Texas: An Efficient, Portable Persistent Store *

Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas USA
oops@cs.utexas.edu

Abstract

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*, which exploits existing virtual memory features to implement large address spaces efficiently on stock hardware, with little or no change to existing compilers. Long pointers are used to implement an enormous address space, but are transparently converted to the hardware-supported pointer format when pages are loaded into virtual memory.

Runtime type descriptors and slightly modified heap allocation routines support pagewise pointer swizzling by allowing objects and their pointer fields to be identified within pages. If compiler support for runtime type identification is not available, a simple preprocessor can be used to generate type descriptors.

This address translation is largely independent of issues of data caching, sharing, and checkpointing; it employs operating systems' existing virtual memories for caching, and a simple and flexible log-structured storage manager to improve checkpointing performance.

Pagewise virtual memory protections are also used to detect writes for logging purposes, without requiring any changes to compiled code. This may degrade checkpointing performance for small transactions with poor locality of writes, but *page diffing* and *sub-page logging* promise to keep performance competitive with finer-grained checkpointing schemes.

Texas presents a simple programming interface; an application creates persistent object by simply allocating them on the persistent heap. In addition, the implementation is relatively small, and is easy to incorporate into existing applications. The log-structured storage module easily supports advanced extensions such as compressed storage, versioning, and adaptive reorganization.

1 Introduction

Texas is an object-oriented persistent storage system implemented as a C++ library. An application linked with the Texas library can create and manipulate two varieties of objects, *transient objects* and *persistent objects*. A transient object's lifetime is bounded by the duration of the program execution in which it was created. That is, when the program terminates, it destroys all transient objects created during that execution. Persistent objects, however, are automatically written to disk so that these objects may be accessed during subsequent program runs; in this respect, persistent objects have properties of file data, while preserving object-and-pointer semantics transparently [1, 2, 3].

In Texas, an object's persistence is orthogonal to its type, in a manner analogous to C or C++ "storage classes." A persistent object is simply one that is allocated in the persistent heap, as opposed to the conventional (transient) heap, or in the activation stack or static area.

For both efficiency and portability, Texas uses a standard C++ compiler; the compiler emits code in the usual way, without distinguishing between transient and persistent objects.

A running program cannot directly manipulate persistent objects on disk; instead, Texas transparently loads objects into virtual memory. Because persistent object storage is intended to replace entire user-level file systems for most purposes, a persistent object can contain references to any of a potentially huge number of other persistent objects. In the current implementation, persistent objects are stored in a single

*From **Proc. Fifth Int'l. Workshop on Persistent Object Systems, San Miniato, Italy, September 1992.** This and other papers on heap management, memory hierarchies, and persistence are **available via anonymous internet ftp** from cs.utexas.edu, in the directory pub/garbage.

persistent storage file.[1] References between them (*persistent pointers*) are represented as 64-bit file offsets. When persistent objects are loaded into virtual memory, these file offsets are translated into virtual memory addresses (*swizzled pointers*).

This *pointer swizzling at page fault* time efficiently supports 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 for millions of machines with terabytes of data apiece.[2]

1.1 Features

The Texas persistent store includes the following features:

- *Transparency* A program accesses transient and persistent objects in the same way, because all objects "seem" to reside in virtual memory. If client code doesn't need to distinguish between transient and persistent objects, it is not forced to.
- *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.
- *Scalability* Repeated touches to a page incur no extra overhead in address translation; the page fault-time costs should decrease as memory sizes increase and thus the number of instructions between faults increases.
- *Robustness* Texas uses logging techniques to provide checkpointing and crash recovery facilities.
- *Portability* Texas can be used with most C++ compilers and modern operating systems, and does not require any special system privileges or resources.
- *Compatibility* The implementation is compatible with existing code libraries. Recompilation is only necessary if these libraries create persistent objects. Moreover, only minimal source code modifications are necessary for a program to take full advantage of Texas' persistent storage and recovery facilities. (Texas' address translation scheme could also be used to reconcile data formats when sharing data between heterogeneous machines [4] and/or merging previously distinct address spaces.)
- *Pay-as-you-go costs* Texas' pointer swizzling costs are incurred only by programs that need them, rather than by all programs. The costs can be reduced to the vanishing point if the hardware-supported address space approaches (or exceeds) the amount of data in use; it provides efficient backward compatibility with narrow machines and extensibility to very large networks of machines with very large amounts of storage.
- *Modularity* Texas is composed of a set of largely orthogonal modules, with address translation, caching, and checkpointing handled in nearly disjoint code. This has made development easy, and allows experimentation and enhancement by reimplementing small parts of the system.

1.2 Overview

This paper describes the design of Texas, and how its modules implement the above features. Note, however, that several of the algorithms are straightforward, and could be easily replaced with similar algorithms which are better suited to specific applications. Although Texas is currently implemented for C++, it could be used with other languages by replacing the heap manager and type descriptor interface. (see Figure 1).

Several key ideas underlie the design of Texas. The system uses an efficient pagewise address translation technique to provide convenient access to persistent data and to implement huge address spaces.

[1] It would not be difficult to remove this restriction and support a system spread across multiple files and/or devices.
[2] Despite the fact that we actually live in a hilly area, the name "Texas" is intended to suggest a very large, flat space.

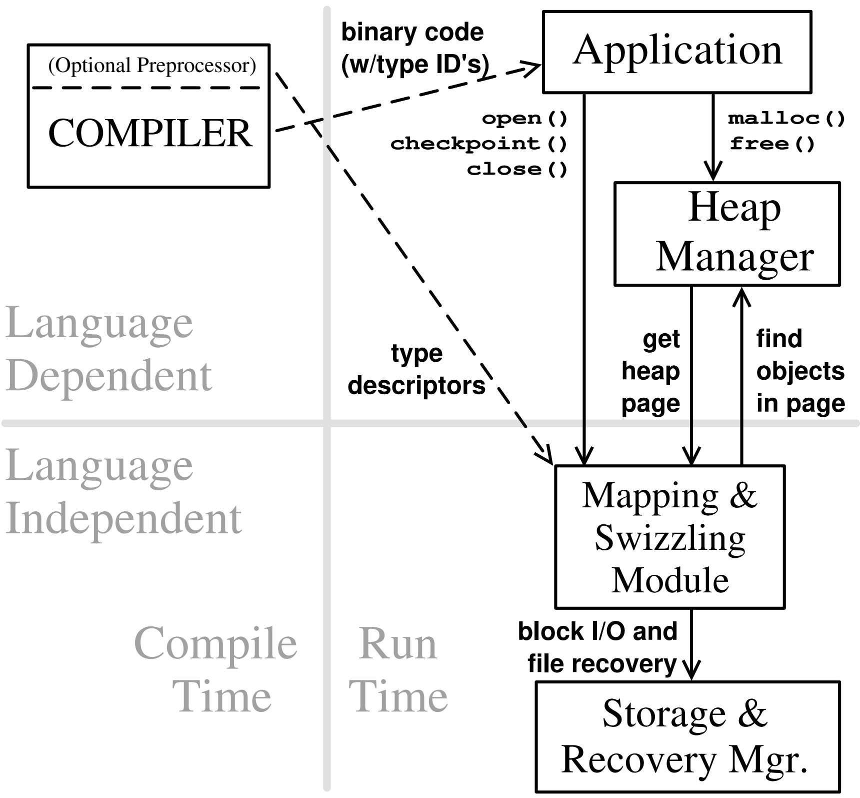

Figure 1: Modular Structure of Texas

Conventional virtual memory is used in the usual way to cache persistent pages—caching is therefore part of the underlying system, and needn't be implemented for Texas at all.[3] Logging and recovery are implemented separately. Only the pointer swizzling module depends on the format of data objects; caching, disk management, and recovery are done in terms of uninterpreted blocks of data.

