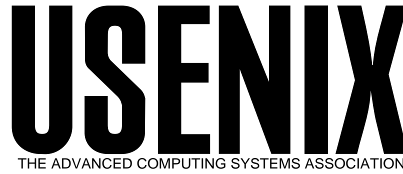

The following paper was originally published in the

5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '99)

San Diego, California, USA, May 3–7, 1999

Address Translation Strategies in the Texas Persistent Store

Sheetal V. Kakkad
Somerset Design Center, Motorola

Paul R. Wilson
University of Texas at Austin

© 1999 by The USENIX Association
All Rights Reserved

Rights to individual papers remain with the author or the author's employer. Permission is granted for noncommercial reproduction of the work for educational or research purposes. This copyright notice must be included in the reproduced paper. USENIX acknowledges all trademarks herein.

For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738
Email: office@usenix.org WWW: http://www.usenix.org

Address Translation Strategies in the Texas Persistent Store

Sheetal V. Kakkad[*]
Somerset Design Center
Motorola
Austin, Texas, USA
svkakkad@acm.org

Paul R. Wilson[†]
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas, USA
wilson@cs.utexas.edu

Abstract

Texas is a highly portable, high-performance persistent object store that can be used with conventional compilers and operating systems, without the need for a preprocessor or special operating system privileges. Texas uses *pointer swizzling at page fault time* as its primary address translation mechanism, translating addresses from a persistent format into conventional virtual addresses for an entire page at a time as it is loaded into memory.

Existing classifications of persistent systems typically focus only on address translation taxonomies based on semantics that we consider to be confusing and ambiguous. Instead, we contend that the *granularity choices* for design issues are much more important because they facilitate classification of different systems in an unambiguous manner unlike the taxonomies based only on address translation. We have identified five primary design issues that we believe are relevant in this context. We describe these design issues in detail and present a new general classification for persistence based on the granularity choices for these issues.

Although the coarse granularity of pointer swizzling at page fault time is efficient in most case, it is sometimes desirable to use finer-grained techniques. We examine different issues related to fine-grained address translation mechanisms, and discuss why these are not suitable as general-purpose address translation techniques. Instead, we argue for a mixed-granularity approach where a coarse-grained mechanism is used as the primary address translation scheme, and a fine-grained approach is used for specialized data structures that are less suitable for the coarse-grained approach.

We have incorporated fine-grained address translation in Texas using the C++ *smart pointer* idiom, allowing programmers to choose the kind of pointer used for any data member in a particular class definition. This approach maintains the important features of the system: persistence that is orthogonal to type, high performance with standard compilers and operating systems, suitability for huge shared address spaces across heterogeneous platforms, and the ability to optimize away pointer swizzling costs when the persistent store is smaller than the hardware-supported virtual address size.

1 Introduction

The Texas Persistent Store provides portable, high-performance persistence for C++ [16, 8], using pointer swizzling at page fault time [23, 8] to translate addresses from persistent format into virtual memory addresses. Texas is designed to implement and promote *orthogonal persistence* [1, 2]. Orthogonal persistent systems require that any arbitrary object can be made persistent without regard to its type; that is, persistence is viewed as the storage class[1] of an object rather than as a property of its type. 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. In contrast, *class-based* persistent systems require that any type or class that may be instantiated to create persistent objects *must* inherit from a top-level abstract "persistence" class, which defines the *interface* for saving and restoring data from a persistent object store.

Texas uses *pointer swizzling at page fault time* as the primary address translation technique. When a page is brought into memory, all pointers in the page are identified and translated (or swizzled) into raw virtual ad-

[*] The work reported in this paper was performed as part of the author's doctoral research at The University of Texas at Austin.

[†] This research was supported by grants from the IBM Corporation and the National Science Foundation.

[1] 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 object is typically allocated on the data stack, and its lifetime is bounded by the scope in which it was allocated.

dresses. If the corresponding referents are not already in memory, virtual address space is *reserved* for them (using normal virtual memory protections), allowing for the address translation to be completed successfully. As the application dereferences pointers into non-resident pages, these are intercepted (using virtual memory access protection violations) and the data is loaded from the persistent store, causing further pointer swizzling and (potential) address space reservation for references to other non-resident data. Since running programs only see pointers in their normal hardware-supported format, conventionally-compiled code can execute at full speed without any special pointer format checks.

