Platypus: Design and Implementation of a Flexible High Performance Object Store.

Zhen He[1], Stephen M Blackburn[2], Luke Kirby[1], and John Zigman[1]

[1] Department of Computer Science
Australian National University
Canberra ACT 0200, Australia
{Zhen.He, Luke.Kirby, John.Zigman}@cs.anu.edu.au
[2] Department of Computer Science
University of Massachusetts
Amherst, MA, 01003, USA
steveb@cs.umass.edu

Abstract. This paper reports the design and implementation of Platypus, a transactional object store. The twin goals of flexibility and performance dominate the design of Platypus. The design includes: support for SMP concurrency; standalone, client-server and client-peer distribution configurations; configurable logging and recovery; and object management which can accommodate garbage collection and clustering mechanisms. The first implementation of Platypus incorporates a number of innovations. (1) A new recovery algorithm derived from ARIES that removes the need for log sequence numbers to be present in store pages. (2) A zero-copy memory-mapped buffer manager with controlled write-back behavior. (3) A data structure for highly concurrent map querying. We present performance results comparing Platypus with SSM, the storage layer of the SHORE object store. For both medium and small OO7 workloads Platypus outperforms SHORE across a wide range of benchmark operations in both 'hot' and 'cold' settings.

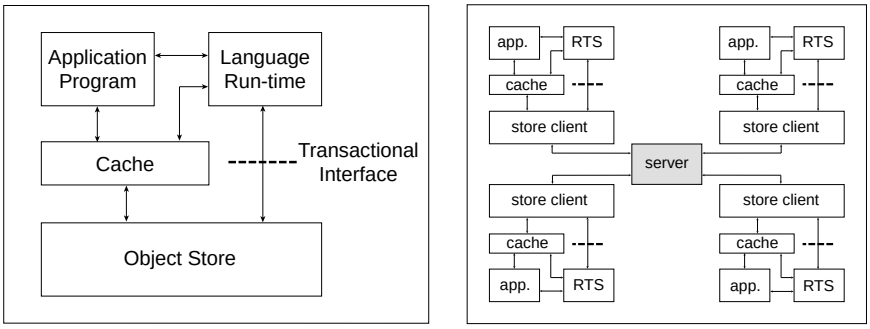
1 Introduction

This paper describes the design and implementation of an object store with a flexible architecture, good performance characteristics, and a number of interesting implementation features such as a new recovery algorithm, an efficient zero-copy buffer manager, and scalable data structures. Furthermore, this work represents a step towards our goal of both meeting the demands of database systems and efficiently supporting orthogonally persistent run-time systems.

The key to orthogonal persistence lies in the power of *abstraction over persistence*, the value of which is being recognized by a wider and wider audience. The potential for orthogonal persistence in mainstream data management appears to be massive and yet largely unrealized. As appealing as orthogonal persistence is, it must be applicable to the hard-edged data management environment in which the mainstream operates before it will see uptake in that world. Traditionally, object stores for orthogonally persistent languages have not strongly targeted this domain, so strong transactional semantics and good transaction throughput have not normally been goals or features of such stores.

On the other hand, relational, object relational, and object oriented databases have not focused strongly on the demands of a tightly coupled language client and so tend to provide a particularly inefficient foundation for orthogonally persistent systems. One result is a lack of systems that can effectively bridge that gap. We have developed Platypus in an attempt to address that gap.

The backdrop to the design of Platypus is the *transactional object cache* architecture which strongly separates storage and language concerns [Blackburn 1998]. Central to the architecture is the concept of both store and runtime sharing *direct access* to an object cache, where access is moderated by a protocol manifest in a transactional interface. This architecture provides a layer of abstraction over storage while removing impedance between the store and the language runtime implementations.



(a) The transactional object cache architecture. The cache is directly accessible shared memory.

(b) Underlying store architectures are transparent to the client runtime (RTS).

Fig. 1. The transactional object cache tightly couples the store and runtime/application through *direct access* to a cache, while abstracting over the store architecture.

A focus on abstraction is reflected in the design of Platypus, which has the flexibility to support a range of storage layer topologies (such as stand-alone, client-server, and client-peer). The design also includes replaceable modular components, such as the object manager, which is responsible for providing a mapping between the underlying *byte store*[1] and the *object store* projected to the client. This flexibility is critical to the utility of Platypus as it provides a single solid framework for prototyping and analyzing most aspects of store design.

The focus on minimizing store/runtime impedance impacts strongly on the implementation of Platypus and led directly to the development of a zero-copy buffer cache implementation. The zero-copy buffer cache presents a number of technical hurdles, notably with respect to the recovery algorithm and controlling page write-back.

[1] We use the term *byte store* to describe untyped, unstructured data.

The construction of a flexible, robust, high performance transactional store is a major undertaking, and in our experience is one filled with problems and puzzles. In this paper we touch on just a few aspects of the construction of Platypus. We address the architecture of the system because we believe that it embodies decisions that were key to delivering both flexibility and performance. We also address some major performance issues that arose in the course of implementation, as we believe that the solutions we have developed are novel and are likely to find application in other systems. For other details of the construction of Platypus which are not novel, such as the intricate workings of the ARIES [Mohan 1999; Mohan et al. 1992] recovery algorithm and the choice of buffer cache eviction policies, the reader is referred to the substantial literature on database and store design and implementation.

In section 2 Platypus is related to existing work. Section 3 describes the architecture of Platypus. Sections 4 and 5 focus on implementation issues, first with respect to the realization of an efficient zero-copy buffer manager, and then techniques that greatly enhance the scalability of Platypus. Section 6 presents and analyzes performance results for Platypus and SHORE using both the OO7 benchmark [Carey et al. 1993] and a simple synthetic workload. Section 7 concludes.

2 Related Work

This paper is written in the context of a large number of object store design and implementation papers in the POS workshop series and a vast number of database and OODB design and implementation papers in the broader database literature.

Architecture A number of papers have reported on the design of object stores targeted specifically at orthogonally persistent programming languages. Brown and Morrison [1992] describe a layered architecture that was used as the basis of a number of persistent programming languages, including Napier and Galileo. Central to the architecture is an abstract *stable heap* (also a feature of the Tycoon architecture [Matthes et al. 1996]). This model offered a clean abstraction over persistence in the store architecture, and the architectures were sufficiently general to admit a wide range of concurrency control policies. However the stable heap interface is fine grained both temporally and spatially (*read_word*, *write_word*, etc.) and so incurs substantially more traversals of the interface than a transactional object cache, which is coarse grained spatially (per object *c.f.* per word) and temporally (per transaction *c.f.* per access).

Of the many store architectures in the literature that do not directly target orthogonal persistence, most address specific optimizations and improvements with respect to OODB design. The THOR architecture from MIT [Liskov et al. 1999] has been the context for OODB research in the areas of buffer management, recovery algorithms, cache coherency and garbage collection. Other projects have supported a similar breadth of work in the context of an OODB architecture. The system that comes closest to the architectural goals of Platypus is SHORE [Carey et al. 1994], a successor to EXODUS and E [Carey et al. 1988], which set out to be flexible (with respect to distribution, for example), focussed on performance, and supported a persistent programming lan-

guage. SHORE is publicly available and has been used as the point of comparison in our performance analysis of Platypus (section 6).

By contrast to the systems built on the stable heap [Brown and Morrison 1992; Matthes et al. 1996], Platypus is built around the transactional object cache abstraction [Blackburn and Stanton 1998], which allows direct access to a cache shared by storage and language runtime layers (although it limits the language-side client to *transactional* concurrency control). Platypus is thus closer to orthodox OODBs than such systems. One notable distinction is that while many OODBMs focus on scalability of client-server (and in the case of SHORE, client-peer) architecture, Platypus's approach to scalability extends towards SMP nodes—either stand-alone, or in client-server or client-peer configurations.

Buffer Cache Implementation Most recent work on buffer management has focused on efficient use of buffers in a client-server context. Kemper and Kossmann [1994] explored adaptive use of both object and page grain buffering—object buffering incurs a copy cost, but may be substantially more space-efficient when objects are not well clustered. The THOR project [Liskov et al. 1999] has included a lot of work on buffer management in client-server databases, mostly focusing on efficiently managing client, server, and peer memory in the face of object updates at the clients. Platypus's buffer cache is in most respects orthodox (implementing a STEAL/NO-FORCE [Franklin 1997] buffer management policy), and is amenable to such techniques for managing data cached via a network. The novelty in our buffer manager lies in its use of memory mapping to address efficiency with respect to access of data resident on *disk*. We address the problem of controlling write-back of mapped pages which have become dirty (any such write-back must be strictly controlled to ensure recoverability). We also outline an extension to the ARIES [Mohan et al. 1992] recovery algorithm which obviates the need for log sequence numbers (LSNs) to be included on every page.

