may be used in different modules of an application, type information may be duplicated across various object files, and hence across mapping tables generated from these files. The `tdlink` utility performs duplicate elimination when it merges type descriptor records from multiple modules into a single table for the application. Duplicate elimination can be done based either on *name equivalence* or on *structural equivalence*. As the names imply, *name equivalence* considers two entities (types) to be equivalent if both have the same name, while *structural equivalence* uses the structure of the entities to compare them.

6.4 RTTD for C++

Section 6.3 discussed the high-level issues involved in generating and manipulating RTTD for any conventional source language, while avoiding an elaborate discussion of some language-specific issues such as the implementation of backpatching variables. In this section, we describe our case study implementation of RTTD for C++. We currently use this implementation in the Texas Persistent Store and a real-time garbage collector for C++ [WJ93].

As described earlier, type descriptor generation relies on the existence of debugging information in object files, and as such, does not directly depend on the source language. In contrast, associating type descriptor records with appropriate instances requires modification of allocation sites, and is obviously language-dependent because it involves the language interface.

Below, we first provide a high-level overview of the whole system in Section 6.4.1 before describing the implementation details in Section 6.4.2. We also discuss issues about handling multiple compilation units (Section 6.4.3) and the use of types names for added flexibility in some applications (Section 6.4.4). Finally, we briefly discuss some possible complications and and enhancements in Section 6.4.5.

6.4.1 Overview

Our approach exploits some common features of C++, while minimizing dependencies on any specific compiler implementation. Our goal is to provide a language interface that requires minimal changes to existing source in order to facilitate ease of use; most of the complex processing is performed behind the curtains while providing a simple front-end to the user.

Recall that the basic strategy is to provide a mechanism to associate type descriptor records with the corresponding objects when they are instantiated; this can be accomplished by modifying the allocation site to make type identifiers available to the allocator, which in turn stores them with the newly created object. For C++ implementation, the main mechanism for modifying the allocation site and the underlying allocator is to change the C++ `new` operator;[13] we *overload*[14] the operator to expect the type identifier as an additional argument and store it in the header of the object. In addition, we also provide a macro interface which encapsulates the (slightly awkward) syntax of the overloaded `new` operator.

For languages such as C, where concrete type information is directly available at the allocation site, a literal type identifier may be passed to the allocator. However, because C++ supports nested and parameterized types, information about the concrete type being instantiated is not always available directly at the source level, and hence a literal type identifier

[13] C++ provides a `new` operator to dynamically allocate objects in an application's memory.

[14] C++ allows a programmer to *overload* a function or an operator by providing different implementation with the same name, as long as the function signatures can be used to distinguish the implementations.

cannot be passed to the allocator. The type information usually becomes available only after the source has been compiled and the compiler has resolved the concrete type at each allocation site. Hence we need a mechanism to backpatch the overloaded `new` operator call-sites to provide the appropriate type identifier value after compilation. However, we also want to avoid object code modification for this purpose. Therefore, we implement backpatching by introducing an extra level of indirection, and leveraging the compiler's own type processing and the linker's natural name resolution as follows. We use (and reference) special backpatching variables, rather than literals, for the type identifiers at each allocation site; when the source is compiled, the compiler generates undefined references to these variables in the object code. Once the appropriate concrete type information has been extracted from the object code using the type descriptor generator, we generate auxiliary source code to initialize the variables to their appropriate type identifier values. This initialization code is then compiled and linked with the application object code, and the linker resolves the undefined references appropriately. In effect, this achieves the desired backpatching without requiring object code modification.

Note that each backpatching variable holds the type identifier for only a single type because each allocation site can instantiate only one type at any given time. In other words, there is a one-to-one mapping between a type and the corresponding backpatching variable. This observation allows us to have one backpatching variable for each type, rather than one for each allocation site; if the same type is instantiated at another allocation site, the same variable can be used at that allocation site without introducing any errors.

A simple approach for providing backpatching variables—which does not work in the general case—is to ensure that each type instantiated has a special class variable[15] which plays the role of the backpatching variable. However, this approach cannot handle builtin types (for example, `int`, `float`, etc.) in C++ because the language allows definition of class variables only for aggregate types. In addition, it requires modification of user-defined types (to add the class variable) which would burden the application programmer.

However, we can get essentially the same effect by modifying the solution slightly. We ensure that, for each unique type instantiated, there exists a *wrapper class* that contains the above-mentioned class variable; we can then use the variable in the wrapper class to hold the type identifier of the corresponding instantiated type. In order to maintain the one-to-one mapping between the instantiated type and the corresponding class variable, the wrapper class is parameterized such that a unique wrapper class is created for each type instantiated by the overloaded `new` operator. In effect, we have added a one-to-one mapping between a wrapper class and the associated type.

Use of a wrapper class provides several benefits for RTTD:

- *It allows easy and automatic creation of class variables for non-aggregate types.*

  Since a separate class is used to hold the class variable, this approach works for builtin types and does not require modification of user-defined types.

- *It allows easy generation of references to class variables in the overloaded* `new` *operator.*

  The macro interface can simply refer to the class variable in the wrapper class using the member reference operator (`::`), and the compiler automatically resolves the reference.

[15] A variable associated with a class, rather than instances of the class, is known as a *class variable.*

The instantiation of the parameterized wrapper class (to create a concrete wrapper class) occurs at compile time, and the run-time reference to the class variable of the wrapper class is fast.

- *It ensures that debugging information is generated by the compiler for all types allocated via the overloaded* `new` *operator.*

  C++ compilers may sometime optimize away the generation of debugging information for predefined types or types which may not be used in certain ways. Use of the wrapper class forces the inclusion of the type information of the instantiated type with that of the wrapper class.

- *It allows easy identification of types for which type descriptor records need to be generated.*

  Since a new concrete wrapper class is instantiated for each "interesting" type (i.e., a type for which RTTD is desired, because it was instantiated via the overloaded `new` operator), the type descriptor generator can use this information to generate type descriptor records only for selected types.

(Readers familiar with advanced object-oriented languages and first-class classes may recognize that a wrapper class for a given type acts as a class metaobject, holding information about instances of that type.)

Note that while the parameterized wrapper class is instantiated for each "interesting" type to create a concrete wrapper class, the resulting concrete wrapper class does *not* need to be instantiated to create unnecessary instances. Since the backpatching variable is a class variable, it is not associated with any specific instances and can directly be referenced using the class name.

6.4.2 Implementation Details

The previous section provided a general overview of the approach used to implement the RTTD mechanism for C++. This section describes the important components of the actual implementation based on operator `new` overloading and our macro interface to the overloaded `new` operator.

We first describe some background information about C++ `new` operator and how it is overloaded, before moving on to explain how we use it to associate type identifiers to appropriate objects at run time.

The C++ `new` Operator

The normal behavior of the `new` *operator* may be described as conceptually two distinct steps, even if code generated by a particular compiler may not explicitly distinguish between them:

1. a compiler-defined `operator new()` *function* is called to obtain storage for the object, and

2. the constructor for the type is called to initialize the object appropriately.

It is important to note the distinction between `new` *operator* and `operator new()` *function* in the above description. The `new` *operator* calls both the `operator new()` *function* and the constructor for a type.[16]

C++ allows a programmer to overload the `new` *operator* for providing additional arguments to the `operator new()` *function*. Note that overloading the `new` operator is different from other C++ operator overloading, in that the entire behavior of this operator is not redefined—only the first step above is overloaded while the constructor for the type is still invoked as in the normal behavior. Although this is referred to as "overloading the `new` operator" in C++ terminology, we are *actually* overloading only the `operator new()` *function*, without changing the constructor-invocation semantics of the `new` *operator*.[17]

Overloading the `new` Operator for RTTD

Recall that we need to associate a type identifier with each object that is allocated by the application. We achieve this by storing the type identifier as part of the bookkeeping information (maintained by the allocator) for the object so that the corresponding type descriptor record can later be referenced. We overload the `new` operator (actually the `operator new()` *function*) and use the so-called *placement syntax*[18] to pass the type identifier as an additional argument to the underlying allocator which can then store it with the appropriate bookkeeping information. Thus the syntax of the `new` operator call is as follows:

```
    X *obj = new (<typeid>) X (<constructor arguments>);
```

The constructor arguments are optional depending on whether a default constructor[19] is provided for type `X`. The expression "`<typeid>`" is a reference to the backpatching variable that will contain the corresponding type identifier value at run time.

Macro Interface for Overloaded `new` Operator

It would be burdensome to require programmers to remember the exact syntax required for the overloaded `new` operator, including the reference to the backpatching variable. Instead, we define a C preprocessor macro that encapsulates the syntax of the overloaded `new` operator call to provide a simple interface to the programmer. As will be evident from later sections, the scheme for generating a reference to the backpatching variable can result in a complicated syntax which may be awkward to use. The purpose of the macro interface is to make the entire mechanism of providing the type identifier transparent to the programmer. An additional benefit of using a macro is that it allows us to modify the underlying implementation for providing the type identifier without requiring modifications to existing application source.

The normal syntax for calling the `new` operator is:

[16] In the remainder of the chapter, we use the phrase "`new` operator" to refer to the *operator* and "`operator new()`" to refer to the storage allocation *function*.

[17] An excellent discussion about the `new` operator and semantics of overloading can be found in [Lip91].

[18] This term was originally used to provide "placement" information for the object as an additional argument to the underlying allocator. It is now a misnomer because the same syntax can be used to pass *any* additional arguments to the allocator.

[19] A constructor that takes no arguments is called a *default constructor* in C++, and is automatically generated for a class if no other constructors have been declared for that class.

```
X *obj = new X (<constructor arguments>);
```

Note that the constructor arguments are optional, and the number of such arguments is not predetermined. As shown earlier, the corresponding syntax for calling the `new` operator using placement arguments is:

```
X *obj = new (<typeid>) X (<constructor arguments>);
```

The expression "`<typeid>`" corresponds to the placement arguments to the `new` operator, and in the current description, it is a shorthand for an actual expression (reference to the backpatching variable) which evaluates to the type identifier value for type `X` at run time. In general, the number of placement arguments is not restricted but it must be predetermined for defining the overloaded `new` operator.

We provide a macro, `rttd_new`, that encapsulates the unusual placement syntax. The definition is of the form:

```
#define rttd_new(type)      new (<typeid>) type
```

where the expression "`typeid`" is derived in some way from the `type` argument provided to the macro. Using this definition, the syntax for calling our macro is:

```
X *obj = rttd_new (X) (<constructor arguments>);
```

Note that this is similar to, but not quite the same as, the syntax for a call to the standard `new` operator without any placement arguments. Specifically, an "extra" set of parentheses is required around the type name. This is necessary because the C/C++ macro preprocessor semantics are very weak; there is no support for macros with variable number of arguments, or for arguments not enclosed in parentheses. In our definition, the macro expects only one argument, the type name, and the (variable number of) constructor arguments are *not* part of the macro definition.

Note that the macro definition matches only the initial "`rttd_new (X)`" part of the statement. If the constructor arguments are provided (as in our example above) after the predefined part, the macro will not affect them; they will be in the "right place" after macro expansion and will function as expected. If the constructor of type `X` does not require any parameters, the user may either provide an empty set of parentheses, or not provide anything at all, in place of "`(<constructor arguments>)`" in the above example. The macro definition handles these cases correctly without additional user intervention. In effect, we simulate a macro with variable number of arguments for the trivial case where these arguments are not transformed or processed in any way during the macro expansion; the macro is defined such that it only recognizes the first argument (the type) and after macro expansion, the call resembles the placement syntax call. (Also, note that no special processing is required from the C preprocessor for this definition, and the scheme works correctly regardless of whether constructor arguments are provided.)

The Wrapper Class

We use the C++ template facility to implement a parameterized wrapper class that is automatically instantiated once for every unique type used in calling the `rttd_new` macro to create a corresponding unique concrete wrapper class.[20] The template wrapper class definition has the following form:

```
template <class T> class TypeDescriptorWrapper
{
  private:
    T *wrappee;
  public:
    static int typeid;
  ...
};
```

For every type `X` instantiated using the `rttd_new` macro, a corresponding concrete wrapper class `TypeDescriptorWrapper<X>` is created by instantiating the template `TypeDescriptorWrapper` with `X`. The class member `typeid` is used to hold the type identifier of `X` at run time, and the data member `wrappee` may be used to selectively generate type descriptor records for only those types for which RTTD is desired, that is, those instantiated via the macro.

We associate the `typeid` with the *class* `TypeDescriptorWrapper<X>` itself rather than *instances* of the class because there is always a one-to-one mapping between type `X` and corresponding wrapper class `TypeDescriptorWrapper<X>`. We use static data members to implement `typeid`s because they represent the C++ mechanism for implementing class variables. As mentioned earlier, the concrete wrapper class itself is not instantiated because we only access its static data member, and no other normal instance variables..[21]

Based on the above parameterized wrapper class definition, a simplified version of the `rttd_new` macro definition, using static data member from the appropriate concrete wrapper class for accessing the `typeid`, is as follows:

```
#define rttd_new(type)     \
    new (TypeDescriptorWrapper<type>::typeid) type
```

The reference to `TypeDescriptorWrapper<type>` in the macro expansion automatically instantiates the template `TypeDescriptorWrapper` for that type without any user intervention, and the one-to-one mapping between type `X` and type `TypeDescriptorWrapper<X>` guarantees that the static data member will always hold the type identifier for the correct type `X`.

Finally, we have to ensure that static data members are defined and initialized correctly. This is necessary because static data members in C++ are considered to be only declared, *not* defined, when they are specified in the class definition. As described in Section 6.3.3, we generate an auxiliary source file containing these definitions and initializations. This file is then compiled and linked in with the application object code. For each type `X`, we generate a line of the following form in the auxiliary source file:

```
int TypeDescriptorWrapper<X>::typeid = <typeid for X>;
```

[20] In C++, when we *instantiate a template* by providing actual arguments (type or value arguments) for the template arguments, a concrete type is created from the parameterized (template) type.

[21] As might be expected of class variables, static data members in C++ can be accessed without referring to any specific instance.

The expression "`<typeid for X>`" represents the actual type identifier value for type `X`. The use of a wrapper class to explicitly identify types for which RTTD is desired (Section 6.4.1) can be used in a similar way to generate the appropriate initializations.

6.4.3 Handling Multiple Compilation Units

When multiple sets of type descriptor records from different compilation units are merged together to generate a single set corresponding to the application, we need to perform duplicate elimination to avoid making the final table too large. In our C++ implementation, we use *name equivalence* for this purpose. We do not need to use *structural equivalence* because standard linkers for C/C++ consider types with the same name in distinct object files to be the same type.[22] In other words, when the object files are linked together to generate a single executable, the linker resolves references to types using their names rather than their structure.

Each execution of the type descriptor generator creates a mapping table that maps *type name tokens* to type descriptor records. At link time, a single mapping table is generated after eliminating duplicates using type name tokens as keys. The type identifier for a specific type is then given by the index of the corresponding type descriptor record in the final mapping table, and the static data member initializations can be generated by performing a table lookup and replacing the expression "`<typeid of X>`" above with the actual type identifier value.

Note that any value can be used for a type name token as long as it uniquely identifies the same type across all compilation units. In the current implementation, we use *fully qualified type names*[23] because they exhibit the exact property required for the tokens. We made this choice because it is easy to access fully qualified type names and they also provide added flexibility for applications such as persistence as described in the next section.

6.4.4 Using Type Names for Added Flexibility

Conceptually, fully qualified type names can be used as type identifiers, instead of using integer values. However, because string manipulation is slow, we use integer type identifiers as an optimization. Initially, we use the fully qualified type name to look up the type descriptor record in the type descriptor mapping table. However, once we have performed the lookup, the integer index of the type descriptor record is cached as the type identifier thereby avoiding expensive table lookup when the type is instantiated again. Thus we pay the cost of string manipulation only the first time a type is instantiated using `rttd_new`; all future instantiations (anywhere in the application) will use the cached type identifier value. As described later in this section, the wrapper class is an ideal candidate for caching the type identifier due to its one-to-one correspondence with the wrapped type. In addition, integer type identifiers also allow us to quickly look up the corresponding type descriptor record by simple array indexing.

Fully qualified type names are also useful for our persistent object storage system because they provide a robust mechanism to resolve type information between a single persistent

[22] Note that C++ type names do include some structural information about the way classes are composed because of scoping and name mangling.

[23] In C++, the fully qualified name of a type is the string representation of the name after all template instantiations, if any, have been performed, and nested class and namespace scope information has been completely resolved.

store and multiple applications. The primary goal is to ensure that when an application accesses objects from a persistent store, the types of these objects from the application's point of view are the same as those from the persistent store's point of view.

Type descriptor records corresponding to types in a persistent store are saved in a mapping table similar to the one generated for an application. Depending on the locations of type descriptor records (for a given type) in both mapping tables, the integer type identifier value from the application side (we call this the *transient* `typeid`) is likely to be different than the type identifier value from the persistent store side (that is, the *persistent* `typeid`). Thus we cannot perform type equivalence test between types in the persistent store and types in the application based solely on the value of type identifiers. Instead, fully qualified names of types are a better choice because they will be same in both tables since they are independent of locations of the type descriptor records. In general, unlike standard C/C++ linkers, we must consider types to be identical only if they have both the same name *and* the same structure. We cannot rely only on name equivalence like standard linkers because a persistent store may be manipulated by many applications which may potentially use same names for types with different structures and semantics. Deferring the type identifier lookup and caching until run time, as described above, allows us the flexibility to use fully qualified type names to search both mapping tables (i.e., *name equivalence*), locate the appropriate type descriptor records, and compare them (i.e., *structural equivalence*) to resolve types between an application and a persistent store.

Unlike an application's type descriptor records table that is built by the type descriptor generator, the table for a persistent store is built incrementally as objects of different types are allocated in that persistent store. First, we use the fully qualified type name at an allocation site to look up the corresponding persistent `typeid`. If no such `typeid` is found, we then look up the corresponding transient `typeid` and copy the relevant information to the mapping table for the persistent store, thereby creating a new persistent `typeid`. On the other hand, if a persistent `typeid` is found in the initial lookup, it is stored with the object, after performing type equivalence tests as described above.

Generating Fully Qualified Type Names

It is important to devise a general-purpose scheme to generate fully qualified types names for using them as described above. An obvious approach—which does not work in the general case—might be to simply use the string representation of the type name at the time of instantiation. The idea is based on the use of standard C preprocessor "stringification" operator in the definition of the `rttd_new` macro to capture the fully qualified type name during macro expansion. This solution works for languages like C that do not support parameterized types, and hence the fully qualified name of the type is trivially available in the source. However, because of parameterized and nested types, stringification is not a general solution for C++; macro expansion and stringification happen during the preprocessing stage before the compiler has instantiated templates and resolved scoping, whereas the fully qualified type names must be generated *after* the templates have been instantiated and the scoping has been resolved. In other words, preprocessor stringification is "too early" in the overall compilation process to extract fully qualified type names.

Instead, we choose to use the type descriptor generator itself to generate fully qualified type names; since it is already a general-purpose tool for building type descriptor records indexed by fully qualified type names, we can provide a natural extension to save fully qualified type names as part of this process. Recall that the type descriptor generator relies on debugging information extracted from object files that are created after the compiler has instantiated templates and resolved all scoping. Therefore, all type names encountered in the debugging information will be fully qualified.

As described in Section 6.4.1, the type descriptor generator selectively generates type descriptor records only for "interesting" types for which RTTD is desired. We extend this model to selectively generate a separate table containing the fully qualified names of these types. This is trivial because we are already using the fully qualified concrete wrapper class names to find the "interesting" types; we can simply discard the wrapper-specific part of the name to obtain the fully qualified name of the instantiated type. Now, in a manner similar to duplicate elimination and merging of mapping tables for all object files (Section 6.4.3), we can generate a single table of fully qualified type names for all types in the application for which RTTD is desired. (This table is saved in a file whose name is suffixed with `.tni` as shown in Figure 6.1.)

Accessing Fully Qualified Type Names

Once the fully qualified type names have been generated from object files, we can make them available to the application at run time using static data members similar to the ones used for type identifiers. We extend the template wrapper class to contain an additional static data member `typename` that holds the fully qualified name of the type. The definition of the `rttd_new` macro is updated such that the type name is now passed as an additional argument, along with the `typeid`, to the overloaded `new` operator:

```
#define rttd_new(type)    \
    new (TypeDescriptorWrapper<type>::typename,   \
         TypeDescriptorWrapper<type>::typeid)     \
      type
```

The change in the definition of the macro does not affect the programmer interface in any way.[24] Consider the following code fragment (same as the earlier example) used to allocate an object of type `X`:

```
X *obj = rttd_new (X) (<constructor arguments>);
```

Using the new macro definition, the above code fragment would now be expanded by the C preprocessor as follows:

```
X *obj = new (TypeDescriptorWrapper<X>::typename,
              TypeDescriptorWrapper<X>::typeid)
            X (<constructor arguments>);
```

[24] Recall that one of the motivations for providing the macro interface was to allow us freedom in changing the underlying mechanism without affecting existing user source code.

As with the typeid, we have to ensure that static data members for type names are initialized correctly. Since we have already generated a single table of type names for all "interesting" types in the application (saved in a .tni file), it is trivial to generate appropriate definitions and initializations of the corresponding static data members in the auxiliary source file using this information. For each type X, we now generate the following initialization code:

```
int   TypeDescriptorWrapper<X>::typeid = <typeid for X>;
char *TypeDescriptorWrapper<X>::typename = "X";
```

The expression "<typeid for X>" may be replaced by the index of the corresponding type descriptor record in the final mapping table. However, there is no added cost to defer the table lookup for initializing the typeid until run time and, as described earlier in this section, it provides benefits for some applications such as persistence. For such cases, the typeid data member is initialized with an invalid value (e.g., -1) to signal the run-time environment to perform the appropriate table lookup.

6.4.5 Complications and Enhancements

The implementation described above is a fairly portable and robust mechanism for generating and manipulating type descriptor records in C++ applications. However, given the complexity of the language, it is almost impossible to provide a completely elegant interface to the low-level features. Below we sketch some of the complications related to our implementation and also describe possible enhancements.

Forcing Generation of Debugging Information

Some compilers may optimize away the generation of debugging information for a given type based on whether that type is instantiated to create any actual objects and whether those objects are used later. This may be especially problematic for the template wrapper class, TypeDescriptorWrapper, which is used only to generate concrete types that are themselves *never* instantiated to create any actual objects.