This page-wise address translation scheme has several advantages. One is that it exploits spatial locality of reference, allowing a single virtual memory protection violation to trigger the translation of all persistent addresses in a page. Another is that off-the-shelf compilers can be used, exploiting virtual memory protections and trap handling features available to normal user processes under most modern operating systems.

However, as with any other scheme that exploits locality of reference, it is possible for some programs to exhibit access patterns that are unfavorable to a coarse-grained scheme; for example, sparse access to large indexing structures may unnecessarily reserve address space with page-wise address translation than with more conventional pointer-at-a-time strategies. It is desirable to get the best of both worlds by combining coarse-grained and fine-grained address translation in a single system.

In Texas, we currently support a fine-grained address translation strategy by using *smart pointers* [17, 7, 12] that can replace normal pointers where necessary. Such pointers are ignored by the usual swizzling mechanism when a page is loaded into memory; instead, each pointer is individually translated as it is dereferenced using overloaded operator implementations. The mixed-granularity approach works well, as shown by experimental results gathered using the OO1 benchmark [4, 5].

The remainder of this paper is structured as follows. In Section 2, we describe existing well-known address translation taxonomies put forth by other researchers, and motivate the need for a general classification of persistence presented in Section 3. In Section 4, we discuss issues about fine-grained address translation techniques, and why we believe that a pure fine-grained approach is not suitable for general use. We describe the implementation of mixed-granularity address translation in Texas in Section 5 and the corresponding performance results in Section 6, before wrapping up in Section 7.

2 Address Translation Taxonomies

Persistence has been an active research area for over a decade and several taxonomies for pointer swizzling techniques have been proposed [13, 9, 11, 19]. In this section, we describe important details about each of these taxonomies and highlight various similarities and differences among them. We also use this as a basis to provide motivation for a general classification of persistent systems based on granularity issues, which we describe in Section 3.

2.1 Eager vs. Lazy Swizzling

Moss [13] 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. That is, the need to check 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. 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 applications.

Other researchers [9, 11] have also used classifications along similar lines in their own studies. However, we consider this classification to be ambiguous and confusing for general use. 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 an application is never allowed to "see" any untranslated pointers. There is no need to explicitly check the format of a pointer before using it, making pointer swizzling at page fault time an eager swizzling scheme. On the other hand, the basic approach is incremental in nature; 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, but eager at the granularity of objects because it swizzles multiple objects—an entire page's worth—at one time. As such, we contend that the granularity at which address translation is performed is the fundamental issue.

2.2 Node-Marking vs. Edge-Marking Schemes

Moss also describes another classification based on the strategy used for distinguishing between resident and non-resident data in 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. The address translation mechanisms are then classified as either *node-marking* or *edge-marking* schemes.

Figure 1 shows the basic structure for node-marking and edge-marking schemes. As the name suggests, *edge-marking* schemes mark the graph edges—the pointers between objects—to indicate whether they have been translated into local format and reference resident objects. In contrast, *node-marking* schemes guarantee that all references in resident objects are always translated, and the graph nodes themselves are marked to indicate whether they are non-resident. In other words, edges are guaranteed to be valid local references but the actual referents may be non-resident. Note that the marking applies only to non-resident entities, that is, either to nodes that are non-resident or to (untranslated) edges that reference non-resident nodes.

Figure 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 while pointers to non-resident objects are swizzled into references to proxy objects. When the application follows a reference to a proxy object, the system loads the referent (F in the figure) from the database and updates the proxy object to reference the newly-resident object (Figure 2b). Alternatively, the proxy object 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 object may (eventually) be reclaimed by the system. Note, however, that the compiled code must still check for the presence of proxy objects on *every* pointer dereference because of the possibility that any pointer *may* reference a proxy object. This adds continual checking overhead, even when all pointers directly reference data objects without 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 normal approach—unlike the classic implementation, there are no *explicit* proxy objects for non-resident in pointer swizzling at page fault time. Instead, access-protected virtual address space pages *act* as proxy objects.[2] As the application progresses and more data is loaded into memory, the pages that were previously protected are now unprotected because they contain valid data. The major advantage of this approach is that there is no need to reclaim proxy objects (because none exist); consequently, there are no further indirections that must be dealt with by compiled code, avoiding continual format checks that would otherwise be necessary.