1.3 Organization of the Paper

Section 2 discusses pointer swizzling at page fault time, followed by Section 3's discussion of heap management; heap management is implemented on top of the pointer-swizzled memory layer, but provides a hook so that the swizzling code can find pointers within heap objects. Section 4 explains how an existing virtual memory is straightforwardly exploited to provide caching. Section 5 discusses log-structured storage, including its support for pagewise checkpointing and recovery; Section 6 describes several extensions intended to reduce the volume of checkpointing writes. Sections 7 and 8 describe the current status of the system and our plans for future work, and Section 9 concludes the paper.

2 Pointer Swizzling at Page Fault Time

The modular structure of Texas is strongly influenced by the use of pointer swizzling at page fault time. This section describes the algorithm briefly, but see [4] for a more detailed treatment.

[3] It would be simple to implement our own caching module, given a facility such as Mach's external pagers. We intend to do just this, to experiment with different caching techniques, but normal virtual memory caching works just fine.

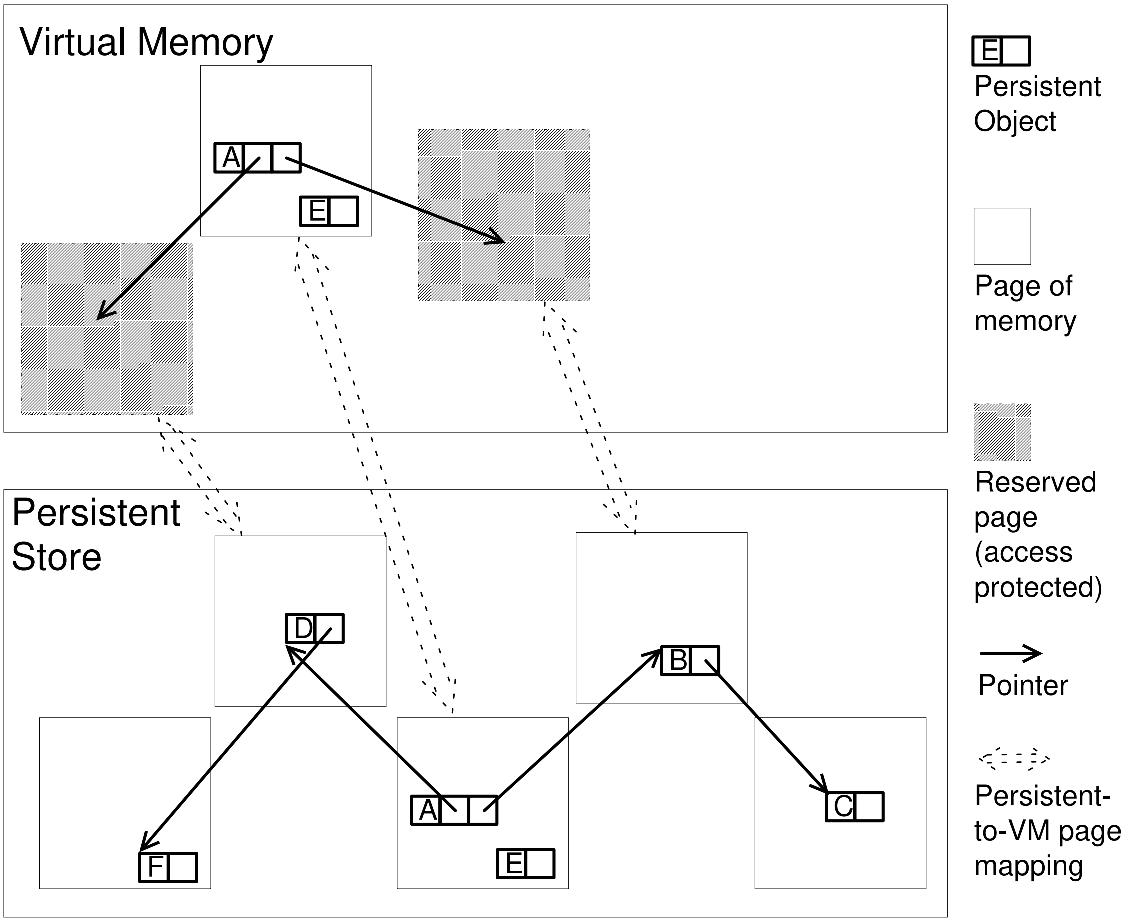

Figure 2: Persistent pages reserved in virtual memory

Texas uses conventional virtual memory access protections to ensure that the first touch to any page is intercepted. Faulting on a page causes it to be loaded and scanned for pointers, which are then translated into actual addresses. This may require other pages of virtual memory to be reserved for referred-to persistent pages, so that the addresses can be resolved.

2.1 Algorithm Description

When a program gains access to a persistent store, it can request pointers to one or a few special rooted objects, which can be retrieved by name. When a rooted object is requested by name, Texas reserves and access protects a page of virtual memory, which acts as a placeholder for the persistent page containing the object. It then returns an actual virtual memory pointer into that page, i.e., a pointer to the rooted object's position in the page. The persistent page containing the rooted object is not actually loaded into memory until it is actually referenced.

Figure 2 shows the state of the system after traversing a root pointer (to object A) and faulting one page into memory. The page containing object A has already been faulted into virtual memory, had its pointers swizzled, and been made accessible to the program. That page holds pointers into two reserved pages; those pages are reserved for the persistent pages containing objects D and B, respectively, as shown by the dashed double arrows.

If the application program traverses the pointer from object A to object B, it incurs an access protection