Our solution to this problem is as follows. We "fool" the compiler into thinking that we *may* instantiate the concrete wrapper class at run time—though we *never* actually do—by modifying the definition of the rttd_new macro. We include an expression that conditionally instantiates the concrete wrapper class to create an object and calls a dummy method on that object:

```
#define rttd_new(type)    \
    new (TypeDescriptorWrapper<type>::typename,        \
         TypeDescriptorWrapper<type>::typeid,          \
         (dummy_test_condition                         \
            ? (new TypeDescriptorWrapper<type>)->nop() \
            : 0)) type
```

Using the above definition, the code fragment from the earlier example would now be expanded as follows:

```
X *obj = new (TypeDescriptorWrapper<X>::typename,
              TypeDescriptorWrapper<X>::typeid,
              (dummy_test_condition
                 ? (new TypeDescriptorWrapper<X>)->nop()
                 : 0)) X (<constructor arguments>);
```

The last expression in the macro expansion fools the compiler into thinking that the concrete wrapper class `TypeDescriptorWrapper<X>` may be instantiated to create an actual object, and a method (`nop`) would be invoked on that object. We ensure that the dummy condition used in the expression will *always* be false at run time, so that we do not actually create unnecessary objects. However, the compiler cannot predict the outcome of the condition at compile time, and cannot optimize away the instantiation. A simple way to ensure this is by using the value of a global variable as the condition and initializing this variable to false in a separate compilation unit. This forces the compiler to generate debugging information for the concrete wrapper class, and consequently we can generate a type descriptor record for the appropriate type in the application (type `X` in our example).[25]

Interaction with Template Repository Mechanisms

Complex template handling schemes, such as template repository mechanisms, may be another source of problems. In template repository implementations, actual code generation for the template methods is typically deferred until link phase. The linker automatically detects missing template instances, generates them as necessary and invokes the compiler for each of them. The purpose of this mechanism is to ensure that only a single copy of template code is instantiated and included in the final executable, thus avoiding code bloat and duplication. With such implementations, we need a way to generate type descriptor records after the templates are instantiated and compiled but before the final linking is done.

One possible solution is to have the template repository mechanisms provide "hooks" so that additional actions (such as calls to the type descriptor generator) may be inserted when the template code is instantiated and compiled into object code. However, we are not aware of any template repository mechanisms that currently provide such a capability. Another possible solution is to have the compiler generate the debugging information normally in the object files where the templates are used and only defer instantiation of *code* for the template methods until link time.

In general, we propose that there be some type of a published "contract" between a compiler and a debugger regarding contents of the debugging information. If the compiler follows some pre-specified guidelines for debugging information generation and contents, tools such as the type descriptor generator can be implemented portably based on those guidelines.

[25] Note that we pass the value of the expression (which should always be zero) as an additional argument to the overloaded `new` operator. This additional argument is present only as a harmless side effect of our solution and is ignored by the overloaded `new` operator.

Handling Nested Types and Classes

Nested types[26] in C++ pose a special difficulty for generating the auxiliary source file that contains the definitions and initializations of the class variables used for backpatching. Consider the following definition and initialization in an auxiliary source file:

```
char *TypeDescriptorWrapper<X>::typename = "X";
```

The language requires that the auxiliary file must have a forward declaration (or the actual definition) for type X before it can be used in the definition and initialization as shown above. This is necessary so that the compiler recognizes the name as a valid type, and does not generate an error. Typically, a forward declaration is sufficient if information about the size, structure or behavior of the type is not required. A full definition may always be provided in lieu of (or in addition to) the forward declaration but it is not always necessary.

For the auxiliary initialization file, a forward declaration is usually sufficient because the type is not instantiated or used in any other fashion. Unfortunately, C++ does not allow forward declarations for nested types. As a result, we cannot generate the definitions and initializations of backpatching variables for nested types without some user intervention to provide the definition of the nested types. Currently, our system requires some programmer cooperation to solve this problem; the programmer must provide a list of header files that contain the definitions of the nested types. These files will be included in the auxiliary source file (and no forward declarations will be generated) so that the compiler can recognize names as valid type names when used in the definition and initialization.

The right solution, however, for this problem is to modify the language semantics to allow the forward declarations of nested types in the same way as for non-nested types. We believe that C++ can provide such a mechanism without any significant impact on the rest of the language definition.

Handling Virtual Function Tables

For some applications such as persistence, we need to treat C++ virtual function table pointers specially.[27] Unlike normal data pointers in an application, the virtual function table pointers point into code representing the executable program. That is, these pointers point into the load image of an executable, not into the data heap.

For data that may be operated on by multiple programs (or by recompiled versions of the same program), we need a symbolic representation of virtual function table pointer values. We achieve this by translating these pointer values (unswizzling) into indexes into a table of name strings when the data are saved, and later translating the indexes back (swizzling) into addresses of the corresponding virtual function tables of the new process that reloads the data.

[26] We use the term *nested types* to refer to both classes and non-aggregate types that may be nested within other classes.

[27] A virtual function represents the C++ mechanism for implementing dynamic method dispatch (run-time polymorphism). Each instance of a class that defines one or more virtual functions contains a virtual function table pointer, which is a pointer to a table of function pointers generated automatically by the compiler.

6.5 Storage Model

As described earlier, the low-level type information extracted from the application object and executable files is maintained in type descriptor records. Various utilities that need access to this type information manipulate the type descriptor records by loading them from disk into memory and "decoding" them at run time.

Depending on the application requirements, type information can be stored in one of two formats. By default, type descriptor records are generated and maintained in a *hierarchical format* that resembles a type graph containing complex or aggregate types as interior nodes and basic types as leaf nodes. It is possible to convert the hierarchical format into a *flat format* based on the specific requirements of the application. In the remainder of this section, we describe each of these formats in detail as well as our motivation for developing such formats. Although we will focus primarily on the formats used for in-core storage, the basic discussion also applies to disk storage. Further, the choice of in-core storage format also directly affects performance characteristics at run time (as discussed in Section 6.6).

6.5.1 Hierarchical Format

As the name implies, the hierarchical format maintains type information in a hierarchy of types implemented essentially as a type graph. Basic types are represented as leaf nodes in the graph, and are composed together to form aggregate types which form the interior nodes. Hierarchies are typically created by language semantics such as containment (one object contained inside another) or inheritance relationships (from the object-oriented programming domain). As such, a representation that maintains the notion of hierarchies maps well into the natural type structures enforced by the language.

We describe the hierarchical format by using a simple example. Consider the following two type definitions from some user application:[28]

```
struct Pet                  struct Owner
{                           {
  short tag;                  char *name;
  char *name;                 void *userdef;
};                            short numpets;
                              Pet pets[2];
                            };
```

The type descriptor records generated for the above definitions can be conceptually represented as a type graph shown in Figure 6.2. Each node in the type graph essentially represents a type descriptor record for a specific type; the leaf nodes represent type descriptors for basic builtin types such as `short` and `char` while the interior nodes represent complex or aggregate types. Each node has two labels—the top label is the name of the actual data structures used by the type descriptor generator, and the lower label is the name of the type that is being represented. Directed edges indicate that the source node (i.e., the node where the edges originate) represents an aggregate type and the destination nodes (i.e., nodes where

[28] For the sake of simplicity, we use C syntax here but the basic idea is applicable to C++ or other languages with aggregate types.

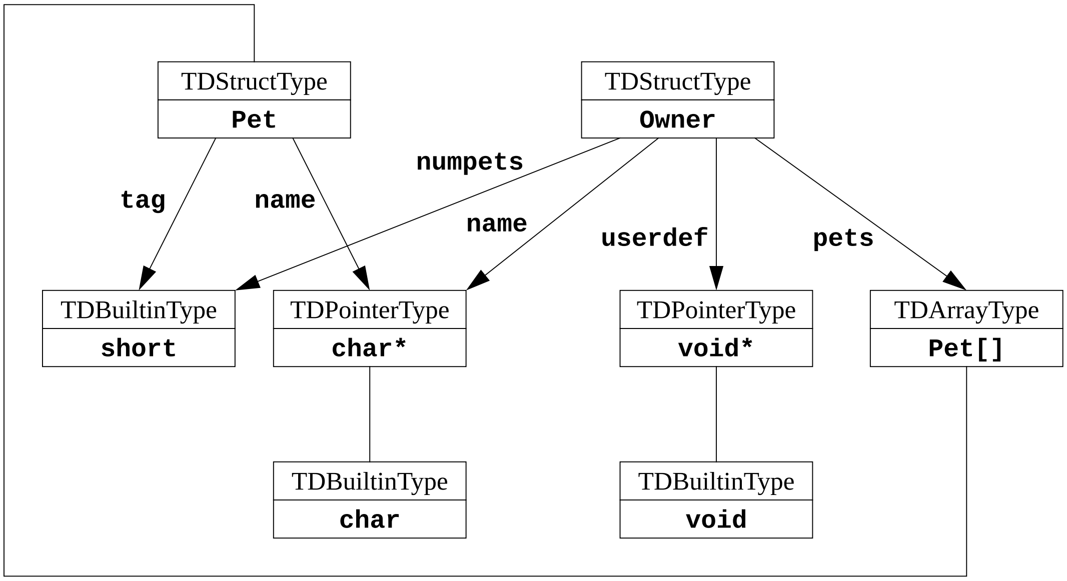


Figure 6.2: Type graph

the edges terminate) represent the types of fields of the aggregate type. The actual names of the fields are given as edge labels. Note that pointers and arrays are also treated as complex types composed of other types (`char` and `Pet`, respectively, in our example); we use undirected (and unlabeled) edges in the graph to denote this type composition.

Note that the type graph shown in Figure 6.2 is highly simplified on purpose to allow easier explanation of the basic concepts. Specifically, we have excluded information such as type sizes, field offsets, etc. that is obviously necessary to fully describe the type structure. The actual data structures used to represent the hierarchical type graph are indeed more complex and contain all necessary information to fully describe the types at run time; see Appendix A for a detailed description of these structures.

6.5.2 Flat Format

It is evident from the foregoing discussion that the hierarchical type descriptor format contains *all* possible information about a given type in the application, and is therefore necessarily complex. This complexity is justifiable because of the need to maintain generality for a variety of possible uses. However, depending on specific application requirements, only a subset of the information maintained in the hierarchical format may be interesting. For example, when implementing pointer swizzling at page fault time, we are primarily interested in locating only pointer fields within objects, disregarding all other information about the various types. It is possible to decode the hierarchical type descriptor records to obtain only the required information while ignoring the rest. However, in the interest of run-time performance, it may be preferable to transform the hierarchical format into a flat format up front, thus reducing the decoding effort required at run time. In essence, compile-time complexity and cost (i.e., format conversion) is traded for run-time efficiency and performance (i.e., faster decoding).

For pointer swizzling at page fault time, we convert the hierarchical format into a flat format that essentially contains a list of field offsets (corresponding to pointer fields) within the objects. The actual data structure, however, is slightly more complex; it consists of two parts, a *fixed part* and a *variable part*. The latter is necessary to support a commonly-used memory model in many C/C++ programs—by default, the language does not check for bounds violations, and hence it is possible to allocate a chunk larger than the size of the object and use the additional memory (at the end of the object) as an extension to the object. This technique is typically used to allocate an "inline" array within an object such that the array is "growable."[29] In terms of implementation, the array is usually declared as the last component of the object and dynamic allocation is used to allocate (or reallocate) a chunk of memory for the object; any excess memory past the (language-defined) end of the object can be used as if it was a part of the array, thereby changing (extending) its size. Although this heuristic is not defined in the official language specification, we support it because many programs actually rely upon this behavior. Note that this heuristic can be realized only by repeating some fixed sub-structure, that is, the structure of at least one element of the array must be known.

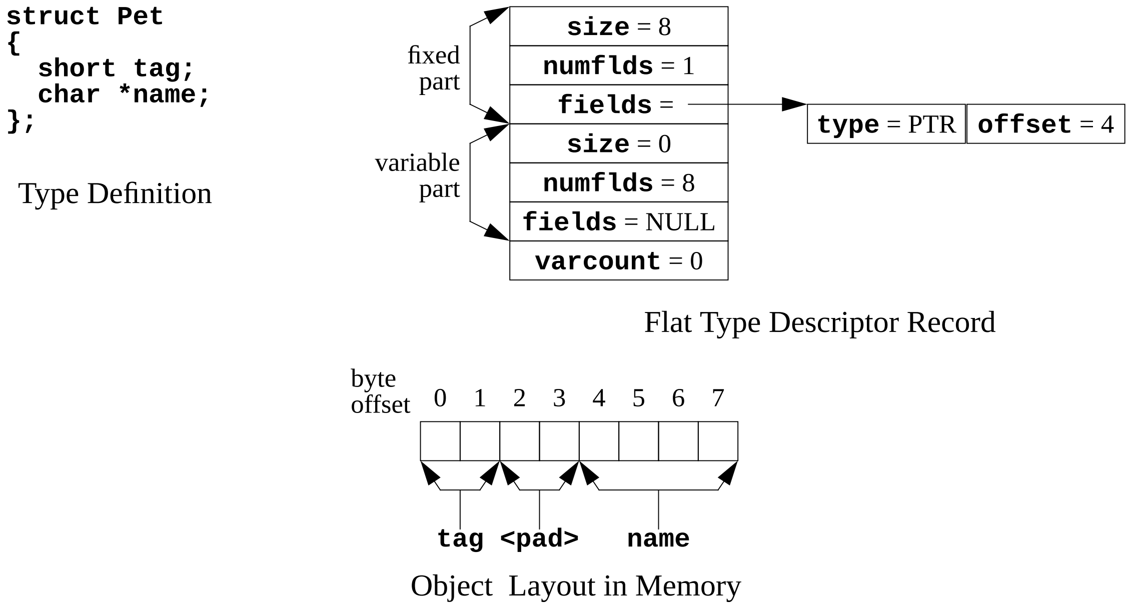
A flat format type descriptor record contains a fixed part, a variable part and an integer that maintains the statically-declared size of the inline array. Each part contains a pointer to an array of integers that represents the pointer offset values within the object. In addition, there are two integers used to maintain the count of entries in the array and the compiler-determined size corresponding to the part. Although the two parts are identical in structure, the semantics of each are quite different; the fixed part maintains information about all components of the type that cannot change in size at run time, while the variable part maintains information about only *one* element of the repeated sub-structure of the object. That is, the variable part maintain information about a *single* element of the inline array, irrespective of the original (statically-declared) size which is maintained separately.

Recall the type definitions provided as part of the example in the previous section. The flat type descriptor record and in-memory object layout for type `Pet` are shown in Figure 6.3. The type does not contain any variable-sized arrays, and hence only the fixed part of the type descriptor record is applicable. It shows that the object is 8 bytes in size, and contains only one pointer at offset 4.[30] This information matches with the in-memory layout of a sample object also shown in the figure. Note that the 2-byte "padding" is automatically added by the compiler to maintain alignment constraints.

Now consider the type `Owner` from our example. This type has a variable-sized array (`pets`) and hence will contain valid information in both fixed and variable parts as shown in Figure 6.4. The information in the fixed part is similar to that described above and can be matched with the in-memory object layout also shown in the figure. We are more interested in the variable part; as can be seen, the information in the variable part for type `Owner` is identical to the information in the fixed part for type `Pet`. This corresponds to the definition of variable parts, which requires them to contain information about a single element of the repeated sub-structure (type `Pet` in our example). Finally, note the last field in the type descriptor record which maintains the statically-declared size of the array. This field is necessary in situations

[29] An array is said to be "inline" if it is contained within the object; such arrays are typically fixed in size because the size of the containing object is fixed (and determined) at compile time.

[30] We assume that word size is 4 bytes, size of short integers is half-word and usual alignment constraints requiring word-size fields to be aligned on word-size boundaries.



```
struct Pet
{
  short tag;
  char *name;
};
```

Figure 6.3: Flat format type descriptor records (simple)

where objects of type `Owner` are themselves used in a variable part of some other object; in such cases, layout rules dictate that the `pets` array in owners *cannot* be variable-sized and must be decoded using the statically-declared array size.

6.6 Performance Characteristics

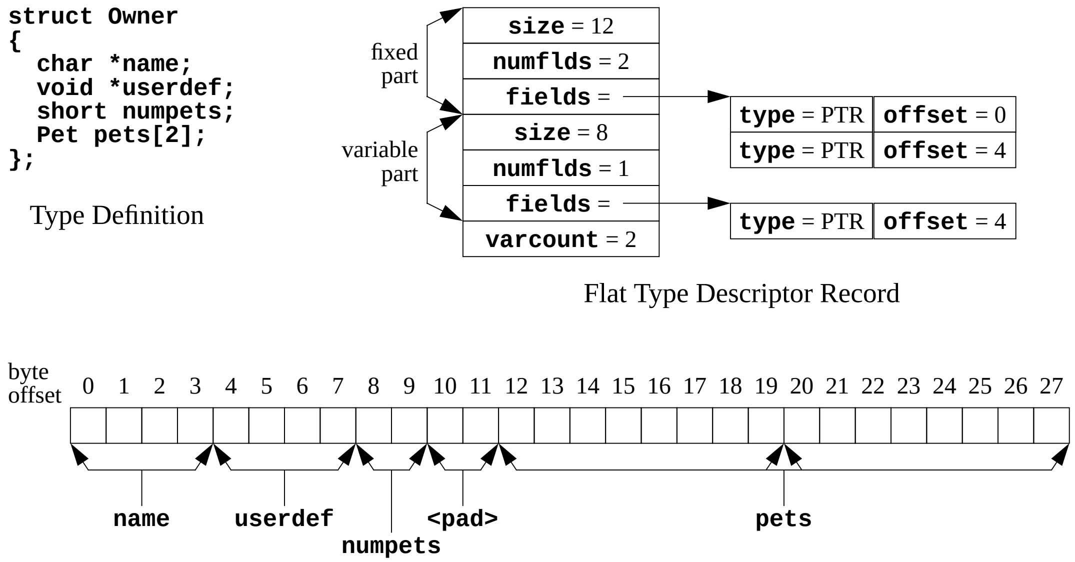
The performance of our RTTD mechanism can be measured in terms of *space* and *time costs* of the system. We divide each of these costs into a *compile-time* component and a *run-time* component based on "when" the cost is incurred. Below we describe these costs in detail and sketch the performance characteristics of our system based on some preliminary results.

6.6.1 Compile-Time Costs

As the name suggests, the compile-time component includes costs that are incurred during "compilation," that is, before the application is actually executed.[31] The compile-time component of the time cost is typically the time required to run the type descriptor generator over the application object file(s) to generate type descriptor records. The space required to store these type descriptor records (typically on disk) constitutes the corresponding compile-time component of the space cost.

As long as compile-time cost components are within "reasonable" limits, they are usually less important than the corresponding run-time cost components. An obvious comparison point for establishing reasonable limits is the corresponding costs for compiling and linking the application itself. That is, the RTTD compile-time costs should be comparable to the cost of

[31] We use the term "compilation" loosely to indicate all steps other than running the application. For example, compiling and linking the application, as well as generating the type descriptor records from the object files, are examples of actions that belong to the compilation phase.



```
struct Owner
{
  char *name;
  void *userdef;
  short numpets;
  Pet pets[2];
};
```

Type Definition

Flat Type Descriptor Record

Object Layout in Memory

Figure 6.4: Flat format type descriptor records (complex)

compiling individual application object files and linking them together to build the executable. In practice, we have found that our costs are significantly less than the actual compilation and linking costs, in terms of both space and time costs.

We measured the costs of building the OO1 benchmark both with and without including the type descriptor generation actions (i.e., the steps shown in Figure 6.1). We found that the normal compilation and linking time for the benchmark took roughly 9 seconds of wall-clock time on a 200MHz Pentium Pro processor running Linux 2.0.x. When type descriptor generation was included in the overall build process, the total time went up to approximately 12 seconds, an increase of 33%. The type descriptor generation actions included ten runs of `o2tdesc` and one run each of `tnamemap`, `tdlink` and `tdfgen`. The ten runs of `o2tdesc` examined and generated hierarchical type graph for about 1700 types, although only 20 types (those that were instantiated persistently) were eventually saved to disk. After linking and duplication elimination through `tdlink`, there were a total of eight types for which type information was saved (both in hierarchical and flat formats).

In terms of space costs, the benchmark executable file was approximately 500KB when debugging information was preserved, and 150KB after recompiling with flags that specified no debugging and additional optimization. In contrast, the final flat format type descriptor records only required roughly 3KB of storage, or 2% of the executable size. Of course, during the actual generation process over multiple object files, the maximum storage requirement was higher (about 20KB) but it amounted to only 4% of the size of the executable with debugging information included (as it must be for type descriptor generation). It should be emphasized that we report the storage cost as a percentage of executable size *only* as a comparison point; in general, the number of type descriptor records and their space requirements will be directly proportional to the number (and corresponding sizes) of types for which run-time type description is desired.

In general, we can conclude that our system does not impose any significant overhead at compile time, both for generating and storing type descriptor records, for an arbitrary user application.

6.6.2 Run-Time Costs

By definition, run-time costs are incurred when the application is actually executed. The run-time component of the time cost usually includes the time to load the type descriptor records into memory as well as the time to access information stored in these type descriptor records. The actual memory usage of the data structures that maintain the type descriptor records in memory is part of the run-time component of the space cost.

With respect to the time cost, the initial loading of type descriptors is considered to be part of the startup costs and is relatively minor compared to other startup costs such as dynamically linking and loading libraries, handling shared objects, etc. The cost of actually decoding the in-memory type descriptor records is the most important cost at run time. The approach used to access information from the type descriptor records is very dependent on the data structures used to represent these records in memory. Section 6.5 described the various formats used to store type information.

We estimate the cost for an aggregate type while temporarily ignoring the variable part in the interest of simplicity. Recall that a fixed part (Figures 6.3 and 6.4) contains two integers and a pointer to an array of integers (i.e., field offsets). Thus, for a type with n pointer fields, the approximate storage cost for the fixed part would be $((2 * sizeof(integer) + 1 * sizeof(pointer)) + (n * sizeof(integer)))$ bytes. Taking the variable part into consideration makes the cost estimation a little trickier. Recall that the variable part is used to maintain information about a single element of the repeated sub-structure; thus, the overhead of including the variable part in the type descriptor record (for the top-level aggregate type) is equivalent to the cost of the fixed part for one element.

6.6.3 Making Decoding Costs Negligible

For pointer swizzling at page fault time, we use flat type descriptor records described earlier to maintain information about the locations of pointer fields within various objects. Due to the nature and structure of the flat representation, the decoding is highly optimized by allowing direct access to the pointer fields based on the offsets recorded in fixed parts. Decoding variable parts is only slightly more complex because it requires iterating over the relevant portion of the object using the type information in the variable part during each iteration; this iterating itself does not add any significant overhead compared to the cost of the actual decoding.