2.3 General Classification for Persistence

We have seen that existing classifications focus only on address translation techniques. While address translation is an important issue, it constitutes only one of several design issues that must be considered when implementing persistence. We have identified a set of design issues that we believe are fundamental to efficient implementation of any persistence mechanism. We believe that a specific combination of these issues can be used to characterize any particular implementation. In effect, we are proposing a general classification scheme based on *granularities of fundamental design aspects*.

A classification based on "eager" and "lazy" swizzling is ambiguous, because it does not attack the problem at the right level of abstraction. The real issue in the distinction between lazy and eager swizzling 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's "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 Moss's "pure eager swizzling" approach).[3]

[2] In fact, unmapped virtual address space pages can also serve the same purpose.

[3] While crude, this is actually not uncommon. Traditionally, Lisp

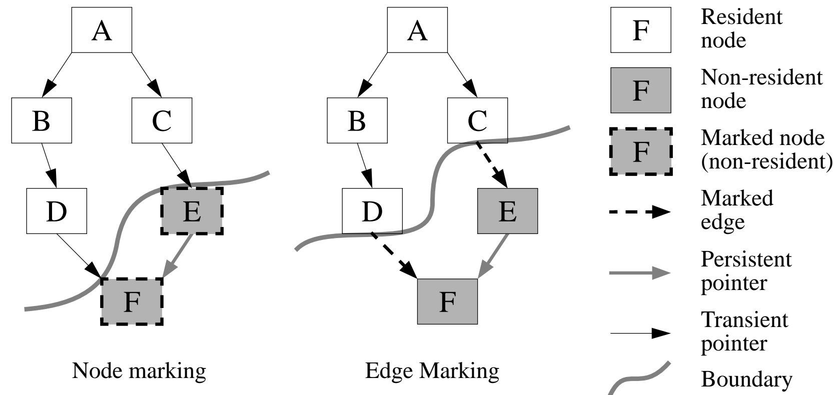

Figure 1: Node-marking and edge-marking schemes

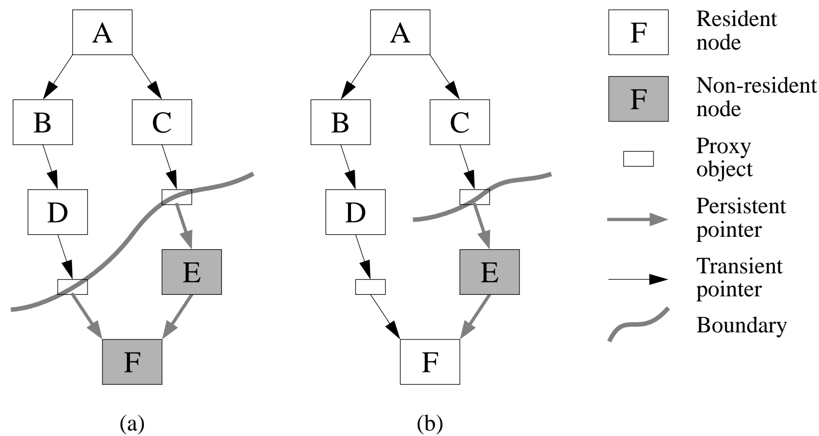

Figure 2: Node-marking scheme using proxy objects

We believe that it is preferable to consider address translation (and other design issues) from the perspective of a *granularity choice* rather than an *ad hoc* classification based on confusing translation semantics. In fact, the ambiguity arises primarily because the classifications either do not clearly identify the granularity, or, because they unnecessarily adhere to a single predetermined granularity. Discussing all design issues in terms of granularity choices provides a uniform framework 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 many schemes are both "eager" and "lazy" at different scales, along several dimensions.

and Smalltalk systems have supported the saving and restoring of entire heap images in a "big inhale" relocation.

3 Granularity Choices for Persistence

We have identified a set of five design issues (including address translation) that are relevant to the implementation of a 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 the different issues can then be used to characterize persistent systems. The specific design issues that we describe in this section are the granularities of *address translation*, *address mapping*, *data fetching*, *data caching* and *checkpointing*. In the remainder of this section, we define and discuss each issue in detail[4] and also present the rationale behind the granularity choices for these issues in our implementation of orthogonal persistence in Texas.

[4]Note that while we describe each issue individually, these granularity choices are strongly related. It is possible (and quite likely) that a system may make the same granularity choice on multiple issues for various reasons.