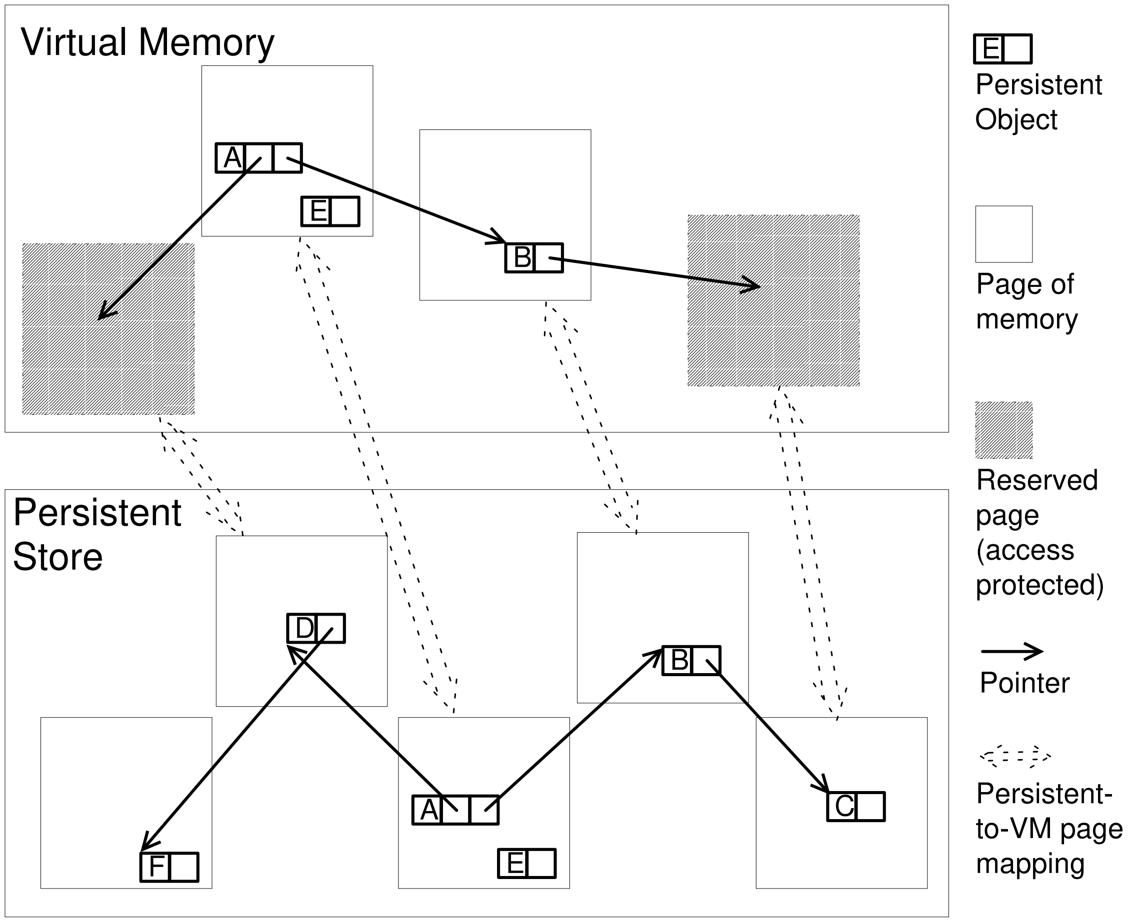

Figure 3: After dereferencing a pointer into a reserved page

exception, and the faulting-and-reserving process repeats. The protection fault handler loads the faulted-on page with the corresponding persistent page, and translates all of the persistent pointers it holds into swizzled pointers.

Even if the object referenced by a persistent pointer does not yet reside in virtual memory, the handler must still provide the eventual address for object. To do this, it reserves (and access protects) another place in virtual memory for the persistent page holding the pointed-to object, C, as shown in Figure 3.

(The mapping between the persistent page and the virtual page is established by recording it in a table. It is then trivial to translate the pointers into the page; it is only necessary to replace the page number bits of the persistent pointer with the (shorter) page number of the reserved virtual memory page.)

Pages of virtual memory are thus reserved "one pointer ahead" of the actual page referencing behavior of the program, in a a pagewise wavefront, just ahead the actual access patterns of the running program. Any page directly reachable from a page that is touched must have space reserved for it in virtual memory. When a reserved page is touched, the wavefront is extended past it, to the pages it holds pointers into.

This wavefront preserves a crucial constraint: the running program is never allowed to see a page containing persistent pointers. It only sees pages containing pointers represented as normal hardware-supported pointers—i.e., actual addresses of objects at particular locations in virtual memory. Pointers to previously-touched pages can be dereferenced in a single load or store machine instruction, just as in a normal virtual memory. This is quite different from conventional pointer swizzling schemes, which require frequent checks to see whether persistent pointers have been converted to actual addresses [5, 6]. The space cost for mapping tables is also small, because pagewise translation requires only an entry per page, rather than an entry per object.

(This incremental process of reserving pages is a variation on Appel, Ellis, and Li's incremental copying garbage collection scheme, which is itself a variant on Baker's incremental copying scheme [7, 8, 9, 4]. The Appel-Ellis-Li system uses page protections to trigger scanning of pages and relocation of referred-to objects, while our system uses them to trigger scanning of pages and relocation of referred-to *pages*.)

Our implementation of pointer swizzling requires no special hardware and no special operating system features to perform well. In modern versions of UNIX, virtual memory pages can be access-protected and protection fault handlers can be set using regular system calls.[4]

We believe page-fault-time swizzling 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 do not have this property, because their checking overheads are directly tied to the rate of program execution.

Another convenient property of our system is that storage and data transfer requirements are essentially the same as those of a conventional virtual memory. Pages of data are transferred on demand, and reserved pages do not require any physical storage (neither RAM nor disk) until they are actually used.[5]

2.2 Address Space Reuse

Despite the fact that reserved pages don't require storage, exhausting the virtual memory address space is still a potential problem. If too many pages are reserved, eventually there will be no more uncommitted pages of address space, and pointers to new persistent pages cannot be translated.

Currently, we do not address this issue explicitly; we assume that programs will be broken into individual transactions that do not exhaust the address space. A program that touches and/or reserves gigabytes of pages may have to flush its virtual memory cache back to the persistent store between transactions, and start the faulting-and-reserving wavefront over again for the next transaction.

We intend to address this issue by implementing an incremental *address space reuse* algorithm, which periodically rebuilds the wavefront in a less expensive manner [4]. Rather than actually evicting all pages, only to fault the working set back in immediately, the actual evictions are deferred so that the working set can be kept in transient memory during the rebuilding of the mappings. Incremental address space reuse exploits the fact that caching and address translation are essentially orthogonal—protecting pages and rebuilding address mappings by faulting needn't cause the actual data in the pages to be moved back and forth.

For the present, however, this address space reuse algorithm does not appear to be necessary; we expect that few applications will require it. Even for persistent stores containing many gigabytes of data, few programs are likely to touch (or reserve pages) more than a gigabyte during a run.

2.3 Sharing and Compatibility

In the long run, as computers become more powerful and ever-larger storage is required, we expect the reuse algorithm to deal satisfactorily with most applications. Its performance is dependent on various kinds of locality, however, and it is conceivable that programs with poor locality on a huge scale could suffer significant overhead from rebuilding the wavefront. We believe that such programs will be very unusual for many years, and are likely to be run on very powerful 64-bit machines in any event.

Our pointer swizzling scheme can reconcile 64- and 32-bit addressing, so that most data can be shared

[4] We use `mprotect()` to access-protect pages and `signal()` to set protection fault handlers. We do not currently use the `mmap()` system call because its behavior varies significantly between different flavors of UNIX; similarly, we avoided `mlock()` because it requires super-user privileges.

[5] The current configuration does actually allocate swap disk for pages of virtual memory, but future versions will avoid this by using of `mmap()`.

between machines with different word sizes.[6] Only programs which actually exercise a 64-bit address space during a run would need to be run on 64-bit hardware. Eventually, it may also be desirable to link many 32-bit and 64-bit machines into a network with a conceptually flat 128-bit address space implemented in software; implementing huge address spaces in software is easy using pointer swizzling at page fault time [4].

In [10], it was claimed that pointer swizzling schemes (including ours) have several drawbacks; we believe these claims to be based on misconceptions, or to be much less troublesome than they seem at first glance. In particular it was claimed that sharing is inhibited by pointer swizzling, and that swizzling costs are unavoidable.

Sharing within a node is not in fact inhibited; it is only necessary for different processes (or protection domains) on the same node to share the same set of persistent page mappings, rather than each process keeping its own set. (The basic protection domain scheme advocated in [10], is entirely compatible with such sharing.)

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.

Equally important, the costs of pointer swizzling could be optimized away in those cases where they are 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 "pre-swizzled" format, with no translation costs whatsoever.

One advantage of pointer swizzling in such a network is that it would provide backward compatibility with smaller machines, as well as extensibility for network growth and future machines which might eventually exhaust the 64-bit address space [4]. Small machines would incur costs in swizzling pointers from the globally shared format to their local, narrow format, but this would not affect the performance of the larger-address machines at all. Similarly, the ability to change mappings and swizzle pointers needn't incur any cost until the address space is exhausted remapping actually begins—that is, if you never need it, you never pay for it.