Although this decoding mechanism is relatively fast, it still adds a few tens of instructions *per* object to the overall run-time costs of using type descriptor records. In an effort to further minimize the costs, we have developed an approach that reduces the decoding costs to negligible levels (on the order of cost of a procedure call). The basic idea can be explained with the following analogy. If type descriptor records are viewed as "bytecodes," then the run-time decoding process can be thought of as "bytecode interpretation." Bytecodes can usually be compiled into native (binary) code allowing execution at full speed without requiring any run-time interpretation. Similarly, as described below, we can "compile" the type descriptor records such that the generated code does not require any run-time decoding.

As described in Section 6.3.3, we generate auxiliary source containing initializations of backpatching variables. In a similar vein, when hierarchical type descriptor records are being converted into flat format, we generate code for a procedure corresponding to each type. This procedure embodies the run-time decoding of type descriptor records—for each field of interest, we generate an invocation of a "callback" function that will be responsible for handling the specific actions for that field. The auxiliary code containing these procedures is then compiled and linked into the application, along with the other initialization code. When the type information for an object is required at run time, the top-level procedure for that object's type is invoked instead of actually decoding the corresponding type descriptor record. Note that the end result is still the same as decoding the type descriptor record at run time, but instead of run-time interpretation we have "compiled the type descriptor record" thereby reducing the decoding costs to a single procedure call.

Type descriptor record "compilation" is compatible with variable-sized object even though the actual size of such objects is not known until run time. The "compilation" process is the same as that described above for the fixed part of the type descriptor record. The variable part is also easily handled. By definition, the variable part of the type descriptor record describes a single element of the variable-sized array and hence we simply generate a loop around the callback function invocations corresponding to the "compiled" variable part; the loop termination condition is based on actual run-time object size guaranteeing correct behavior. Again, this approach does not add any more overhead compared to run-time decoding of the type descriptor record.

There is one minor disadvantage of using type descriptor record "compilation" as described above. The basic approach works well as long as entire objects are manipulated, starting at the beginning of each object. However, it is not suitable for objects that must be partially processed (e.g., large objects) because of the static ordering imposed on the callback function invocations. In such situations, the basic approach must be augmented to allow additional control over the callback functions. Of course, a simple workaround is to just fall back to run-time decoding of the type descriptor records. For pointer swizzling at page fault time, because the basic swizzling unit is one page, the problem is likely to arise only for large objects that cross page boundaries; all objects smaller than a page are always swizzled in entirety.

6.7 Current Status and Future Work

We have implemented the type descriptor generator and other mechanisms as described in this chapter. We use these in our Texas Persistent Store (Chapter 4) and a real-time garbage collector for C++ [WJ93]. Currently, we have two versions of the type descriptor generator available: one for most modern Unix systems and the other for OS/2, the only difference between the two being the platform-specific code to parse the debugging information.

Since the debugging information format on different Unix systems varies significantly, we have leveraged code from the GNU debugger, `gdb`, to extract the debugging information. This approach is portable because `gdb` understands several different kinds of object file formats and debugging information formats for various architectures and compilers. Using `gdb` for the platform-specific operations makes the type descriptor generator instantly portable to all architectures supported by `gdb`. Our code uses standard `gdb` routines to parse the debugging information for all types used in the application into in-memory data structures; these data structures are then transformed into type descriptor records.

Note that it is *not* necessary to always use `gdb` for this purpose; it is feasible to implement platform-specific code that extracts the debugging information directly from object files. This approach has been used for adapting our type descriptor generator for OS/2 and IBM VisualAge compiler.[32] Of course, if the compiler already provides low-level object layout information in some form, "adaptor" code can be written to transform the compiler-specific information into type descriptor records.

Our code modules for implementations based on using debugging information are relatively small. For example, the new code that we added for the `gdb`-based type descriptor generator is only around 600 executable lines[33] of C++. Similarly, the OS/2 version is approximately 2000 executable lines of C++. The source code for both versions is publicly available (under GNU GPL) at `ftp://ftp.cs.utexas.edu/pub/garbage/texas`.

Our system has been designed to be easily portable to other compilers and platforms. The only major effort required for porting is providing the platform-specific code to parse the debugging information in object files. We intend for the system to be ported to a variety of PC-based operating systems and compilers. We believe that this should be relatively easy because most PC-based compilers can usually generate debugging information in more than one format; by choosing a few representative formats and providing specific code to parse those formats, it should be possible to support multiple compilers.

6.8 Related Work

Several other techniques have been proposed and implemented for providing RTTD. However, most of these techniques are either specific to a source language or incur significant additional run-time overheads, and do not provide an efficient, general-purpose mechanism to generate and manipulate RTTD.

Two other systems similar to ours have been developed, to our knowledge, both independently; unfortunately, no published descriptions exist for either. Marc Shapiro and his collaborators at INRIA have developed a type description facility, also using code from `gdb`.[34] It also appears that Object Design, Inc. (ODI) developed a similar facility for its ObjectStore persistent object storage system for C++, but details on this proprietary system are not available. (ODI also provides a preprocessor supporting other extensions of C++.)

Other researchers have proposed using special-purpose preprocessors and precompilers in conjunction with user intervention to provide support for RTTD. Edelson [Ede92a] proposes using a precompiler to automatically augment a C++ source program with additional RTTD information for garbage collection. The underlying idea is based on using smart point-

[32] Thanks to Tom Porcaro of IBM–Austin for implementing the platform-specific part for OS/2.

[33] The count excludes blank lines, comments and source lines that do not get compiled into executable code.

[34] Marc Shapiro, personal communication, May 1996.

ers [Ede92b] which are class objects that emulate normal (raw) pointers. The precompiler performs two tasks that are necessary for safe garbage collection for C++, namely finding the *root set*[35] and accurately identifying internal pointers within objects. Because the system uses smart pointers to identify roots, the first task of the precompiler is to parse the source code and define appropriate smart pointer classes to be used (instead of raw pointers) for the root set. The second task of the precompiler is to parse the type definitions and emit a member function for each garbage-collected type to identify the internal pointers within that type.

Detlefs [Det92] describes a modified scheme that is also based on smart pointers. This scheme extends the smart pointer definition further and constrains the programmer to use this interface for all garbage-collected objects. The extended smart pointer definition provides additional actions to be performed when standard pointer operations are invoked on the smart pointer objects. Garbage collection can then be implemented using this extended functionality of smart pointers.

Both these schemes require placing additional restrictions on the user and incur additional run-time overheads for manipulating smart pointers. In addition, Edelson's precompiler is quite similar to a preprocessor because it needs to parse type definitions from the source code, and hence is susceptible to the same problems we described earlier for preprocessors.

Interrante and Linton [IL90] proposed a *Dossier* class as a standard interface for run-time type information in C++. A (preprocessor-style) dossier generator is used to create *Dossier* objects from the source code. Interrante and Linton propose that the language be extended to automatically generate a virtual function for each class to access the appropriate dossier object. If this feature is not provided as part of the language, programmers can provide the information manually for each class. As before, this scheme requires a preprocessor and is also closely dependent on C++ language implementation features (that is, virtual functions).

6.9 Conclusions

We have introduced the term Run-Time Type Description (RTTD) to denote availability of low-level object layout information at run time in contrast to Run-Time Type Identification (RTTI) which is used to access language-level information at run time. We have also described a portable, general-purpose mechanism for generating and manipulating RTTD for high-level languages such as C, C++ and Ada which do not provide this information as a language feature. Our approach does not require special compiler cooperation and allows the programmer to use off-the-shelf high-performance conventional compilers.

We have presented type descriptor records for representing the low-level layout information at run time. We have developed a novel approach to build type descriptor records by using the debugging information generated by modern compilers. Since debugging information format typically does not depend on a specific source language or compiler, our approach works for combinations of different languages and normal compilers.

We have implemented a type descriptor generator for C++ to illustrate the issues involved in providing RTTD for a specific language. For Unix systems, we have leveraged code

[35] An application must maintain entry pointers into various data structures; these pointers are known as *roots* and collectively referred to as the *root set*.

from the GNU debugger, gdb, to provide the platform-specific part of the type descriptor generator. This approach is portable because gdb understands several different debugging information formats on various architectures. A version of the type descriptor generator for OS/2 (and VisualAge compiler), which uses non-gdb code for parsing the debugging information, is also available.

We described the storage model and various formats used to store type descriptor records in memory at run time. Depending on the application requirements, it is possible to convert full-fledged (hierarchical) type descriptor records into a simpler, flat format to allow for faster decoding. We have also presented an approach that can be used to effectively reduce the run-time decoding costs to zero. Preliminary performance results of our untuned implementation have shown that both compile-time and run-time costs of the RTTD mechanism are not excessive.

In general, we believe that a facility for accessing implementation-level type information at run time is useful, and possibly quite necessary, for a variety of utility system extensions. The RTTD mechanism as described in this chapter provides a framework for implementing run-time type description for a variety of high-level languages that do not offer it as a standard, builtin language feature. We believe that our approach of using the debugging information is highly portable and is preferable to other techniques because it does not depend on knowledge of the source language or specific compilers.

Chapter 7

Interactions with Operating Systems

7.1 Introduction

It is evident from earlier discussion that Texas interacts strongly with low-level features provided by the operating system. Most of this interaction stems from the use of virtual memory protection and access-protection violation handling for implementing coarse-grained address translation. In addition to Texas, there are many other useful system-level extensions and libraries (for example, garbage collectors [AEL88, Wil97], distributed shared virtual memory systems [Li86, LH89], virtual memory tracing and compression facilities [WKBK97, WKB97a], advanced profiling, etc.) that closely interact with the operating system.

We believe that operating system implementors should take interactions of such low-level systems and libraries into consideration when designing new systems. That is, a modern operating system should provide sufficient "hooks" to allow various extensions to be implemented efficiently outside the kernel but still be able to exploit the low-level features. This would greatly improve the portability of such systems while maintaining their general high performance characteristics.

In this chapter, we discuss several areas of operating system interactions that are important for efficiently implementing system-level extensions. Although this discussion is based mostly on our experience with implementing pointer swizzling at page fault time in Texas, the same issues should also be applicable to other systems mentioned earlier. We are primarily interested in interactions with the virtual memory system, that is, memory allocation, protection, and replacement. Most operating systems allow some degree of flexibility in this area, but further control is desirable and achievable. We also briefly describe other related issues such as efficient handling of access-protection violations that are generated by attempts to access protected memory. Although we focus mostly on Unix-like operating systems, the basic ideas are also applicable to other modern operating systems.

7.1.1 Background: Virtual Memory

Virtual memory [Den70, KELS82] was originally designed simply to manage two distinct levels of memory hierarchies—main memory and secondary storage—while giving the applications an illusion of a single level of storage that is larger than the size of the available physical memory.

Today, however, virtual memory is essential for smooth operation of computer hardware and software. Modern operating systems provide various primitives that allow application programs to interact with (and exploit) the virtual memory system; such interactions can range from a simple allocation (and deallocation) model to advanced interfaces such as shared memory [Li86, LH89] and memory inheritance as supported by Mach [BKLL93].

In the absence of a virtual memory system, if an application program outgrew the size of available memory, it was the programmer's responsibility to manually split the code into *overlays* that had to be explicitly controlled. In contrast, a virtual memory system provides seamless program execution by automatically handling data transfer between main memory and secondary storage as necessary without involving the programmer.

7.1.2 Basic Terminology

Virtual memory is typically implemented by allowing each process to have its own *virtual address space* that is distinct from all other processes. Pages of the virtual address space are mapped into the *physical address space* (i.e., main memory) as an application accesses data on those pages. The application is allowed to access data only via virtual addresses, and is never made aware of the actual physical location of the data in main memory. This makes the pages easily relocatable depending on the availability of the physical memory. It is the job of the operating system's memory manager to maintain current mappings between virtual and physical addresses and automatically translate between the two as necessary.

The collection of pages accessed "together" (i.e., at roughly the same time) by an application is usually called the *working set*. As the application continues execution through different phases, its working set changes because some pages are no longer accessed and new pages, not referenced before, are accessed. In order to maximize main memory utilization, the memory manager implements a *page replacement policy* for pages in main memory. As the name suggests, old pages that are no longer in use are removed from main memory and replaced with newer pages. This is usually accomplished as follows. Each page of virtual address space referenced by an application typically has a corresponding page of *backing store* associated with it; the backing store, which may be assigned lazily (i.e., only when necessary), is where the page is actually stored when it is replaced from main memory. A special disk partition, called *swap space* or *paging space*, is usually configured on secondary storage to serve as the backing store although it is certainly possible to use a conventional file or the main memory (or even secondary storage) of a remote host on the network for this purpose. The process of transferring data between memory and backing store to maximize the memory utilization is often known as *paging* (or *remote paging*, if the backing store is on a different host).

7.2 Virtual Memory Allocation

The fundamental principle behind virtual memory allocation is to allow applications to allocate more memory than is physically available on the machine. Theoretically, it is possible to allocate as much virtual memory as is addressable by the hardware word size. In practice, however, this varies based on the operating system implementation and is usually limited by the maximum secondary storage configured as swap space on the system.

We use the phrase *virtual memory allocation* quite loosely throughout this chapter. The usual connotation refers to allocation of *both* the virtual address space and the corresponding backing store. However, we are also interested in allocation of only the address space without having any physical memory associated with it; we highlight the distinction as necessary in the rest of the chapter.

This section discusses various issues related to virtual memory allocation, including the differences between storage space and address space allocation, and various primitives that are available for this purpose. Specifically, we describe standard Unix primitives for virtual memory allocation and their overall performance characteristics for two major operating systems, Linux and Solaris. Section 7.3 contains further details about these primitives and swap space allocation.

7.2.1 Storage Space vs. Address Space Allocation

Although the distinction is typically not exposed to normal users, we believe that it is very important for system implementors to distinguish between allocation of virtual *storage space* and allocation of virtual *address space*. This distinction is very relevant to pointer swizzling at page fault time, and possibly also to other low-level system algorithms that benefit from additional control over virtual memory mechanisms.

We define *virtual storage space allocation* as allocation of both the virtual address space *and* the corresponding backing store for that address space. In contrast, *virtual address space allocation* simply allocates the address space, but does *not* actually assign any backing store for that chunk of address space. Later, if the allocated space needs to be used by the application, a page of backing store is allocated and assigned (either by the operating system or by the application itself) before the address space can be referenced. Alternatively, a lazy approach is to "reserve" a page of backing store but not actually allocate it until the page is ready to be written out; an even lazier approach is to wait until the page is ready to be evicted.

Although pointer swizzling at page fault time functions correctly with either address space or storage space allocation strategies, the former is preferable for pages that are reserved (access-protected) during the normal course of operation. This is because many reserved pages may never be referenced by the application, thus not requiring any data to be loaded into those pages—allocating backing storage for such pages would obviously be wasteful because it will never be used. In contrast, virtual address space allocation strategy works well with our basic swizzling mechanism. Pages that are referenced by the application cause our protection fault handler to be invoked; the handler then assigns backing store for the faulted-on page and loads the data from the persistent store before returning control back to the application. Using this approach, backing storage is assigned only for pages that are actually used by the application.

7.2.2 Virtual Memory Primitives

We are primarily interested in low-level *primitives* (for example, the `sbrk` primitive available on various Unix systems) that are used to allocate virtual memory from the operating systems. We are not interested in high-level standard library *routines* such as `malloc` or `free` which represent an implementation of some allocation policy (such as first-fit or best-fit [WJNB95]) on top of the low-level primitives. Of course, the flexibility and features of the underlying virtual memory primitives are likely to guide the implementation choices of these high-level allocation mechanisms.

Most modern Unix flavors provide two standard primitives, `sbrk` and `mmap`, for virtual memory allocation. Historically, most Unix variants have always supported `sbrk` in some form or other, while `mmap` is a newer feature that originally existed in 4.2BSD but has since appeared in other variants within the last decade [GMS87]. All currently popular flavors of Unix for workstations and PCs now support `mmap`, albeit with minor differences in the interface. The exact behavior of the primitives may be slightly different across variants, but the basic functionality remains the same.[1]

Although we classify both `sbrk` and `mmap` in the same class of primitives for the purposes of the current discussion, their interface and implementation details vary significantly compared to each other. We briefly describe each of them below before comparing the two with respect to their flexibility and features. Note that the rest of this section focuses primarily on primitives provided in Unix-based operating systems, but the basic ideas are also applicable to virtual memory primitives in other modern operating systems such as Windows NT and OS/2.

The `sbrk` Primitive

Most high-level allocation mechanisms commonly use `sbrk` as the underlying virtual memory primitive because it provides a simple mechanism to extend the data region of an application. The interface is very simple—the caller requests allocation in number of bytes; the request is then satisfied by allocating as many bytes as necessary using normal swap space as the backing store for the allocated address space.

Figure 7.1 shows the classic view of a process' virtual address space. It is typically divided into four regions (sometimes also called *segments*[2]), namely *text*, *data*, *heap*, and *stack*. The text region maintains the code segment of the process and the data segment contains both the initialized and uninitialized data for the process. The heap segment represents the dynamically-allocated data; as shown in the figure, it starts beyond the data segment and grows "upwards" in address space (i.e., increasing address values). In contrast, the stack segment usually starts at a high address and grows "downwards."

The last-allocated position in address space is always represented by the *break* point, also shown in the figure. The `sbrk` actions are very closely related to the notion of this break point. As more memory is dynamically allocated from the operating system using `sbrk`, the break point is moved appropriately to maintain the invariant as per its definition. Note that the break point moves monotonically upwards in address space as more memory is allocated from the operating system.

[1] This is typical of general Unix programming where one has to deal with minor but tedious incompatibilities across different variants. The usual approach is to use preprocessor directives (e.g., `#ifdef`, etc.) to customize the application source for each variant.

[2] It should be emphasized that a segment in this context refers to a mapping between a process' address space and the backing store, and *not* to segmented addressing.

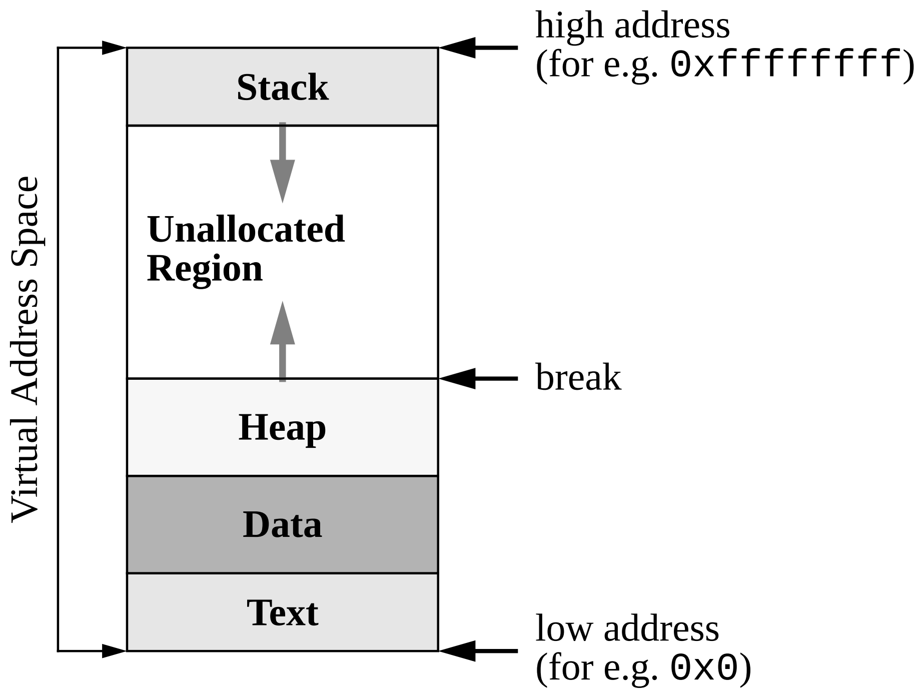


Figure 7.1: Address space of a process

The `mmap` Primitive

Unlike `sbrk`, `mmap` is more flexible and provides several additional features. The basic interface allows applications to "map" a file (or parts of a file) into the virtual address space such that the file itself acts as backing store for that address space. The data, if any, from the file is made available directly in the corresponding virtual address space without requiring any explicit I/O requests on the file. When the file is eventually unmapped from memory, in the usual case, any modified data in the virtual memory is automatically updated in the file. In addition to this mapping facility, `mmap` also offers other features such as memory protection (*a la* `mprotect`), mapping the file copy-on-write, etc. These features, although useful, are not immediately relevant to the current discussion and are ignored for the moment.

Some operating systems (for example, Linux and AIX) provide additional features that allow applications to map *anonymous regions* instead of files. An anonymous region is usually just a chunk of normal swap space that is used as backing store instead of a file in the file system.[3] Use of anonymous regions allows essentially the same semantics as `sbrk`, but with the additional capabilities of `mmap`.

Note that `mmap` actions are not directly related to the break point shown in Figure 7.1.[4] Instead, `mmap` can be used to map a file (almost) anywhere in the heap segment of the address space. There are two ways of selecting the address range for the mapping. By default, the operating system selects the address range using its own heuristics and the existing mappings.

[3] The term "anonymous" is used because the backing store (i.e., swap space) cannot be referenced through a file, and hence does not have a "name."

[4] For some systems, `sbrk` may actually be implemented in terms of `mmap` in the kernel, but this fact is not generally apparent or exposed to the user.

In addition, most implementation also allow the programmer to override the default and explicitly specify the *exact* address range for the mapping (typically by using the MAP_FIXED option with mmap). However, the call may fail if the specified address is “unsuitable” for some reason (for example, if it is not page-aligned). The use of this option is usually discouraged because it results in unportable code. In general, it is preferable to let the operating system select the address range in order to avoid conflicts with other allocated data. However, the facility is provided for applications that may require (and benefit from) explicit user selection of the address range.

Comparing sbrk and mmap

It is obvious from the above discussion that sbrk provides a simple functionality and interface, and mmap can provide the same semantics while offering additional capabilities. We believe that mmap is preferable in general because it also provides additional control over the allocated memory. Such control is desirable for many systems (especially, pointer swizzling at page fault time) that interact extensively with the operating system.

Most operating systems also provide a primitive, munmap, that complements the functionality of mmap—the purpose of this primitive is to “unmap” the memory previously mapped by mmap. In other words, it breaks the association between the backing store (which can be either a file or an anonymous region) and the specified range of virtual address space. If the backing store is represented by an anonymous region, then munmap is an excellent way to reclaim unused swap space from the application and return it to the operating system. There is no similar primitive corresponding to sbrk, that is, there is no convenient way to reclaim swap space corresponding to memory allocated using sbrk until the application finishes execution.

Although we prefer mmap over sbrk, we are not necessarily advocating that a persistent store be mapped directly into virtual memory. In fact, it may not be advisable to do this because the persistent store may either be stored in compressed storage or cached across a network, making it unsuitable for direct mapping via mmap. Instead, we favor mmap only for allocating virtual address space with better control over backing storage.