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 runtime software checks in compiled code, taking advantage of user-level memory protection facilities of most modern operating systems. Sometimes, however, it is necessary to change the granularity choice for a particular issue to accommodate the special needs of unusual situations. It is possible to address these issues at a different granularity in a way that integrates gracefully into the general framework of Texas.

3.1 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 use of virtual memory pages has several advantages in terms of overall efficiency because we use 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 protection faults and swizzling entire pages.

As described in Section 5, it is 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 allows fine-grained translation on individual pointers, the granularity of address translation in those cases would be a single pointer.

3.2 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 the 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 any 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.

However, 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.

3.3 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 two granularities presented above, we use a virtual memory page for this purpose in the current implementation of Texas. 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 other design issues in our implementation.

It is possible to change the granularity of fetching without affecting any other granularity choices. In essence, we can implement our own prefetching to preload data from the persistent store. This may actually be desirable for some applications when using raw unbuffered I/O instead of normal file I/O [8]. Depending on the access characteristics of the application and the dataset size, the overall I/O costs can be reduced by prefetching several (consecutive) pages instead of a single faulted-on page. In general, the granularity of data fetching is intimately tied to the I/O strategy that is selected in the implementation.

3.4 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 mem-

ory page, because Texas relies exclusively on the virtual memory system for caching persistent data.

A persistent page is usually cached in a *virtual memory* page as far as Texas is concerned. The virtual memory system determines whether the page actually resides in RAM (i.e., physical 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 RAM and on disk*; that is, virtual memory caching is left up to the underlying virtual memory system, which does its job in the normal way.

It is, of course, possible to change this behavior such that Texas directly manages 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 advantageous for most purposes because it prevents any particular application from monopolizing system resources (RAM in this case). As such, applications using Texas are just normal programs, requiring no special privileges or resources; they "play well with others" rather than locking up large amounts of RAM as many database and persistent systems do.

3.5 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.

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 write-ahead 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 [16].

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 [16]. 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 locality characteristics. The granularity of checkpointing in this case is equivalent to the size of the "diffs" which are saved to stable storage.[5]

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 can intervene to perform some inline compression using specialized algorithms tuned to heap data. Further research has been initiated in this area [24] and initial results indicate that the I/O cost can be reduced by about a factor of two, and that data can be compressed fast enough to double the effective disk bandwidth on current machines. As CPU speeds continue to increase fast than disk speeds, the cost of compression shrinks exponentially relative to cost of disk I/O. Further reduction in costs is also possible with improved compression algorithms and adaptive techniques.

4 Fine-grained Address Translation

There are several factors that motivated us to develop a coarse-grained mechanism over a fine-grained approach when implementing pointer swizzling at page fault time in Texas. The primary motivation is the fact that we wanted to exploit existing hardware to avoid expensive residency checks in software. However, we believe that there are also other factors against using a fine-grained approach as the primary address translation mechanism. In this section, we discuss 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 most current fine-grained schemes appear to be slower than pointer swizzling at page fault time in terms of the basic address translation performance.

[5] The basic "diffing" technique has been implemented in the context of QuickStore [19]; preliminary results are encouraging, although more investigation is required.

4.1 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 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 translated into valid local format or not[6] 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 is directly managed by the persistence mechanism. This cost is usually directly proportional to the rate of execution because it requires a read barrier.[7] We discuss this further in the next subsection.
- As resident objects are evicted from memory, 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 directly proportional to the rate of eviction and 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.[8] 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.

4.2 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.

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. 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

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

[7] The term *read barrier*, borrowed from garbage collection research [21], 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.

[8] Some systems use crude replacement and/or checkpointing policies to simplify integration with persistence and garbage collection mechanisms. These may incur additional costs due to the choice of suboptimal policies.

[9] *Interior pointers* are those that point inside the bodies of objects rather than at their heads.

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

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 references to the object being evicted. Kemper and Kossman [9] solve 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 [11] use a different data structure, called the *swizzle table*, a fixed-size hash table that maintains a list of all swizzled pointers in the system. Both these approaches are generally unfavorable because they increase the storage requirements (essentially doubling the number of pointers at the minimum) and the implementation complexity.

4.3 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 [14, 15]) 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 execution speed is improved up by a factor of ten using a state-of-the-art compiler.

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 transient 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 [14, 15] 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 [19], E is about 48% slower than transient C/C++ for hot traversals of the OO1 database benchmark [4, 5].[11] This is a fairly significant considering that the overhead of our system is *zero* for hot traversals and much smaller (less than 5%) otherwise [8].