Scalable Data Structures There exists an enormous literature on data structures and scalability. Knuth [1997] provides an excellent discussion of sorting and searching that proved to be very helpful in our implementation efforts. Kleiman, Smalders, and Shah [1995] give an excellent treatment of issues relating to threads and concurrency in an SMP context, describing numerous techniques for maximizing scalability which were relevant to our store implementation. One of the scalability problems we faced was in the efficiency of map structures. After exploring splay trees [Sleator and Tarjan 1985], and hashes (including dynamic hashes [Larson 1988]) and their various limitations, we developed a new data structure which is a simple hybrid, the *hash-splay*. The hash-splay data structure is explained in full in section 5.1.

3 Architecture

At the core of the Platypus architecture are three major separable abstractions: *visibility*, which governs the inter-transactional visibility of intra-transactional change, *stability*, which is concerned with coherent stability and durability of data, and *availability*

which controls the availability of data (i.e., its presence or absence in a user-accessible cache) [Blackburn 1998]. Each of these corresponds to one or more major functional units (called 'managers') in the Platypus architecture. On top of these three concerns there exists an *object* manager, which projects an object store from an underlying 'byte store', and a *transaction* manager, which orchestrates visibility, stability and availability concerns on account of user requests across the transactional interface. This architecture is illustrated schematically in figure 2.

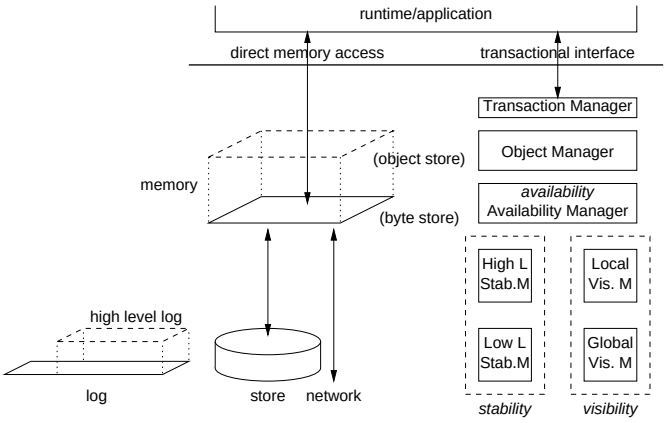


Fig. 2. Platypus architecture schematic. Managers (right) oversee the structure and movement of data between memory, a store, a log, and remote stores (left). Through the transactional interface the client runtime gains direct access to the memory it shares with the store.

3.1 Visibility

Isolation is a fundamental property of transactions [Härder and Reuter 1983]. While changes made by an uncommitted transaction must be isolated from other transactions to ensure serializability, serializability also demands that those changes be fully exposed once the transaction is successfully committed. Visibility is concerned with controlling that exposure in the face of particular transactional semantics. Visibility becomes somewhat more complex in the face of distribution, where a range of different strategies can be employed in an effort to minimize the impact of *latency* [Franklin 1996]. Since these strategies are concerned solely with efficiently maintaining coherent visibility in the face of distribution-induced latency, they are not relevant to purely local visibility issues. We thus carefully separate visibility into two distinct architectural units: a *local visibility manager* (LVM), and a *global visibility manager* (GVM).

The LVM is concerned with the mediation of transactional visibility *locally* (i.e., within a single uniformly addressable memory space, or 'node'). In the absence of network latency, optimism serves little purpose, so the LVM can be implemented using

an orthodox lock manager (as found in most centralized database systems).[2] The LVM operates with respect to *transactions* and abstract *visibility entities*, which the transaction manager would typically map directly to objects but which could map to any other entity over which visibility control were required. Through the transaction manager, transactions acquire locks from the LVM, which ensures that appropriate transactional visibility semantics are preserved and detects and reacts to deadlocks when necessary.

The GVM manages transactional visibility *between* the LVMs at distributed nodes (i.e., nodes which do not share a uniformly addressable memory). The GVM may implement any of the considerable range of transactional cache coherency algorithms [Franklin 1996]. The GVM operates between distributed LVMs and works with respect to *transactions*, *visibility entities*, and *sets of visibility entities*. By introducing *sets of visibility entities*, interactions can occur at a coarser grain. Just as visibility entities may be mapped to objects, sets of visibility entities may be mapped to pages, allowing global visibility to operate at either page or object grain (or both [Voruganti et al. 1999]). The GVM allows access rights over visibility entities (objects) and sets of visibility entities (pages) to be cached at nodes. Once rights are cached at a node, intra-node, inter-transaction visibility is managed by the LVM at that node. The GVM exists only to support transactional visibility between *distributed* LVMs, so is only necessary in the case where support for distribution is required.

3.2 Stability

Durability is another fundamental property of transactions. Härder and Reuter [1983] define durability in terms of *irrevocable stability*, that is, the effects of a transaction cannot be undone once made durable. An ACID transaction can only be undone logically, through the issue of another transaction (or transactions) which counters the first transaction. There are other important aspects to managing stability, including crash recovery, faulting of data into memory and the writing of both committed (durable) and uncommitted data back to disk. Given the pervasiveness and generality of the write ahead logging (WAL) approach to recovery [Mohan 1999], we model stability in terms of a store and a stable log of events (changes to the store)[3]. This model allows uncommitted store data to be safely stabilized, as the log always maintains sufficient information to bring the store data to a state that coherently reflects all committed changes to the store. Stability functionality is divided into two abstract layers managed respectively by the *high level stability manager* (HSM) and the *low level stability manager* (LSM).

The HSM is responsible for maintaining and ensuring the recoverability of 'high level' logs which stably reflect changes to the store induced by user actions (as opposed to actions internal to Platypus's management). For example a user's update to

[2] Optimism is a computational device for hiding latencies associated with the in-availability of information on which a decision must be made (typically due to physical dislocation or dependencies on incomplete concurrent computations). In a context where the information is available—such is the case with shared memory concurrency control—optimism is unnecessary and in fact deleterious (there is no point in computing 'optimistically' when information indicating the futility of that computation is immediately available).

[3] There were other important reasons for choosing WAL, not least of these is that unlike shadowing, WAL admits the possibility of a zero-copy buffer cache (see section 4.1).

an object, O, is recorded by the HSM in relatively abstract terms: 'transaction T mutated object O thus: $O' = O + \Delta$'. The HSM's log of changes is robust to lower level events such as the movement of an object by a garbage collector operating within the store because the HSM log does not record the physical location of bytes changed, but rather records deltas to abstractly addressed objects. Another role of the HSM is that it manages 'before-images' of objects which the user has stated an intention to update. Before images serve two important roles. They are necessary for the calculation of change deltas ($\Delta = O' \oplus O$), and they allow updates to the store to be undone. The latter is made necessary by our desire for a zero copy object cache and the possibility of implementing a STEAL policy in the buffer cache (i.e., that uncommitted mutated pages might be written to the store [Franklin 1997]). The HSM ensures that deltas are made stable at transaction commit time, and that before-images are made stable prior to any page being 'stolen' from the buffer cache.

The LSM has two roles. It maintains and ensures the recoverability of 'low level' logs which stably reflect changes to store meta-data and store structure (such as the movement of an object from one disk page to another or the update to some user-transparent index structure). The LSM also encapsulates HSM logs. Recovery is primarily the responsibility of the LSM. When the LSM finds a log record containing a HSM log record, it passes responsibility for that update to the HSM. This separation has two major benefits. First, it allows any low-level activities such as changes to object allocation meta-data to be completely separated from the high level object store behavior, which allows modular, independent implementation. Second, it opens the way for a two-level transaction system which can reduce or remove concurrency conflicts between low-level activities and high level activities (this is essential to minimizing conflicts between concurrent user transactions and garbage collection). For example a high-level transaction maybe updating an object while it is being moved by a low-level transaction. The two-level system allows both transactions to proceed concurrently.

3.3 Availability

The *availability manager* (AM) has a number of roles, all of which relate to the task of making data available (to the user and the store infrastructure). The centerpiece of the AM is thus the buffer cache—the (bounded) set of memory pages in which the store operates. For each node there exists one AM.[4] The AM manages three categories of pages: *store pages* which are mapped from secondary storage (through the LSM) or from a remote node (through the GVM), *log pages* which are mapped from secondary storage (through the LSM) and *shared meta-data* which is common to all processes executing at that node. Whenever there is demand for a new page, a frame is allocated in the buffer cache (possibly causing the eviction of a resident page). In the case of a store page, the LSM or GVM then faults the page into the new frame. Evicted pages are either swapped to their corresponding store page (a page STEAL), or to a special swap area on disk (for non-store pages).[5]

[4] The AM is managed cooperatively by all store processes executing concurrently on the node.

[5] This is simply achieved by 'mmap'ing non-store pages to a swap file.

The AM operates at two grains: *available regions*, which are typically mapped to objects, and *pages*, which is the operating system's unit of memory management. Through the transaction manager, the AM keeps track of which memory regions are in use (pinned) by the user, and which regions are dirty, information which is used when making decisions about which pages to evict.

3.4 Objects

An *object manager* (OM) is responsible for establishing (and maintaining) a mapping between an underlying 'byte store' (unstructured data) and the object store projected to the user. The nature of this mapping and the sophistication of the OM is implementation dependent and may vary enormously, from a system that treats objects as untyped byte arrays and simply maps object identifiers to physical addresses, to a system that has a strong notion of object type and includes garbage collection and perhaps a level of indirection in its object addressing. The strength of this design is that while the object manager is a relatively small component of the store in implementation terms, it largely *defines* the store from the user perspective. Thus making this a modular, separable component greatly extends the utility of the remainder of the store.

3.5 Transactions

The *transaction manager* is a relatively simple architectural component that sits between the transactional interface and the remainder of the architecture. It executes user transactions by taking user requests (such as readIntention, writeIntention, new, commit, and abort), and orchestrating the appropriate transactional semantics through the OM (creating new objects, looking up OIDs), AM (ensuring that the object is in cache), LVM (ensuring that transactional visibility semantics are observed), and HSM (maintaining the persistent state of the object).

Having described the Platypus architecture, we now discuss aspects of its implementation.

4 Implementing a Zero-Copy Buffer Cache

At the heart of the transactional object cache architecture is the concept of both store and runtime sharing *direct access* to an object cache, where access is moderated by a protocol manifest in a transactional interface (see figure 1). This contrasts with interfaces that require that objects be *copied* from the store to the runtime and back again. In our recent experience with a commercial database product and the SHORE [Carey et al. 1994] object database, neither would allow direct ('in-place') updates to objects.[6] Such a requirement is understandable in the face of normal user access to an object database. If direct, unprotected, access is given to a cached database page the grave possibility exists of the user *breaking* the protocol with an update that (inadvertently) occurrs outside

[6] We were able to circumvent this limitation in SHORE—*at the expense of recoverability* (this is described in section 6).

an object's bounds and overwrites critical metadata or an object for which the transaction had not declared a write intention. Furthermore, the efficient implementation of a direct access interface presents some challenges.

In pursuit of performance, Platypus is implemented with a zero-copy buffer cache. By zero-copy we mean that disk pages are memory mapped (avoiding copying that occurs in `stdio` routines), and that the store client (language runtime) is given direct access to the mapped memory, client updates being written directly into the mapped file. Of course a particular language runtime may well choose to copy objects before updating them (this happens in our orthogonally persistent Java implementation, where each object must be copied into the Java heap before it can be used). However, we have nonetheless removed one level of copying from the store's critical path.

Unfortunately the implementation of a zero-copy buffer cache in the context of a standard Unix operating system presents two significant hurdles: controlling the movement of data between memory and disk, and dealing with log sequence numbers (LSNs) in multi-page objects.

4.1 Controlled Write-Back in a Memory Mapped Buffer under POSIX/UNIX

While the `mmap` system call provides a means of accessing the filesystem without copying, `mmap` does not allow re-mapping (i.e. once data is mapped from disk address d to virtual address v it can only be written to some other disk address d' via a copy). Furthermore, while `msync` can be used to *force* a mapped page to be written back to disk, the only means of *stopping* data from being written back in a conventional user-level Unix process is through `mlock`, which is a privileged command that binds physical memory to a particular mapping. This presents a serious hurdle. Without using `mlock`, any direct updates to an mmapped file could at any time be written back to disk, destroying the coherency of the disk copy of the page.

We see Platypus's primary role as the back-end to a persistent language runtime system, and therefore do not think that it would be satisfactory for store processes to require root privileges. A standard solution in Unix to problems of this kind is to use a privileged *daemon process* to perform privileged functions as necessary. Using this approach, as soon as a *non-resident* page is faulted in, an *mlock daemon* could be triggered which would lock the page down, preventing any write-back of the page until the lock was released.[7] Control would then revert to the store process.

The use of a daemon process comes at the cost of two context switches (context must move from the store process to the daemon and back). Given our focus on performance, this could present a significant problem. Fortunately, the context switch can be completely overlapped with the latency associated with faulting in the page which is to be locked. Thus `mlock` semantics are attained without requiring store processes to be privileged and without an appreciable performance penalty. The approach is sketched in figure 3. When a store process requests a page to be faulted into the store, it first signals a condition variable that the `mlock` daemon is waiting on, then it triggers the I/O. The I/O causes the store process to block and thus giving the `mlock` deamon (which is now no longer blocked on a condition variable) a chance to be switched in. A lock-protected

[7] Through the use of `mlock` and `mprotect`, residency is under the control of Platypus.

place request in mlock circular buffer		*blocked on condition variable*
call mmap		*blocked on condition variable*
signal mlock daemon		*condition variable unblocked*
trigger I/O	*– control →*	*inspect circular buffer*
blocked on I/O		for each non-empty entry in circular buffer
blocked on I/O		call mlock & set done flag
I/O completes	*← control–*	*finished [time slice consumed]*
check mlock done flag		*blocked on condition variable [sleep]*
[block on condition variable	*– control →*	*continue processing circular buffer]*
store process		**mlock daemon**

Fig. 3. Overlapping I/O with context switches to a daemon to efficiently use `mlock` in a non-privileged store process. Access to the mlock circular buffer is done in a mutually exclusive manner. Events in brackets (*[]*) depict scenario where daemon does not complete request.

circular buffer is used to deal with the possibility of multiple store processes simultaneously making requests to the mlock daemon, and a flag is used to allow the store process to ensure that the `mlock` did in fact take place while it was blocked on the I/O. If the daemon has not run and the mlock has thus not occurred, the store client blocks on a condition variable which is signalled by the daemon when it completes the `mlock`. The `mlock` daemon is central to the successful implementation of the zero-copy buffer cache in Platypus.

4.2 Avoiding LSNs in a Write-Ahead Logging Recovery Algorithm

Write-ahead logging (WAL) algorithms (such as ARIES [Mohan 1999; Mohan et al. 1992]) revolve around two key persistent entities, a *store* and a *log*. The log is used to record changes (deltas) to the store. This allows I/O to be minimized by only writing store pages back to disk opportunistically or when no other choice exists.[8] Abstractly, the WAL protocol depends on some means of providing linkage between the store and the log at recovery time. Such linkage provides the means of establishing which of the changes recorded in the log have been propagated to the store and so are essential to the recovery process. Concretely, this linkage is provided by log sequence numbers (LSNs), which are essentially pointers into the log [Mohan et al. 1992]. The natural way to implement LSNs is to include them in store pages—a store page can then be read upon recovery and its state with respect to the log quickly established by checking the LSN field of the page.

LSNs have the less desirable side-effect of punctuating the persistent storage space with low-level recovery-related information. This perturbation is particularly noticeable in the context of objects that are larger than a page because the object will have to be broken up to make way for an LSN on each of the pages that it spans (a coherent object

[8] Note that it is because WAL maintains only one store page copy (unlike shadow-paging) that a zero-copy buffer cache is possible. Shadow-paging essentially requires a remapping of the memory↔store relationship each time a page is committed, which precludes zero-copy use of `mmap` as described above.

image will thus only be constructible by copying each of the parts into a new contiguous space). More generally, this use of LSNs precludes zero-copy access to objects of any size that span pages. Motivated by our goal of a zero-copy object store and the possibility of an object space not interrupted by low-level recovery data, we now describe an alternative approach that avoids the need for explicit LSNs to be stored in store pages without loosing the semantics that LSNs bring to widely used WAL algorithms such as ARIES.

Our approach follows an interesting remote file synchronization algorithm, *rsync* [Tridgell 1999; Tridgell and Mackerras 1998], which makes clever use of *rolling checksums* to minimize the amount of data required to synchronize two files. Checksums are used by rsync to identify which parts of the files differ (so that minimal deltas can be transmitted), and the *rolling* property of rsync's checksum algorithm allows checksums to be cheaply and incrementally calculated for a large number of offsets within one of the two files being compared.

At recovery time in WAL systems the log may contain many updates to a given store page. Thus there exists a many-to-one relationship between log records and store pages. A critical part of the recovery process is reducing this to a one-to-one relationship linking each store page with the log record corresponding to the last update to that page before it was flushed. An LSN stored in the first word of the store page provides a trivial solution to the problem—the LSN is a pointer to the relevant record. By associating a checksum with each log record, the problem is resolved by selecting the log record containing a checksum which matches the state of the store page. This amounts to a minor change to the recovery process and depends on only a small change to the generation of log records during normal processing. Each time a log record is produced, the checksums of all updated pages are calculated and included in the log record. Thus the use of checksums in place of LSNs is in principle very simple, yet it avoids the fragmentation effect that LSNs have on large objects (ones that span multiple pages). However this simple change to the WAL algorithm is made more complex by two issues: the cost of checksum calculation, and the problem that checksums may not *uniquely* identify page state. We now address these concerns in turn.

Cheap Checksum Calculation The 32-bit Adler rolling checksum [Deutsch 1996] efficiently maintains the checksum (or '*signature*') for a window of k bytes as it is slid across a region of memory one byte at a time. The algorithm calculates each new checksum as a simple function of the checksum for the last window position, the byte that just entered the window, and the byte that just left the window. It is both cheap and incremental.

In an appendix to this paper, we generalize the Adler checksum to *partial checksums* (which indicate the contribution to the total checksum made by some sub-part of the checksummed region). We then show that in addition to its property of incrementality, the Adler checksum has a more general *associative* property that allows checksums to be calculated using *delta checksums*. A delta checksum is simply a partial checksum calculated with respect to the byte-wise difference between old and new values. The new checksum for a modified region is equal to its old checksum plus the delta checksum of the bytes that were modified.

Thus rather than recalculate the checksum for a whole page, it is only necessary to establish the delta checksums for any updated objects and add them to the page's old checksum. Furthermore, the calculation of the checksum is very cheap (involving only an add and a subtract for each byte), and can be folded into the calculation of the delta which must be generated from the before and after images of the changed object prior to being written to the log at commit time. A derivation of the incremental checksum and a pseudo-code description are included an appendix to this paper.

Dealing With Checksum Collisions It is essential that at recovery time the state of a store page can be unambiguously related to exactly one and only one (of possibly many) log records describing changes to that page. Checksum collisions can arise from two sources. First, although any two consecutive changes must, by definition, leave the page in different states, a subsequent change could return the page to a previous state (i.e. $A \rightarrow B \rightarrow A$). Secondly, two store page states can produce the same checksum. The 32-bit Adler checksum used in Platypus has an *effective bit strength* of 27.1 [Taylor et al. 1998], which means that the probability of two differing regions sharing the same checksum is approximately $1/2^{27}$. So a checksum alone is inadequate.

To this end we introduce page flush log records (PFLR) to the basic WAL algorithm. Positive identification of a page's log state can be simplified if for each time a page is flushed to disk a PFLR is written to the log—each flush of a store page is followed by a log record recording that fact. The problem then reduces to one of determining which of *two* states the store page is in. Either both the flushed page and the associated PFLR which follows it are on disk or only the flushed page made it to disk before a failure occurred. If the PFLR contains the page's checksum, determining which state the system is in is trivial unless there is a checksum collision. However, a checksum collision can be trivially identified at the time that a PFLR is created by remembering the checksum used in the last PFLR for the page in question and identifying any occurrence of consecutive PFLRs which contain the same page checksum. As long as consecutive PFLRs for a given page hold unique checksums, the state of a store page can be unambiguously resolved.

If the page state is the same (i.e. the state of the page in consecutive flushes is unchanged), then the page need not actually be flushed and a flag can be set in the second PFLR indicating that the page has been logically flushed although its state has not changed since the last flush. When collisions do occur (i.e. same checksum but different store state), they can be disambiguated by appending the second (conflicting) PFLR with the full state of all regions of the page that have been modified since the last page flush. A conflict between the two PFLRs is then simply resolved by checking whether the store page state matches with the state appended to the second PFLR. While in the worst case this could involve including the entire page state in the log record (if the entire page were changed), checksum collisions occur so infrequently that any impact on overall I/O performance will be extremely small (on average only 1 in 2^{27} page flushes could be expected to generate any such overhead, and the WAL protocol already minimizes the occurance of any page flushes by using the log rather than page flushes to record each commit).

The common-case I/O effect of using checksums instead of in-page LSNs is negligable as the total number of bytes written to disk in both cases is similar. In the case of the checksum, store page utilization improves slightly because there is no space lost to LSNs, an effect that is compensated for by including the checksum in each log record.[9] The store page and PFLR flushes can be asynchronous so long as the PFLR is flushed from the log before the next flush of the same store page. This is guaranteed, however, because a subsequent page flush can only arise as a result of an update to that page, and under the WAL protocol any such update *must* be logged before the page is flushed, and the logging of such an update will force all outstanding PFLRs in the log to be written out. However, on the rare occasion when collisions occur, the PFLR *must* be written to the log and the log flushed *prior* to flushing the store page.

Although the checksum approach requires that checksums be recalculated upon recovery, the time taken to perform an Adler checksum of a page is dominated by memory bandwidth, and so will be small in comparison to the I/O cost associated with bringing the page into memory. The checksum technique will therefore not significantly effect performance on the rare occasion that recovery does have to be performed.

Page checksums and PFLRs can thus be used to replace intrusive LSNs for the purposes of synchronizing store and log state at *recovery time*. However, LSNs may be used for other purposes during *normal execution* such as reducing locking and latching [Mohan 1990]. The use of conventional LSNs during *normal execution* is not effected by the use of PFLRs because LSNs may still be associated with each page while the page resides in memory (in data structures which hold other volatile metadata maintained for each page). Thus algorithms like Commit_LSN [Mohan 1990] remain trivially applicable when checksums and PFLRs are used instead of in-page LSNs.

4.3 Checksum-Based Integrity Checks

A side effect of maintaining checksums for each page is that page integrity checks can be performed if desired. Such checks can be used for debugging purposes or simply to increase confidence in the store client's adherence to the transactional object cache protocol. The first of these goals is met by a function that checks pages on demand and ensures that only areas for which the client had stated a writeIntention were modified. Someone debugging a store client could then use this feature to identify any pages which had inadvertently been corrupted. The second goal is met by checking each page prior to flushing it to disk, checking that only those regions of the page with a declared writeIntention had been updated. The associative property of the Adler checksum (see above and the appendix to this paper), makes the efficient implementation of such partial checksums efficient and straightforward. Because of the checksum's probabilistic behavior, such checks can only be used to check integrity to a *very high level of confidence*, not as *absolute* checks of integrity.

[9] LSNs are not normally explicitly included in log records because they are encoded as the *log offset* of the log record [Mohan et al. 1992].

5 Maximizing Scalability

Performance is a key motivator for Platypus, so it follows that scalability is a major implementation concern. After briefly outlining the approach we have taken to scalability, we describe in some detail two techniques that were significant in delivering good performance results for Platypus.

At the highest level, the store architecture must be able to exploit scalable hardware. For this reason the Platypus architecture supports different distributed hardware topologies including client server and client peer (figure 2 and section 3.3). The second of these has been demonstrated to scale extremely well on large scale multicomputers [Blackburn 1998]. In addition to distributed memory machines, the Platypus architecture is also targeted at SMP platforms, where memory coherency is not a problem but resource contention can be a major issue. Through the use of daemon processes all store processes on a node can share the common data and meta-data through a buffer cache that is `mmap`ed to the same address in each process and cooperatively maintained.

At a lower level, we found two primary sources of in-scalability when building and testing Platypus, *data structures that would not scale well*, and *resource contention bottlenecks*. We successfully addressed all major resource contention bottle necks by using techniques such as *pools* of locks, which provide a way of balancing the considerable space overhead of a POSIX lock with the major contention problems that can arise if locks are applied to data structures at too coarse a grain [Kleiman et al. 1995]. We now describe the *hash-splay*, a specific solution to a key data structure scalability problem.

5.1 Hash-Splay: A Scalable Map Data Structure

Map data structures are very important to databases and object stores. Platypus implements a number of large map structures, most notably the LVM has a map relating object identifiers (OIDs) to per-object meta-data such as locks, update status, and location. Prior to the development of Platypus, we implemented a PSI-compliant object store based on the SHORE [Carey et al. 1994] object database, and found that map lookups were a bottleneck for SHORE. Consequently we spent time analyzing the scalability characteristics of common map structures, most notably hash tables and splay trees [Sleator and Tarjan 1985], each of which have some excellent properties. Knuth [1997] gives a detailed review of both hashes and binary search trees.

Primary features of the memory resident maps used by Platypus and other object stores are that access time is at a premium *and* they have key sets of unknown cardinality. The same map structure may need to efficiently map no more than a dozen unique keys in the case of a query with a running time in the order of a few hundred microseconds, and yet may need to map millions of keys for a large transaction that will take many seconds. The challenge therefore is to identify a data structure that can satisfy both the variability in cardinality and the demand for fast access. Another characteristic of these maps is that they will often exhibit a degree of locality in their access patterns, with a ‘working set’ of w keys which are accessed more frequently (such a working set corresponds to the locality properties inherent in the object store client’s object access patterns).

While binary search trees are ideal for mapping key sets of unknown cardinality, their access time is $O(\log n)$ for n keys. On the other hand, hash tables can deliver access time performance of $O(1)$, *but* are not well suited to problems where the cardinality of the key set is unknown.

Our initial experiments were with splay trees, which are self-adjusting binary search trees. Splay trees re-adjust at every access, bringing the most recently accessed item to the root. While this provides excellent performance for access patterns of the type $\{aaaaabbbbbba\ldots\}$, splay trees perform particularly poorly with an access pattern like $\{abababababab\ldots\}$. We quickly found that by using 'sampling' (the tree is re-adjusted with some probability p), the re-adjusting overhead dropped appreciably with no appreciable degradation to access performance, providing a substantial overall performance improvement. This idea has been independently developed and analyzed by others [Fürer 1999].

Others have experimented with *linear hashing*[10] which adapts to the cardinality of the key set. This was originally developed for out-of-memory data [Litwin 1980; Larson 1980] and subsequently adapted to in-memory data [Larson 1988]. Aside from the question of key set cardinality, the reality of non-uniform hash functions means that a significant tradeoff must be made between space and time for any hash which takes the conventional approach of linear-probing on its buckets. Access time can be minimized by reducing collisions through a broad (and wasteful) hash space, *or* space efficiency can be maximized at the cost of relatively expensive linear searches through buckets when collisions do occur. Significantly, the impact of a *access locality* on the behavior of splays and hashes appears not to have been well examined in the literature, rather, performance analyses tend to address random access patterns only.

These observations lead us to the *hash-splay*, a simple data-structure which combines the $O(1)$ access time of a hash with the excellent caching properties and robustness to key sets of unknown cardinality that splay trees exhibit. The hash-splay consists of a hash where the hash targets are splay trees rather than other more traditional bucket structures.[11]

The hash-splay data structure has *excellent* caching properties. The combination of the re-adjusting property of the splay trees and the $O(1)$ access time for the hash means that for a working set of size w, average access times for items in the set is approximately $O(\log(w/k))$ where k represents the hash breadth. By contrast, for a simple splay it is approximately $O(\log w)$, and for a simple linear probing hash it is $O(n/k)$. If k is chosen to be greater than w, and a suitable hash function is chosen, working set elements should be at the roots of their respective buckets, yielding near-optimal access for these items. Significantly, k is selected as a function of w, not n, allowing k to be substantially smaller.

Platypus successfully implements the hash-splay to great effect. We find that our data structures perform very well for both huge transactions (where many locks and objects are accessed), as well as for small transactions (see section 6.2).

---

[10] Not to be confused with the standard technique of *linear probing* in hash tables.

[11] *c.f. hash-trees* [Agrawal and Srikant 1994], which are trees with hashes at each node.

6 Performance Analysis of Platypus

We conducted a number of experiments to assess the performance of Platypus. For each of the experiments we use SHORE as a point of comparison. After describing the experimental setup, we compare the overall performance of the two stores. We then compare the performance of each store's logging system. The comparative read fault performance of the two stores was the next experiment conducted. Finally we compare the scalability of the data structures used by each store for reading and locking objects.

6.1 Experimental Setup

All experiments were conducted using Solaris 7 on an Intel machine with dual Celeron 433 Mhz processors, 512MB of memory and a 4GB hard disk.

Platypus The system used in this evaluation was a first implementation of Platypus. The features it implements include: a simple object manager which features a direct mapping between logical and physical IDs; a local visibility manager; both low level and high level logging; and an availability manager. The page buffer implemented in the availability manager supports swapping of store and meta-data pages, and implements the NO-FORCE/STEAL recovery policy [Franklin 1997]. The features described in sections 4 and 5 have all been implemented with the exception of checksum-based integrity checks. The system is SMP re-entrant and supports multiple concurrent client threads per process as well as multiple concurrent client processes. Limitations of the implementation at the time of writing include: the failure-recovery process is untested (although logging *is* fully implemented); the availability manager does not recycle virtual address space; the store lacks a sophisticated object manager with support for garbage collection; no GVM has been implemented, precluding distributed store configurations.

SHORE SHORE is a persistent object system designed to serve the needs of a wide variety of target applications, including persistent programming languages [Carey et al. 1994]. SHORE has a layered architecture that allows users to choose the level of support appropriate for a particular application. The lowest layer of SHORE is the SHORE Storage Manager (SSM), which provides basic object reading and writing primitives. Using the SSM we constructed PSI-SSM, a thin layer providing PSI [Blackburn 1998] compliance for SSM. By using the PSI interface, the same OO7 code could be used for both Platypus and SHORE. SSM comes as a library that is linked into an application either as a client stub (for client-server operation) or as a complete system—we linked it in to PSI-SSM as a complete system to avoid the rpc overhead associated with the client-server configuration. We also bypass the SSM's *logical* record identifiers, using *physical* identifiers instead in order to avoid expensive logical to physical lookups on each object access. This simple mode of object identification closely matches the simple object manager currently implemented in Platypus. We perform *in-place* updates and reads of objects[12] which further improves the SHORE results reported in this section.

[12] SSM does not support *in-place* updates, so when a write intention is declared with respect to an object, PSI-SSM takes a before-image of the object to enable the undo of any updates if the

The PSI-SSM layer also has the effect of caching handles returned by SSM on a read request, reducing the number of heavy-weight read requests to SSM to one per-object per-transaction, as subsequent read requests for the same object are trapped by PSI-SSM. PSI-SSM uses the hash-splay data structure to implement its light-weight map. We used SHORE version 2.0 for all of the experiments reported in this paper.

OO7 The OO7 benchmark suite [Carey et al. 1993] is a comprehensive test of OODB performance. OO7 operations fall into three categories, traversals which navigate the object graph, queries like those typically found in a declarative query language and structural modifications which mutate the object graph by inserting and deleting objects. The OO7 schema is modeled around the components of a CAD application. A hierarchy of assembly objects (referred to as a module) sits at the top level of the hierarchy, while composite part objects lie at the bottom of the schema. Each composite part object in turn is linked to the root of a random graph of atomic part objects. The OO7 database is configurable, with three sizes of database commonly used—*small*, *medium*, and *large*. In our study we use small (11,000 objects) and medium (101,000 objects) databases. The results of OO7 benchmark runs are considered in terms of *hot* and *cold* results, corresponding to transactions run on warm and cold caches respectively. Operations are run in batches and may be run with *many* transactions per batch (i.e., each iteration includes a transaction commit), or as *one* transaction (i.e., only the last iteration commits). All of the results reported here were run using the *many* configuration. For a detailed discussion of the benchmark schema and operations, the reader is referred to the paper and subsequent technical report describing OO7 [Carey et al. 1993]. We use the same implementation of the OO7 benchmark for both Platypus and SHORE. It is written in C++ and has explicit read and write barriers embedded in the code. This implementation does not depend on store support for indexes and maps, but instead implements all necessary maps explicitly.

While we acknowledge the limitations of using any benchmark as the basis for performance analysis, the operations included in the OO7 suite emphasize many different aspects of store performance, so the performance of a system across the entire suite of operations can give a reasonable indication of the general performance characteristics of that system.

6.2 Results

Overall Performance Figure 4 shows the relative performance of Platypus with respect to SHORE for the twenty-four operations of the OO7 benchmark for both hot and cold transactions with small and medium databases. Of the 92[13] results, Platypus performs significantly worse in only five 5 (5.4%), the results are marginal in 9 (9.8%), and

transaction is aborted. While this is safe with respect to aborts, a STEAL of a page updated this way will lead to store corruption. This optimization therefore comes at the expense of recoverability in a context where swapping (STEALing) is occurring.

[13] We were unable to use the large database configuration (and unable to get results for t3b and t3c on the medium configuration) because of failures in PSI-SSM and/or SSM once the database size exceeded the available physical memory. This behavior may well be related to our use of in-place updates which is unsafe in the face of page STEAL.

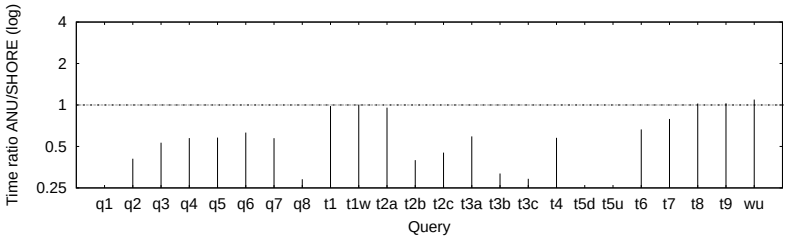
Platypus performs significantly better in 78 (84.8%). The geometric means of the time ratios for the four sets of results range from 0.580 to 0.436, corresponding to Platypus offering average performance improvements over SHORE from 72% to 129%. The hot results spend proportionately more time in the application and performing simple OID to virtual address translations (corresponding to application reads). Since both systems execute the same application code and the PSI-SSM layer performs OID to virtual address translations using a hash-splay (saving SHORE from performing such lookups), there is appreciably less opportunity for the advantages of Platypus over SHORE to be seen in the hot results.

Logging Performance Table 1 outlines the results of a comparison of logging performance between Platypus and SHORE. The results were generated by re-running the OO7 benchmark suite using both systems with logging turned off (i.e., without write I/O at commit time), and then subtracting these results from those with logging turned on. The difference between the results yields the logging (write I/O) cost for each system. The table reveals that Platypus performs logging far more efficiently than SHORE, which may be attributed to the zero copy nature of `mmap` (used to allocate memory for log records and used to flush log records to disk in Platypus). The better results for the small database suggests that logging in Platypus is particularly low latency (any latency effect will be diminished in the medium database results, where bandwidth may dominate). This result also supports the claim that the new recovery algorithm (including incremental checksum calculations) presented in section 4.2 is efficient in terms of logging costs.

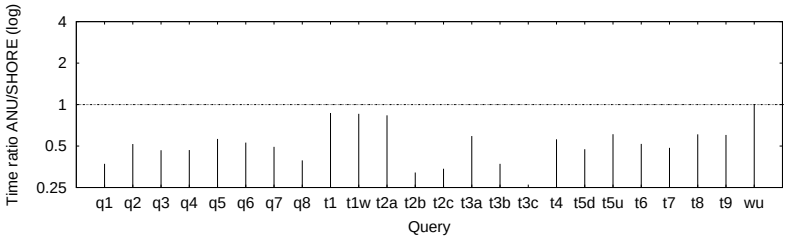
	Time Ratio Platypus/SHORE	
Store Size	Hot	Cold
Small	0.290 (0.0141)	0.396 (0.0169)
Medium	0.362 (0.0225)	0.401 (0.0181)

Table 1. Performance comparison of logging between Platypus and SHORE, showing the geometric means of ratios of logging times for selected operations for each system. Only times for the benchmark operations where at least 1% of execution time was spent on logging were included in the results (8 operations for medium results and 15 operations for small results). The results shown are average results from 3 repetitions. The values in brackets () depict the coefficient of variation of the corresponding result.

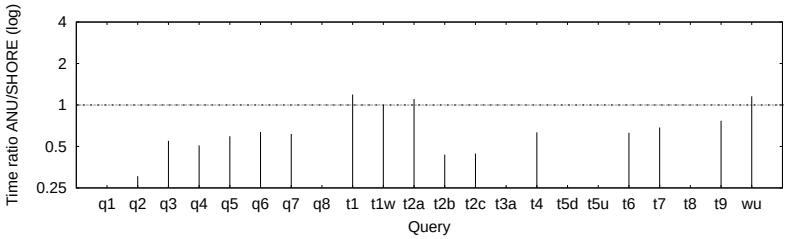
Read Fault Performance To compare how well the two systems perform read I/O, we subtracted the hot time from the cold time for each benchmark operation. This difference represents the cost of faulting pages into a cold cache, and any incidental per-transaction costs. The geometric means of the ratios between the results for each store are presented in table 2. The use of `mmap` both reduces the number of pages copied during page fault to zero and also reduces memory consumption. Reducing memory consumption has the secondary effect of deferring the need for swapping and so further reducing both read and write I/O.



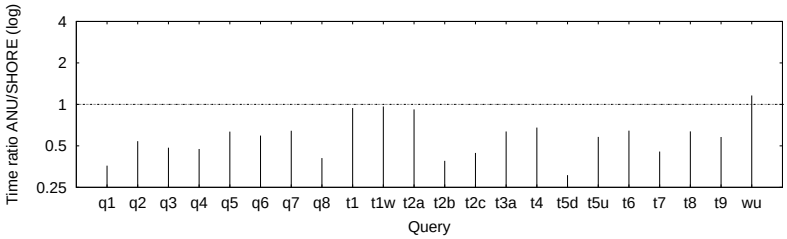
(a) Execution time ratios for *cold* benchmark runs over the OO7 *small* database (geometric mean = 0.495, with 0.0156 coefficient of variation).



(b) Execution time ratios for *hot* benchmark runs over the OO7 *small* database (geometric mean = 0.517, with 0.0141 coefficient of variation).



(c) Execution time ratios for *cold* benchmark runs over the OO7 *medium* database (geometric mean = 0.436, with 0.0142 coefficient of variation).



(d) Execution time ratios for *hot* benchmark runs over the OO7 *medium* database (geometric mean = 0.580, with 0.0176 coefficient of variation).

Fig. 4. Execution time comparison of Platypus and SHORE for the OO7 benchmark. Each impulse represents the ratio of average execution time for Platypus and the average execution time for SHORE for a particular operation in the OO7 suite. Average results are generated from 3 repetitions. A result less than 1 corresponds to Platypus executing faster than SHORE.

Store Size	Time Ratio Platypus/SHORE
Small	0.775 (0.0156)
Medium	0.338 (0.0176)

Table 2. Performance comparison of read fault time between Platypus and SHORE, showing the geometric means of ratios of read fault times for each system. The results shown are average results from 3 repetitions. The values in brackets () depict the coefficient of variation of the corresponding result.

Data Structure Scalability This experiment explores the scalability of map and locking data structures used in Platypus and SHORE. We did not use the OO7 benchmark, but constructed a trivial synthetic benchmark that involved no writes and did not read any object twice. These constraints allowed us to bypass the caching and before-image creation functionality of the PSI-SSM layer, and thereby get a direct comparison of the Platypus and SSM data structure performance. The synthetic benchmark operates over a singly-linked list of 100,000 small objects.[14] Before each test, a single transaction that reads all of the objects is run. This is done to bring all objects into memory and so remove any I/O effects from the timing of the subsequent transactions. After the initial transaction, many transactions of various sizes (traversing different numbers of objects) are run. To measure the 'per-read' cost, we subtract from the time for each transaction the time for an empty (zero reads) transaction, and divide the result by the number of objects read. Figure 5 depicts a comparison of the data structure scalability for Platypus and SHORE. As the number of objects read increases, the amount by which Platypus outperform SHORE also increases, indicating the data structures used in Platypus are substantially more scalable than those used in SHORE.

7 Conclusions

The power of abstraction over persistence is slowly being recognized by the data management community. Platypus is an object store which is specifically designed to meet the needs of tightly coupled persistent programming language clients and to deliver the transactional support expected by conventional database applications. The design of Platypus embraces the theme of abstraction, abstracting over store topology through a flexible distribution architecture, and encapsulating the mechanisms by which the object store abstraction is constructed from an untyped byte store. In addition to a high level of flexibility, Platypus is extremely functional. It is multi-threaded and can support a high degree of concurrency, both intra and inter process and intra and inter processor on an SMP architecture. The implementation of Platypus resulted in the development of a simple extension to the ARIES recovery algorithm and a simple combination of existing data structures, both of which perform well and should have application in other

[14] The objects contain only a pointer to the next object. The smallest possible object size was used to ensure that all objects would fit each store's cache and so avoid the possibility of swapping effects impacting on the comparison.

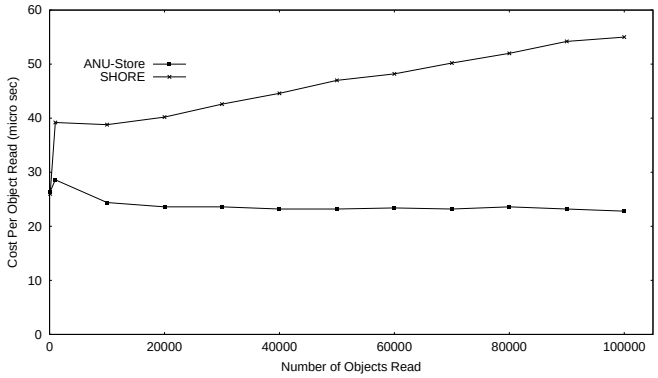


Fig. 5. Data structure scalability comparison between Platypus and SHORE. The results shown are average results from 5 repetitions. The coefficient of variation for the results shown above range from 0.0667 to 0.112.

object store implementations. Performance analysis shows that Platypus performs extremely well, suggesting that the combination of orthodoxy and innovation in its design and implementation has been a successful strategy.

The most obvious area for further work is to complete the construction of Platypus. The completed store would include a more sophisticated object manager that allows a garbage collector to be implemented, a fully tested failure-recovery process and a GVM (allowing for a distributed store configuration). Further experiments that may be conducted include assessing the performance of the store when the number of concurrent client processes are increased in a stand-alone configuration, measuring the scalability of the store in a distributed configuration and a comparative performance evaluation of using checksums for recovery versus LSN.

Acknowledgements

The authors wish to acknowledge that this work was carried out within the Cooperative Research Center for Advanced Computational Systems established under the Australian Government's Cooperative Research Centers Program.

We wish to thank Andrew Tridgell and Paul Mackerras for many helpful suggestions and comments on this work, and would like to particularly acknowledge the inspiration provided by their package 'rsync' for the checksum-based recovery algorithm used by Platypus.

Bibliography

[Agrawal and Srikant 1994] AGRAWAL, R. AND SRIKANT, R. 1994. Fast algorithms for mining association rules in large databases. In J. B. BOCCA, M. JARKE, AND C. ZANIOLO Eds., *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile* (1994), pp. 487–499. Morgan Kaufmann.

[Blackburn 1998] BLACKBURN, S. M. 1998. *Persistent Store Interface: A foundation for scalable persistent system design*. PhD thesis, Australian National University, Canberra, Australia. http://cs.anu.edu.au/~Steve.Blackburn.

[Blackburn and Stanton 1998] BLACKBURN, S. M. AND STANTON, R. B. 1998. The transactional object cache: A foundation for high performance persistent system construction. In R. MORRISON, M. JORDAN, AND M. ATKINSON Eds., *Advances in Persistent Object Systems: Proceedings of the Eighth International Workshop on Persistent Object Systems, Aug. 30–Sept. 1, 1998, Tiburon, CA, U.S.A.* (San Francisco, 1998), pp. 37–50. Morgan Kaufmann.

[Brown and Morrison 1992] BROWN, A. AND MORRISON, R. 1992. A generic persistent object store. *Software Engineering Journal 7*, 2, 161–168.

[Carey et al. 1994] CAREY, M. J., DEWITT, D. J., FRANKLIN, M. J., HALL, N. E., MCAULIFFE, M. L., NAUGHTON, J. F., SCHUH, D. T., SOLOMON, M. H., TAN, C. K., TSATALOS, O. G., WHITE, S. J., AND ZWILLING, M. J. 1994. Shoring up persistent applications. In R. T. SNODGRASS AND M. WINSLETT Eds., *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994*, Volume 23 of *SIGMOD Record* (June 1994), pp. 383–394. ACM Press.

[Carey et al. 1993] CAREY, M. J., DEWITT, D. J., AND NAUGHTON, J. F. 1993. The 007 benchmark. In P. BUNEMAN AND S. JAJODIA Eds., *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993* (1993), pp. 12–21. ACM Press.

[Carey et al. 1988] CAREY, M. J., DEWITT, D. J., AND VANDENBERG, S. L. 1988. A data model and query language for exodus. In H. BORAL AND P.-Å. LARSON Eds., *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, June 1-3, 1988* (1988), pp. 413–423. ACM Press.

[Deutsch 1996] DEUTSCH, P. 1996. RCF 1950 ZLIB compressed data format specification version 3.3. Network Working Group Request for Comments: 1950. http://www.faqs.org/rfcs/rfc1950.html.

[Fletcher 1982] FLETCHER, J. 1982. An arithmetic checksum for serial transmissions. *IEEE Transactions on Communications 30*, 1 (Jan.), 247–253.

[Franklin 1996] FRANKLIN, M. J. 1996. *Client Data Caching: A Foundation for High Performance Object Database Systems*, Volume 354 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, Boston, MA, U.S.A. This book is an updated and extended version of Franklin's PhD thesis.

[Franklin 1997] FRANKLIN, M. J. 1997. Concurrency control and recovery. In A. B. TUCKER Ed., *The Computer Science and Engineering Handbook*, pp. 1058–1077. CRC Press.

[Fürer 1999] FÜRER, M. 1999. Randomized splay trees. In *Proceedings of the tenth annual ACM-SIAM Symposium on Discrete Algorithms, 17-19 January 1999, Baltimore, Maryland.* (1999), pp. 903–904. ACM/SIAM.

[Härder and Reuter 1983] HÄRDER, T. AND REUTER, A. 1983. Principles of transaction-oriented database recovery. *ACM Computing Surveys 15*, 4 (Dec.), 287–317.

[Kemper and Kossmann 1994] KEMPER, A. AND KOSSMANN, D. 1994. Dual-buffering strategies in object bases. In J. B. BOCCA, M. JARKE, AND C. ZANIOLO Eds., *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile* (1994), pp. 427–438. Morgan Kaufmann.

[Kleiman et al. 1995] KLEIMAN, S., SMALDERS, B., AND SHAH, D. 1995. *Programming with Threads*. Prentice Hall.

[Knuth 1997] KNUTH, D. E. 1997. *The Art of Computer Programming* (second ed.), Volume 3. Addison Wesley.

[Larson 1980] LARSON, P.-Å. 1980. Linear hashing with partial expansions. In *Sixth International Conference on Very Large Data Bases, October 1-3, 1980, Montreal, Quebec, Canada, Proceedings* (1980), pp. 224–232. IEEE Computer Society Press.

[Larson 1988] LARSON, P.-Å. 1988. Dynamic hash tables. *Communications of the ACM 31*, 4 (April), 446 – 457.

[Liskov et al. 1999] LISKOV, B., CASTRO, M., SHRIRA, L., AND ADYA, A. 1999. Providing persistent objects in distributed systems. In R. GUERRAOUI Ed., *ECCOP'99 - Object-Oriented Programming, 13th European Conference, Lisbon, Portugal, June 14-18, 1999, Proceedings*, Volume 1628 of *Lecture Notes in Computer Science* (1999), pp. 230–257.

[Litwin 1980] LITWIN, W. 1980. Linear hashing: A new tool for file and table addressing. In *Sixth International Conference on Very Large Data Bases, October 1-3, 1980, Montreal, Quebec, Canada, Proceedings* (1980), pp. 212–223. IEEE Computer Society Press.

[Matthes et al. 1996] MATTHES, F., MÜLLER, R., AND SCHMIDT, J. W. 1996. Towards a unified model of untyped object stores: Experiences with the Tycoon Store Protocol. In *Advances in Databases and Information Systems (ADBIS'96), Proceedings of the Third International Workshop of the Moscow ACM SIGMOD Chapter* (1996).

[Mohan 1990] MOHAN, C. 1990. Commit_LSN: A novel and simple method for reducing locking and latching in transaction processing systems. In D. MCLEOD, R. SACKS-DAVIS, AND H.-J. SCHEK Eds., *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings* (1990), pp. 406–418. Morgan Kaufmann.

[Mohan 1999] MOHAN, C. 1999. Repeating history beyond ARIES. In M. P. ATKINSON, M. E. ORLOWSKA, P. VALDURIEZ, S. B. ZDONIK, AND M. L. BRODIE Eds., *VLDB'99, Proceedings of 25th International Conference on Very Large Data*

Bases, September 7-10, 1999, Edinburgh, Scotland, UK (1999), pp. 1–17. Morgan Kaufmann.

[Mohan et al. 1992] MOHAN, C., HADERLE, D. J., LINDSAY, B. G., PIRAHESH, H., AND SCHWARZ, P. 1992. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *TODS 17*, 1, 94–162.

[Sleator and Tarjan 1985] SLEATOR, D. D. AND TARJAN, R. E. 1985. Self-adjusting binary search trees. *Journal of the ACM 32*, 3, 652–686.

[Taylor et al. 1998] TAYLOR, R., JANA, R., AND GRIGG, M. 1998. Checksum testing of remote synchronsation tool. Technical Report DSTO-TR-0627 (March), Defence Science and Technology Organisation, Canberra, Australia. http://www.dsto.defence.gov.au.

[Tridgell 1999] TRIDGELL, A. 1999. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University.

[Tridgell and Mackerras 1998] TRIDGELL, A. AND MACKERRAS, P. 1998. The rsync algorithm. Technical report, Australian National University. http://rsync.samba.org.

[Voruganti et al. 1999] VORUGANTI, K., ÖZSU, M. T., AND UNRAU, R. C. 1999. An adaptive hybrid server architecture for client caching ODBMSs. In M. P. ATKINSON, M. E. ORLOWSKA, P. VALDURIEZ, S. B. ZDONIK, AND M. L. BRODIE Eds., *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK* (1999), pp. 150–161. Morgan Kaufmann.

Appendix: Calculation of Page Checksums Using Differences

We note the following definition of an 'Adler' signature $r(x,k)$ corresponding to a range of bytes $x_0, x_1, \ldots, x_{k-1}$, where M is an arbitrary modulus (usually 2^{16} for performance reasons) [Deutsch 1996]:[15]

$$\begin{aligned} r_1(x,k) &= (\sum_{i=0}^{k-1} x_i) \bmod M \\ r_2(x,k) &= (\sum_{i=0}^{k-1} (k-i)x_i) \bmod M \\ r(x,k) &= r_1(x,k) + M r_2(x,k) \end{aligned}$$

Calculating Partial Signatures The *partial checksum*, $p(x,k,s,e)$, corresponding to the contribution to $r(x,k)$ made by some sub-range of bytes $x_s, x_{s+1}, \ldots, x_e$ ($0 \le s < e \le k-1$) is then:

$$\begin{aligned} p_1(x,s,e) &= (\sum_{i=s}^{e} x_i) \bmod M \\ p_2(x,k,s,e) &= (\sum_{i=s}^{e} (k-i)x_i) \bmod M \\ p(x,k,s,e) &= p_1(x,s,e) + M p_2(x,k,s,e) \end{aligned}$$

Calculating Delta Signatures Consider the Adler signature $r(x,k)$ for a range of bytes $x_0, x_1, \ldots, x_{k-1}$. If the bytes in the range $x_s \ldots x_e$ are changed $x'_s \ldots x'_e$ ($0 \le s \le e \le (k-1)$), the components $r_1(x',k)$ and $r_2(x',k)$ of the new checksum $r(x',k)$ are then:

$$\begin{aligned} r_1(x',k) &= (r_1(x,k) - p_1(x,s,e) + p_1(x',s,e)) \bmod M \\ &= (r_1(x,k) - (\sum_{i=s}^{e} x_i) \bmod M + (\sum_{i=s}^{e} x'_i) \bmod M) \bmod M \\ &= (r_1(x,k) + (\sum_{i=s}^{e} (x'_i - x_i)) \bmod M) \bmod M \\ r_2(x',k) &= (r_2(x,k) - p_2(x,k,s,e) + p_2(x',k,s,e)) \bmod M \\ &= (r_2(x,k) - (\sum_{i=s}^{e} (k-i)x_i) \bmod M + (\sum_{i=s}^{e} (k-i)x'_i) \bmod M) \bmod M \\ &= (r_2(x,k) - (\sum_{i=s}^{e} (k-i)(x'_i - x_i)) \bmod M) \bmod M \end{aligned}$$

[15] Adler-32 is a 32-bit extension of the Fletcher checksum [Fletcher 1982].

If we further consider a *delta image*, $\bar{x}$ such that $\bar{x_i} = x'_i - x_i$ for the range of bytes corresponding to $x_s \ldots x_e$, and a partial checksum $p(\bar{x},k,s,e)$ with respect to $\bar{x_s} \ldots \bar{x_e}$, we note that the above can be further simplified:

$$\begin{aligned} r_1(x',k) &= (r_1(x,k) + p_1(\bar{x},s,e)) \bmod M \\ r_2(x',k) &= (r_2(x,k) + p_2(\bar{x},k,s,e)) \bmod M \\ r(x',k) &= (r(x,k) + p(\bar{x},k,s,e)) \bmod M \end{aligned}$$

Thus the checksum (signature) $r(x',k)$ for some modified page of size k can be calculated from its previous checksum $r(x,k)$, and the partial checksum of the delta image for the region of the page that was changed, where the delta image is a simple byte-wise subtraction of old from new values.

Simplifying Signature Calculation A naive implementation of the signature calculation would see a multiply in the inner loop. The following analysis shows how the (partial) signature implementation can be done with just a load, an add and a subtract in the inner loop.

$$\begin{aligned} p_2(x,k,s,e) &= (\sum_{i=s}^{e}(k-i)x_i) \bmod M \\ &= (\sum_{i=0}^{e-s}(k-i-s)x_{s+i}) \bmod M \\ &= ((k-s)\sum_{i=0}^{e-s}x_{s+i} - \sum_{i=0}^{e-s} i x_{s+i}) \bmod M \\ &= ((k-s)\sum_{i=0}^{e-s}x_{s+i} - \sum_{i=0}^{e-s}\sum_{j=0}^{i-1} x_{s+i}) \bmod M \end{aligned}$$

From the above, we note the following recurrence relations:

$$f_p(x,e,i) = \begin{cases} f_p(x,e,i-1) + x_{e-i} & \text{if } i > 0 \\ x_e & \text{if } i = 0 \\ \textit{undefined} & \text{otherwise} \end{cases}$$

$$f(x,e,i) = \begin{cases} f(x,e,i-1) + f_p(x,e,i-1) & \text{if } i > 0 \\ 0 & \text{if } i = 0 \\ \textit{undefined} & \text{otherwise} \end{cases}$$

We therefore note the following expressions for p_1, p_2 and p:

$$\begin{aligned} p_2(x,k,s,e) &= ((k-s)f_p(x,e,e-s) - f(x,e,e-s)) \bmod M \\ p_1(x,s,e) &= f_p(x,e,e-s) \bmod M \\ p(x,k,s,e) &= f_p(x,e,e-s) \bmod M + M(((k-s)f_p(x,e,e-s) - f(x,e,e-s)) \bmod M) \end{aligned}$$

which leads us to the following pseudo-code:

```
int signature(char *x,  /* start of block */
              int L,    /* block length */
              int s,    /* start index of signature region */
              int e     /* end index of signature region */
              )
{
    for (int i = e, p_1 = 0, p_2 = 0; i > s; i++) {
        p_1 += x[i];
        p_2 -= p_1;
    }
    p_1 += x[s];
    p_2 += (L-s) * p_1;
    p_1 &= (1<<16)-1;  /* mod 2^16 */
    p_2 &= (1<<16)-1;  /* mod 2^16 */
    return  p_1 | (p_2 << 16);
}
```

Fig. 6. Pseudo code for partial Adler-32 signature calculation.