7.2.3 Performance of Virtual Memory Primitives

So far, we have discussed the differences between sbrk and mmap in terms their functionality and the flexibility afforded by different features of each primitive. We now describe results based on experiments that we conducted for measuring the performance of each primitive.

Since both primitives allocate address space from the operating system, every call to either primitive causes control to cross kernel boundary, which in turn causes the execution to switch from normal user mode to a privileged mode. This is obviously more expensive than non-kernel calls that operate only in unprivileged mode. One solution to reduce the performance penalty is to amortize the cost of multiple calls by *batching* several requests into a single large request. Such batching can be implemented by using simple application-level *buffering* as described below.

The application maintains a batch of (contiguous) virtual address space pages in a *batch* *buffer* of predetermined size (the *batch size*).[5] The batch buffer is empty on startup; the first allocation request invokes the chosen primitive to allocate as many pages of address space as the selected batch size, and the batch buffer is now full. The original request is then satisfied by removing one (or more) pages from the batch buffer, and control returns to the application. As future allocation requests arrive, we first check whether there are enough pages available in the batch buffer. If so, the request can be satisfied from the batch buffer and there is no need to switch into kernel mode. However, if the request cannot be satisfied from the batch buffer, or if the batch buffer is empty, the full cost of invoking the primitive is incurred.

The choice of batch size involves a tradeoff between space and time costs. The larger the batch size used, more memory is "preallocated" as part of the batch buffer even though the application may not use it all. On the other hand, the smaller the batch size, more frequent is the need to cross kernel boundary and switch execution modes, affecting the overall performance. We studied various choices for the batch size and found that relatively small batch sizes (e.g., 4 to 8 pages) provide significant benefits over no batching, and a moderate batch size (e.g., 32 or 64) usually provides almost all the benefit of batching, approaching diminishing returns for larger sizes. Below, we present the experimental design and results in further detail.

Experimental Design

We used two benchmark suites for measuring overall performance of the virtual memory primitives and the effect of batching. One of these is a set of microbenchmarks designed specifically to measure the absolute performance of the primitives with several different batch sizes (ranging from 0 through 128). The other comprises of standard OO1 benchmark forward traversal operations that were also used extensively to measure the performance of Texas (Chapter 5). We use the results from the microbenchmarks to select a set of "interesting" batch sizes for the OO1 benchmark traversal operation experiments.

The microbenchmarks are relatively simple, designed to only measure the performance of the primitives in isolation. Each run of the benchmark repeatedly calls the specific primitive under consideration in a tight loop, measures the time for the entire loop, and finally divides the total time by the number of iterations to obtain the average time per invocation of the primitive. Although each iteration of the loop always requests only a single page of virtual address space, for batch sizes greater than one, we implemented the batching mechanism described earlier to group multiple small requests into a single large one. We iterate for a total of 5000 times, and record the time using a clock cycle timer to get an accurate measurement.[6] Each microbenchmark is run multiple times, such that the batch size is one larger than in the previous run, starting at 1 (i.e., no batching) up to a maximum of 128.

Using the results from the microbenchmarks, we select a set of *four* "interesting" batch sizes (specifically, 1, 4, 16 and 64) for each of the two primitives, for a total of eight possible

[5] Note that we buffer the address space allocation itself, (i.e., only the virtual addresses), *not* the actual contents of virtual memory, as it would be in traditional buffering techniques.

[6] Although the batching affects mostly system time only (by reducing the kernel boundary crossings), we use a real-time cycle timer because its resolution is *significantly* better than any CPU-time timer. We expect real time to be a very close approximation of CPU time in this case because the microbenchmarks do not incur any system- or user-level overhead other than the primitives.

combinations for allocating virtual address space during the OO1 benchmark traversal operation. For each of these combinations, we generated a unique version of Texas that uses the specific primitive and batch size for allocating new address space. Each version is then linked with the same benchmark code, giving us a suite of eight benchmark traversal operations. As before, we ran the benchmark for both the small and large databases, and measured the time for each traversal.

We use the same hardware (a 200MHz Intel Pentium Pro processor with 32MB RAM) and operating systems (Linux 2.x and Solaris 2.5) that were used for Texas performance measurements (Chapter 5). We first present the results from our microbenchmarks before presenting the results from the OO1 benchmark traversals using the representative set of batch sizes.

Experimental Results: Microbenchmarks

In addition to the plots for the `sbrk` and `mmap` primitives (labeled accordingly), the microbenchmarks results also include a third plot labeled `fixedmmap`. This corresponds to a variant of the `mmap` primitive that allows a programmer to explicitly specify the address range to be used for the mapping via the `MAP_FIXED` option. Note that we provide the performance results for fixed-map variant of `mmap` only for the sake of comparison. The use of this variant is not practical because it places additional restrictions on the application and results in unportable code. Specifically, there is no portable way to determine a "safe" address range across different operating systems, or even for the same operating system on different platforms, and hence the use of `MAP_FIXED` is usually discouraged.

Figure 7.2 shows the overall performance of both primitives (and the fixed-map variant). The Y-axis represents the average cost (in clock cycles) of calling the primitive for different batch sizes enumerated along the X-axis. Note that even relatively small batch sizes (e.g., between four and eight) reduce the average per-call cost by a factor of 4 to 10. As expected, the cost decreases as the batch size is further increased, exponentially reaching a point of diminishing returns. Zooming in on the lower part of the curves (Figure 7.3), we notice that the costs of both primitives are similar, thus making `mmap` preferable to `sbrk` because of its added flexibility.

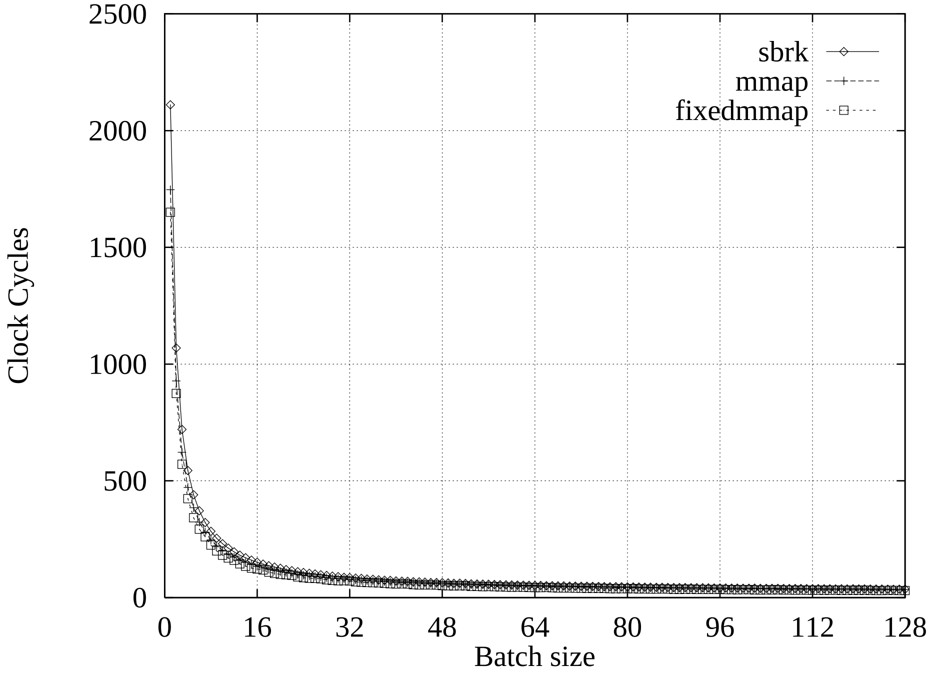


Figure 7.2: Performance of virtual memory primitives (Linux)

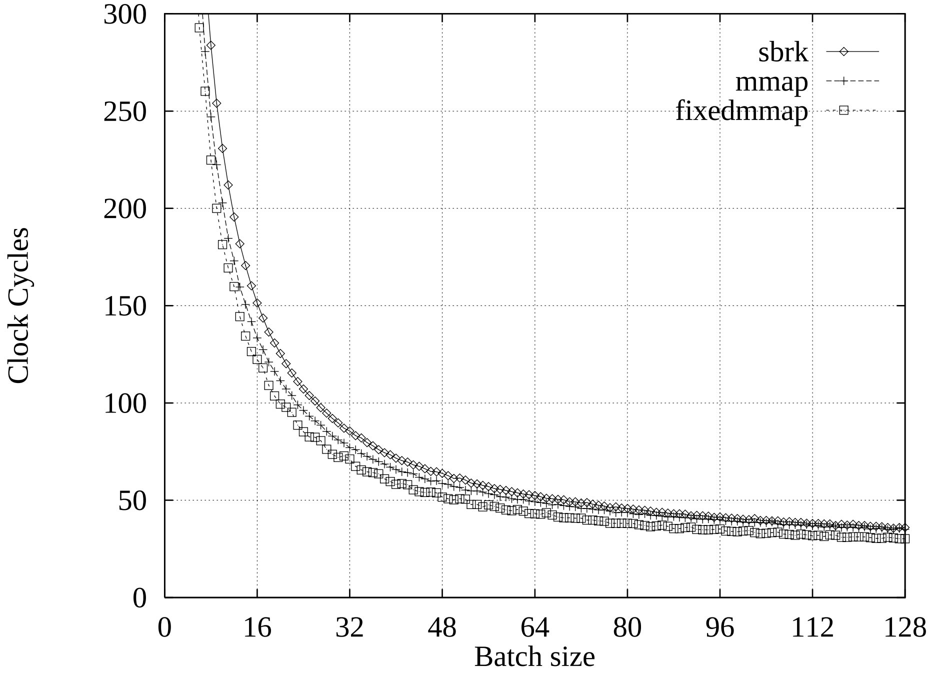


Figure 7.3: Performance of virtual memory primitives, zoom (Linux)

Next, we present the same results for Solaris, and note some surprisingly unusual behavior for both primitives. Figure 7.4 shows the overall performance for all different batch sizes. The first thing to notice from the figure is that, with no batching, `mmap` is extremely expensive on Solaris; the overall cost is approximately *20 times* that of either `sbrk` or the fixed-map variant of `mmap`. However, a batch size as small as four is sufficient to reduce this high cost to within a factor of two of the cost of `sbrk` without batching. These results, when compared to the Linux results (Figure 7.2), also clearly show that Solaris is uniformly slower as far as pure performance of the virtual memory primitives is concerned. We believe that this is probably due to some gross inefficiency in the kernel implementation that should not be too difficult to overcome.

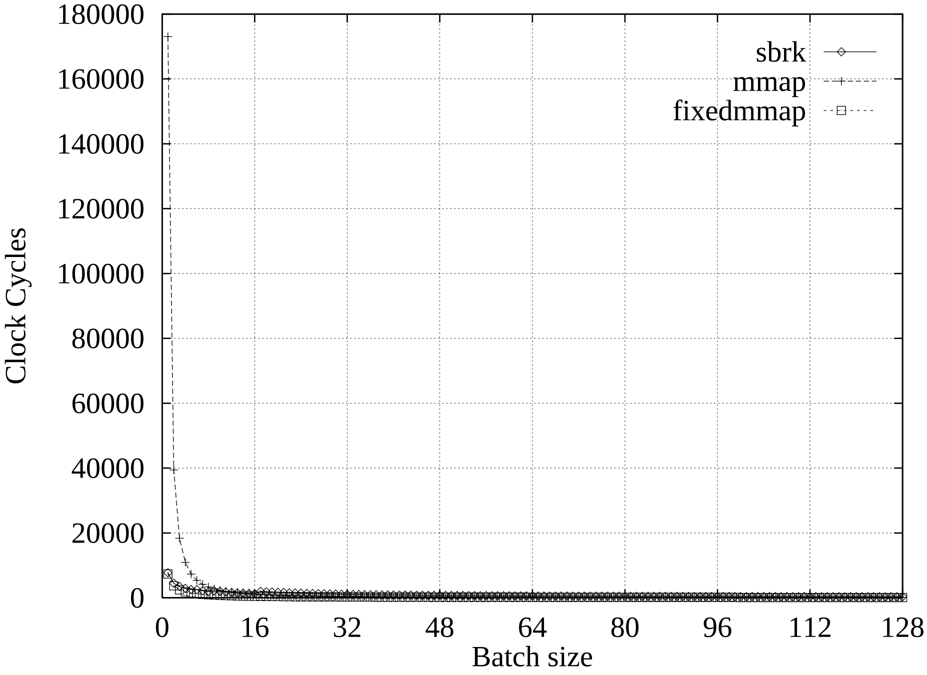


Figure 7.4: Performance of virtual memory primitives (Solaris)

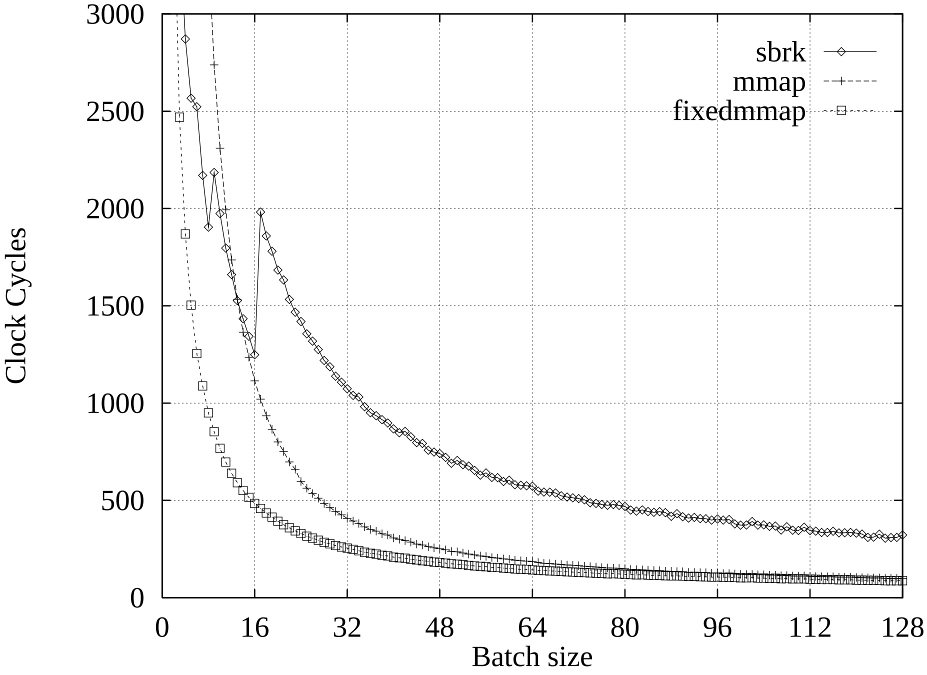


Figure 7.5: Performance of virtual memory primitives, zoom (Solaris)

As before, we zoom in on the lower part of the curves (Figure 7.5). However, unlike the Linux results (Figure 7.3), we notice some unusual behavior in the plot corresponding to the performance of `sbrk`. In particular, at batch sizes 9 and 17, there are sudden jumps in the average cost per call of the primitive, forming "sawtooth-like" shapes in the plot. This is quite an unusual feature, and we believe that it is closely related to either an internal data structure or a particular algorithm used in the implementation of the primitive. Given access to kernel sources or other related implementation details, it should be relatively easy to confirm this hypothesis.

Finally, there is one more useful piece of information that can be derived from the results. Specifically, we notice that both variants of `mmap` eventually converge to approximately the same average cost per call at a moderately large batch size. This is quite reassuring because it indicates that normal `mmap` usage can have a performance that is equivalent to the faster fixed-map variant which is not suitable for practical use. At the same time, `sbrk` continues to be about twice as expensive as `mmap`, even with a batch size as large as 128. We believe that these observations further justify our position that `mmap` should be selected as the primitive of choice for virtual memory allocation.

Experimental Results: OO1 Benchmark Traversal Operations

The microbenchmarks results have clearly shown that batching multiple allocation requests together provides a significant improvement in the overall performance of the virtual memory primitives. In this section, we present the results for the OO1 benchmark forward traversal operations using batched variants of the two primitives for address space allocation. In particular, we present the results for traversal on the large database on Solaris using four different batch sizes for each primitive.

Before showing the actual traversal results, we briefly revisit some characteristics of the OO1 benchmark that must be kept in mind when comparing performance of the virtual memory primitives and their batched variants. Recall that, on average, every tenth pointer in the benchmark database references an arbitrary part object because of the randomized interconnections. This causes many pages of address space to be reserved during the initial traversals as a lot of new data is loaded into memory and swizzled. However, as the execution progresses into later warm traversals, we find that pages corresponding to most newly-swizzled pointers have already been either reserved or loaded into memory during an earlier traversal and no further action is necessary. Hence, most of the new address space reservation happens only during the first few traversals when pointers into new pages (not seen before) are swizzled.

Figure 7.6 plots the number of pages swizzled and number of pages reserved for each traversal of a traversal set run on the large database. It is clear from this figure that new pages are swizzled throughout the entire set of warm traversals but most of the address space reservation indeed occurs only within the first 15 traversals. As such, these early traversals are obviously the most relevant for performance comparison of the virtual memory primitives and their batched variants.

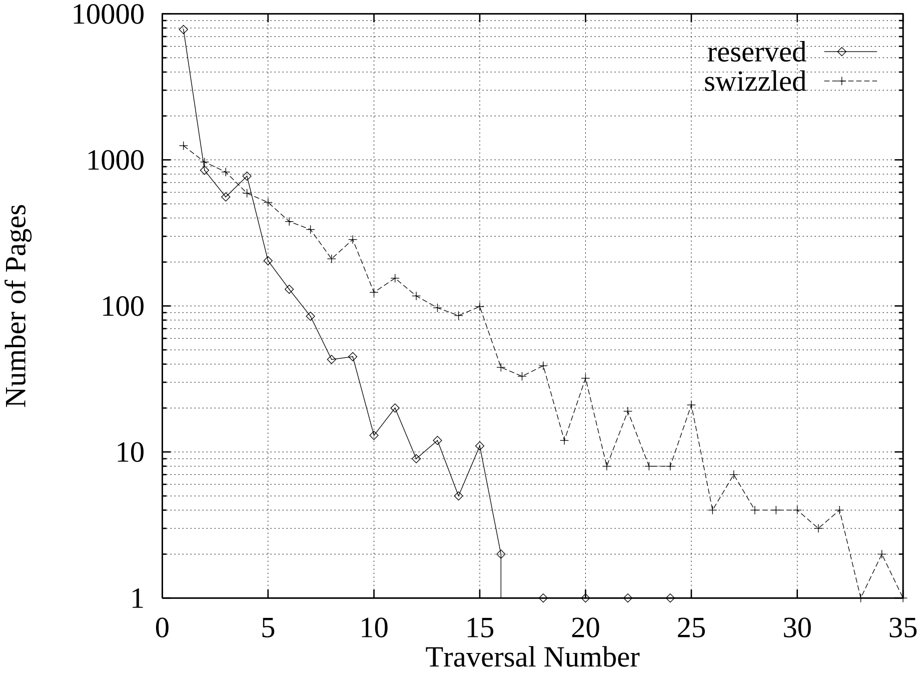


Figure 7.6: Pages swizzled/reserved during all traversals, large database (Solaris)

Given this background, we now present the traversal results for each of the primitives. In the various figures below, the X-axis represents different traversals (i.e., cold, warm and hot iterations) of a single traversal set and the Y-axis represents the CPU time[7] (in milliseconds) for each traversal. We plot CPU time instead of total real time because the batching affects mostly system time (and to a minor extent, user time) since it primarily reduces the number of switches between user and kernel mode. Unfortunately, most operating systems do not provide a high-resolution timer for measuring CPU time; typical resolutions are on the order of several milliseconds, which is very coarse for our purposes since most of the overheads are fairly small. This is not as severe as it seems at first because we are mostly interested only in the initial traversals which contain a lot of swizzling and new address space reservation such that the cumulative times are large enough and therefore less likely to be affected by the coarse granularity of the timers.[8]

Figure 7.7 shows the CPU time for all traversals of the OO1 benchmark forward traversal set for the large database, using `mmap` with different batch sizes for allocating virtual address space during swizzling. It is obvious that `mmap` with no batching is very expensive, and even a batch size as small as 4 improves the performance by a factor of six (or more). This observation is in line with earlier results obtained from the microbenchmarks (Figure 7.4).

[7] We refer to the sum of *user* and *system* time as the *CPU time*.

[8] Note that using real time, as we did for the microbenchmarks, is not sufficient in this situation because the traversal contains computation and I/O components that interfere with performance measurements of only the primitives.

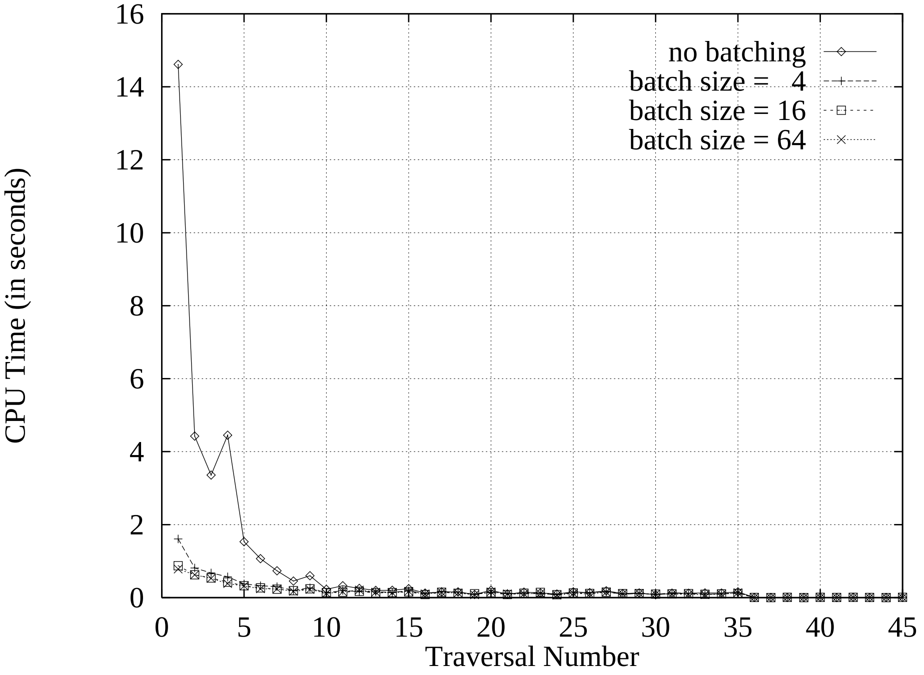


Figure 7.7: CPU times for all traversals using `mmap`, large database (Solaris)

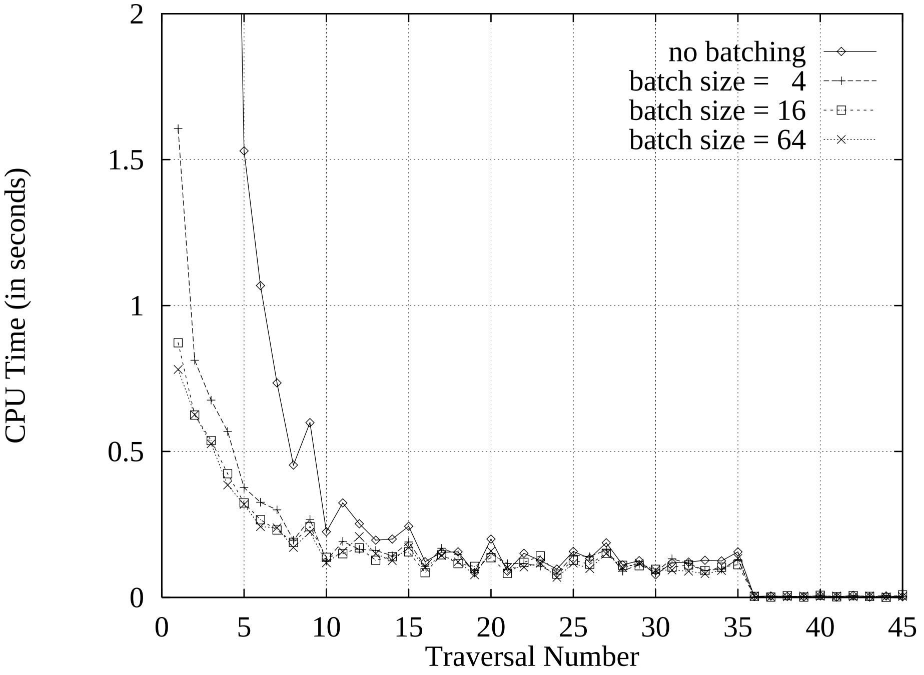


Figure 7.8: CPU times for all traversals using `mmap`, large database, zoom (Solaris)

Zooming in on the lower part of the curves (Figure 7.8) confirms that increasing the batch size reduces the overall cost as expected, eventually reaching diminishing returns. Again, recall that the first 10 to 15 traversals are the most relevant for new address space reservation and meaningful comparison of performance improvements due to batching.