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 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, 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]
- Complex data-flow analysis and code generation techniques are required to optimize some of the costs associated with the read barrier used in the implementation. Furthermore, such extra optimizations may cause unwanted code bloat.
- Although the residency property can be treated as a type so that Self-style optimizations [6] can be applied to eliminate residency checking, it is not easy to do so; unlike types, residency may change across procedure calls depending on the dynamic run-time state of the application. As such, residency check elimination is fundamentally a non-local problem that depends on complex analysis of control flow and data flow.

Based on these arguments, we believe that fine-grained translation techniques are comparatively 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

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

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

advantages. A primary one is the potential savings in I/O because fine-grained schemes can 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 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. Inevitably, this will incur an appreciable run-time cost, even if that cost is "billed" to multiple desirable features. Such costs may be reduced in the future if fine-grained checking is supported in hardware.

5 Mixed-granularity Address Translation in Texas

Pointer swizzling at page fault time usually provides good performance for most applications with good locality of reference. However, 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 from the root. If the tree nodes are large 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. However, 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 actual translation of a pointer may be triggered by one of two events—either when it is "found"[13] or when it is dereferenced.

There are many ways of implementing a fine-grained (pointer-wise) address translation mechanism as we described above. We have selected an implementation strategy that remains consistent with our goals of portability and compatibility with existing off-the-shelf compilers, by using the C++ *smart pointer* abstraction [17, 7, 12]. Below, we first briefly explain this abstraction and then describe how we use it for implementing fine-grained translation in Texas. We also discuss how both fine-grained and coarse-grained schemes can coexist to create a mixed-granularity environment.

5.1 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 using a C++ class with overloaded operators supporting these pointer operations, it is possible to execute arbitrary code as part of any such operation. While smart pointers were originally used in garbage collectors to implement write barriers [22, 21], they are also suitable for implementing address translation; the overloaded pointer dereference operations (via the "*" and "->" operators) can implement the necessary translation from persistent pointers into transient pointers.

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
  T *operator -> (); // dereference
  operator T * ();   // cast to 'T *'
  ...
};
```

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

[13] A pointer is "found" when its location becomes known. This is similar to the notion of "swizzling upon discovery" as described in [20].