Pointer swizzling at page fault time is thus effectively a "no-cost option," even if one assumes that 64-bit machines will rapidly dominate the market and displace the installed base of 32-bit machines.

In addition, pointer swizzling at page fault time has a large advantage in that it can serve as a reconciliation layer to resolve conflicts between different address spaces. Even in a world of 64-bit hardware, this is highly desirable. For example, consider the case of merging two local-area networks, each with its own flat shared address space (*a la* [10]). 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 subnets) to map the same data to different local virtual addresses. It therefore requires no clairvoyance on the part of system administrators to ensure that no conflicts arise between systems that might eventually be merged, e.g., when an organization is restructured or one company acquires another.

The only remaining concern from [10] is the complexity added by having the memory system rely on pointer-finding within heap data. We believe this to be a very small cost; as the next section will show, our interface to GNU C++ does not require modifying the compiler at all. True "higher-level" languages (e.g., Lisp, Smalltalk, Eiffel, Modula-3) would be even easier to interface with the memory system.

2.4 Type Descriptors

In order to perform pointer swizzling, Texas must be able to accurately locate the pointers within a faulted-on page of data. Although *conservative* pointer identification techniques are usually sufficient for non-copying garbage collection [11], they are unsuitable for pointer swizzling. Pointer swizzling must *accurately* translate between persistent and swizzled pointers; the misidentification of a pointer could result

[6] It is only necessary to use 64-bit pointer fields uniformly. On 32-bit machines, the pointers can be swizzled to 32-bit addresses that actually occupy only half the field. It is even easy to design compatible instruction sets so that binary code can exploit whichever hardware address size is available [4].

in the erroneous modification of a nonpointer value. (As we describe in [4], conservative techniques *can* deal with pointers from the stack, which is convenient when implementing incremental address space reuse.)

To support identification of objects within a page, we use a slightly modified version of the standard GNU `malloc()` (described more fully in Section 3).

To accurately identify pointers within a page, Texas first finds the objects within the page, and then uses *runtime type identification* information in the objects' header fields to identify the pointer fields. Unfortunately, there is no standard runtime type ID feature in C++ yet. In the meantime, we enhance C++ by defining a *type descriptor* object for every class declared by the program. When a persistent object is created, the memory allocation routine (`malloc()`) augments the object with a type descriptor pointer. This pointer references the type descriptor object that describes the layout of objects of that class.

Type descriptors can be generated automatically by a compiler or by a separate preprocessor. The Free Software Foundation's GNU C++ 2.2 compiler provides a command-line switch to optionally create type-descriptor objects [12]. (Other type descriptor systems have also been proposed and/or implemented, and it is likely that one will eventually become standard for C++. For our purposes, any of them is acceptable.) We are also writing a preprocessor that scans a program's source code for all class declarations and generates corresponding type descriptor definitions; this preprocessor will make Texas independent of any particular C++ compiler.

2.5 Related Work

A commercial object oriented database management system (ObjectStore, from Object Design, Inc.) uses pointer swizzling at page fault time to provide uniform access to transient and persistent objects [13]. Fundamentally, ObjectStore and Texas use similar techniques to implement persistence (developed independently), but few technical details are available about ObjectStore's implementation. The systems probably differ in a variety of ways, not least in their approaches to generating type descriptor information. ObjectStore is closely tied to a proprietary version of AT&T's `cfront` compiler; Texas is compatible with most off-the-shelf C++ compilers through the use of a preprocessor to generate type descriptor information.

Recently, we learned[7] that the basic idea of pointer swizzling at page fault time was apparently discovered and abandoned in the late 1970's, during the development of persistent Algol. A system running under VMS used pointer swizzling at page fault time, but performance was poor, and the technology was never fully developed; the initial disappointing performance was due in part to comparatively high trap overheads and page fault frequencies, given the small memories and slow processors of that time. We also believe that the basic technique may have been devised more or less independently in several places over the last decade. The Moby address space system for LMI Lisp Machines [14] used pagewise relocation, but pointers were swizzled on discovery (one at a time) using the hardware's support for tagging. (A similar Scheme was later developed for the TI Explorer, and Moby may also have inspired Object Design's system.) A stock-hardware version of a capability-based system [15] developed at Monash University may also use similar techniques, but we have been unable to obtain details as of this writing.[8] A form of pointer swizzling at page fault time was also used in a recent experimental version of the Comandos [16] operating system[9], though apparently it was never fully developed and nothing was published about it.[10] (We welcome correspondence on earlier implementations of pointer swizzling, especially those using pagewise swizzling and/or mapping.)

In any event, simple pointer swizzling has been used in many places; we intend to show in this paper that page fault-time swizzling with pagewise relocation is especially attractive given current hardware and software trends, and that it can be implemented with surprisingly little change to other aspects of a system.

[7] Malcolm Atkinson, personal communication, 1992.

[8] From [15], it would appear that this system swizzles object identifiers, rather than persistent addresses, requiring per-object translation tables. The version documented in [15] (for specialized hardware) swizzled pointers *on discovery* [6], rather than at page fault time, as our system does.

[9] Pedro Sousa, personal communication, 1992.

[10] Comandos also uses object identifiers rather than persistent addresses, and requires per-object mappings.

3 Heap Management

Texas lets programs access multiple persistent stores, each with its own heap; programs may also create transient objects on a normal transient heap. A naive memory manager would create separate heap areas for persistent and transient objects. A persistent heap would start at an arbitrary address, and each heap would grow and shrink independently. This design requires an *ad hoc* static partitioning of a process' virtual address space. Our memory manager avoids statically partitioning the address space, and thus 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.

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 are themselves stored as persistent data structures, so that free space within partially-filled persistent pages can be reallocated during subsequent program runs.

The free list for each heap is actually structured as a vector of separate free lists for different *size classes*. A size class is simply a small range of object sizes; objects are allocated in free chunks of memory large enough to hold the actual size of object, but possibly with wasted space if there's not an exact fit.

When an object must be allocated and the free list for the appropriate size class is empty, a new page is allocated to that heap, and it is immediately divided into a page full of uniform-sized chunks, which are linked onto the free list.

Like the standard GNU `malloc()`, our heap allocator expends little effort attempting to coalesce free blocks into larger blocks. (In the case of the GNU malloc, this is because it favors speed over space costs: it is often unnecessary to coalesce blocks, which often end up being split again anyway, wasting time both ways.)

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. 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, and cross page boundaries. Rather than recording the actual size class for such pages, the page is flagged as being part of a large object. Large objects' boundaries are stored in a special table, so that their starting addresses (and type descriptors) can be found.

Using a different heap management algorithm (such as first-fit or best-fit) would complicate the process of finding object's headers and pointer fields, but we believe it could be done with little additional overhead—the necessary techniques are already well-developed for use in garbage collection.

4 Caching

Pointer swizzling at page fault time implements address translation on top of the 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 to backing store and back in again as necessary. Pages containing swizzled pointers may be paged in and out, independently of the transfer of pages between virtual memory and the persistent store.

(There is no explicit cache module in Figure 1, because virtual memory does 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 is not concerned with whether pages are memory resident or not.)

This approach is appropriate for many applications such as typical CAD databases, or simply replacing conventional files in conventional 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 storage file and the local backing disk's swap area. It would also reduce the possibility of dirty pages being paged out to swap disk, only to be subsequently paged back in so that they can be written to the persistent store.[11] 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 pageouts. This could be achieved using something like Mach's external pager facility, but it would make the system somewhat more complex and less portable.