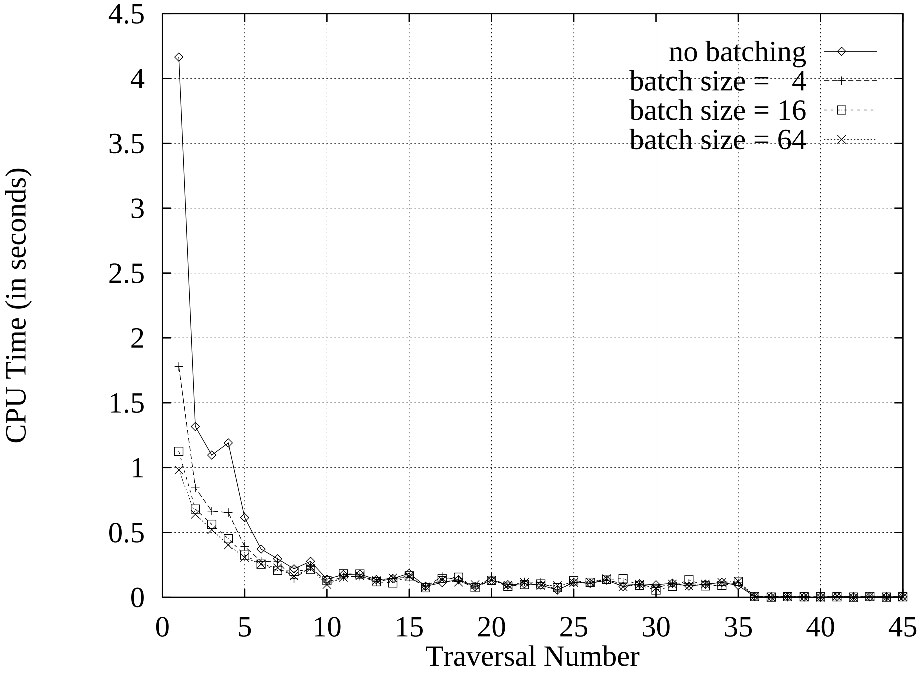


Figure 7.9: CPU times for all traversals using `sbrk`, large database (Solaris)

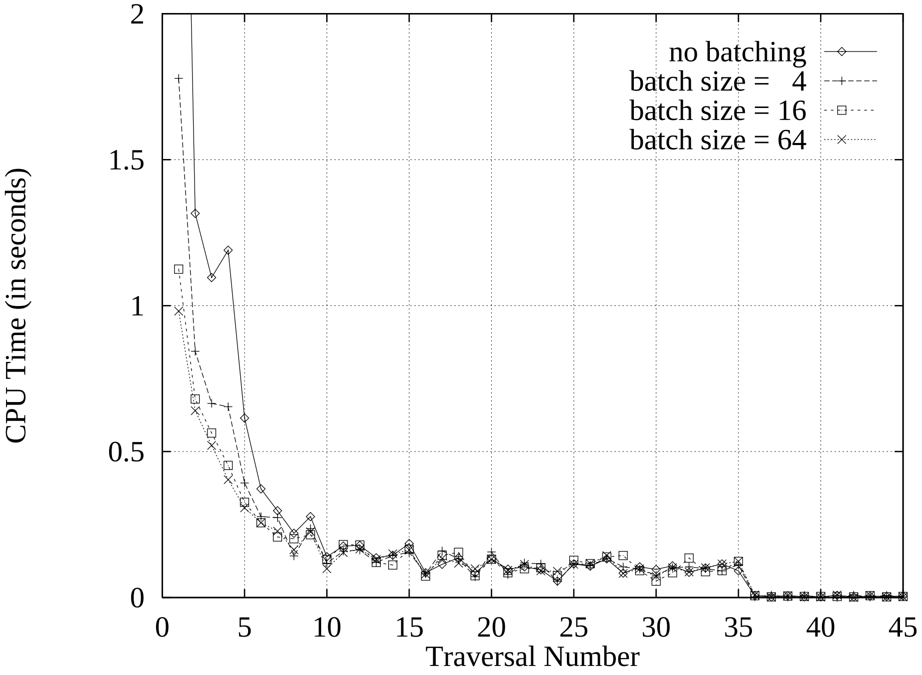


Figure 7.10: CPU times for all traversals using `sbrk`, large database, zoom (Solaris)

Figures 7.9 and 7.10 show the corresponding traversal results using `sbrk` instead of `mmap` for allocating the virtual address space for reserved pages. As before, we note that `sbrk` without any batching is more expensive than when any batching—even as small as four pages—is used. Again, this corresponds to the results obtained from the corresponding microbenchmarks (Figure 7.4). However, the performance improvement for `sbrk` is only between a factor of 2 and 4, smaller than the absolute numbers for mmap. However, this is not surprising because we know from the microbenchmarks results that the performance of mmap without batching is *extremely* bad compared to that of sbrk. This is also evident from the plots labeled "nobatch" in Figures 7.7 and 7.9—at least for the first ten traversals—where the mmap version is consistently more expensive than the sbrk version by a factor of 3 (or more).

The results for traversals on the small database are not reported here because of two reasons. First, they do not show any interesting behavior that is not already obvious from the results presented for the large database and second, only the first four traversals on the small database are relevant for performance comparisons (i.e., only those traversals had new address space reservation actions) making it harder to measure any significant performance improvements. In the same vein, we do not present any Linux results either, because they showed similar overall behavior with one important distinction. The Linux results for both sbrk and mmap did not show as large a difference as the results obtained on Solaris; instead, the results were mostly similar for both primitives. Of course, this is also in line with the conclusions derived from the microbenchmarks results presented earlier.

Discussion

The empirical results presented above clearly show that the *benefit* of batching multiple small requests into a single larger request is measurable performance improvement over using individual requests. Further, we notice that batch sizes do not have to be very large to be effective. Specifically, a batch size of 16 pages usually provides most of the benefit, while a batch size of 64 provides almost all the benefit of batching and quickly approaches the point of diminishing returns for larger sizes beyond 64.

The *cost* of batching is the amount of memory "wasted" due to preallocation when the application uses only part of (or none of) the preallocated memory. With a typical virtual memory page size of 4KB on current systems, batching would preallocate anywhere between 64KB (16 pages) and 256KB (64 pages) of backing store. These numbers are much smaller compared to today's typical main memory sizes of 16MB to 64MB (or more), and swap space sizes that range in hundreds of megabytes. Thus we conclude that batching is useful and can provide substantial performance improvements without equivalent increases in the overall cost. Note that some high-level allocation library routines (e.g., malloc, free, etc.) may already be using a variation of the batching technique described above. However, in general, it would be useful to provide some batching directly through the virtual memory primitives themselves because their implementation is more tightly coupled with the operating system.

Another interesting conclusion that can be derived from the empirical results is regarding the performance of Solaris. We have seen that the virtual memory primitives under Solaris are several times slower than those under Linux, on identical hardware setups. It can be argued that this is not a fair comparison because Solaris implements a layered VM architecture [GMS87] which affords cleaner abstractions and better portability, but also has an impact on the overall performance. However, we believe that even after allowing for layering and abstractions, Solaris is likely to be slower than Linux (which has also been ported to several different architectures) by a factor of at least two. We have also found that user-level protection fault handling is another area where Solaris performance is worse than expected.

7.3 Issues in Swap Space Allocation

As described earlier, we actively distinguish between allocation of virtual address space and allocation of virtual storage space for a given application based on whether backing store is associated with the allocated address space. Most user applications do not know (or care) about this distinction because their typical use always requires swap space to be allocated. However, the basic nature of pointer swizzling at page fault time warrants additional control over virtual memory allocation, and lazy allocation of backing store.

A naive approach would allocate swap space as soon as a page of address space is allocated, irrespective of whether the application has referenced the page or has any data on the page; we call this *eager allocation* of swap space. In contrast, a smarter approach would recognize that it is unnecessary to save a page to backing store as long as the page has not been referenced by the application. Based on this heuristic, swap space allocation can be deferred until the page has been referenced (read or written); this is called *lazy allocation.* A potential problem with lazy allocation is that it may cause resources (in this case, the swap space) to be "overcommitted." Applications may fail if they attempt to access all of the allocated memory and sufficient swap space is not available to provide the required backing storage.

In this section, we discuss how the additional control over backing storage allocation is beneficial for our purposes (specifically for pointer swizzling at page fault time), and useful in general for other low-level algorithms which interact with the virtual memory system. We also describe how existing implementations of virtual memory primitives on various modern operating systems handle allocation of swap space. Finally, we end with suggestions for some possible improvements that would be useful for various applications.

7.3.1 PS@PFT and Swap Space Allocation

As described in Chapter 3, we use virtual memory protection techniques to access-protect the address space corresponding to reserved pages.[9] The memory protection ensures that an application cannot attempt to use data from these pages without causing an access-protection violation. This violation is fielded by our handler which then locates and loads the corresponding data from the persistent store. Thus we are guaranteed that reserved pages can never be used by the application without our handler receiving a notification of such access.

It follows that when pages are reserved during the course of swizzling, it is *not* necessary to allocate any backing store for those pages. That is, we need to allocate only virtual address space for reserved pages, without consuming any real memory or swap space for those (unused) pages. Since we are guaranteed that the fault handler will always gain control before any attempt to access data from a protected page, we can arrange to have the necessary backing store assigned at that time. One easy way to associate backing store with the faulted-on page is to `mmap` an anonymous region to that page.

Lazy allocation of swap space is particularly useful for pointer swizzling at page fault time because coarse-grained swizzling can be done without worrying about wasted swap space for unused pages. Virtual address space is still consumed for all reserved pages; however,

[9] Recall that reserved pages are those pages that are referenced from swizzled pages, and are used as "placeholders" because the data has not been loaded yet from the persistent store.

address space is a less scarce resource compared to swap space. Furthermore, if the swap space is allocated lazily using `mmap` (or similar), we can potentially reclaim it using `munmap` (or similar) for pages that are no longer in use by the application. Of course, if swap space is reclaimed, the address space must still be retained because it may be referenced from anywhere in the application data structures. We reprotect such address space to guard against (and to get notification of) future attempts to access the data in that space.

7.3.2 Survey of Existing Implementations

A wide variety of Unix-like operating systems are currently available in the market. Each of these implements standard virtual memory primitives that provide the basic functionality as described earlier, but with minor variations compared to others. Following is a brief survey of implementations of the virtual memory primitives with respect to swap space allocation on some popular modern operating systems.

SunOS 4.1.x, Solaris 2.x and Ultrix 4.2

The virtual memory primitives on each of these operating systems allocate swap space lazily. However, although the actual allocation is done only when required, swap space is "reserved" eagerly. That is, for every page of address space allocated, the operating system "sets aside" corresponding swap space but does not actually generate the physical-to-virtual mapping entry in the page table.

This eager reservation of swap space is designed to avoid the problem of overcommitment associated with lazy allocation of swap space. By reserving swap space at the same time when the page is allocated, the operating system guarantees that there will always be sufficient swap space for all allocated pages. However, for our purposes, the overall effect is the same as that of eager swap space allocation, that is, we still "waste" swap space for pages that are not used by the application.

Solaris 2.5 defines a new option, `MAP_NORESERVE`, that can be specified when using `mmap`. As the name suggests, this flag lets the application indicate to the operating system that no swap space should be reserved (or allocated) for the newly-mapped address range, and that the application will be responsible for ensuring that adequate backing storage is available when the address space is accessed. This is exactly what we would like to have for efficiently implementing our coarse-grained pointer swizzling mechanism. It would be useful to also have such control when `sbrk` is used for situations where `mmap` cannot be used.

AIX 3.2 and AIX 4.1

By default, swap space is allocated lazily, that is, only when the application actually uses (reads from or writes to) the page. However, unlike any of the systems above, users are allowed to dynamically modify this behavior to cause early allocation of swap space for specific sessions. This is done by setting a user environment variable, `PSALLOC`, to a special value, `early`. All applications executed after this environment variable has been set will default to eager allocation of swap space.

In addition, there exists a special signal SIGDANGER that is sent to all running processes when the available swap space falls below a certain threshold. This signal is designed to allow graceful handling of swap space exhaustion as far as possible; if processes ignore the signal, the operating system will eventually kill one or more processes to avoid a complete "meltdown."

It is certainly debatable whether having such a signal and arbitrarily killing processes is a good design decision in general. We believe that having a mechanism that allows process notification for controlling memory usage is a useful idea in principle. Unfortunately, there are no good heuristics for selecting processes that are amenable to termination. For example, the current heuristic selects processes with the largest memory allocation. At first glance, this seems reasonable because getting rid of the largest processes should provide the biggest benefit. However, it is also possible that these large processes are the most critical or long-running processes and terminating them effectively wastes the CPU utilization that has already been devoted to their execution.

We believe that it is extremely difficult to find a process selection strategy that would be acceptable in all situations. Of course, it would be very helpful if programmers were better educated about the existence and behavior of SIGDANGER so that more applications will be implemented to gracefully handle the signal, thereby reducing the need for drastic actions from the operating system.

Finally, there is one minor quirk of AIX that should be highlighted. The operating system divides the 32-bit process address space into sixteen *segments*, each of which is 256MB in size.[10] Every application is then classified as supporting either a *small allocation model* or a *large allocation model*. By default, all applications fall into the former category which restricts maximum virtual address space allocation to a single segment. It is possible to build applications that support the large allocation model by providing a special option to the linker, increasing the upper limit of address space allocation to 2GB (i.e., 8 segments).

OS/2 Warp

The default swap space allocation model is eager allocation. As with AIX, it is possible to explicitly control this behavior to achieve lazy allocation. However, the operating system supports the lazy allocation model on a per-request basis, that is, it is possible to select lazy allocation by using a special flag during an allocation request. This selection is in effect only for that particular allocation request. Note that this is quite different from AIX where the environment variable affects allocation requests in all applications that are executed after the variable is set.

Linux 2.x

Linux appears to be the most unusual regarding its swap space allocation model compared to all other operating systems described above. The default model is completely lazy allocation—unlike a normal lazy allocation scheme where swap space is allocated when the page is referenced for the first time, Linux defers allocation of swap space even further by waiting until

[10] This is unlike classic segmented addressing because applications are not aware of segments and use direct virtual addresses, but the operating system internally uses segment register addressing to locate the data in the larger (52-bit) hardware-supported address space.

the page is ready to be swapped out. In essence, swap space is treated as an "extension" to main memory rather than its backing store. This approach allows configurations *without any swap space* as long as all running processes consume less than the total available main memory. Note that Linux provides no mechanisms to "turn off" the lazy allocation model.

7.3.3 Discussion

It is evident from the above discussion that there are no standard heuristics for swap space allocation—each operating system implements the virtual memory primitives differently and provides its own variant on the mechanism for configuring lazy or eager allocation of swap space. We believe that the lazy allocation model is preferable in general, as long as applications are aware of—and are willing to handle—situations where possible overcommitment of backing store could occur. An ideal situation would have the default as lazy allocation with an application-configurable option for switching to eager allocation. With such a setup, applications that truly need eager allocation can ensure that backing storage is allocated along with the address space.

Using Unmapped Address Space

In addition to the inherent lazy allocation supported by `sbrk` and `mmap` implementations, another possible way to achieve the same effect would be through the use of *unmapped* address space, that is, virtual address space that has never been mapped and consequently has no corresponding entries in the page table. With this approach, applications can use unmapped address space (instead of access-protected address space) and handle unmapped memory violations (instead of access-protection violations) to detect accesses to the "protected" region. The main advantage of this approach is that it supports *true* lazy allocation since the address space is not even mapped until it is accessed.

A naive implementation of the above scheme would simply select a range of addresses when reserving address space without ever involving the operating system in the selection process. At first glance, this seems reasonable because, by definition, address ranges that are not known to the operating system are unmapped. However, there are at least two obvious problems with this approach:

1. Other modules of the application may be allocating memory in cooperation with the operating system; it is possible that the operating system may accidentally select address ranges that clash with the "reserved" addresses because it did not know that those ranges were already "in use."

2. There is no portable way to determine "safe" address ranges across multiple operating systems, or even for a single operating system across different hardware platforms or kernel versions, such that these ranges do not clash with the normal allocation. (This is the same problem as the one that we face when using the `MAP_FIXED` option of `mmap`.)

The basic solution is to *disallow* the operating system from adding new mappings into our "reserved" area. For example, if we let the operating system map a file or a dynamically linked (possibly shared) library into memory, it may select an address range that either partially overlaps or completely clobbers the pages that we have already reserved as part of the pointer swizzling process. This is potentially even more serious problem for simple `mmap`'ed persistent storage systems that do not use pointer swizzling because the actual persistent object store is very tightly coupled to specific virtual address ranges where it *must* be mapped.

It is clear that, in order to solve these problems effectively, the operating system must be involved in the address space reservation process. One approach for this might be to provide a new virtual memory primitive, say `mreserve`, that can be used to "inform" the operating system that a certain address range is now "in use" by the application but no backing store needs to be allocated as yet. Alternatively, the primitive can be designed to "ask" the operating system for an unmapped address range on behalf of the application. The implementation of this primitive can be made very fast, in part because of the minimal actions required to implement the simple functionality. A fast primitive would also reduce the overhead of reserving address space during swizzling; currently, either `sbrk` or `mmap` must be used for this purpose; this adversely impacts the overall performance of the system.[11]

7.4 Pointer Swizzling and Virtual Memory Management

Pointer swizzling at page fault time uses existing virtual memory hardware and protection facilities supported by the operating system to detect references to non-resident object and to trigger loading and swizzling of pages containing these objects. Once a page has been faulted in from the persistent store, it is cached in the virtual memory and *no distinction is made between the page in memory and on secondary storage.* In other words, the virtual memory system is free to move swizzled pages between main memory and swap space without affecting the normal operation of Texas. This is unlike some other systems which enforce that some or all pages corresponding to persistent memory must be "pinned" in RAM.

In general, pointer swizzling at page fault time is independent of page replacement policies of the underlying virtual memory system. Of course, if necessary, it can exploit any additional control that may be offered by the operating system (*a la* Mach-style external pagers). By default, however, Texas "plays nice" with other applications running on the system because it does not impose any special considerations on the virtual memory system.

In this section, we discuss various issues related to interactions between the virtual memory system and pointer swizzling at page fault time in the context of control over memory management. We first revisit the mistaken-dirty-pages problem (described earlier in Chapter 3) which arises due to the extremely loose coupling between pointer swizzling at page fault time and the virtual memory system. Using this example, we describe the possible spectrum of interactions between an application and the virtual memory system, and provide some directions for additional application-level control over memory management, including paging.

7.4.1 Control over Memory Management

Recall that the mistaken-dirty-pages problem arises because the virtual memory system cannot distinguish between modifications by Texas (for pointer swizzling) and those by the application

[11] The batching mechanism is one simple way to amortize the cost of crossing kernel boundaries.

itself. As a result, pages are "erroneously" marked dirty by the virtual memory system and paged out if necessary. The basic resolution for the problem lies in providing mechanisms that allow additional interactions between the virtual memory system and the application regarding the latter's access characteristics and status of its pages. A general solution might allow applications to have complete control over the virtual memory management of the operating system. However, it is not necessary to have such extensive control—a simpler *virtual file system interface* can be exploited for this purpose at the expense of some operating system-specific implementation.

It is possible to imagine a spectrum of different levels of interaction between an application and the virtual memory system, ranging from simple (i.e., no interaction) to complete application control over various virtual memory management policies. The default situation falls at one end of the spectrum where the application does not exert any control over the virtual memory system and therefore cooperates with other applications. At the other end of the spectrum is the situation where an application has full control over the virtual memory system, including the replacement policy and mechanism. In this case, the application has special privileges compared to other applications on the system whose performance can be adversely affected by the actions of the privileged application. For example, if one application pins some pages in RAM, the memory available to other applications is effectively reduced, possibly causing extra paging activity. In general, any situation where resources are exclusively assigned to a single privileged process will adversely affect other less-privileged processes.

Finally, between the two extremes, there are a variety of levels that afford different degrees of interaction with the virtual memory system. We briefly describe some of these approaches and discuss how they can be applied to solve the mistaken-dirty-pages problem.

Special-purpose Virtual Memory Primitive(s)

Narasayya *et al.* [NNM$^+$96], who originally identified the mistaken-dirty-pages problem, propose a special system call that allows the application to clear the dirty status bit of a page. The idea is to use this system call for pages that have not been modified by the application; later, if these pages need to be evicted from memory, they can simply be discarded without any unnecessary page-outs. The corresponding virtual address space must also be reprotected so that future references to that space will be intercepted by the normal fault handling mechanism.