```
class Node;          // assume defined
Node *node_p;        // regular pointer
Ptr<Node> node_sp; // smart pointer
...
node_p->some_method();
node_sp->some_method();
```

Note that we have only shown some of the operators in the declaration. Also, we avoid describing the private data members of the smart pointer because the interface is much more important than the internal representation; 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 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 limited (compile-time) reflection [10] into C++ for builtin data types (i.e., pointers).[14] However, as described in [7], 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 unspecified implementation dependence. Thus, we do not advocate smart pointers for arbitrary usage across the board, but they are useful in situations where further control is required over pointer operations.

5.2 Fine-grained Address Translation

In order to implement fine-grained address translation in Texas, we must swizzle individual pointers, instead of entire pages at a time, thereby reducing the consumption of virtual address space for sparsely-accessed data structures with high fanout. By using smart pointers for this purpose, we allow the programmer to easily choose 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 fetching is still a page, not individual objects, to avoid excessive I/O costs. Below we describe at least two possible ways to handle fine-grained address translation, and discuss why we choose one over the other.

5.2.1 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.[15] 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 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 (*a la* the C++ == operator)—when two smart pointers are compared, the comparison can only be made after ensuring that both 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 check can complete. Thus, a simple equality check, on average, can become more expensive than desired.

One solution is to make the pointer field large enough to store both persistent and virtual address values, as in E [14, 15]. 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 implemented by simply comparing persistent addresses without regard to swizzling, at the expense of additional storage.

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 has to be cached in the pointer field (either E-style or otherwise), that is, we must *modify* the pointer. Since virtual memory protections are used to detect updates initiated by the application for checkpointing purposes, updating a smart pointer to cache the swizzled address will generate "false positives" for updates, causing unnecessary checkpointing. We could work around this problem by first resetting the permissions on the page, swizzling (and caching) the pointer,

[14] C++ already provides limited reflective capabilities in the form of operator overloading for user-defined types and classes. However, this fails to support completely transparent redefinition of pointer operations in arbitrary situations.

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

and then restoring the permissions on the page. However, this is very slow on average because it requires kernel intervention to change page protections.

5.2.2 Translations at Each Use

We have seen that a simple fine-grained swizzling mechanism is not as desirable because of its unusual interactions with the operating system and the virtual memory system. However, we can slightly modify the basic technique and overcome most of the disadvantages without losing any of the benefits.

The solution 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 overhead because the pointer fields are always in the same representation and can be compared directly.

Pointer dereferences also do not incur any additional checking overhead. The cost of translating at each use does not add much 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 computation with the resulting target object before dereferencing another smart pointer.

The advantage of this approach is that the pointer fields do not need to be modified because the translated address values are never cached, and all unwanted interactions with checkpointing and the virtual memory system are avoided. Of course, this approach is still unsuitable as a general swizzling mechanism compared to the pointer swizzling at page fault time for reasons described in Section 4.

5.3 Combining Coarse-grained and Fine-grained 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 through the use of the smart pointer abstraction. The details of a fine-grained address translation scheme are hidden, making the approach partially reflective.

We have implemented mixed-granularity address translation in Texas by combining a fine-grained approach using smart pointers that are translated at every use, along with the standard coarse-grained approach. This allows better programmer control over the choice of data structures for which fine-grained address translation is used, while maintaining the overall performance of pointer swizzling at page fault time.

6 Performance Measurements

We present our experimental results for different address translation granularities using the standard OO1 database benchmark [4, 5] with some minor variations as the workload for our experimental measurements. We first briefly explain the rationale for choosing the OO1 benchmark for our performance measurements, then describe the experimental design followed by the actual results, and finally end with a summary.

6.1 Benchmark Choice

Most performance measurements and analysis of persistent object systems (and object-oriented database systems) have been done using *synthetic benchmarks* instead 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.

The OO1 and OO7 [3] benchmarks have become quite popular among various benchmarks, and have been used widely for measuring the performance of persistent systems. However, we posit that these benchmarks are *not representative* of typical real-world applications, because they have not been validated against applications in the domain they represent; other researchers [18] have also reached similar conclusions. As such, the experimental results from these benchmarks should be interpreted with caution. The apparently "empirical" nature of these "experimental" results is likely to lull people into relying on the results more than appropriate. It is important to always remember that while the results are obtained empirically, they are ultimately derived from a synthetic benchmark and are only as good as the map-

ping of benchmark behavior onto real applications.

Although OO1 is a crude benchmark and does not strongly correspond to a real application, we use it 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 is also 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. Further discussion on sythetic benchmarks and their applicability is available in [8].

6.2 Experimental Design

The benchmark database is made up of a set of *part objects* 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—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.[16] 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.

We use the OO1 benchmark traversal operation (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) for 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).[17] 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.

6.3 Experimental Results

We present results for the OO1 traversal operations corresponding to different address translation granularities for the data structures used during the traversals. In particular, we are interested in three different address translation granularities, namely *coarse-grained*, *mixed-granularity* and *fine-grained* strategies. The following table describes the types of pointers used for each granularity and the corresponding key in the results.

Granularity	Type(s) of pointers	Key
coarse	all language-supported	all-raw
mixed	smart for index	smart-index
fine	all smart	all-smart

We use CPU time[18] instead of absolute real time because the difference in performance is primarily due to differences in faulting and swizzling, and allocating address space for reserved pages. Unfortunately, CPU-time timers on most operating systems have a coarse granularity (typically in several milliseconds), 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. Thus, we use an older SPARCstation ELC, which is slow enough to offset the coarse granularity of the timers, while providing reasonable results.

Figure 3 presents the CPU time for all traversals in an entire traversal set run on a large database. As expected, the cost for "all-raw" case (coarse-grained address translation) 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

[16] 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.

[17] This is different from the standard benchmark specification containing only 20 traversals (split as 1 cold, 9 warm, and 10 hot traversals); we run more warm traversals because we believe that 9 traversals are not sufficient to provide meaningful results, especially for the large database case.

[18] We refer to the sum of *user* and *system* time as the *CPU time*.

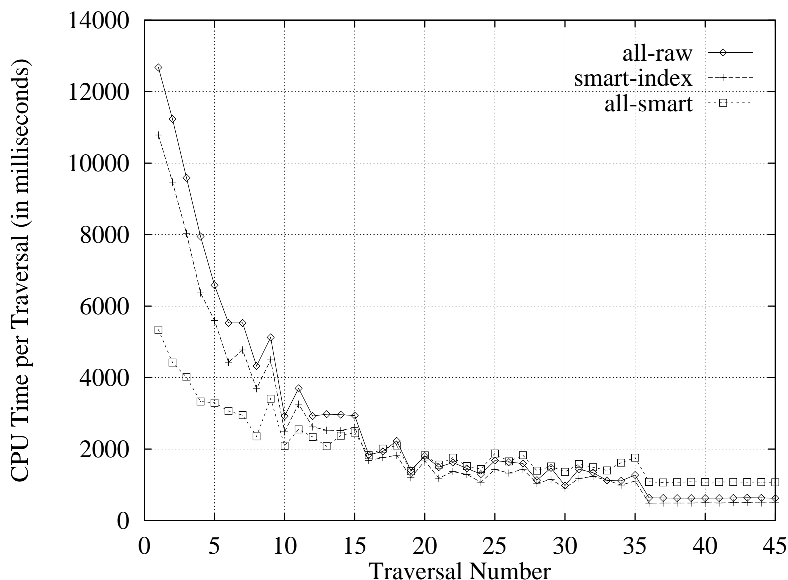

Figure 3: CPU time for traversal on large database

used by the application. This is exacerbated by the poor locality of reference in the benchmark traversals as 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.

Note that the cost for the "all-smart" case (fine-grained address translation) 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 the "smart-index" case (mixed-granularity address translation) falls between the other two cases for the first 15 traversals. This is also reasonable because only the 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 (36 through 45). 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, and this cost is incurred even if the target object is resident. In contrast, the "all-raw" case has zero overhead for hot traversals.[19]

Figure 4 shows the corresponding results for the small database, where only the first 3 or 4 traversals contain faulting and swizzling.[20] Once again, a phenomenon similar to the one in large database results can be seen in the current results, but only for the initial traversals. In particular, the CPU time is highest for the "all-raw" case and lowest for the "all-smart" case. Also as before, the two granularities swap their positions for the hot traversals; the "all-smart" case is more expensive because of the continual translation overhead at every use. Finally, as expected, the "all-raw" and "smart-index" results are identical for hot traversals because no index pointers are dereferenced.

6.4 Summary

The results presented above support our assertion that fine-grained address translation can be effectively used for data structures with high fanout that are less conducive for a coarse-grained scheme. At the same time, a pure fine-grained approach is not the best performing as the primary address translation mechanism in Texas because of various overheads associated with it.

One problem with using the OO1 benchmark is that the operations do not perform any real computation (unlike

[19]The "smart-index" results should be identical to the "all-raw" results for hot traversals because there is no index lookup, and no smart pointers need to be translated. We attribute the difference between the hot results in these two cases to caching effects.

[20]Most of the database is memory-resident within the first few traversals because of the extremely poor locality characteristics in the connections.

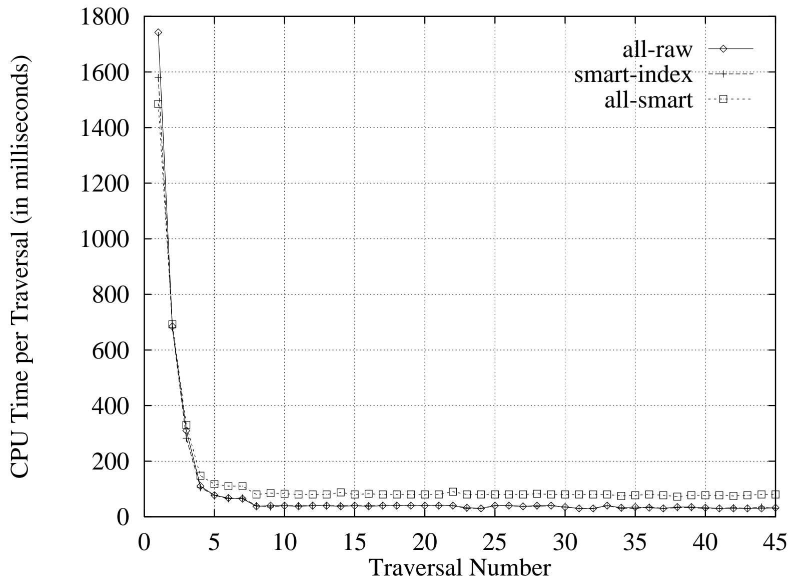

Figure 4: CPU time for traversal on small database

in actual applications) with objects that are traversed. As such, the cost of fine-grained translation is highlighted as a larger component of the total cost than it typically would be in an actual application that performs real "work" on data objects as they are traversed.

7 Conclusions

We presented a discussion on address translation strategies, both in the context of the Texas persistent store and for general persistence implementations. We also proposed a new classification for persistence in terms of granularity choices for fundamental design issues rather than using taxonomies based only on address translation semantics, and discussed each choice that we made in Texas.

We also discussed issues related to fine-grained address translation, including their inherent costs that make them unsuitable as the primary address translation mechanism in a persistence implementation. Instead, we discussed how a mixed-granularity approach can be used to selectively incorporate fine-grained address translation in the application.

We presented our implementation of mixed-granularity address translation in Texas which combines the C++ smart pointer idiom for the fine-grained translation component with the normal pointer swizzling at page fault time mechanism for the coarse-grained translation component, while maintaining portability and compatibility of the system.

Our basic performance results using the OO1 benchmark have shown that the mixed-granularity approach works well for applications with data structures that do not provide the best performance with a pure coarse-grained approach. However, further performance measurements are necessary, especially using real applications instead of synthetic benchmarks which do not always model reality very well.

References

[1] 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.

[2] Malcolm P. Atkinson and Ron Morrison. Orthogonally Persistent Object Systems. *VLDB Journal*, 4(3), 1995.

[3] 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.

[4] R. G. G. Cattell. An Engineering Database Benchmark. In Jim Gray, editor, *The Benchmark Hand-*

book for Database and Transaction Processing Systems. Morgan Kaufmann, 1991.

[5] Rick G. G. Cattell and J. Skeen. Object Operations Benchmark. *ACM Transactions on Database Systems*, 17(1):1–31, March 1992.

[6] 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.

[7] Daniel Ross Edelson. Smart Pointers: They're Smart, But They're Not Pointers. In *USENIX C++ Conference*, pages 1–19, Portland, Oregon, August 1992. USENIX Association.

[8] Sheetal V. Kakkad. *Address Translation and Storage Management for Persistent Object Stores*. PhD thesis, The University of Texas at Austin, Austin. Texas, December 1997. Available at `ftp://ftp.cs.utexas.edu/pub/garbage/kakkad-dissertation.ps.gz`.

[9] 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.

[10] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Massachusetts, 1991.

[11] 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.

[12] Scott Meyers. Smart Pointers. *C++ Report*, 1996. Article series published from April through December. Available at `http://www.aristeia.com/magazines.html`.

[13] 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.

[14] 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.

[15] Daniel T. Schuh, Michael J. Carey, and David J. DeWitt. Persistence in E Revisited—Implementation Experiences. In Alan Dearle, Gail M. Shaw, and Stanley B. Zdonik, editors, *Proceedings of the Fourth International Workshop on Persistent Object Systems*, Martha's Vineyard, Massachusetts, September 1990.

[16] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: An Efficient, Portable Persistent Store. In Antonio Albano and Ron Morrison, editors, *Fifth International Workshop on Persistent Object Systems*, pages 11–33, San Miniato, Italy, September 1992. Springer-Verlag.

[17] Bjarne Stroustrup. The Evolution of C++, 1985 to 1987. In *USENIX C++ Workshop*, pages 1–22. USENIX Association, 1987.

[18] Ashutosh Tiwary, Vivek R. Narasayya, and Henry M. Levy. Evaluation of OO7 as a System and an Application Benchmark. In *OOPSLA Workshop on Object Database Behavior, Benchmarks and Performance*, Austin, Texas, October 1995.

[19] Seth J. White. *Pointer Swizzling Techniques for Object-Oriented Database Systems*. PhD thesis, University of Wisconsin, Madison, Wisonsin, 1994.

[20] 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.

[21] Paul R. Wilson. Garbage Collection. *ACM Computing Surveys*. Expanded version of [22]. Available at `ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps`. In revision.

[22] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In Yves Bekkers and Jacques Cohen, editors, *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 1–42, St. Malo, France, September 1992. Springer-Verlag.

[23] Paul R. Wilson and Sheetal V. Kakkad. Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Addresses on Standard Hardware. In *International Workshop on Object Orientation in Operating Systems*, pages 364–377, Paris, France, September 1992. IEEE Press.

[24] Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis. The Case for Compressed Caching in Virtual Memory Systems. In *Proceedings of the USENIX Annual Technical Conference*, Monterey, California, April 1999. USENIX Association.