When using the virtual memory system for caching, it is important to avoid caching the persistent storage file in the filesystem cache; once a page has been loaded from the persistent storage file into the virtual memory, it is implicitly cached. Caching it in a file system buffer is a waste of resources, so the persistent storage file should be stored in an uncached area of disk.

5 Disk Storage Management and Recovery

Like any useful persistent store, Texas supports checkpointing and recovery; with a persistent store, the conventional heap/file distinction is lost, and explicit checkpointing must take the place of "saving changes to a file." We have implemented two different checkpointing schemes for Texas. The first was a conventional *write-ahead logging* scheme; that is, changed pages were written "off to the side" in a log. (The persistent storage file itself was only modified after all changes were safely written to the log, so that a crash would not leave the store in an inconsistent state.[12]) We have recently replaced this with a more flexible *log-structured storage module.*

5.1 Log-structured Storage System (LSS)

Rather than refining our crude write-ahead logging scheme, we chose to replace both the log and the persistent storage file with a log-structured storage system which supports checkpointing directly and efficiently. The LSS is essentially the lower levels of a *log structured file system* [17]; and manipulates a single large uncached file (currently, a raw UNIX disk partition).

In a log-structured store, the entire disk is used as a log, and the log itself acts as the final repository of data pages. Rather than "updating" blocks that reside in fixed locations on disk, logical blocks can migrate, with no "home" location. The "current" version of a block is simply the last one written to the log, and changes to a file can be committed by updating indexing information to point to the new versions of changed blocks.

This "write anywhere" property has many advantages for implementing advanced functionality. It has been promoted as a means of increasing effective write bandwidth (by allowing writes to proceed nearly continuously to a contiguous area of disk) [17], but we believe a well-implemented conventional logging scheme can perform similarly.[13]

For our purposes, the real advantages of "write anywhere" disk management lie elsewhere. While we are interested in high write bandwidth, we are equally interested in improving read latency and bandwidth, especially for programs that manipulate very large amounts of data and whose performance may not be dominated by checkpointing writes. As we will discuss in more detail in Section 8, block mobility makes

[11] 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 5.

[12] Once all the writes are on stable storage, they can be safely propagated to the persistent store; a crash during this propagation phase only requires repeating the phase until it succeeds; repeated writes due to retries will have no lasting effect.

[13] Where a conventional logging scheme must reclaim log space by writing changes to the "home" locations of blocks, a log-structured file system must reclaim log space by some form of garbage collection. The efficiency of either scheme depends (at least in part) on the lifetime distribution of block versions. If too many new but long-lived versions of blocks are created, they must be migrated to free log space (in write-ahead logging) or to compact the log (in log-structured storage), and performance may suffer.

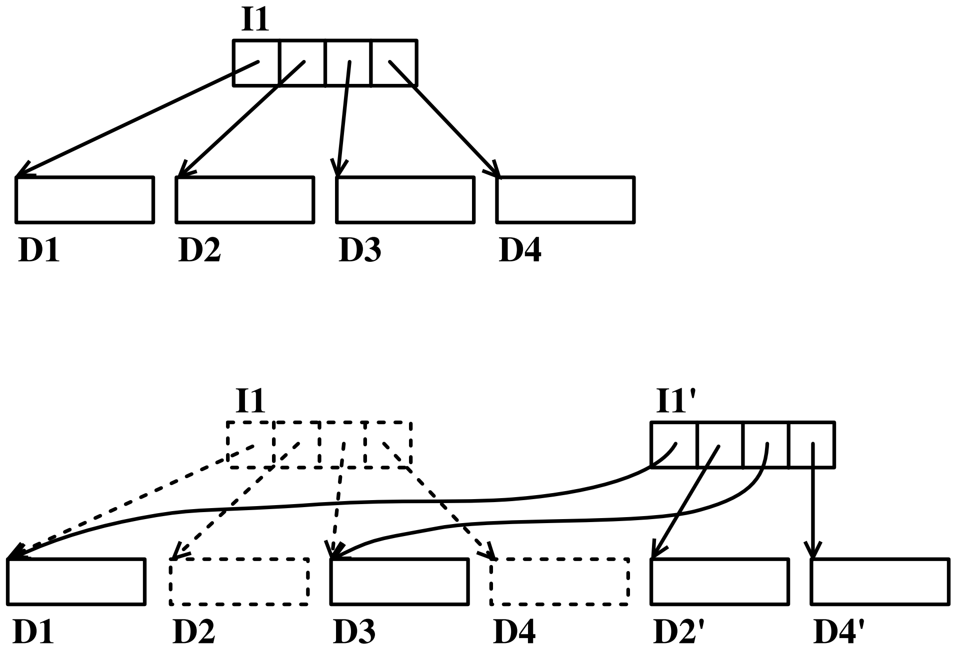

Figure 4: A tree-indexed file

it easy to implement several attractive advanced features: sub-page logging, adaptive disk reorganization, compressed storage, and multi-versioned storage.

5.2 LSS File Structures

In the LSS, each file is stored as a tree-indexed sequence of data blocks. That is, a file is actually structured as a multiway search tree, somewhat like a B-tree, with the data blocks as the leaves. This structure is shown in the top part of Figure 4. When data blocks are changed, new versions are written. This requires updating their parent (indexing) blocks.

Unfortunately, updating parent blocks in the conventional way would require seeks to write *their* new data, which is precisely what log-structured storage is intended to avoid. Rather than seeking and over-writing parent blocks with updated pointers to moved children, the parent blocks are treated in exactly the same way as the children—new versions are written to the disk as part of the log-writing stream. This problem recurs, because any pointers to the parent node must be written out as well; so each ancestor (all the way up to the file's top-level indexing node) must be written. As shown in the lower part of Figures 4, when data blocks D2′ and D4′ are written, indexing block I1′ is also written. The old versions, (D2, D4, and I1) simply become garbage. Once the new version of the top-level indexing node has been written, the changes to a file (or several files) can be committed atomically by writing a special commit record. The commit record simply signifies that the new top-level indexing node(s) should be used, rather than the old one(s).

This is essentially an application of functional (side-effect free) programming technique to the representation of tree structures on disk. When leaf nodes (data blocks) are replaced with new versions, a new copy of the file's whole indexing tree is made. The copy is optimized by using shared structure—any nodes that aren't changed are simply shared, instead of copied.

Because this copying is nondestructive, it can easily be extended to keep multiple versions of any file or set of files. It is only necessary to retain multiple top-level indexing nodes, and to avoid reclaiming any

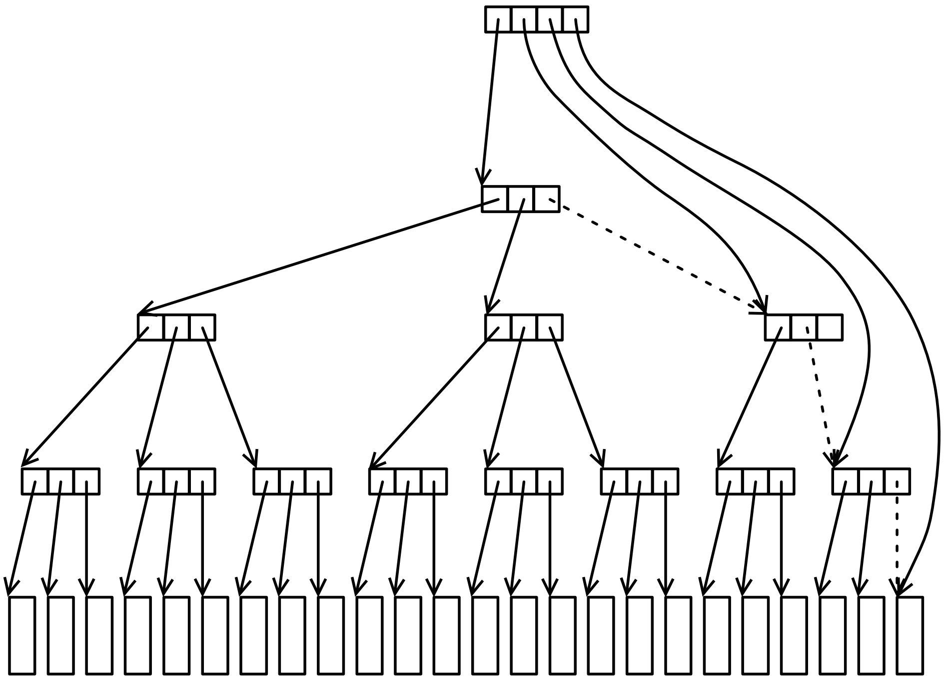

Figure 5: Right-shallow file indexing trees

physical block version that is part of any valid file version.[14]

In our LSS, the trees are *right shallow*, meaning that incomplete rightmost subtrees of the whole tree hang directly off of the top-level indexing node (see Figure 5). When a subtree fills because of file "appends," the full subtree is pushed down and to the left to become a rightmost subtree of a larger subtree.[15]

The advantage of this structure is that trees are very bushy, because only the rightmost subtrees are ever incomplete, and the tree as a whole is never more than one node deeper than a perfectly symmetrical tree. This minimizes the number of ancestor nodes that must be written when a data block is written, and appends are especially cheap. For small- to medium-sized files, only one or two ancestors must be written for any write to an arbitrarily chosen block. Most append operations only require modifying the top-level indexing node, because the rightmost several data blocks hang directly off of it.[16] This structure is therefore equally suitable for random search and update patterns, and also for serial file construction, e.g., writing event logs.

5.3 Log Reclamation

Log reclamation in a log-structured storage system [17] is essentially a matter of garbage collection [18, 9]. Committing a new version of a file disconnects the top-level indexing node from the current state of the file system; it is therefore garbage (subject to reclamation). Any lower-level indexing and data blocks that are not reachable from a valid top-level node are likewise garbage and may be reclaimed.

Our log reclamation scheme is quite straightforward, relying on reference counting of data and indexing blocks. A special file (itself stored in the LSS) holds a bitmap of free blocks. The normal checkpointing of

[14] This requires only a slight modification to the reference-counting scheme used to keep track of free disk blocks.

[15] This is quite different from conventional UNIX file structures, which are left-shallow and do not require rebalancing as they grow larger.

[16] Occasionally, subtrees must be pushed down, but this rebalancing is infrequent and the more expensive rebalancings are extremely infrequent.

the LSS keeps this bitmap consistent with the checkpointing of data files.

Currently, we use a very simple bitmap scanning technique to find mostly-free areas of disk for writing new versions of blocks. This is rather different from the copying compaction employed by the Sprite LFS [17]. Our technique is more akin to the sweep phase of mark-sweep collection, and we believe it may be more appropriate than copying compaction under most circumstances. (We currently have no compactor at all, though we intend to implement one for situations in which fragmentation costs are high.)

It is interesting to note that the *garbage detection* problem is trivially solved using reference counting, and this gives tremendous flexibility in the choice of garbage reclamation and reallocation strategies. Rosenblum and Ousterhout use a technique that is analogous to generational copying collection [17]—that is, they take advantage of the fact that most block versions live only a short while, because they are superseded by subsequent writes of the same logical block. They use a copying compactor to reclaim the space for obsolete versions of logical blocks. While the generational heuristic is likely to be effective in most circumstances, it is unnecessarily indirect and ill-informed.

The information provided by the reference counting garbage detector can easily inform more *opportunistic* strategies, which take advantage of complete information about which blocks are free.[17] This opportunism can avoid expending effort where it is unnecessary, and might be used to dynamically choose between mark-sweep style reuse and copying compaction.

Mark-sweep is advantageous when fragmentation is naturally low, as we expect it to be in most cases, due to locality effects. Rosenblum and Ousterhout have reported [17] that many moderate-sized areas of disk[18] contain no live block versions at all by the time their collector "cleans" (copy collects) them. In those cases, no special effort should be required to find long runs of contiguous or nearly-contiguous disk space. Our bitmap sweeping technique should work well in such situations.

Copying compaction has the advantage of eliminating fragmentation, but incurs additional, less predictable costs in updating younger blocks holding pointers to relocated blocks. In the worst case, it appears to require cleaning *all* segments younger than the oldest segment being cleaned, because creating new versions of indexing blocks may itself cause fragmentation.[19]

6 Sub-page Logging to Reduce Checkpointing Writes

Pagewise pointer swizzling is attractive because it can exploit spatial locality, but pagewise operation also has disadvantages, particularly in checkpointing short transactions. If transactions have poor locality of writes, pagewise checkpointing will save too much unchanged data; a single write to a page during a transaction will cause the entire page to be written to disk.

For long-running transactions which manipulate considerable quantities of data, this cost is probably not unacceptable; simply reading large amounts of data requires disk seeks which are usually more expensive than the log writes. For small transactions, however, the working set of the program is likely to reside in memory, and the log writes may well be the limiting performance factor. In this case, the actual limitation is likely to be disk *bandwidth* (rather than latency), since log writes can be written as a continuous stream with few seeks.

White and DeWitt have compared ObjectStore (a pagewise scheme similar to ours) and several versions of the E system, using finer-grained (objectwise) swizzling and checkpointing [6]. Their results appear to bear out the thesis that pagewise schemes work best for long transactions—because they can typically exploit spatial locality—but exhibit poorer performance when transactions are short and locality of writes is poor.

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. To do this, we are

[17] This is analogous to opportunistic garbage collection for general-purpose systems, as in [19] and especially some proposals by Barry Hayes [20].

[18] I.e., "segments"—the units of contiguous allocation in their system.

[19] This corresponds to the typical approach employed in generational garbage collectors, where some generation is chosen for garbage collection, and all younger generations are also collected during that collection [21, 9]. This is unnecessary with a non-moving collector where liveness is known beforehand.

implementing *sub-page logging*, so that we can checkpoint areas of memory that are smaller than pages. Rather than writing out entire dirty pages, we can often write out only those parts of a page that have actually changed. In order to do this, we need the ability to detect which sub-page units have changed and which have not.

We are investigating two possible approaches to this fine-grained checkpointing. One is to keep dirty bits for small areas, either fine-grained hardware dirty bits (as in the ARM 600 [22]), or *card marking* techniques, which maintain dirty bits in software [19, 23]. Another is to use *page diffing* (i.e., word-by-word comparison with a clean copy) to find out which parts of dirty pages have actually changed.

All of these approaches are independent of our pagewise pointer-swizzling scheme, and exploit the orthogonality of address translation and data storage; checkpointing is carried out at a different resolution than address translation.

Currently, we intend to use page diffing, not fine-grained hardware dirty bits or (software) card marking, because it requires no change to the hardware, operating system, or compiler, and we believe it will perform quite well, given current ratios between CPU and disk speeds.

6.1 Page Diffing

Page diffing is also conceptually straightforward—it is only necessary to keep a clean version of each modified page, and compare the modified and unmodified pages at checkpoint time. Only the changed parts need to be written to the log.

We are currently implementing page diffing using write-protection traps to trigger a copy upon the first write to each page; the write protection trap handler copies the page into a *clean version buffer*, unprotects the original, and lets the program resume. This scheme incurs costs proportional to the number of dirtied pages per transaction—each written page causes a trap, a page copy, and a page diffing operation. In addition, there is a space cost of one page of clean version per page written.

It may seem that these costs are quite high—an isolated write to a 4 KB page, for example, will cost several thousand instructions at checkpoint time. We believe this cost is quite acceptable for the majority of applications. The main cost of checkpointing small transactions is not the cost of executing instructions in memory, but the cost of disk operations to commit data to nonvolatile storage. It is attractive to expend thousands of instructions to avoid writing an entire page of data.

Consider a typical high-performance workstation disk with a bandwidth of roughly 4 MB/sec for large writes, e.g., streaming data to a contiguous area with few seeks. Four megabytes per second is 4 KB per millisecond—or roughly a page per millisecond.

On the other hand, consider CPU operations. Common workstation and PC speeds are in the tens of MIPS, soon to be in the hundreds. A 100 MIPS processor can execute 100,000 instructions per millisecond. It will therefore be worthwhile to spend roughly 100,000 instructions to save one page's worth of data writes.

(Because page copying and diffing are memory-intensive, their speed may not scale directly with processor speeds. They have excellent sequential locality, however; they are likely to exploit increases in memory bandwidth quite well, rather than being closely tied to memory latency. It should also be noted that the diffing of pages can be done in a pipelined fashion, in parallel with the writes. If transaction commits are the limiting performance factor, most page diffing can be done in the idle time while other pages are being written. The actual page diffing is therefore likely to be effectively free, though the trapping and copying is not.)

Current operating system implementations incur high trap overheads, usually around 10,000 or 20,000 instruction cycles per trap. Even so, this is only a fraction of the instructions necessary to negate the advantage of page diffing. It appears that page diffing will eliminate most of the cost of worst-case write locality. Improved trap handling performance would naturally increase the attractiveness of this scheme. (We believe that trap handling overheads are amenable to significant reductions, and that operating system implementors are likely to achieve those reductions as more applications exploit virtual memory facilities; see [24, 25, 26] for more detailed discussions of these issues.)

A caveat is in order here: the trade-offs involved are very sensitive to changes in CPU and storage technologies. If, for example, nonvolatile RAM is used for checkpointing, checkpointing writes may become much cheaper, and applications may exploit that by checkpointing more frequently. In that case, the time spent diffing pages may go up significantly, and fine-grained dirty bit schemes will become correspondingly more attractive.[20]

6.2 Limiting the Space Costs of Diffing

Besides the time cost of page diffing, there is a space cost, namely the cost of keeping clean versions of modified pages. For short transactions, where checkpointing writes are the limiting factor, this is not generally a problem—the problem is the time required to commit a moderate number of dirty pages to disk, not the storage required for a large number of dirty pages.

For longer transactions, on the other hand, the sheer volume of dirty pages may be a significant space cost. We have chosen to limit this cost by keeping a relatively small buffer of versions of dirty pages, and writing out pages early if the buffer fills. The buffer is managed in roughly least-recently-changed fashion, so that the most stable versions are written first. When a changed page version is written from the buffer, the corresponding page is access protected so that subsequent changes can be detected, and a new copy made.

We believe that this strategy will be good for the performance of both long and short transactions. Writing out changes prematurely may increase disk bandwidth requirements for long transactions, but is likely to increase overall performance by decreasing commit latency—blocks that are written prematurely (and which do *not* change again before the transaction commits) need not be written out commit time. Given some locality of writes, this should increase total traffic somewhat, but make it less bursty.

7 Current Status

Currently, Texas' log-structured storage scheme is still under development, but a version of Texas is up and running using a simple write-ahead logging scheme.

As reported in [4], the swizzling scheme performs as expected: for programs with good locality, the cost is essentially zero, while for programs with poor locality, paging costs dominate address translation costs.

Our type descriptor system requires some hand-coding. This is due to bugs in the GNU C++ compiler, rather than any problem with Texas itself; we hope and expect that these bugs will be fixed soon. We are also developing our preprocessor to make the system compiler-independent. The preprocessor currently parses C++ and recognizes class definitions, but the actual generation of runtime descriptors is not complete.

Checkpointing performance, even without the log-structured file system and sub-page logging, is acceptable. Preliminary benchmarks for short transactions, using code from White and DeWitt's variant of the OO1 benchmark, show performance to be comparable to the systems they measured, on comparable hardware. Exact comparisons are not possible because of differences in hardware configurations.[21]

With a very preliminary version of the log-structured store, performance appears to be significantly better, and in fact our system superficially outperforms most of the systems reported in [6].

These results are hard to interpret, however, both because of the inexact match in hardware used, and because our system is a single-workstation system, while the systems tested in [6] were two-workstation client-server configurations. In addition, our storage system is incomplete (especially in terms of its log-reclamation and compaction strategies), so its eventual performance may be somewhat different. On the other hand, we expect our page diffing techniques to reduce costs further, so we are quite pleased with our preliminary results.

In a test corresponding to White and DeWitt's Figure 7, our system outperformed all of their E variants and ObjectStore (version 1.2) by at least a small margin. In a test corresponding to their Figure 8, with finer-grained transactions, Texas consistently outperformed two E variants by a small margin, while two

[20] We believe that fine-grained hardware dirty bits are a good idea in general [26], but do not want to depend on them at present for reasons of simplicity and portability.

[21] Like White and Dewitt, we are using SPARCstation ELC's, but with different disk drives; we don't believe the difference in drives would make a major difference to performance, but we can't be certain.

other E variants consistently outperformed our system, again by a small margin; all of these systems outperformed ObjectStore. Again, we must stress that these are very preliminary results, and the systems are not exactly comparable. (In particular, ObjectStore is a commercial product with many advanced features, rather than a stripped-down minimal system like ours.) Qualitatively, though, our system appears to perform extremely well in terms of address translation costs [4] and to be competitive with other systems in terms of checkpointing.

8 Future Work

In the near term, our main goals are to complete the type descriptor generator and log-structured storage mechanisms so that we can test and benchmark Texas more fully. After a period of debugging and tuning, we will then issue a beta release, probably early in 1993.

After Texas is stable, we intend to release it generally as free software, most likely in the public domain. Soon thereafter, we intend to create a client-server version of Texas, for distributed applications. (We may also try a distributed virtual memory approach.) We are exploring several issues related to distributed systems, including having independently sharable and protectable heap areas within a single store. We are currently leaning toward a scheme that is similar to the protection domain scheme of the Opal operating system [10].[22]

A major goal of this project is to produce a useful, high-performance persistent store and give it away, so that we can develop a significant user base of actual programs that will fully exploit the potential of our pointer swizzling and storage schemes. Our decision to support C++ was largely motivated by the desire to have a significant number of applications that could be easily ported to our system, so that we can properly evaluate it.

Once we have a user base, we also intend to use Texas as a vehicle for basic research in locality of reference in large sets of data. Conventional filesystem traces are difficult to interpret, because the references to file blocks are a very complex function of program's conceptual referencing behavior; domain-level objects may move from file block to file block, perhaps being in multiple places at once, and unrelated objects may map to the same logical or physical file block over time. Because a persistent store preserves object identity, reference behavior should be somewhat easier to interpret and exploit.

After gathering reference traces and running them through simulators we have developed [27], we intend to experiment with actual implementations of novel replacement policies, and prefetching policies exploiting adaptive reorganization. Adaptive replacement will require using an external pager mechanism (or something similar) to override the default caching of the underlying virtual memory. Adaptive reorganization for prefetching [28, 26] can straightforwardly be added to the log-structured store, due to its write-anywhere policies; this will also require an external pager facility to exploit it for the local caching mechanism.

Log-structured storage also lends itself to *compressed* storage of data blocks. Given that blocks can move around on the disk anyway, it is a small step to let them vary in size, depending on how well they happen to compress.[23]

Previous work in compressed file storage has focused primarily increasing effective disk capacity for typical file data. They therefore focus on byte string-oriented compression techniques such as bytewise Lempel-Ziv variants [29, 30].

Our goals are quite different. We are focusing on decreasing effective memory *latency* for typical *heap* data. Given that we page objects in a heap format, and also store them in an only slightly different format in the persistent store, we are much more interested in heap data layouts than conventional file I/O formats. We would like to add a level of *compressed in-memory storage* to the memory hierarchy, intermediate in

[22] Our subheaps would take the place of their low-level notion of contiguous "segments," and subheaps might act as sets of objects with distinct policies controlling migration, replication, coherence, and garbage collection.

[23] This is particularly easy if copying compaction is used to reclaim log space, because contiguous allocation regions make it easy to allocate variable-sized blocks contiguously. It's not much harder for a non-moving log reuse policy, however, because data blocks don't need to be stored entirely contiguously; this greatly simplifies packing problems.

performance between normal (non-compressed) RAM and disk [26].[24]

We are developing adaptive compression techniques that exploit the typical low information content and word-wise alignment of heap data fields, and the strong alignment and grouping regularities imposed by typical heap allocation schemes. In our initial tests, we have gotten compression factors for C++ heap data that vary between 30 and 90 percent with fast single-pass algorithms, which can compress a 4 KB page in a very few milliseconds on a relatively slow (Sun SPARC ELC) processor. We are optimistic that for typical programs we will achieve compression factors of two to four, with compression taking a fraction of a millisecond on next-generation processors.

9 Conclusions

Texas provides persistence with good performance for C++ programs running on stock hardware and stock operating systems. It is surprisingly simple and flexible, however, and requires little or no change to existing compilers.

The current version of our persistent store, excluding the log-structured storage module, is less than four thousand lines of C++ code. Log-structured storage adds less than two thousand lines. While we expect these figures to grow somewhat, and we expect to add more modules (e.g., for concurrency control), this is a gratifyingly small amount of code for a persistent storage mechanism with high-performance address translation and checkpointing.

This simplicity and flexibility is largely due to the use of pointer swizzling at page fault time, which lets us treat address translation, caching, and recovery as nearly orthogonal features. Address translation can be performed by access protection trap handlers, with a conventional virtual memory performing data caching. Write protection handlers support logging writes to a modular log-structured storage subsystem.

Page fault-time address translation has essentially zero overhead for programs with good locality, and appears to be competitive with other schemes in general. Likewise, pagewise logging performs very well when write locality is good, and sub-page logging promises to keep it competitive when write locality is poor.

Our modular approach appears to work well, allowing us to easily change storage management schemes. Texas and its log-structured storage system are promising both as an easy-to-use, high-performance persistence mechanism and as a flexible testbed for basic research in locality, distributed systems, and storage techniques.

Acknowledgements

We would like to thank the organizers and participants of POS-V, as well as several seminar students who helped design Texas—Roberto Bayardo, Atif Chaudry, Shankar Krishnamoorthy, Rajiv Kumar and Rekha Singhal. Thanks also to Seth White, for sharing the benchmark code from [6]; to Mark Johnstone, for last-minute programming help with the log-structured storage system; and to Janet Swisher for help with the figures and text.

References

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

[2] W. Cockshott, M. Atkinson, K. Chisholm, P. Bailey, and R. Morrison. Persistent object management system. *Software Practice and Experience*, 14(1):49–71, January 1984.

[24] A similar approach has been taken for compressing compiled code in the Acorn system, though it operates on a program-wise basis rather than a pagewise basis.

[3] 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).* Morgan Kaufman, Martha's Vineyard, Massachusetts, September 1990.

[4] 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.

[5] J. Eliot B. Moss. Working with objects: To swizzle or not to swizzle? Technical Report 90–38, University of Massachusetts, Amherst, Massachusetts, May 1990.

[6] 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, Canada, October 1992. To appear.

[7] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[8] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent garbage collection on stock multiprocessors. In *Proceedings of SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 11–20. SIGPLAN ACM Press, June 1988. Atlanta, Georgia.

[9] Paul R. Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management*, pages 1–42, St. Malo, France, September 1992. Springer-Verlag Lecture Notes in Computer Science vol. 637.

[10] Jeffrey S. Chase, Henry M. Levy, Edward D. Lazowska, and Miche Baker-Harvey. Lightweight shared objects in a 64-bit operating system. In *ACM SIGPLAN 1992 Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '92)*, Vancouver, British Columbia, Canada, October 1992.

[11] Hans-Juergen Boehm and Alan Demers. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, September 1988.

[12] David Wallace. Runtime type support in C and C++. Technical Report revision 1.1, Cygnus Reports, 1992.

[13] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, October 1991.

[14] Richard Greenblatt. Unpublished technical report on the Moby address space. Technical report, Lisp Machines, Incorporated.

[15] M. Anderson, R. D. Pose, and C. S. Wallace. A password-capability system. *Computer Journal*, 29(1), 1986.

[16] Jose Alves Marques and Paulo Guedes. Extending the operating system to support an object-oriented environment. In *ACM SIGPLAN 1989 Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '89)*, pages 113–122, New Orleans, Louisiana, October 1989.

[17] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Thirteenth ACM Symposium on Operating Systems Principles*, pages 1–15, Pacific Grove, California, October 1991.

[18] Jacques Cohen. Garbage collection of linked data structures. *Computing Surveys*, 13(3):341–367, September 1981.

[19] Paul R. Wilson and Thomas G. Moher. Design of the opportunistic garbage collector. In *ACM SIGPLAN 1989 Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '89)*, pages 23–35, New Orleans, Louisiana, October 1989.

[20] Barry Hayes. Using key object opportunism to collect old objects. In *ACM SIGPLAN 1991 Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '91)*, pages 33–46, Phoenix, Arizona, October 1991. ACM Press.

[21] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[22] Walter R. Smith and Robert V. Welland. A model for address-oriented software and hardware. In *Proc. 25th Hawaii Int'l Conference on Systems Sciences*, January 1991.

[23] 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.

[24] John Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *USENIX Summer Conference*, pages 247–256, Anaheim, California, June 1990. IEEE Press.

[25] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 96–107, April 1991. Santa Clara, CA.

[26] Paul R. Wilson. Operating system support for small objects. In *International Workshop on Object Orientation in Operating Systems*, Palo Alto, California, October 1991. IEEE Press. Revised version to appear in *Computing Systems.*

[27] Paul R. Wilson, Shubhendu S. Mukherjee, and Sheetal V. Kakkad. Anomalies and adaptation in the analysis and development of prepaging policies. *Journal of Systems and Software*, 1992. Technical Communication, to appear.

[28] 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.

[29] 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)*, pages 200–209, Santa Clara, California, April 1991.

[30] Michael Burrows, Charles Jerian, Butler Lampson, and Timothy Mann. On-line data compression in a log-structured file system. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, September 1992.