This solution can be generalized to provide new special-purpose primitive(s) that can be used by an application to communicate various types of information to the underlying virtual memory system. Using such primitives, the paging policy for mistaken-dirty pages can easily be configured as either local paging or remote paging (i.e., from the persistent store) depending on the current application-specific conditions, and the virtual memory system behavior can be controlled accordingly.

Some operating systems such as Solaris 2.x and AIX 4.x already support a primitive, `madvise`, that is essentially a simpler version of a full-fledged primitive for communicating with the virtual memory system. Currently, `madvise` can only be used for memory that has been `mmap`'ed, and supports a limited amount of information that can be communicated. It is possible to extend the functionality of `madvise` and generalize it for arbitrary pages in the address space.

External Memory Management

We envision an *external memory management* mechanism that allows user-level code to control various memory management operations such as paging, address mapping, etc. Some operating systems such as Mach [BKLL93], Choices [Rus91], and Chorus [ARG89] already provide some form of external memory management.

Mach allows user-level tasks to act as external pagers that fully control the use of memory within the process address space. A pager may control all or part of the address space and multiple pagers can coexist, each managing a different part of the address space. External pagers also handle the page-in and page-out requests generated by the kernel, and are free to save and restore data using any arbitrary mechanism. Mach also provides default pagers (also called *internal pagers*), which are used if no external pagers are configured.

An external page-in facility can be combined with our pointer swizzling mechanism to create an intermediate "loading and swizzling module" that executes as a separate thread and handles all requests to load data from the persistent store. It is the job of this module to locate the page, swizzle it and *then* load it into memory. Since pages are modified *before* being loaded into memory, the virtual memory system cannot erroneously consider a page dirty due to swizzling only. This provides a general solution that allows pointer swizzling at page fault time to coexist with the virtual memory system.

Using a Virtual File System

Although a builtin, operating system-supported external page-in facility can be used as a dedicated swizzling module, a similar mechanism can be implemented by exploiting the *virtual file system* (VFS) interface provided by most modern operating systems, at the expense of requiring some system-specific implementation and superuser privileges for mounting a new file system. In essence, we can implement a simplified external pager-like facility through a virtual file system.

The basic idea is to implement a simple, special-purpose "file system" that manages *only* two specific files. One of these acts as the backing store for data that is loaded into the heap from the persistent store, and the other is a pseudo-file that is actually used to communicate information from the application to the special file system. We use the term *file system module* for code that implements the actual file system mechanism by interacting with the VFS interface. In contrast, the term *file system instance* corresponds to a unique set of the two special files managed by a file system module. We have only one file system module that must be loaded and/or mounted specially into the kernel as a new virtual file system; however, we can have multiple file system instances, one for each persistent store manipulated by an application. In other words, we have a one-to-one mapping between persistent stores and file system instances that "manage" them. Apart from managing the paging for a particular persistent store, each file system instance also acts as the "swizzling server" for that persistent store, swizzling the data as necessary and maintaining the information required for mapping between persistent and virtual addresses.

When an application opens a persistent store, the first step is to `mmap` a page of virtual address space to the special backing storage file, *without actually loading any data from the* *persistent store*.[12] By definition, this page corresponds to the first page of the persistent store, and the persistent store entry pointers can be swizzled into corresponding virtual addresses on this page. As with the current implementation, this concludes the bootstrap phase.

As the application attempts to dereference the entry pointer(s), a normal virtual memory (kernel) fault is generated because the data for the first page is not yet available.[13] The virtual memory system transfers control to our special file system for loading the data into memory. The file system then locates and loads the page from the actual persistent store (stored in a regular file system), swizzles it appropriately while maintaining the necessary mappings between persistent and virtual addresses, and then provides the swizzled page to the virtual memory system, which then finishes handling the virtual memory fault by making the data available to the application. Notice that the page is delivered to the virtual memory system *after* it has been swizzled, ensuring that it will be considered clean by the virtual memory system. Any other pages that need to be reserved during swizzling can be handled by mapping address space to the special backing storage file and making corresponding entries in the mapping table.

When the virtual memory system needs to evict pages as part of the page replacement policy, it can simply discard the clean (but swizzled) pages without any further intervention from our file system. If the application references this page again, a new kernel fault is generated which is handled as before, by loading the page from the persistent store and swizzling it within the special file system. On the other hand, if pages that are actually modified by the application (the *truly-dirty pages*, as opposed to the mistaken-dirty pages) need to be evicted, they will be paged out to our file system, which can write them to local swap space (as with normal virtual memory paging). Alternatively, depending on the checkpointing and logging mechanism, the truly-dirty pages may be written directly to the log, which may be unified with the persistent store itself if a log-structured storage system is being used.

Notice that we avoid the mistaken-dirty pages problem completely by providing swizzled pages to the virtual memory system. By implementing our own "file system" to handle the backing storage for a persistent store, we are able to intervene at the right level of abstraction (similar to an external pager) without requiring extensive modifications to the operating system. Referring back to system architecture shown in Figure 4.1 (Chapter 4), we have effectively migrated the swizzling and mapping module into a separate entity—the special file system. Of course, the type descriptor information must still be communicated to the swizzling module as before, requiring some additional communication—handled using the pseudo-file—between the application and the special file system.

Another benefit of this approach is that user-level fault handling is no longer required for implementing pointer swizzling at page fault time. Instead, we manage the loading and swizzling entirely using normal kernel-level faults generated by the virtual memory system. Even pages that are actually modified by the application can be distinguished easily because only the truly-dirty pages will be paged out to our file system, while the clean pages will directly be discarded by the virtual memory system. Finally, this approach avoids the traditional

[12] This is very similar to the current bootstrapping mechanism. One optimization may be to use batching as currently implemented—we `mmap` multiple pages of address space at a time rather than individual pages on demand.

[13] This is very similar to normal kernel-level faults for loading data from files mapped into memory.

database system strategy of “wiring down” a chunk of RAM and managing it explicitly. This is similar to our current faulting and swizzling approach, where Texas “plays nice” with other applications in terms of virtual memory management.

The approach works across most modern operating systems; we can achieve a high degree of portability by ensuring that all actions are in terms of a well-known file interface (i.e., reads and writes to files). Some operating system-specific implementation will still be required for interacting with the virtual file system interface which is likely to be specific to each operating system. Another issue is the need for superuser privileges to mount the special “file system” appropriately such that the kernel recognizes it as a valid module. However, this is necessary only to mount the single file system module; running a normal application (that implicitly interacts with file system instances) does not require any special privileges. We believe that these minor disadvantages are acceptable costs for the portability and performance benefits gained by avoiding the mistaken-dirty-pages problem.

7.4.2 Discussion

Address mapping and virtual memory management are two *separate* issues that should be managed separately. Unfortunately, most current operating systems combine these two together in their implementations, leading to unexpected interactions with low-level mechanisms such as pointer swizzling at page fault time and compressed virtual memory. The mistaken-dirty-pages problem, and the associated additional page-outs that follow, provide an excellent example of such unexpected (and unnecessary) interactions. The right solution is to improve operating system implementations to provide better *separation of concerns* and additional control to the programmer. Some modern operating systems provide an extended memory management model that separates the fundamental issues and allows programmers to externally control the behavior of various operating system components.

We have seen from the earlier discussion that there are many issues that must be resolved in order to provide a flexible mechanism that user-level applications can interact with and control. An approach that implements the basic operating system functionality in a microkernel core and layers the rest on top via external modules is probably the best way to handle such interactions. We envision a flexible external memory management mechanism, similar to Mach-style external paging, for supporting additional user-level control over operating system functionalities.

Besides the actual mechanics of page-ins and page-outs, *paging policy* is another important aspect of virtual memory paging. The paging policy is used by the virtual memory system to select *which* pages should be replaced and *when* they should be replaced. Although Mach supports user-level pagers for handling the mechanics of the actual paging itself, control over paging policy is still retained by the operating system and external pagers typically cannot affect any policy decisions.

There is a wide spectrum of applicability corresponding to different levels of control over the virtual memory system’s behavior. For most applications, having control over only the paging mechanism is likely to be sufficient; for others, it may not even be necessary to have any control. In contrast, for some special-purpose applications, a full custom replacement policy and complete control will probably be a significant win over any of the default policies.

Therefore, it is important to approach and resolve a problem at the right level of abstraction to achieve the cleanest solution. For example, it is easy to solve the mistaken-dirty-pages problem simply by using Mach-style external pagers without requiring complete control over (and arbitrarily changing) the replacement policies of the virtual memory system or diving too deep into low-level implementation issues.

There are, of course, a few disadvantages of allowing user-level control over operating system functionalities. Specifically, the overall performance may be significantly affected if the user-level code makes “bad” decisions. However, the benefits appear to outweigh the potential performance problems. An ideal setup would provide sufficient “hooks” into various operating systems facilities along with default processing so that applications that truly need additional control (and are aware of the consequences) can indeed do so relatively easily. Of course, there must still be centralized control for certain critical issues to ensure fairness.

7.5 Other Operating System Features

Most of our discussion until this point has concentrated on issues related to the virtual memory system and how existing features can be exploited for efficiently and portably implementing coarse-grained address translation. In this section, we briefly discuss other operating system features that are useful for implementation of low-level system extensions. In particular, we discuss the need for efficient and flexible exception handling as well as issues in selection of virtual memory page sizes and extended page protection facilities. Finally, we also briefly discuss raw I/O which allows unbuffered access to block devices for applications with special I/O characteristics.

7.5.1 Exception Handling

We use the term *exception* for any kind of fault (or signal) that occurs during program execution. These exceptions may be classified into two types, *asynchronous* and *synchronous*. Asynchronous exceptions may be generated due to external events that are not under an application's control—for example, when the user types Ctrl-C (for interactive programs) or some preset alarm goes off. On the other hand, synchronous exceptions are triggered by events that are directly related to the execution of the program—for example, when a divide-by-zero operation is performed.

Historically, exception handling has been used to permit graceful handling of serious error conditions that are encountered during application execution. For example, an interactive application may choose to handle user interrupts[14] by releasing all resources and terminating cleanly with a useful error code. Similarly, a floating point exception, typically generated because of a divide-by-zero operation, can be handled such that a helpful error message is printed before execution is terminated.

Since the usual response to an exception is termination of execution, most operating system implementations do not consider efficiency to be an important factor in exception handling. However, current day usage of exception handling has advanced significantly beyond the

[14] Most Unix system generate the `SIGINT` signal when the user types Ctrl-C.

simple “graceful error handling” usage model. An obvious example is our pointer swizzling at page fault time technique, which uses virtual memory access-protection violations to avoid the software overhead of checking pointer formats. Examples of other applications include garbage collectors, distributed shared virtual memory systems, and compressed virtual memory. With a wide variety of “non-traditional” uses for exceptions [AL91], it is becoming increasingly necessary that operating system implementors recognize the need for efficient exception handling and improve their implementations appropriately.

Experimental Design

We have measured the performance of access-protection violation handling on both Solaris and Linux using our clock cycle timer. Specifically, we measured the time elapsed between the point where a signal is raised (as the application attempts to access a protected page) and the point where the user-level protection fault handler gains control of execution. This approximates the overhead imposed by the operating system for servicing an exception and transferring control to the user-specified handler.

For this purpose, we use a microbenchmark that is set up as follows. We attempt to access a single page that is access-protected, generating an appropriate signal (SIGSEGV) which is then handled by a a user-level signal handler. However, the handler simply increments a counter (originally initialized to zero) and returns without changing the protections on the page. When the faulting instruction is restarted, it immediately generates another signal because the page is still access-protected and the same sequence of events occurs again. Eventually when the counter reaches a predefined maximum (5000 in our microbenchmark), the fault handler unprotects the faulted-on page and returns, allowing the application to successfully complete the original access without generating any further signals. We measure the total time for the entire sequence (starting from the first attempt to access the protected page, until the page is actually unprotected) and divide by the total number of faults to obtain the time taken by the operating system for servicing a single fault.

Note that this approach really gives us a *lower bound* on the operating system cost for handling a single access-protection violation because of the loop-like nature of the microbenchmark. That is, a large number of faults are generated in quick succession, much like iterations of a loop, and we are likely to see effects of caching; in particular, the relevant kernel data structures and fault handling code may be cached in a second-level cache after the first fault. However, an application which is not generating protection faults heavily is unlikely to benefit from such caching and consequently may incur higher overhead per fault.

Experimental Results

We ran our microbenchmark on both Linux and Solaris systems, using identical underlying hardware (a 200MHz Intel Pentium Pro processor with 32MB RAM) as before for both systems. Table 7.1 shows the actual cost for each operating system. The results clearly show that exception handling on Linux is several times faster than on Solaris. Both SunOS 4.x and Solaris 2.x implement a layered VM architecture [GMS87] that is substantially different from the 4.3BSD memory management architecture. In particular, it is more modular and requires

Operating system	Cost (in cycles)
Linux 2.0.x	2,500
Solaris 2.5.1	17,500

Table 7.1: Cost of handling an access-protection violation

many function calls, sometimes indirected via a function table lookup. This will obviously impact the overall performance, and indeed Chen *et al.* [CBL90] have shown that layering of components adds a 20% overhead to fault handling. However, the empirical results that we have obtained indicate a slowdown factor of six, definitely much larger than what can be accounted by the extra 20% penalty.

Discussion

Other researchers have also recognized the problem with slow exception handling on various operating systems. Thekkath and Levy [TL94] contend that it is easier to improve performance of handling synchronous exceptions than it is for asynchronous exceptions because the information needed for servicing the former is already available in the user space of the process. They describe both hardware (architectural changes) and software approaches (kernel changes) for this purpose and present encouraging results for their software approach. Implemented by modifying the DEC Ultrix 4.2A kernel, their approach requires only 8 microseconds for handling a null exception compared to 80 microseconds taken by the unmodified kernel.

Another great example of fast exception handling mechanism is the L3 (and subsequently, L4) microkernel [Lie95, Lie96]; the full cost of a kernel call in the L3 microkernel is between 123 and 180 cycles [Lie93], or *less than one microsecond on a modern 200MHz processor*. This is an extremely impressive result, given that our best performance on Linux is more than an order of magnitude slower. Anderson *et al.* [ALBL91] have discussed the interaction between hardware architecture and operating systems, including virtual memory and fault handling.

7.5.2 Virtual Memory Page Size and Sub-page Protections

Pointer swizzling at page fault time generally benefits from the use of a smaller page size because it reduces the amount of swizzling that is required at page faults. In addition, the address space consumption rate is also reduced as fewer pointers are swizzled for every page loaded from disk.

In general, virtual memory page size plays an important role in the implementation and performance of systems that exploit virtual memory features to implement new abstractions. Unfortunately, opposing forces are usually at work when a page size needs to be selected [HP96]. On one hand, larger sizes are preferable from the perspective of reducing hardware costs—page tables and Translation Lookaside Buffers (TLBs) can be made smaller—and for improving efficiency of data transfer between memory and secondary storage because larger units of transfer reduce the effect of latency. On the other hand, smaller sizes provide additional flexibility in terms of allocation and memory protection, and for reducing internal fragmentation.

One possible approach for resolving the page size selection conflict is to choose a size that provides a balance between the opposing forces without considering pointer swizzling at page fault time (or other systems), and *then* support operations—mainly page protections and signal handling—at a sub-page granularity for those systems that benefit from smaller page sizes. This approach is likely to have minimal impact on the normal operation of a virtual memory system because page sizes are selected based only on the *relevant* trade-offs. At the same time, it works well for coarse-grained address translation; with sub-page protections, only parts of reserved pages are swizzled, effectively providing the same benefit as if the pages were smaller.

We believe that with a little hardware support, operating systems should be able to provide sub-page protection facilities relatively easily. In fact, the ARM600 processor provides such support [SW91] facilitating the easy implementation of sub-page protections in the Apple Newton. The 801 prototype RISC machine also incorporated support for protections at the sub-page granularity [CM88]. Finally, Thekkath and Levy [TL94] have shown that a little kernel support can be used to emulate sub-page protections in a simple manner at a higher level of abstraction.

7.5.3 Support for Raw I/O

In any computer system, disk I/O is usually much slower compared to the performance of other components, most notably the CPU. Typical disk latencies are on the order of five to ten milliseconds, which is several orders of magnitude slower than main memory speeds. Therefore, it is advisable to minimize the number of disk I/O operations to avoid a major impact on general performance. Most Unix systems implement this by providing *file system caching* (sometimes also known as the *buffer cache*). As with any cache, the basic idea is to store recently accessed disk blocks in fast storage speculating that the same data will be read again *soon*. Most operating systems also implement a readahead mechanism to prefetch additional data into the cache with the hope that expensive disk seeks can be avoided if the prefetched data is accessed soon.

Traditional Unix implementations have typically used a fixed-size, independent area of memory specially designated for the buffer cache. However, most modern systems integrate the buffer cache with the virtual memory system and dynamically control the fraction of memory assigned to each component. When a `read` is issued, the kernel first maps the specified part of the file into its own (protected) address space, faults the data in, and then copies it to the user-specified buffer. The procedure for handling a `write` follows a similar model, also performing an extra copy from user space to kernel space.

However, for specialized application with well-understood (but possibly unusual) I/O behavior, the extra copy from/to the buffer cache may adversely affect the performance both in time and space. For example, database-style applications that transfer large amounts of data to and from disk are likely to benefit from bypassing the buffer cache because they usually implement their own caching mechanisms. In the case of Texas, the data is fetched only once from the disk and is subsequently cached in virtual memory. A buffer cache is undesirable in this situation because it will compete with the virtual memory system for real memory, effectively reducing the memory available to the application.

Most Unix systems provide a facility called *raw I/O* (or *direct I/O*) that allows unbuffered access to a block device, avoiding the extra copy because the data is faulted in directly into user space. The interface is still the same (normal `read` and `write` system calls), although the requests must be made to the appropriate character device corresponding to the block device;[15] the kernel internally handles the requests differently. Another alternative for eliminating the extra copy is to use `mmap` instead of standard file system read/write interface; this is, however, less attractive for two reasons: `mmap` is not as portable, and it presents semantics that are different from the `read` and `write` system calls.

As operating system implementations become more open, it would be extremely useful to have application-configurable disk I/O characteristics. In particular, applications should be able to perform either normal, buffered I/O or direct I/O for arbitrary files in the normal file system based on their specific file access characteristics and semantics. It appears that various operating system implementations are indeed moving in that direction. For example, Irix (from Silicon Graphics, Inc.) supports a special flag (`O_DIRECT`) that can be specified when opening a file (via the `open` system call) to notify the operating system that direct I/O must be used for reading and writing that file. Similarly, the new Solaris 2.6 release supports a new library routine, `directio`, that can be used to dynamically switch access characteristics for the specified file. Alternatively, the operating system also allows a file system to be mounted such that I/O to all files in that file system will be direct.[16]

7.6 Conclusions

In this chapter, we have described various aspects of operating system interactions that are relevant when implementing low-level system extensions such as distributed shared virtual memory and pointer swizzling at page fault time. Since our system relies heavily on virtual memory hardware, our discussion is primarily focused on interaction with the virtual memory system, but we also briefly discussed some other useful features of the operating system.

We have shown that `sbrk` and `mmap`, the two virtual memory primitives most commonly used for allocation, differ significantly in terms of their features and interface, and overall performance. The flexibility of `mmap` with respect to additional control over swap space allocation and reclamation is likely to provide a major benefit over `sbrk`. As such, we believe that `mmap` should be used as the basic primitive for implementing high-level allocation policies.

We also discussed issues regarding external memory management and control over virtual memory paging as related to pointer swizzling at page fault time. In this context, we described various levels of interactions with the virtual memory system that are possible depending on the capability of the underlying operating system kernel. It should be emphasized again that although pointer swizzling at page fault time can exploit additional support from the virtual memory system, it is *not a requirement* for normal and correct operation in general.

Other than virtual memory system interactions, we also briefly described some related issues in operating system development. The most important of these is the support for

[15] For example, on Solaris, the name `/dev/dsk/c0d0s7` represents a block device, so the corresponding raw device would be named `/dev/rdsk/c0d0s7`.

[16] Gavin Maltby, personal communication, July 1997.

efficient exception handling. We have measured exception handling costs on both Linux and Solaris and shown that although the latter is slower by a factor of about six, it has potential to be improved. We believe that fast microkernel implementations are likely to be the winners in this category. Another important issue is the support for virtual memory protections and extensions to allow protections at the sub-page granularity. The latter would be useful for various systems that rely on virtual memory facilities and would benefit from smaller page sizes. Finally, we make a case for providing additional control to the programmer with respect to file I/O and ability to bypass file system caching.

In conclusion, operating system implementations should be more open and reflective, allowing "responsible" user-level applications to control some of the key features to suit their needs. In other words, operating systems should provide the basic building blocks that can be assembled together by user-level facilities to implement useful and extensible systems efficiently. Overall, it would be beneficial if the operating systems had more separation of concerns, thereby allowing systems (such as ours) to approach and solve problems at the right level of abstraction, without getting distracted by tedious (and irrelevant) implementation issues. There is evidence that other researchers also have similar goals for improving operating system implementations [KLM+93].

Chapter 8

Future Work

8.1 Introduction

The previous chapters have described the design and implementation of Texas, our portable persistent storage mechanism based on high-performance coarse-grained address translation using pointer swizzling at page fault time. Although Texas is an actual system that has been used in both commercial and non-commercial systems, we also intend for it to be used as a research testbed for exploring various avenues in address translation and advanced storage management issues. In this chapter, we describe some of these research directions, concentrating mainly on storage management and briefly discussing other related extensions to Texas.

8.2 Storage Management

The current design of Texas is flexible in terms of the implementation of underlying storage management. We have implemented an abstraction layer that can be used to implement different kinds of storage mechanisms without disturbing any of the other functionality. In the current implementation, the storage mechanism is designed such that the persistent store can be saved either to a regular file in the file system or to a raw disk partition. It is possible to implement a log-structured storage system that simplifies the checkpointing and recovery mechanism, while improving flexibility and performance.

We are also interested in studying prefetching and compressed in-memory storage as two issues important to storage management. Prefetching can be viewed as a way to *improve I/O performance* by reducing the time spent in waiting for I/O to complete. In contrast, compressed in-memory storage is a way to *avoid I/O* by attempting to keep more data in memory. Finally, we also discuss adaptive techniques for both prefetching and compressed in-memory storage.

8.2.1 Log-structured Storage System

As described in Chapter 4, Texas currently implements a simple checkpointing and recovery mechanism. We can replace the simple logging mechanism with a more flexible *log-structured storage system* that supports additional functionality.

A log-structured storage system (LSS) is essentially the lower levels of a log-structured file system [RO91]. The storage system typically manages a single raw disk partition, although a normal file could be used instead. We choose not to implement an entire file system because the complexity is not needed for simple persistent storage management. Instead, only the storage functionality is implemented at the lowest layer and the upper layers may build additional facilities such as files and directories, access permissions, etc. any way they choose.

In a log-structured storage system, the entire disk (or file) being managed is used as a log, and the log itself acts as the final repository of data pages (i.e., the persistent store). In other words, there are no separate entities corresponding to the persistent store and the log that is used for checkpointing. By definition, blocks (i.e., pages) in a log-structured store do not have a single, fixed "home" location on disk. Instead, logical blocks in the system are allowed to "migrate" by simply writing a new version of the block at some different location on the (managed) disk; the "current" version of the block is the last one written to the log. Since blocks do not have a fixed location, changes to a file are committed by updating the index structures to point to the new data.

It is easy to see how a log-structured storage system can be used to maintain the persistent data and support the necessary checkpointing. In fact, since multiple versions of data can exist on disk, we can save multiple checkpoints and support rollback to older checkpoints. Given the "write anywhere" strategy of log-structured systems, writes can also be clustered such that related data are stored consecutively on disk, improving the read latency and bandwidth for future access. Our original paper on Texas [SKW92] describes further details about the log-structured storage system, including a description of its data structures.

8.2.2 Adaptive Prefetching

Modern computer memory systems are *hierarchical*, being composed of several levels of memory [HP96]. The lower levels (e.g., magnetic disks and tapes) are inexpensive and therefore large but slow, while the higher levels (e.g., RAM and caches) are expensive but small and fast. As memory systems become more hierarchical by growing "downward" to become persistent object stores, it becomes increasingly important to make good decisions about which data should be in fast memory at any given time. The most commonly used policy in current systems is *demand prefetching* combined with an approximation of LRU replacement. Demand prefetches are those that occur only in conjunction with *demand fetches*, that is, real page faults.

There are several choices to be made when selecting a prefetching policy. One is the policy of *which* pages to prefetch, while another is *how many* pages to prefetch, and yet another is *how long* prefetched pages are retained in memory if they are not immediately referenced by the program. The most common prefetching rule is *one-block lookahead*—when a page (or block) is faulted on, the next *consecutive* page is prefetched. For example, if page number 237 is faulted on, page number 238 is brought into memory as well. One-block lookahead is an attractive policy because it is easy to implement; consecutively-numbered pages are usually consecutive on disk, and can be brought into memory without an additional seek.

Whether to Prefetch: The “fool-me-once” Rule

We believe Horspool and Huberman’s work on prefetching [HH87] to be among the most interesting, though not for the reasons they intended. They experimented with a variation of one-block lookahead that was designed to be easy to simulate efficiently. Our interpretation of their data, however, is that they inadvertently simulated an *adaptive* prefetching policy, which is more interesting than what they were actually attempting to approximate.

Horspool and Huberman modified one-block lookahead to preserve the *inclusion property*; this property allows simulation of many memory sizes in one pass through a trace. Inclusion is a well-known property of LRU replacement [MGST70]—it guarantees that pages in a memory of a given size are always a subset of pages that would be in a memory of any larger size, and therefore the misses for a given memory are a subset of the misses for any smaller size. For efficient simulation, a single queue of pages can be maintained as a trace is analyzed, as though memory were arbitrarily large. This queue shows the recency ordering of *all* pages that have been touched, independent of any actual memory size.

LRU inclusion means that moving a page to the head of the queue will be interpreted as a hit for a particular memory size (as well as for all larger sizes), and a miss for smaller memories. Horspool and Huberman’s innovation is to devise a prefetching scheme, similar to conventional one-block lookahead, while preserving the inclusion property. Their policy reorders pages in the queue in ways that are independent of any particular memory size. These reordering will be interpreted as prefetches for some sizes of memory, but as reordering of in-memory pages for larger sizes of memory. The particular ordering rule they use is this: pages that are actually touched are always brought to the head of the queue, as though it were an LRU queue; the next (lookahead) page in memory is *also* brought to the head of the queue, *if and only if* it was nearer the head of the queue than the page that was actually touched. Details of the scheme can be found in [HH87].

Horspool and Huberman were surprised to find that their algorithm actually *outperformed* conventional one-block lookahead. We believe that in preserving the demand prefetch policy, Horspool and Huberman inadvertently simulated an adaptive prefetching policy, which approximates what we call the “fool-me-once” rule—if a page is prefetched but not referenced by the application, it is not prefetched the next time.

Unfortunately, the details of Horspool and Huberman’s algorithms introduce unexpected anomalous properties [WKM94]. In particular, their policies are not properly *timescale relative*—events occurring on a timescale that should only matter to some sizes of memory adversely affect replacement decisions for memories of very different sizes. As we describe in [WKM94], slight changes to the algorithms can restore timescale relativity and make them much better-behaved.

What to Prefetch

Horspool and Huberman’s policy decides *whether* to prefetch based on previous observations of reference behavior. It is equally interesting *what* to prefetch. One possibility is to dynamically reorganize small pages within larger units of disk transfer. As a program accesses different pages, the ordering of those accesses can be recorded. When the pages are (eventually) paged out, they can be written to disk in a new order that reflects the recent access ordering. If future access orderings are correlated with past orderings, then this enables a very convenient form of prefetching.

In the mid-1970's, Baer and Sager [BS76] simulated a prefetching policy that relied on reorganizing pages on disk using the order of initial accesses by the program. Unfortunately, their results were disappointing; even though preliminary measures of locality indicated that previous page fault orderings were a good indicator of subsequent access patterns, the policy they actually simulated was unsuccessful, and did not improve performance. We believe that this negative result was due to subtle and interacting flaws in the design of their experiment, and that their data are actually quite encouraging when properly interpreted.

For example, Baer and Sager's reorganization policy used the LRU queue ordering of pages to determine the order in which pages were written out. One problem with this policy is that the LRU ordering is the order of *last* reference to the pages in question, *not the page fault ordering that originally brought the pages in.* While these two grouping principles are probably strongly correlated, they are not identical. We believe that further research and analysis in this area is necessary, and is likely to yield interesting results.

8.2.3 Compressed In-memory Storage

Current trends in hardware configurations indicate that there is a huge gap in performance—about five orders of magnitude—between main memory latencies and disk latencies. A cost effective approach to bridge this gap is to introduce a new level into the memory hierarchy. *Compressed in-memory storage* uses part of main memory as a cache for compressed pages [Wil90, WLM91, Wil91, AL91, Dou93]; this divides the main memory into partitions for uncompressed pages and compressed pages. The use of compressed in-memory storage can improve overall system performance because "paging" from the compression cache may be faster than paging from disk.

The performance of this scheme depends on the relative costs of processor cycles and disk transfers, and on the efficiency of the compression algorithm. Consider the fact that currently it is reasonable to expect machines that can execute over 200 million instructions per second on average and have disks with 8 millisecond latencies. This means that in the time it takes to perform a single disk operation, the processor can execute over 1.6 million instructions. Therefore, as long as each 1.6 million instructions of compression and uncompression saves one disk seek, it is worthwhile to use the compression cache. As disk speeds lag further and further behind processor speeds, compressed in-memory storage becomes increasingly attractive.

Novel Compression Techniques

We avoid the common trap of adapting text-oriented compression algorithm to compress in-memory data. Instead, we aim towards using *domain-specific compression algorithms* for heap data to take advantage of the knowledge about the data representation in memory.

In-memory data typically show different kinds of regularities than character data from files. This is due to the demands of computer architectures, which favor word-sized fields aligned on word and double-word boundaries. Therefore, we use words as the basic unit of matching, refined by discriminating between high-order bits (which are mostly stable) and low-order bits (which are more likely to differ). This is quite effective for both integers and pointers; integers are likely to be small and similar to other small integers, while pointers are likely to be similar to nearby pointers. We can also tailor our techniques to compress floating point data which usually show regularities in the exponent and high order bits of the mantissa.

Discussion

Preliminary experiments have shown a compression factor between two and three, and a time cost of only one-third of a millisecond to compress a 4KB page on a 200MHz Pentium Pro processor running Linux. While more refined experiments and a wider selection of test programs are required, we believe that the early results are very promising. We expect to reduce the time cost by another factor of two based on more fine-tuning of the basic implementation. Furthermore, our virtual memory trace-gathering tool [WKBK97] can also be extended to gather compressibility information on the fly, and our simulators can be modified to evaluate adaptive compressed paging techniques based on information from the traces.

It should be noted that unlike other compression-based memory management schemes, our primary goal is not to increase the available disk storage, but to increase system throughput by reducing the average memory latency and increasing effective performance. Our system should also give benefits similar to those of file-compression schemes [CG91, BJLM92]. Further research is currently underway and detailed results will be presented in an upcoming paper [WKB97a].

8.3 Advanced Issues

In Chapter 1, we briefly mentioned some advanced issues that are beyond the scope of this dissertation. Specifically, we discussed issues related to *distribution, concurrency control and fault tolerance*, *schema evolution*, and *security.*

It should be emphasized that none of these issues are fundamentally in conflict with the basic pointer swizzling at page fault time technique and the implementation of orthogonal persistence in Texas. In fact, some of these have been resolved specifically in the context of Texas, as well as in other related systems. Of particular interest is the issue of distribution and concurrency control. Our current implementation of Texas does not support either of these; however, we are aware of at least one system, MC-Texas [BS96], implemented on a Fujitsu AP1000 multicomputer as a precursor to developing a reference architecture for distributed persistence. Blackburn and Stanton report encouraging results regarding the overall scalability of Texas, modulo a couple of situations related to false sharing of implementation meta-data that should be relatively easy to resolve.

As persistent storage becomes more popular, data security will also become increasingly important because persistent data must be protected against unauthorized access. Previous work has been done in this area and various solutions are possible (e.g., *protection domains* in Opal [CLLBH92] and *areas* in ObjectStore [LLOW91]).

Chapter 9

Conclusions

Coarse-grained address translation techniques have been disregarded in the past as a viable alternative to traditional fine-grained address translation techniques for building efficient persistence mechanisms in general-purpose languages. An overall goal of this dissertation was to "set the record straight" regarding the performance and flexibility of coarse-grained translation techniques and their potential as high-performance address translation mechanisms.

As part of this research, we have implemented Texas, a high-performance persistence storage system for C++ that uses pointer swizzling at page fault time, a coarse-grained address translation technique. In the foregoing chapters, we described various aspects of building such systems to provide efficient orthogonal persistence in general-purpose languages. In particular, we discussed pros and cons of different address translation approaches, presented a new classification scheme for persistent systems based on granularity issues, introduced the concept of run-time type description (RTTD) for accessing implementation-level type information at run time, and discussed lessons that we learned in interacting with operating systems. We also provided research directions in storage management for persistent object stores, including a log-structured storage strategy and compressed in-memory storage.

9.1 Address Translation

We have shown that a coarse-grained approach to address translation does not necessarily constrain the performance requirements for most applications that incorporate orthogonal persistence. By exploiting the virtual memory facilities of modern operating systems and existing virtual memory hardware on modern computers, we have implemented a coarse-grained address translation scheme that runs on stock hardware and has minimum overheads in the usual case. We rely on locality of reference (usually exhibited by most applications) to amortize the cost of translating an entire page over repeated accesses to that page (which incur no further overhead).

We have empirically validated our competitive argument for coarse-grained swizzling techniques by using controlled measurements with existing benchmarks. Specifically, we have demonstrated that the direct cost of our approach is *zero* for normal CPU-bound operations that manipulate in-memory data, and *very small* (between 1 and 5 percent) for I/O-bound operations when the data is being loaded into memory from stable storage. In general, the address translation costs are much smaller than the corresponding I/O costs incurred during loading the data itself. We expect these overheads to decrease even further because CPU speeds typically improve faster than I/O speeds.

While the direct costs of pointer swizzling at page fault time are minimal, there are some indirect costs related to unexpected interactions with the virtual memory system leading to unnecessary page-outs of swizzled pages. Although not directly related to pointer swizzling, these costs may affect the total performance of an application because of extra paging and therefore must be accounted for in the overall measurements. Fortunately, the costs are bounded and also incurred only once per swizzled page. There are also several ways to avoid the problem altogether if the operating system provides additional virtual memory management support (e.g., external pagers in Mach).

9.2 Granularity Choices for Persistence

We have identified a set of design issues that we believe are fundamental to the implementation of any persistent system. The choice of granularity for each design issue forms a classification scheme for any persistence implementation. The design issues that we have identified are the granularities of *address translation*, *address mapping*, *data fetching*, *data caching* and *checkpointing*. We believe that using a combination of granularity choices for these design issues provides a better classification mechanism than the existing *ad hoc* taxonomies.

We have chosen the basic unit for all granularity choices in Texas to be a virtual memory page. This is because pointer swizzling at page fault time is a page-wise translation scheme and the implementation relies heavily on the virtual memory facilities of the underlying operating system. However, if necessary, it is possible to temporarily change the granularity to a finer level—for example, we implement pointer-wise address translation for situations where pure coarse-grained approach will not provide the best benefit.

9.3 Run-Time Type Description

Another important contribution of this dissertation is the notion of Run-Time Type Description (RTTD) for making implementation-level type information accessible at run time. It is useful to have such detailed information about layouts of data objects in memory at run time for a variety of applications. For example, in addition to its applicability to address translation and persistence, detailed run-time object layout information is also useful for applications such as garbage collection, advanced tracing and profiling, etc. The RTTD mechanism presented here is designed to generate the layout information at compile time and make it available to the application at run time.

We have shown that the RTTD mechanism can be implemented portably by using compiler-generated debugging information as the basis for extracting the necessary information for RTTD. We chose to use debugging information over the seemingly more obvious approach based on using special-purpose preprocessors for reasons of portability and compatibility—our approach is portable across multiple compilers and operating systems, and is compatible with different source languages because the debugging information format is usually independent of these factors. Our case study implementation for C++, based on the GNU debugger, is fully operational and is currently used in Texas and a real-time garbage collector for C++.

9.4 Operating System Interactions

During the course of implementing and porting pointer swizzling at page fault time and the Texas persistent store to different operating systems, we have learned several interesting lessons about interacting with different operating systems and the subtle differences in their related features. Most of this interaction has been concentrated in the areas of virtual memory management and protection fault handling, the two most important features relevant to the implementation of Texas.

We discussed several aspects of interactions with virtual memory systems. Among these, we presented a comparison of virtual memory allocation primitives and their performance characteristics on Linux and Solaris, described a study of heuristics for swap space allocation on different operating systems, and discussed advanced facilities for external memory management and additional control over paging *a la* Mach. For protection fault handling, we found significant room for performance improvements in terms of operating system support for efficient exception handling.

As part of our analysis, we presented some suggestions that we believe are important for improving operating system implementations, and consequently their interactions with systems such as Texas. In general, we argue for implementations that are more "open" and "reflective," and which provide basic building blocks that can be assembled by higher-level user facilities to tailor the system to their specific needs. We believe that microkernel-based approaches, with additional layering of functionality on top, are likely to provide some of the performance characteristics that are desirable for general usage.

9.5 Storage Management Issues

Most of the discussion in this dissertation has concentrated on the implementation of a high-performance address translation mechanism. However, issues related to storage management for persistent object stores are also important when implementing persistence. In our current implementation of Texas, we have incorporated a simple no-undo/redo write-ahead logging strategy to provide simple checkpointing and crash recovery support. We discussed alternative approaches ranging from simple page "diffing" and sub-page logging techniques to advanced log-structured storage management for the entire persistent store.

We also described other issues related to storage management, specifically adaptive prefetching and compressed in-memory storage. The former is designed to improve the performance of I/O while the latter attempts to minimize the amount of I/O necessary. Compressed in-memory storage can be used for increasing the effective memory size by using part of main memory to store compressed pages. Preliminary results have shown promise, and further research is currently underway.

9.6 Final Words

Orthogonal persistence is becoming increasingly important as applications get more sophisticated and manipulate complex, heap-allocated data structures. In this dissertation, we have shown that the problem of addressing large amounts of data on standard hardware and operating systems can be resolved effectively using coarse-grained address translation techniques, without compromising on issues of performance, portability and compatibility. By this, we hope to enable wider acceptance of persistence as a useful and important feature in general-purpose programming languages.

Appendix A

Hierarchical Type Graph

Chapter 6 (Section 6.5) described the storage model (i.e., both hierarchical and flat formats) for type descriptor records in our RTTD implementation. In discussing the hierarchical format, we simplified the structure of the type graph for ease of explanation of the basic concepts. The actual data structures maintained by the type descriptor are much more complex and contain all the necessary information to fully describe the various types at run time. Below we describe the full hierarchical graph data structures in further detail.

Recall that we used the following sample type definitions in Chapter 6 to describe the storage model:

```
struct Pet                struct Owner
{                         {
  short tag;                char *name;
  char *name;               void *userdef;
};                          short numpets;
                            Pet pets[2];
                          };
```

Figure A.1 shows the structure of the full hierarchical type graph representing these two type definitions. Compare this with the simplified structure shown in Figure 6.2 and it is obvious that the actual data structures store much more information about the types.

Each application type is classified as either a simple type (e.g., basic builtin type such as `short` and `char`) or a complex type (e.g., pointer or aggregate type), and is represented by a unique type descriptor record in the hierarchical type graph. Each type descriptor record object is instantiated from a special RTTD type[1] that represents a specific application type. The RTTD type names follow a simple convention—each name has three parts: predetermined prefix (‘`TD`”) and suffix (“`Type`”), and a middle component that depends on the application type. Thus RTTD type `TDStructType` would be instantiated to generate a type descriptor record that maintains information about a struct (or class) in an application. Using this naming convention, we note that boxes labeled `TDStructType`, `TDBuiltinType`, `TDPointerType` and `TDArrayType` in Figure A.1 represent the type of various type descriptor records in the hierarchical type graph. (By the same token, boxes labeled `TDField` are *not* type descriptor

[1] We use the phrase “RTTD type” to distinguish types defined in our system (i.e., types of various type descriptor record objects) from types in the application, which are represented by the type descriptor records.

records, but just auxiliary data structures used by TDStructType as described later.)

Each type descriptor record must maintain a set of information that is common across all application types; this corresponds to the first four sub-components of each type descriptor record shown in the figure. In terms of implementation, this is accomplished by ensuring that all RTTD types inherit from a single superclass that has the common information. This information includes the size of the application type in bits (the size field) and information about inheritance hierarchy in the application type structure (the numprnts and prnts fields which track information about "parents" of an application type). In addition, since pointer types are also considered to be complex types in our system, each type descriptor record maintains a reference to *another* type descriptor record that represents a pointer type of the application type represented by *this* type descriptor record (the ptrtype field). For example, the type descriptor record that represents type char will reference another type descriptor record that represents type char*; the latter, in turn, may reference yet another type descriptor record that represents type char**, and so on.

In addition to the common information, each RTTD type contains additional information depending on the requirements for the application type that must be represented. For example, type TDBuiltinType maintains a tag whose (predetermined) enumerated values describe the specific builtin type being represented, while types TDPointerType and TDArrayType contain a pointer to another type descriptor record that represents either the *pointed-to* type (for TDPointerType) or the *array-of* type (for TDArrayType). In contrast, type TDStructType has several additional fields which are necessary for fully representing an aggregate type. Apart from its name, we also need to maintain information about each field of an aggregate type; this is done via an auxiliary type TDField that can maintain information about the name and size of a field (in bits), a reference to the type descriptor record that represents the type of the field, and the offset of the field in the overall aggregate type. Note that it is necessary to maintain both size and offset for each field (rather than deriving the offset from the size of the previous field) because the compiler may insert *padding* between fields to comply with specific layout and alignment requirements of the language and/or the operating system.

In summary, we have described the details of a type descriptor record structure, and the various interconnections that make up the full hierarchical type graph created by the type descriptor generator. It is obvious that, in addition to a common set of information that is necessary for describing any application type, each RTTD type must also maintain additional information that is specifically geared towards representing a particular application type, and may be arbitrarily complex. Finally, we have briefly described the structure and behavior of several important RTTD types which represent type structures (builtin type, pointers, arrays and aggregate types) that are common in most applications.

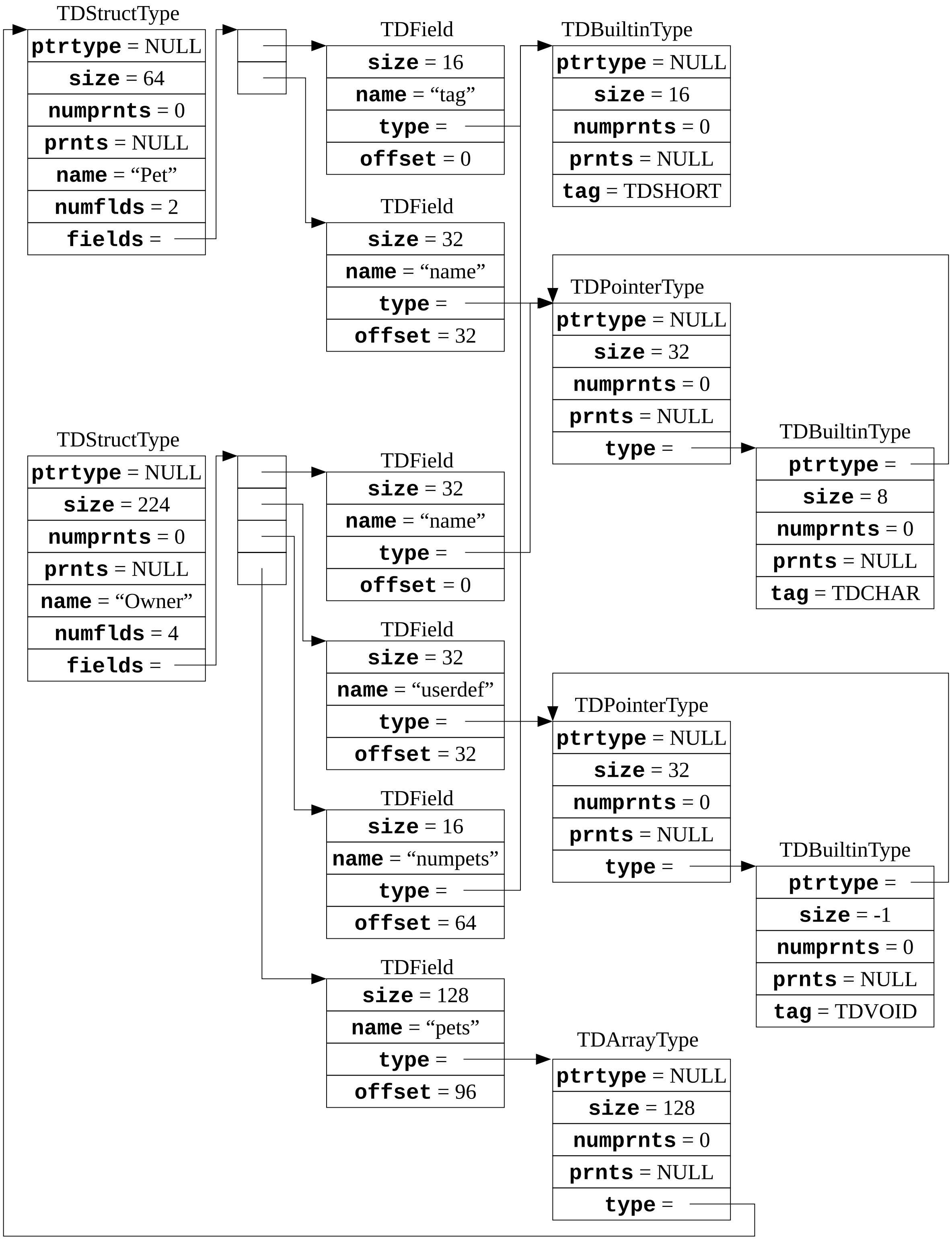


Figure A.1: Full hierarchical type graph

Bibliography

[ABC$^+$83a] Malcolm P. Atkinson, Peter J. Bailey, Ken J. Chisholm, W. Paul Cockshott, and Ron Morrison. An Approach to Persistent Programming. *Computer Journal*, 26(4):360–365, December 1983.

[ABC$^+$83b] Malcolm P. Atkinson, Peter J. Bailey, Ken J. Chisholm, W. Paul Cockshott, and Ron Morrison. PS-Algol: A Language for Persistent Programming. In *Proceedings of the 10th Australian National Computer Conference*, pages 70–79, Melbourne, Australia, 1983.

[ABM$^+$90] T. Lougenia Anderson, Arne J. Berre, Moira Mallison, Harry T. Porter, and Bruce Schneider. The HyperModel Benchmark. In *Proceedings of the International Conference on Extending Database Technology*, pages 317–331, Venice, Italy, 1990.

[ACC82] Malcolm P. Atkinson, Ken J. Chisholm, and W. Paul Cockshott. PS-Algol: An Algol with a Persistent Heap. *ACM SIGPLAN Notices*, 17(7):24–31, July 1982.

[ACCM83] Malcolm P. Atkinson, Ken J. Chisholm, W. Paul Cockshott, and Richard Marshall. Algorithms for a Persistent Heap. *Software Practice and Experience*, 13(3):259–272, March 1983.

[AEL88] Andrew W. Appel, John R. Ellis, and Kai Li. Real-Time Concurrent Garbage Collection on Stock Multiprocessors. In *Proceedings of the 1988 SIGPLAN Conference on Programming Language Design and Implementation*, pages 11–20, Atlanta, Georgia, June 1988. ACM Press.

[AL91] Andrew W. Appel and Kai Li. Virtual Memory Primitives For User Programs. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)* [ASP91], pages 96–107.

[ALBL91] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The Interaction of Architecture and Operating Systems Design. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)* [ASP91], pages 108–120.

[AM92] Antonio Albano and Ron Morrison, editors. *Fifth International Workshop on Persistent Object Systems*, San Miniato, Italy, September 1992. Springer-Verlag.

[AM95] Malcolm P. Atkinson and Ron Morrison. Orthogonally Persistent Object Systems. *VLDB Journal*, 4(3), 1995.

[ARG89] Vadim Abrossimov, Marc Rozier, and Michel Gien. Virtual Memory Management in Chorus. In *Proceedings of Workshop on Progress in Distributed Operating Systems and Distributed Systems Management*, Berlin, Germany, April 1989. Springer-Verlag. Also Chorus systemes TR CS/TR-89-30.

[ASP91] *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, Santa Clara, California, April 1991.

[Bak78] Henry G. Baker, Jr. List Processing in Real Time on a Serial Computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[Bak91] Henry G. Baker, Jr. The Treadmill: Real-Time Garbage Collection without Motion Sickness. In *OOPSLA '91 Workshop on Garbage Collection in Object-Oriented Systems*, October 1991. Position paper. Also appears as *ACM SIGPLAN Notices 27*(3):66–70, March 1992.

[BC92] Yves Bekkers and Jacques Cohen, editors. *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, St. Malo, France, September 1992. Springer-Verlag.

[BJLM92] Michael Burrows, Charles Jerian, Butler Lampson, and Timothy Mann. Online Data Compression in a Log-structured File System. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 2–9, Boston, Massachusetts, September 1992.

[BKLL93] Joseph Boykin, David Kirschen, Alan Langerman, and Susan LoVerso. *Programming under Mach*. Addison-Wesley, Reading, Massachusetts, 1993.

[BL92] Thomas Ball and Jim Larus. Optimal Profiling and Tracing of Programs. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 59–70. ACM Press, January 1992.

[BS76] Jean-Loup Baer and Gary R. Sager. Dynamic Improvement of Locality in Virtual Memory Systems. *IEEE Transactions on Software Engineering*, SE-2(1):54–62, March 1976.

[BS96] Stephen M. Blackburn and Robin B. Stanton. Multicomputer Object Stores: The Multicomputer Texas Experiment. In Scott Nettles and Richard Connor, editors, *Seventh International Workshop on Persistent Object Systems*, Cape May, New Jersey, May 1996. Morgan Kaufmann.

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage Collection in an Uncooperative Environment. *Software Practice and Experience*, 18(9):807–820, September 1988.

[Car89] Michael J. Carey. The EXODUS Extensible DBMS Project: An Overview. In Stanley B. Zdonik and David Maier, editors, *Readings in Object-Oriented Databases*. Morgan Kaufmann, 1989.

[Cat91] R. G. G. Cattell. An Engineering Database Benchmark. In Jim Gray, editor, *The Benchmark Handbook for Database and Transaction Processing Systems*, pages 247–281. Morgan Kaufmann, 1991.

[CBL90] Danny Chen, Ronald E. Barkley, and T. Paul Lee. Insuring Improved VM Performance: Some No-Fault Policies. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 11–22, Berkeley, California, January 1990. USENIX Association.

[CDN93] 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*, pages 12–21, Washington DC., June 1993. ACM Press.

[CG91] Vincent Cate and Thomas Gross. Combining the Concepts of Compression and Caching for a Two-Level File System. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)* [ASP91], pages 200–209.

[Cha92] Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for an Object-Oriented Programming Language.* PhD thesis, Stanford University, March 1992.

[CLLBH92] Jeffrey S. Chase, Henry M. Levy, Edward D. Lazowska, and Miche Baker-Harvey. Lightweight Shared Objects in a 64-bit Operating System. In Andreas Paepcke, editor, *Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '92)*, pages 397–413, Vancouver, British Columbia, October 1992. ACM Press. Published as *ACM SIGPLAN Notices 27*(10), October 1992.

[CM84] George P. Copeland and David Maier. Making Smalltalk a Database System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 316–325, Boston, Massachusetts, June 1984. ACM Press. *ACM SIGMOD Record 14*(2).

[CM88] Albert Chang and Mark F. Mergen. 801 Storage: Architecture and Programming. *ACM Transactions on Computer Systems*, 6(1):28–50, February 1988.

[CRRS93] Khien-Mien Chew, Jyothy Reddy, Theodore H. Romer, and Abraham Silberschatz. Kernel Support for Recoverable-Persistent Virtual Memory. In *Proceedings of the 3rd Symposium on Mach*, pages 215–234, Santa Fe, New Mexico, April 1993. USENIX Association.

[CS92] Rick G. G. Cattell and J. Skeen. Object Operations Benchmark. *ACM Transactions on Database Systems*, 17(1):1–31, March 1992.

[CS93] Khien-Mien Chew and Abraham Silberschatz. The Recoverable-Persistent Virtual Memory Paradigm. Technical Report TR–93–08, The University of Texas at Austin, Austin, Texas, March 1993. Available at `ftp://ftp.cs.utexas.edu/pub/techreports/tr93-08.ps.Z`.

[Den70] Peter J. Denning. Virtual Memory. *ACM Computing Surveys*, 2(3):153–189, September 1970.

[Det92] David L. Detlefs. Garbage Collection and Run-Time Typing as a C++ Library. In *USENIX C++ Conference* [USE92].

[Dou93] Fred Douglis. The Compression Cache: Using On-line Compression to Extend Physical Memory. In *Proceedings of 1993 Winter USENIX Conference*, pages 519–529, San Diego, California, January 1993.

[DSZ90] Alan Dearle, Gail M. Shaw, and Stanley B. Zdonik, editors. *Implementing Persistent Object Bases: Principles and Practice (Proceedings of the Fourth International Workshop on Persistent Object Systems)*, Martha's Vineyard, Massachusetts, September 1990. Morgan Kaufmann.

[Ede92a] Daniel R. Edelson. Precompiling C++ for Garbage Collection. In Bekkers and Cohen [BC92].

[Ede92b] Daniel Ross Edelson. Smart Pointers: They're Smart, But They're Not Pointers. In *USENIX C++ Conference* [USE92], pages 1–19. Technical Report UCSC-CRL-92-27, University of California at Santa Cruz, Baskin Center for Computer Engineering and Information Sciences, June 1992.

[GMS87] Robert A. Gingell, Joseph P. Moran, and William A. Shannon. Virtual Memory Architecture for SunOS. In *USENIX Summer 1987 Technical Conference*, pages 81–94, Phoenix, Arizona, June 1987. USENIX Association.

[GR89] Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison-Wesley, Reading, Massachusetts, 1989.

[HH87] R. Nigel Horspool and Ronald M. Huberman. Analysis and Development of Demand Prepaging Policies. *Journal of Systems and Software*, 7:183–194, 1987.

[HN97] Antony L. Hosking and Aria P. Novianto. Mostly-copying Reachability-based Orthogonal Persistence for C, C++ and Other Intransigents. In *OOPSLA '97 Workshop on Memory Management and Garbage Collection*, October 1997.

[Hos95] Antony L. Hosking. *Lightweight Support for Fine-Grained Persistence on Stock Hardware*. PhD thesis, University of Massachussetts at Amherst, Amherst, Massachussetts, February 1995.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Addison-Wesley, Reading, Massachusetts, 1996. 2nd Edition.

[HR83] Theo Haerder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[IL90] John A. Interrante and Mark A. Linton. Run-Time Access to Type Information in C++. In *USENIX C++ Conference*, Berkeley, California, 1990. USENIX Association.

[JLR$^+$94] Hosagrahar V. Jagadish, Daniel Lieuwen, Rajeev Rastogi, Avi Silberschatz, and S. Sudarshan. Dali: A High Performance Main Memory Storage Manager. In *Twentieth International Conference on Very Large Data Bases*, Santiago, Chile, 1994.

[Joh97] Mark S. Johnstone. *Non-Moving Memory Allocation and Real-Time Garbage Collection.* PhD thesis, The University of Texas at Austin, Austin. Texas, December 1997.

[Kae86] Ted Kaehler. Virtual Memory on a Narrow Machine for an Object-Oriented Language. In *Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '86) Proceedings* [OOP86].

[KC86] Setrag N. Khoshafian and George P. Copeland. Object Identity. In *Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '86) Proceedings* [OOP86], pages 406–416.

[KdRB91] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, Massachusetts, 1991.

[KELS82] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. One-level Storage System. In D. P. Siewiorek, C. G. Bell, and A. Newell, editors, *Computer Structures: Principles and Examples*, pages 135–148. McGraw-Hill, New York, NY, 1982. Originally appeared in IRE Transactions EC-11:(2), 223–235, April 1962.

[KK83] Ted Kaehler and Glenn Krasner. LOOM—Large Object-Oriented Memory for Smalltalk-80 Systems. In Glenn Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*, pages 251–271. Addison-Wesley, 1983.

[KK95] Alfons Kemper and Donald Kossmann. Adaptable Pointer Swizzling Strategies in Object Bases: Design, Realization, and Quantitative Analysis. *VLDB Journal*, 4(3):519–566, July 1995.

[KLM$^+$93] Gregor Kiczales, John Lamping, Chris Maeda, David Keppel, and Dylan McNamee. The Need for Customizable Operating Systems. *Proceedings of the Fourth Workshop on Workstation Operating Systems (WWOS-IV)*, October 1993.

[LAB$^+$81] Barbara Liskov, Russell Atkinson, Toby Bloom, J. Eliot B. Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual.* Number 114 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany, 1981.

[LH89] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[Li86] Kai Li. *Shared Virtual Memory on Loosely-Coupled Processors.* PhD thesis, Yale University, New Haven, Connecticut, 1986.

[Lie93] Jochen Liedtke. Improved IPC by Kernel Design. In *Proceedings of the Fourteenth Symposium on Operating Systems Principles*, pages 175–188, Asheville, North Carolina, December 1993. ACM Press. Published as *Operating Systems Review 27*(5).

[Lie95] Jochen Liedtke. On Microkernel Construction. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, pages 237–250, Copper Mountain Resort, Colorado, December 1995. ACM Press.

[Lie96] Jochen Liedtke. Toward Real Microkernels. *Communications of the ACM*, 39(9):70–77, September 1996.

[Lip91] Stanley B. Lippman. *C++ Primer.* Addison-Wesley, Reading, Massachusetts, 1991. 2nd Edition.

[LLOW91] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore Database System. *Communications of the ACM*, 34(10):50–63, October 1991.

[MBC^{+}89] Ron. Morrison, Alfred L. Brown, Ray Carrick, Richard Connor, Alan Dearle, and Malcolm P. Atkinson. The Napier Type System. In J. Rosenberg and D. Koch, editors, *Third International Workshop on Persistent Object Systems*, pages 3–18, Newcastle, Australia, September 1989. Springer-Verlag.

[MG89] Jose Alves Marques and Paulo Guedes. Extending the Operating System to Support an Object-Oriented Environment. In *Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '89) Proceedings* [OOP89], pages 113–122.

[MGST70] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9:78–117, 1970.

[Mos92] J. Eliot B. Moss. Working with Persistent Objects: To Swizzle or Not to Swizzle? *IEEE Transactions on Software Engineering*, 18(8):657–673, August 1992. Also available as Technical Report 90–38, University of Massachusetts, Amherst, Massachusetts, May 1990.

[MS95] Mark L. McAuliffe and Marvin H. Solomon. A Trace-Based Simulation of Pointer Swizzling Techniques. In *Proceedings of the International Conference on Database Engineering*, pages 52–61, Taipei, Taiwan, March 1995. IEEE.

[Nel91] Greg Nelson, editor. *Systems Programming in Modula-3.* Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[NNM^{+}96] Vivek R. Narasayya, Tze Sing Eugene Ng, Dylan McNamee, Ashutosh Tiwary, and Henry M. Levy. Reducing the Virtual Memory Overhead of Swizzling. In *Proceedings of the FifthInternational Workshop on Object Orientation in Operating Systems*, Seattle, Washington, October 1996. IEEE Press.

[OOP86] *Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '86) Proceedings.* ACM Press, October 1986. Published as *ACM SIGPLAN Notices 21*(11), November 1986.

[OOP89] *Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '89) Proceedings*, New Orleans, Louisiana, 1989. ACM Press.

[RC89] Joel E. Richardson and Michael J. Carey. Persistence in the E Language: Issues and Implementation. *Software Practice and Experience*, 19(12):1115–1150, December 1989.

[RK68] Brian Randell and C. J. Kuehner. Dynamic Storage Allocation Systems. *Communications of the ACM*, 12(7):297–306, May 1968.

[RO91] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pages 1–15, Pacific Grove, California, October 1991. ACM Press. Published as *Operating Systems Review 25*(5).

[Rus91] V. F. Russo. *An Object-Oriented Operating System.* PhD thesis, University of Illinois at Urbana-Champaign, Champaign-Urbana, Illinois, January 1991.

[SCD90] Daniel T. Schuh, Michael J. Carey, and David J. DeWitt. Persistence in E Revisited—Implementation Experiences. In Dearle et al. [DSZ90].

[SKT94] Shinji Suzuki, Masaru Kitsuregawa, and Mikio Takagi. An Efficient Pointer Swizzling Method for Navigation Intensive Applications. In Antonio Albano and Ron Morrison, editors, *Sixth International Workshop on Persistent Object Systems*, pages 79–95, Tarascon, France, September 1994. Springer-Verlag.

[SKW92] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: An Efficient, Portable Persistent Store. In Albano and Morrison [AM92], pages 11–33.

[SL92] Bjarne Stroustrup and Dmitry Lenkov. Run-time Type Identification for C++ (revised). In *USENIX C++ Conference* [USE92].

[Sta82] James William Stamos. A Large Object-Oriented Virtual Memory: Grouping Strategies, Measurements, and Performance. Technical Report SCG-82-2, Xerox Palo Alto Research Center, Palo Alto, California, May 1982.

[Str87] Bjarne Stroustrup. The Evolution of C++, 1985 to 1987. In *USENIX C++ Workshop*, pages 1–22. USENIX Association, 1987.

[SW91] Walter R. Smith and Robert V. Welland. A Model for Address-Oriented Software and Hardware. In *Proceedings of the 25th Hawaii International Conference on System Sciences.* IEEE, January 1991.

[SZ90] Eugene Shekita and Michael Zwilling. Cricket: A Mapped, Persistent Object Store. In Dearle et al. [DSZ90], pages 89–102.

[TL93] Chandramohan A. Thekkath and Henry M. Levy. Limits to Low-Latency Communiation on High-Speed Network. *ACM Transactions on Computer Systems*, 11(2):179–203, May 1993.

[TL94] Chandramohan A. Thekkath and Henry M. Levy. Hardware and Software Support for Efficient Exception Handling. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 110–119, San Jose, California, October 1994.

[TNL95] Ashutosh Tiwary, Vivek R. Narasayya, and Henry M. Levy. Evaluation of OO7 as a System and an Application Benchmark. In *OOPSLA '95 Workshop on Object Database Behavior, Benchmarks and Performance*, Austin, Texas, October 1995.

[USE92] USENIX Association. *USENIX C++ Conference*, Portland, Oregon, August 1992.

[Vah96] Uresh Vahalia. *Unix Internals: The New Frontiers.* Prentice-Hall, Upper Saddle River, New Jersey, 1996.

[VD92] Francis Vaughan and Alan Dearle. Supporting Large Persistent Stores Using Conventional Hardware. In Albano and Morrison [AM92].

[vEBB95] Thorsten von Eicken, Anindya Basu, and Vineet Buch. Low-Latency Communication over ATM Networks using Active Messages. *IEEE Micro*, 15(1):46–53, February 1995.

[WC96] ISO WG21 and ANSI X3J16 Committee. Working Paper for Draft Proposed International Standard for Information Systems—Programming Language C++, December 1996. Document numbers WG21/N1043 (ISO) and X3J16/96-0225 (ANSI). Current public draft available at `http://www.cygnus.com/misc/wp/`.

[WD92] Seth J. White and David J. Dewitt. A Performance Study of Alternative Object Faulting and Pointer Swizzling Strategies. In *18th International Conference on Very Large Data Bases*, Vancouver, British Columbia, October 1992. Morgan Kaufmann.

[WD94] 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, Minneapolis, Minnesota, May 1994. ACM Press.

[Whi94] Seth J. White. *Pointer Swizzling Techniques for Object-Oriented Database Systems.* PhD thesis, University of Wisconsin—Madison, Madison, Wisonsin, 1994.

[Wil90] Paul R. Wilson. Some Issues and Strategies in Heap Management and Memory Hierarchies. In *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems*, October 1990. Also appears in *ACM SIGPLAN Notices 23*(3):45–52, March 1991.

[Wil91] Paul R. Wilson. Operating System Support for Small Objects. In *International Workshop on Object Orientation in Operating Systems*, pages 80–86, Palo Alto, California, October 1991. IEEE Press.

[Wil92] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In Bekkers and Cohen [BC92], pages 1–42.

[Wil97] Paul R. Wilson. Garbage Collection. *ACM Computing Surveys*, 1997. Expanded version of [Wil92]. Draft available at `ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps`. In revision, to appear.

[WJ93] Paul R. Wilson and Mark S. Johnstone. Truly Real-Time Non-Copying Garbage Collection. In *OOPSLA '93 Workshop on Memory Management and Garbage Collection*, December 1993. Available at `ftp://ftp.cs.utexas.edu/pub/garbage/GC93`.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *1995 International Workshop on Memory Management*, Kinross, Scotland, UK, 1995. Springer Verlag LNCS.

[WKB97a] Paul R. Wilson, Scott F. Kaplan, and V. B. Balayogahan. Compressed Paging. In preparation, 1997.

[WKB97b] Paul R. Wilson, Scott F. Kaplan, and V. B. Balayogahan. Current Research in Compressed Virtual Memory. In preparation, 1997.

[WKBK97] Paul R. Wilson, Scott F. Kaplan, V. B. Balayogahan, and Sheetal V. Kakkad. Virtual Memory Reference Tracing Using User-Level Access Protections. In preparation, 1997.

[WKM94] Paul R. Wilson, Sheetal V. Kakkad, and Shubhendu S. Mukherjee. Anomalies and Adaptation in the Analysis and Development of Prepaging Policies. *Journal of Systems and Software*, 27:147–153, November 1994.

[WLM91] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective Static-Graph Reorganization to Improve Locality in Garbage-Collected Systems. In *Proceedings of the 1991 SIGPLAN Conference on Programming Language Design and Implementation*, pages 177–191, Toronto, Ontario, June 1991. ACM Press. Published as *ACM SIGPLAN Notices 26*(6), June 1992.

[WM89] Paul R. Wilson and Thomas G. Moher. Design of the Opportunistic Garbage Collector. In *Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '89) Proceedings* [OOP89], pages 23–35.

[WWH87] Ifor W. Williams, Mario I. Wolczko, and Trevor P. Hopkins. Dynamic Grouping in an Object-Oriented Virtual Memory Hierarchy. In *European Conference on Object Oriented Programming*, pages 87–96, Paris, France, June 1987. Springer-Verlag.

[You89] Michael W. Young. *Exporting a User Interface to Memory Management from a Communication-Oriented Operating System*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 1989. Also available as Technical Report CMU-CS-89-202.

Vita

Sheetal Vinod Kakkad was born in Rajkot (Gujarat), India on June 16, 1968, the son of Jayashree Kakkad and Vinod Kakkad. After completing his work at Rajkumar College, Rajkot, in 1985, he entered the University of Bombay, Bombay, India. He received the degree of Bachelor of Engineering from the University of Bombay in July 1989. During the next two years, he was employed as a lecturer at the R. A. Institute of Technology, New Bombay and as a software engineer at Citicorp Overseas Software Limited, Bombay. In August 1991, he entered the Graduate School of The University of Texas at Austin. He received the degree of Master of Science in Computer Sciences in May 1994.

Permanent Address: 10306 Morado Cove #157, Austin, Texas 78759

This dissertation was typeset with LaTeX 2ε[2] by the author.

[2] LaTeX 2ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin.