This paper appears in *Proceedings of the 1996 ACM SIGMOD Int. Conf. on Management of Data*, Montreal, Canada, June 1996.
Note: the pages are numbered 318–329, as in the proceedings.

Safe and Efficient Sharing of Persistent Objects in Thor

B. Liskov, A. Adya, M. Castro, M. Day†, S. Ghemawat‡, R. Gruber§, U. Maheshwari, A. C. Myers, L. Shrira

Laboratory for Computer Science,
Massachusetts Institute of Technology,
545 Technology Square, Cambridge, MA 02139

{liskov,adya,castro,†,‡,§,umesh,andru,liuba}@lcs.mit.edu

Abstract

Thor is an object-oriented database system designed for use in a heterogeneous distributed environment. It provides highly-reliable and highly-available persistent storage for objects, and supports safe sharing of these objects by applications written in different programming languages.

Safe heterogeneous sharing of long-lived objects requires encapsulation: the system must guarantee that applications interact with objects only by invoking methods. Although safety concerns are important, most object-oriented databases forgo safety to avoid paying the associated performance costs.

This paper gives an overview of Thor's design and implementation. We focus on two areas that set Thor apart from other object-oriented databases. First, we discuss safe sharing and techniques for ensuring it; we also discuss ways of improving application performance without sacrificing safety. Second, we describe our approach to cache management at client machines, including a novel adaptive prefetching strategy.

The paper presents performance results for Thor, on several OO7 benchmark traversals. The results show that adaptive prefetching is very effective, improving both the elapsed time of traversals and the amount of space used in the client cache. The results also show that the cost of safe sharing can be negligible; thus it is possible to have both safety and high performance.

1 Introduction

Thor is a new object-oriented database system intended for use in heterogeneous distributed systems. It provides highly-reliable and highly-available storage so that persistent objects are likely to be accessible despite failures. Thor supports heterogeneity at the levels of the machine, network, operating system, and especially the programming language. Programs written in different programming languages can easily share objects between different applications, or components of the same application. Furthermore, even when client code is written in unsafe languages (such as C or C++), Thor guarantees the integrity of the persistent store.

This paper describes the interface and implementation of Thor and focuses on a novel aspect of Thor in each area. At the interface level, we discuss its *type-safe* heterogeneous sharing. At the implementation level, we describe our novel client-side cache management.

Thor provides a particularly strong safety guarantee: objects can be used only in accordance with their types. It thus provides *type-safe sharing*. Type-safe sharing provides the important benefit of data abstraction. Users can view objects abstractly, in terms of their methods; and they can reason about them behaviorally, using their specifications rather than their implementations.

Type-safe sharing requires that code uses objects *only* by calling their methods. A combination of techniques ensures type-safe sharing. Thor stores objects with their methods, and methods are implemented in Theta [LCD+94], a new, statically-typed programming language that enforces strict encapsulation. When client code is written in an unsafe language, Thor runs it in a separate protection domain. Finally, Theta and Thor provide automatic memory management and therefore avoid dangling references. As discussed in Section 3, we know of no other object-oriented database that provides safe, heterogeneous sharing.

Safe sharing is not without its potential performance costs, and other systems have chosen to forgo safety for improved performance. However, the paper shows how to avoid this choice by presenting techniques that improve application performance without losing safety.

Thor has a distributed implementation in which persistent storage is provided by servers and applications run at client machines. Clients cache copies of persistent objects (rather than copies of pages as in most other systems). We use a new, adaptive prefetching algorithm to bring objects into the cache. This algorithm allows the system to adapt to

---

† M. Day is now at Lotus: Mark_Day@crd.lotus.com
‡ S. Ghemawat is now at DEC SRC: sanjay@pa.dec.com
§ R. Gruber is now at AT&T Labs: gruber@research.att.com

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-91-J-4136. M.Castro is supported by a PRAXIS XXI fellowship.

Copyright ©1996 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that new copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept, ACM Inc., fax +1 (212) 869-0481, or permissions@acm.org.

the application, fetching many objects when clustering is effective, and fewer objects when it is not. Fetch requests include a desired *prefetch group* size; the server attempts to return that many objects. The prefetch group size varies according to the usefulness of previous prefetch groups.

Prefetch groups offer several benefits over fetching whole disk pages. Our approach improves cache utilization and reduces network load, since we fetch a small group of useful objects rather than everything on a disk page (when clustering is good, we fetch more than a page.) Prefetch groups also decouple decisions about client cache management from the properties of server disks and the optimal disk transfer size.

We also present some performance results (and their analyses) for our initial prototype, Thor 0, run on several OO7 benchmark traversals [CDN93]. The results show that Thor does well on these benchmarks, and that adaptive prefetching is very effective, reducing both the elapsed time of traversals and the amount of space used in the client cache. They also show areas where improvement is needed (we plan to fix the defects in our next release). Finally, the results show that techniques for reducing the cost of safe sharing can be very effective; it is possible to have both safe sharing and high performance.

The rest of the paper is organized as follows. In Section 2, describes the application interface of Thor. Section 3 discusses the issue of type-safe sharing in more detail. Section 4 describes the architecture of the Thor implementation. Section 5 covers client cache management, including related work. Section 6 gives performance results. Section 7 summarizes our contributions.

2 Thor Interface

Thor provides a universe of objects. Each object has a state and a set of methods; it also has a type that determines its methods and their signatures. The universe is similar to the heap of a strongly-typed language, except that the existence of its objects is not linked to the running of particular programs. Instead, applications use Thor objects by starting a Thor *session*. Within a session, an application performs a sequence of *transactions*; Thor currently starts a new transaction each time the application ends the previous one. A transaction consists of one or more calls to methods or to stand-alone routines; all the called code is stored in Thor. Clients end a transaction by requesting a commit or abort. A commit request may fail (causing an abort) because the transaction has made use of stale data. If a transaction commits, we guarantee that the transaction is serialized with respect to all other transactions, and that all its modifications to the persistent universe are recorded reliably. If a transaction aborts, any modifications it performed are undone at the client (and it has no effect on the persistent state of Thor).

Method calls return either *values* or *handles*. A value is a scalar such as an integer or boolean. A handle is a pointer to a Thor object that is valid only for the current client session; Thor detects attempts to use it in a different session.

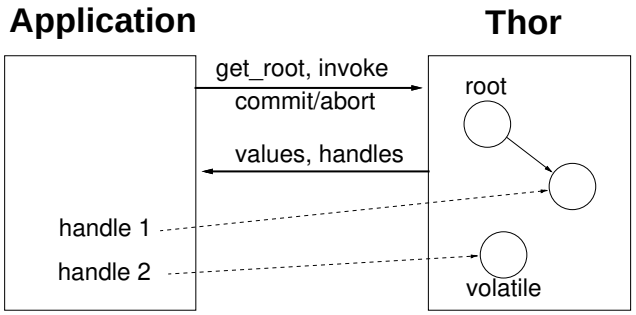

Figure 1: The Thor Interface

The universe has a set of persistent *server roots*, one for each server that stores persistent objects. (This is one way in which distribution shows through to users; the other is a mechanism for moving objects among servers.) All objects reachable by some path from a root are persistent. When an object becomes inaccessible from the roots, and also inaccessible from the handles of all current sessions, its storage is reclaimed automatically by the garbage collector [ML94, ML95].

An application can use volatile Thor objects, such as newly-created objects, during a session. (Of course, an application can also have its own volatile objects outside of Thor.) An initially volatile object may be made persistent by a transaction that makes a persistent object refer to the volatile one. Storage for volatile objects continues to exist as long as they are reachable from a session handle. At the end of a session, all volatile objects disappear.

Server roots have names that allow users to find their persistent objects when they start up sessions. Each server root is a directory object that maps strings to objects, which allows applications to locate their data. Objects of interest can be found by *navigation* or queries. (Thor does not yet support queries, although we have studied support for efficient queries [Hwa94].)

Figure 1 illustrates the Thor interface. Note that Thor objects and code remain inside Thor; this is an important way in which Thor differs from other systems.

2.1 Defining and Implementing Object Types

Object types are defined and implemented in Theta, a new, general-purpose object-oriented programming language [DGLM95, LCD+94]. Although Theta can be used separately from Thor, it was developed to support the type-safe sharing requirement of Thor.

Theta is a strongly-typed language. It distinguishes between specifications (which are used to define the interface and behavior of a new type) and implementations (code to realize the behavior). An object's implementation is encapsulated, preventing external access to the object's state. Theta provides both parametric and subtype polymorphism, and separates code inheritance from the subtyping mechanism. Theta objects reside in a garbage-collected heap, and

```
directory = type [T]

% overview: a directory provides a mapping from strings to
% objects of type T.

  size ( ) returns (int)
    % returns the size of self (number of mappings)

  insert (s: string, o: T)
    % adds the mapping from s to o to self, replacing
    % any previous mapping for s

  lookup (s: string) returns (T) signals (not_in)
    % if there is no mapping for s in self signals not_in
    % else returns the object s maps to

  elements ( ) yields (string, T)
    % yields all the mappings in self

end directory

create_directory[T] ( ) returns (directory[T])
  % returns a new, empty directory
```

Figure 2: Part of the type definition for directory

all built-in types do run-time checks to prevent errors, e.g., array methods do bounds checking.

A type definition provides only interface information; it contains no information about implementation details. An example is given in Figure 2. (The lines headed by % are comments.) The figure defines a parameterized directory type and a stand-alone procedure, create_directory, which creates new directory objects. Example instantiations of this type are directory[int], which maps strings to integers, and directory[any], which maps strings to objects of different types (any is the root of the Theta type hierarchy). Note that methods can be both procedures and iterators (an iterator yields a sequence of results one at a time [LSAS77]), and that calls can either return normally or signal an exception.

The information in type definitions and implementations is stored in Thor, in a *schema library* that can be used for browsing, for compilation of Theta programs, and for producing the programming language veneers that are discussed in the next section.

2.2 Veneers

Applications that use Thor's persistent objects can be written in various programming languages, but Thor objects are defined in Theta. This section discusses how application code makes use of Thor objects.

Applications interact with Thor through a thin interface layer called a *veneer*. The veneer is customized to the particular application language used. So far, no veneers have required compiler support. A veneer provides procedures that can be called to start a session or commit a transaction, and it provides translations between scalars stored in Thor

```
template <class T> class th_directory {
  int size( );
  void insert(th_string s, T v);
  T lookup(th_string s);
  th_generator<struct{th_string s; T v;}> elements( );
};
```

Figure 3: A veneer class declaration for C++

(e.g., integers) and related types in the application language.

The veneer exposes ordinary (non-scalar) Thor objects with a *stub generator*, a program that translates a Theta type definition into a application-language *stub type*. We refer to objects belonging to these stub types as *stub objects*. Stub objects are created in response to calls to Thor; when a call returns a handle, the application program receives a stub object containing the handle. The operations of the stub type correspond to methods of the Theta type; when a operation is called on a stub object, it calls the corresponding Thor method on the Thor object denoted by the handle in the stub object, waits for a reply, and returns to the caller.

For example, the stub generator for C++ produces a class corresponding to a Theta type, with a member function for each method of that type. The C++ class name is the Theta type name, prefixed with th_. Figure 3 gives part of the C++ class declaration for the directory parameterized class shown in Figure 2. The size method returns a C++ int. The insert method takes a th_string as an argument, i.e., it takes a stub object for a Thor string. (A th_string can be constructed by the client from a C++ char*, using a constructor provided by the veneer.) The lookup method does not signal an exception, because many C++ compilers do not yet support exceptions properly. Instead, if the Thor method signals an exception, the associated stub operation records the exception in a special exception object. The application is halted if it does not check for the exception. The elements method returns a *generator* object, which has a next member function used to obtain the next yielded result. An example of code that uses th_directory is

```
th_directory<int> di = th_get_root("thor", "dir");
di.insert("three", 3);
di.insert("six", 6);
th_commit( );
```

which retrieves and uses a directory from the root directory of the server named "thor".

We have defined veneers for C, C++, Perl, Tcl, and Java, none of which required compiler modifications. The existing veneers show that object-oriented features can be mapped easily to languages lacking objects. More information about veneers is available [BL94].

3 Type-safe Sharing

Type-safe sharing provides two important properties. First, because objects can be manipulated only by their methods, users can focus on how objects behave rather than how

they are implemented. This allows users to conceptualize objects at a higher level and to reason about them using their specifications. Second, safe sharing provides the benefits of modularity: we can reason about correctness locally, by just examining the code that implements a type, with the assurance that no other code can interfere.

These properties have proved very useful in programming, especially for large programs. We believe that they are at least as important for object-oriented databases, where the set of programs that can potentially interact with objects in the database is always changing; type-safe sharing ensures that new code cannot cause existing code to stop working correctly. Of course, type-safe sharing is not the only form of safety one should provide, since it guarantees only that individual objects continue to have reasonable abstract states. A constraint mechanism that expresses predicates over many objects would be useful, and access control is clearly needed.

The basis of type-safe sharing is the database programming language. That language must ensure two things: (1) *type correct calls* — every method call goes to an object with the called method, and (2) *encapsulation* — only code that implements an object can manipulate its representation. The first property can be ensured using either compile-time or runtime type checking, but static, compile-time checking is better because it rules out runtime errors and reduces runtime overheads. Static checking ensures that the declared type of a variable is a supertype of the object that variable refers to at runtime. This guarantee might be undermined by explicit memory management (since then a variable might point to nothing or to an object of some other type), unsafe casts, or arrays lacking bounds checking. Given type safety, encapsulation requires only that user code be constrained to interact with objects by calling their methods.

Implementing database objects in a type-safe language is not sufficient to provide type-safe sharing, however. The database also must restrict applications written in unsafe languages to interact with database objects only by calling their methods. Thor does this by running user code in a separate domain from the database and dynamically type checking user calls. This type checking ensures that the call is addressed to a legitimate object which has the called method, and that the call has the right number and types of arguments. Type checking is needed since stub code can be corrupted or bypassed. Thor ensures legitimacy of object references by checking the validity of handles and by managing the persistent heap with a garbage collector.

Other systems do not provide safe sharing. O2 [D+90] and GemStone [BOS91] store methods in the database. However, the languages provided by O2 for method definition are not safe (for example, one of these languages is an extension of C). GemStone does better since programmers use a variant of Smalltalk to define the methods in the database, and one can run a GemStone client in a separate domain from the process that manages its objects, as is done in Thor. However, GemStone exports an unsafe interface to client applications that allows direct access to an object's internal state.

Other object-oriented systems, e.g., SHORE [CDF+94] and ObjectStore [LLOW91], do not store methods in the database. Instead, applications compile or link against appropriate method code that is stored outside the database system. This approach is fragile: it works only if the right version of the method code is linked in. Furthermore, most systems (e.g., ObjectStore) allow method code to be written in unsafe languages, and run applications in the same domain as the persistent objects. In SHORE, the application runs in a separate domain, but objects are copied into client space, and modified objects are copied back later without any integrity checking; thus, SHORE objects can be corrupted by unsafe clients.

Safe sharing requires that someone write the definitions and implementations of persistent object types in the database language (e.g., Theta). Writing type definitions is similar to what other systems require (for example, we could translate ODMG descriptions [Cat94] to Theta), but writing implementations is more work. However, extra work is needed only when an entire application is written in one application language and does not share its objects with applications written in other languages. As soon as there is inter-language sharing, our approach is less work: other approaches require writing methods in each application language whereas we implement them once, in Theta. In fact, the need to write the methods in various languages is an impediment to heterogeneous sharing; an approach with a database language, on the other hand, encourages heterogeneous sharing.

In CORBA [OMG91], objects implemented in different programming languages can call one another. Two differences between Thor and CORBA are that Thor defines a common implementation language and ensures safe sharing.

3.1 Safe Sharing Techniques

Safe sharing requires running applications written in unsafe programming languages in a separate domain from the database and type checking all calls to the database. An application written in a safe language can avoid these costs. These requirements add a substantial cost to short calls (but not to long ones, such as queries). This section briefly discusses three ways to reduce this cost: batching, code transfer, and sandboxing; we have performed a more detailed performance study [LACZ].

Batching reduces total execution time by grouping calls into batches and making a single domain crossing for each batch, amortizing the domain-crossing penalty over all calls in the batch. We have studied batching of straight-line code [BL94] and loops [Zon95]; the latter is especially promising since it allows large batches and reduces the overhead of dynamic type checking: each call in the loop is checked just once, although the loop body may execute many times.

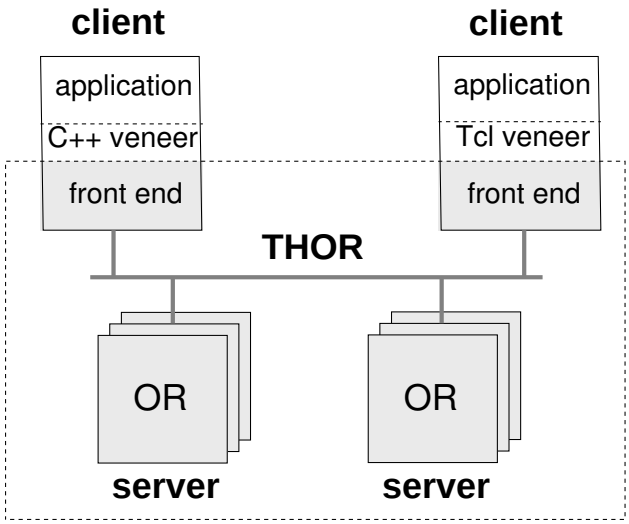

Figure 4: Architecture of Thor Implementation

Code transfer moves a portion of the application into the database. Typically, an application alternates between database and non-database computation; the database computation can be transferred into the database and executed there. Code transfer effectively increases the granularity of application calls. Queries are a code transfer technique; here we are interested in other kinds of code transfers that cannot be expressed in a query language such as OQL [Cat94] or extended SQL [SQL92].

We cannot move a procedure written in an unsafe language into the database. Safe ways of transferring code include writing the procedure in the database language (Theta), translating application subroutines into the database language, and translating application subroutines to type-safe intermediate code such as Java bytecodes [Sun95]. The latter approach is particularly promising since translators from various application languages to Java may be common in the future. The Java code could either be verified and then run inside Thor, or it could be compiled to Theta.

In the *sandboxing* [WLAG93] approach, (unsafe) application code/data is placed inside the database but allowed to access only a restricted range of virtual memory addresses. The restriction is enforced by inexpensive runtime checks. However, every call made from the sandboxed code to the database methods must still be type checked. Since the code transfer technique reduces the number of type-checked calls, sandboxing cannot perform as well as code transfer.

Our performance studies indicate that code transfer techniques [LACZ] offer the best performance. The performance studies presented in this paper are based on code transfer; our benchmark code was written in Theta.

4 System Architecture

Thor has the client/server architecture illustrated in Figure 4. Persistent objects are stored at servers called ORs (object repositories). Each OR stores a subset of the persistent objects; at any moment, an object resides at a single OR, but it can migrate from one OR to another. Objects can refer to objects at other ORs, although we expect inter-server references to be relatively rare. If object x refers to y, and both are created in the same transaction, they are likely to reside at the same OR at disk locations that are close together.

To achieve high availability, each OR is replicated at a number of server machines and its objects have copies stored at these servers. We plan to use a primary/backup replication scheme as in the Harp file system [LGG+91], but replication is not yet implemented.

An application interacts with Thor through a component called the FE (front end), which is created when an application starts a session. Although a typical configuration has an application and its FE running on a client machine that is distinct from servers, an application and its FE could run at a server, or an application could use a remote FE that runs at another client or at a server.

In this architecture, the server and its disk are likely to be the bottleneck. We address this problem in two ways: we offload work from ORs to FEs, and manage the server disk efficiently. We offload work by all application calls at the FE on cached copies of objects; this speeds up many applications since communication with ORs is infrequently required. FE cache management is the subject of Section 5.

We manage the server disk efficiently by a combination of two techniques. Like other databases, we record modifications made by committed transactions on a stable log, and write the new versions back to the database on disk in the background; however, we manage the log and database on disk differently from other systems. Our log is not normally kept on disk; instead, it is kept in primary memory at two servers, a primary and a backup, as in Harp [LGG+91]. The primary and backup each have an uninterruptible power supply that allows them to write the log to disk in the case of a power failure. This approach provides stability at a lower cost than writing the log to disk [LGG+91], and in addition provides highly-available object storage.

We keep new versions of modified objects in a volatile modified object cache (*mcache*) [Ghe95]. This has several advantages over the modified page cache that is used in other systems. First, the mcache allows us to ship objects rather than pages to ORs at transaction commit, making commit messages smaller. Second, we need not do installation reads [OS94] at commit time. Third, we can store many more modified objects in the mcache than modified pages in a page cache of the same size — the cache can absorb more writes, reducing the I/O needed to record new object states onto disk.

Our current prototype, Thor0, is a complete implementation of the architecture described in Sections 4 and 5, except that replication has not yet been implemented.

5 Client Cache Management

We manage the entire FE cache as an object cache: we fetch objects into the cache, and discard objects when more room is needed. Object caching is attractive because the cache only stores objects of interest, without keeping unused objects from the same disk pages. Most objects in an object-oriented database are likely to be considerably smaller than disk pages (e.g., the average in-disk object size for our implementation of the small OO7 database is 51 bytes). Typically, many objects are stored on each disk page. Even when objects are well-clustered on disk pages, no single clustering matches all uses [TN92]. A page cache works well when clustering matches usage, but wastes storage when it doesn't. It also wastes network bandwidth by sending unwanted objects; although this cost may not be large for the future LANs, it will still be significant for WANs (and wireless networks).

Another advantage is that object caching allows us to decouple disk management from fetching. Disks work well with large pages, since seeking to a page is expensive relative to reading it. However, with large pages, few objects per page may be used by a particular application, leading to wasted space at the client if there is a page cache, and wasted time in shipping the page across the network.

We combine an object cache with a new, adaptive prefetching algorithm, which prefetches many objects when clustering matches usage, and few objects when it doesn't. Thus we adapt minimizing the amount of cache space wasted and reducing network communication overheads.

Others have developed caching schemes that are similar to ours in some respects. Some systems (e.g., [Ont92, D+90]) fetch and discard objects, but none has an adaptive prefetching algorithm. Ontos [Ont92] allows the user to specify at object creation time whether object-based or page-based fetching should be used for the new object. Our technique is better: it adapts over time, uses a wide range of possible prefetch group sizes, and does not require programmer involvement.

Some systems, such as SHORE [CDF+94] and a system described by Kemper/Kossman [KK94], use a dual caching scheme, in which pages are fetched into a page cache, and objects are then copied into an object cache. In these systems a page can be discarded while useful objects from the page remain in the object cache. (SHORE copies objects on first use; Kemper/Kossman compare this approach to a *lazy copy* approach that copies the useful objects from a page only when it is about to be removed from the page cache.) However, such systems cannot get the full benefit of object caching: they always fetch entire pages, even when many of the objects on the page are not useful.

Object caches are harder to manage than page caches, however. In a page cache, available memory is divided into page-sized slots, allowing a new page to overwrite an old one. Objects are of variable size and require more complex memory management. The Thor0 implementation uses a variant of copying garbage collection [Bak78] to manage the cache. The FE reads an arriving prefetch group into the beginning of free space. When free space runs low, some persistent objects are evicted from the cache, followed by a copying collection. The collection compacts storage to provide enough contiguous space for incoming prefetch groups. The next section describes cache management issues in more detail, while Section 5.2 discusses our adaptive prefetching algorithm.

5.1 Details

Swizzling and Fetching. Thor objects refer to one another using 64-bit location-dependent names called *xrefs*. An xref is a pair, containing the id of an OR and some OR-dependent information. With a location-dependent name, an FE can easily determine from which OR to fetch an object. The details of our scheme, including support for object migration, are available [DLMM94].

Using xrefs for object addressing at the FE cache would be expensive because each object use would require a table lookup. Like many other systems [SKW92, C+89], we *swizzle* xrefs, replacing them with local addresses. Swizzling is accomplished using a *resident object table* (ROT) that maps the xrefs of resident objects to their local addresses.

A number of studies have compared various approaches to swizzling and object fault detection; e.g., see [Day95, Mos92, HM93, WD92]. Thor0 uses a form of *node marking*. All the references in an object are swizzled at once. If the object refers to an object not present in the cache, we create a *surrogate*, a small object that contains the xref of the missing object; the pointer is swizzled to point to the surrogate, and an entry is made for the surrogate in the ROT. (Our surrogates are similar to *fault blocks* [HM93].)

Fetching occurs when an attempt is made to use a surrogate; if the object named by the xref in the surrogate is not yet at the FE, the FE sends a fetch request to the OR requesting the object identified by the xref. The OR responds with a *prefetch group* containing the requested object and some others. The requested object is swizzled, the surrogate is modified to point to it (and thus becomes a *full surrogate*), and the method whose call led to the fetch is run. We need the level of indirection provided by the full surrogate because objects in the cache may have been swizzled to point to the surrogate. The indirection links are snapped at the next garbage collection.

The other objects in the prefetch group are entered in the ROT but are not swizzled; instead, we swizzle them at first use. This *lazy swizzling* allows us to minimize the cost for prefetching an object that is not used and reduce the number of surrogates.

Transactions. We use an optimistic concurrency control scheme [AGLM95]. As a transaction runs, the FE keeps track of which objects it reads and modifies. Our use of Theta allows us to recognize *immutable* objects; we don't track them, reducing both work while a transaction runs and the amount of information sent to ORs at commit time.

When a transaction commits, the FE sends to ORs the xrefs of all persistent objects the transaction read, and the states of all persistent objects the transaction modified. Modified objects are unswizzled before being sent. While doing unswizzling, the FE may discover pointers to volatile objects that are being made persistent by the transaction; the FE sends them to the ORs as well. The ORs then validate the transaction [AGLM95].

After an OR commits a transaction, it sends *invalidation messages* in the background to all FEs that contain cached copies of objects modified by the transaction. (The tables recording which objects are cached at FEs are kept at a coarse granularity and do not take up much space.) The FE discards invalidated objects by turning them into surrogates and aborts the current transaction if it used a modified object.

Collection. Objects are discarded from the heap only in response to invalidation messages, or when room is needed for new objects. When the heap is almost full, the FE goes through the prefetch groups in FIFO order, discarding all objects in a group that have not been modified by the current transaction, until it reaches a threshold; at present we attempt to recover 50% of the space in from-space, although this is an implementation parameter. Objects are discarded by turning them into empty surrogates. We call this process *shrinking* . Then the FE does a copying garbage collection using the handle table and the modified objects as roots. The collector snaps pointers to full surrogates to point to the actual objects; at the end of collection, there will be no full surrogates in to-space. Since empty surrogates are small, the collector copies them rather than unswizzling all pointers to the surrogate, since that would require that we reswizzle the objects containing the pointer at next use.

In Thor0 we never discard modified objects, which limits the number of objects a transaction can modify and sometimes causes us to break up a transaction into smaller transactions. We will eliminate this restriction in our next prototype.

5.2 Prefetching

The ORs store objects in *segments*, which are similar to large pages. The size of a segment is selected based on disk characteristics; in our current system they are 28.5 KB, the track size on our disks. Each segment holds some number of objects. Objects are not split across segments. (28.5 KB is the default size, objects larger than this are assigned their own segment.)

Related objects are clustered in segments. Currently Thor uses a simple-minded clustering strategy: objects made persistent by the same transaction are placed in the same segment. We plan to study smarter policies.

When the FE sends a fetch request to an OR, the OR reads the segment of the requested object from disk if necessary. Then the OR picks a *prefetch group*, a subset of the segment's objects that includes the requested object, and sends it to the FE.

We use a simple *sub-segment prefetching* policy. We split each segment up into groups containing k objects each: the first k objects belong to the first group, the second k objects belong to the second group, and so on. In response to a fetch request for a particular object, the OR sends the entire group that contains the requested object. (Earlier we used a policy that determined the prefetch group based on the structure of the object graph [Day95]. This policy was discarded because it often generated very small prefetch groups near the fringes of the object graph. Furthermore, the traversal of the object graph added a non-trivial run-time cost to the OR.)

We vary the requested group size k dynamically to improve the effectiveness of prefetching. Each time the FE generates a fetch request, it sends a new value for k to the OR. If prefetching is performing well, i.e., the application has used many of the objects prefetched earlier, the FE increases k; if prefetching is not working well, the FE decreases k.

In order to compute the requested group size, the FE maintains two counters, fetch, which is an estimate of the number of fetched objects, weighted toward recent fetches, and use, which is an estimate of how many objects have been used, weighted toward recent uses. Every time an object is swizzled, i.e. used for the first time after being fetched, use is incremented. When a prefetch group of size N arrives, the FE recomputes fetch and use using exponential forgetting:

```
fetch := fetch/2 + N
use := use/2 + 1
```

The FE also maintains size, the estimate of the right group size; size is constrained to be between 20 and 700. The upper limit is a rough estimate of the maximum number of objects in a segment, and the lower limit is the value of the increment used to adjust the size. When the FE sends a fetch request it computes the new size as follows:

```
if use/fetch < thresh
  then size := max ( size−20, 20 )
   else size := min ( size+20, 700 )
```

The variable thresh is a threshold whose current value of 0.3 was determined empirically.

The benefit of this adaptive scheme is that it works well under most conditions. When the clustering of objects into segments matches the application access pattern, the scheme quickly increases the prefetch group size until a large fraction of a segment is sent in response to each fetch request, reducing the number of network round-trips. When clustering and the application access pattern do not match very well, the dynamic scheme quickly reduces the prefetch group size until only a small number of objects is sent in response to each fetch request. This reduces the number of useless objects sent over the wire, avoids the overhead of caching these objects, and avoids ejecting useful objects to make room for useless ones.

The implementation of the prefetching algorithm has a few subtleties. Since the group size varies, the group sent in one fetch reply may overlap the groups sent in earlier fetch replies. The OR maintains a per-FE table that keeps track of objects

sent recently to that FE and uses it to weed out duplicates before shipping the prefetch group to the FE. The OR also weeds out objects over some threshold size (at present the threshold is 800 bytes); such "big" objects are sent only in response to explicit fetch requests.

6 Performance Studies

This section shows how Thor0 performs on OO7 traversals [CDN93] for the small and medium databases. We also present results comparing adaptive prefetching to fixed-size techniques such as page fetching. Our system does well on most traversals, and adaptive prefetching outperforms fixed-size prefetching on both T1 (where clustering works) and T6 (where it does not).

6.1 Experimental Setup

The experiments ran a single client/FE on a DEC 3000/400 workstation, and an OR on another, identical workstation. Both workstations had 128 MB of memory and ran OSF/1 version 3.2. They were connected by a 10 Mb/s Ethernet and had DEC LANCE Ethernet interfaces. The database was stored on a DEC RZ26 disk, with a peak transfer rate of 3.3 MB/s, an average seek time of 10.5 ms, and an average rotational latency of 5.5 ms. The FE cache was 12 MB. The OR cache was 36 MB, of which 6 MB were used for the mcache. We used 28.5 KB segments (one track on our disk).

The experiments ran the single-user OO7 benchmark [CDN93]. The OO7 database contains a tree of *assembly* objects, with leaves pointing to three *composite parts* chosen randomly from among 500 such objects. Each composite part contains a graph of *atomic parts* linked by *connection* objects; each atomic part has 3 outgoing connections. The *small* database has 20 atomic parts per composite part; the medium has 200. The total size is 6.7 MB for the small database and 62.9 MB for the medium database. We implemented the database in Theta, following the specification [CDN94] closely.

We ran traversals T1, T6, T2a, and T2b. (We also ran T2c but do not report the results since they are identical to those for T2b.) These traversals perform a depth-first traversal of the assembly tree and execute an operation on the composite parts referenced by the leaves of this tree. T1 and T6 are read-only; T1 reads the entire graph of a composite part, while T6 reads only its *root atomic part*. Traversals T2a and T2b are identical to T1 except that T2a modifies only the root atomic part of the graph, while T2b modifies all the atomic parts.

The traversals of the small database and traversal T6 of the medium database were completely implemented in Theta, and consist of a single client method call and transaction. We had to break the other traversals of the medium database into several client calls because we can only do garbage collection between calls. There are a total of 9112 client calls that introduce a negligible overhead (0.37 seconds). We also had to break these traversals into several transactions because we cannot evict modified objects from the FE cache.

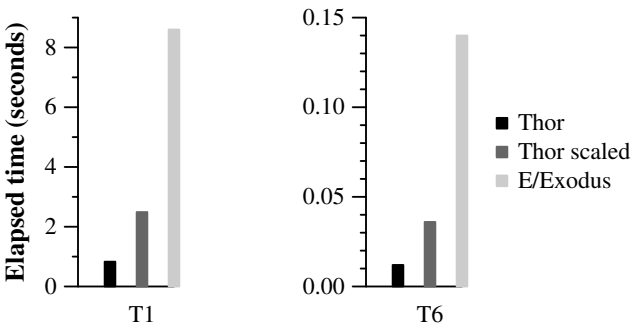

Figure 5: Hot Traversals, Small Database

There are 110 transactions. The overhead introduced by having multiple transactions is bounded by the total commit overhead which is less than 10% for these traversals.

The code was compiled with DEC's CXX and CC compilers with optimization flag -O2. Our experiments followed the methodology in [CDN94]. Hot traversals run five times within a single transaction starting with cold FE and OR caches; the elapsed time is the average of the middle runs, so that start up costs and commit times are excluded (but concurrency control costs are included). Cold traversals run one traversal on cold FE and OR caches; their elapsed times include commit time. The traversals were run in an isolated network; we report the average of the elapsed times obtained in 10 runs of each traversal. The standard deviation was always below 2% of the reported value.

To put our performance in perspective, we compare our results with results for E/EXODUS [RCS93, C$^+$89]. We selected this system because it has the best overall performance among the database systems evaluated in [CDN94]. E/EXODUS is a page-based client-server system that uses 8 KB pages as the unit of cache management, network transfer and disk transfer. We use results for E/EXODUS taken from [WD94]; these results were obtained using a Sun IPX workstation for the server and a Sun ELC workstation for the client, connected by a 10 MB/s Ethernet. We scaled results for Thor0 to account for the CPU differences (but not for the disk differences although our disks are slower); we show both scaled and unscaled results. We scale up our CPU times by a factor of 3.0 at the client and 2.5 at the server. We obtained these factors by running a C++ implementation of traversal T1 of the small OO7 database on the three machines of interest. The C++ code was compiled using g++ with optimization level 2 in all three machines, and the scaling factors were obtained using the average of 10 program runs.

6.2 The Small OO7 Traversals

Figure 5 shows results for the hot small traversals: traversals where no fetching or cache management is needed, so that their performance depends primarily on the efficiency of the traversal code. As pointed out in [WD94], one of the reasons E/EXODUS does poorly on T1 and T6 is because it

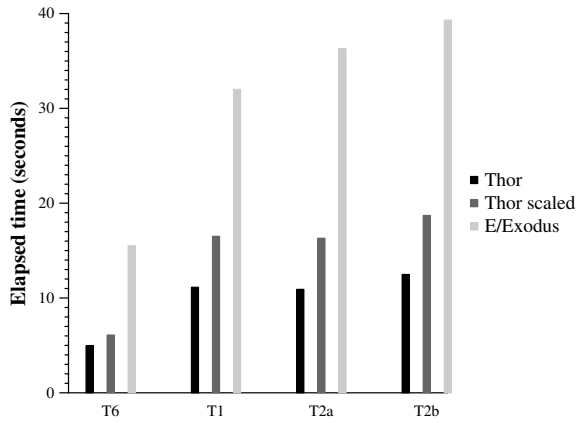

Figure 6: Cold Traversals, Small Database

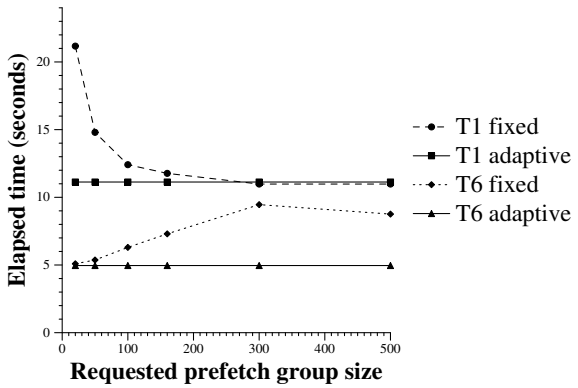

Figure 7: Elapsed Time (in seconds): Fixed Size vs. Adaptive Algorithm

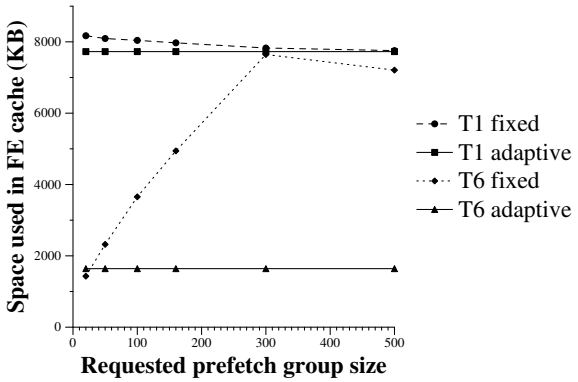

Figure 8: Cache Space: Fixed Size vs. Adaptive Algorithm

spends a significant amount of time executing system code to dereference unswizzled pointers. In contrast, we do not incur significant swizzling overheads in this traversal.

Figure 6 shows the cold small traversals. The difference between our times and those of E/EXODUS is due to our adaptive prefetching, the slow hot times for E/EXODUS as discussed above, our use of large disk reads and the fact that our objects are smaller (our database is 36% smaller than E/EXODUS's). We prefetch 24.5 KB groups for T1 (approximately 485.2 objects per fetch); for T6 the average group size is 25.7 objects (approximately 1434 bytes).

Recall that we achieve stability by keeping the log in primary memory at two servers, each of which has an uninterruptible power supply. To account for the cost of this mechanism (which is not yet implemented), we added 0.1 seconds to the elapsed time for T2a and 0.7 to the elapsed time for T2b. These are the times needed for the FE to communicate with the OR to commit the transaction; they overestimate the time needed for the primary to communicate with the backup, since smaller messages are sent in this case.

To evaluate the effectiveness of adaptive prefetching, we compared our approach to fixed-size prefetching. We ran the cold traversals with a fixed-size prefetch group for several group sizes. Figure 7 shows that the adaptive scheme does better than fixed-size schemes in terms of elapsed time. Figure 8 shows that the adaptive scheme uses much less space than fixed-size schemes for T6, where clustering isn't well matched to the traversal. For T1, where clustering works well, the adaptive scheme is a little better; the reason is that our large prefetch groups result in fewer surrogates.

It is interesting to compare the adaptive scheme with a scheme that always requests 160 objects. For the OO7 database, 160 objects is roughly 8 KB of data, i.e., the amount of data fetched in E/EXODUS. For T6, the adaptive scheme is 32% faster than the fixed-size (160 object) scheme, and uses only 33% of the cache space. The large difference in the amount of cache space used indicates that in traversals where there is cache management the performance difference will increase significantly. For T1, it is 6% faster and uses 97% of the space.

Traversals T1 and T6 have a homogeneous clustering behavior. Therefore, the performances of the adaptive scheme and the best fixed scheme are similar. If the behavior of a traversal changes slowly the adaptive scheme performs significantly better than the best fixed scheme. However, the adaptive scheme may be unable to adapt to fast varying behaviors. We invented a new traversal to evaluate the adaptive prefetching scheme in this worst case. This traversal randomly classifies each composite part as *all* or *root* with uniform probability. It traverses the entire graph of a *all* composite part, and it only visits the root atomic part of a *root* composite part. The results obtained running this traversal show that adaptive prefetching exhibits an averaging behavior. The average prefetch group size is 276.1 which is only 8% greater than the average of the prefetch group sizes observed for T1 and T6. The performance is only 8% worse than the best fixed scheme (100 objects), so the adaptive prefetching scheme performs well even in the worst case.

6.3 Cost of Safe Sharing

As mentioned, our experiments use the code transfer technique. We transfer entire traversals into Thor, and therefore our results are the best possible for safe sharing, allowing us to see what costs are unavoidable.

To measure the cost of safe sharing, we coded the small, hot T1 traversal in C++. This traversal does not incur I/O costs that could hide the safe sharing overheads. To reduce the noise due to misses in the code cache, we used *cord* and *ftoc*, two OSF/1 utilities that reorder procedures in an executable by decreasing *density* (i.e. ratio of cycles spent executing the procedure to its static size). We used cord on both the Theta and C++ executables; as a result the time for Theta is better than the one given in Figure 5.

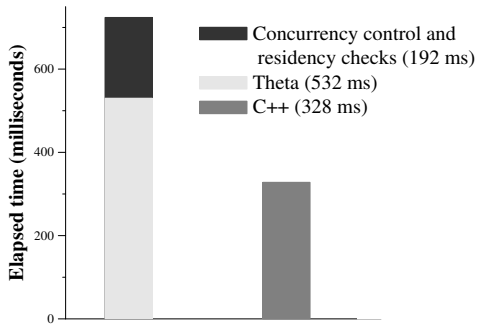

Figure 9: Cost of providing safety in Theta.

The results are presented in Figure 9. The C++ program does not incur any overhead for concurrency control and residency checks, but of course it ought to. Therefore, we break the Theta execution time into overhead that is related to concurrency control and cache residency checks, versus the actual cost of running the traversal in Theta. Table 1 presents a breakdown of the execution time for Theta. The only cost intrinsic to safe sharing is array bounds checking, since it prevents violations of encapsulation; this introduces an overhead of approximately 11%. The remaining Theta cost is for checking exceptions for things like integer overflow; these checks improve safety by preventing commits of erroneous transactions. These results show that safe sharing can be inexpensive. Since our Theta compiler is an experimental prototype, we expect overheads to be reduced as it matures.

Concurrency control	120
Residency checks	72
Exception generation/handling	156
Array bounds checking	36
Traversal time	340
Total	724

Table 1: Elapsed Time (in milliseconds): Hot T1 traversal, small database.

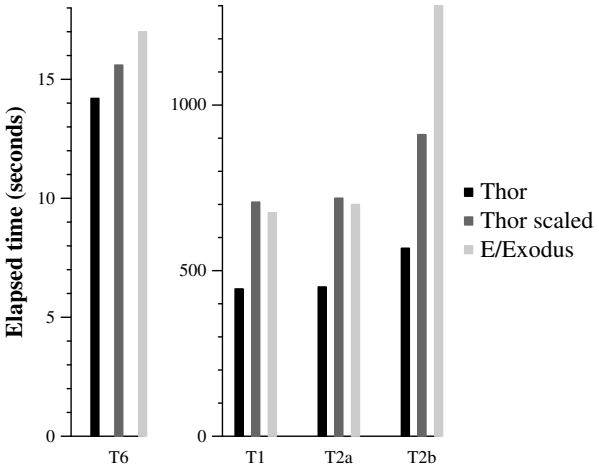

Figure 10: Cold Traversals, Medium Database

6.4 The Medium OO7 Traversals

Figure 10 shows the results for the traversals of the medium database. These traversals run on a cold cache. We perform 8% better than E/EXODUS on T6 mainly because of our efficient adaptive prefetching. This difference is not as significant as for the small database because systems are disk bound on this traversal; Table 2 gives the breakdown for Thor0. Our disks are slow; had we been using the E/EXODUS disks, our disk time would have been around 8.7 seconds, giving us a total of 11.7 seconds (unscaled) or 13.1 seconds (scaled). Furthermore, in the medium database, there is at most one root atomic part per 28.5 KB page, and T6 reads only the root of a composite part graph; thus we frequently read 28.5 KB to obtain one object. E/EXODUS suffers less from this problem because it uses 8 KB disk pages. Of course we are not advocating a switch to 8 KB pages — quite the contrary. With current disks, the time required to read 28.5 KB is very little more than the time required to read 8 KB. Therefore, we do not anticipate that the time needed to read big pages will be a problem for us in the future.

Traversal	0.01
Lazy swizzling	0.08
Commit	0.03
Fetch handling FE	0.3
Fetch communication	2.29
Fetch handling OR	0.36
Disk	11.15
Total	14.22

Table 2: Elapsed Time (in seconds): T6 Traversal, Medium Database.

Thor0 performs 5% and 3% worse than E/EXODUS on

traversals T1 and T2a for the medium database because we currently manage the cache using garbage collection. Table 3 gives the breakdown of the elapsed time in traversal T1 for Thor0 (the breakdown for T2a is very similar). Discarding objects with shrinking plus garbage collection is expensive. If only a small number of objects are shrunk, garbage collection occurs more often, and the amount of live data copied is greater. Therefore we discard very large numbers of objects when we shrink, leading to poor cache utilization:

avg. size of from-space before GC	7.7 MB
avg. size of to-space after GC	1.4 MB
avg. cache space used	4.6 MB
number of collections	54

Thus, we can use only 38% of the cache on average, leading to more misses, and more time in every other part of the breakdown except for commit. Just eliminating the cost of GC and shrinking gives us a scaled time 21% better than E/EXODUS; therefore, with a better cache management scheme, we will perform significantly better.

Traversal	9.2
Lazy swizzling	14.7
Commit	9.4
Shrinking	19.3
GC	39.5
Fetch handling FE	29.4
Fetch communication	252.2
Fetch handling OR	19.1
Disk	52.1
Total	444.9

Table 3: Elapsed Time (in seconds): T1 Traversal, Medium Database.

The results for traversals T2a and T2b include estimates of the time needed for the primary to communicate with the backup of 3.6 seconds for T2a and 29.9 seconds for T2b.

The good performance for T2b is due to other parts of the Thor0 implementation: it costs less to write to the backup as in [LGG+91] than to force the log to disk at commit, plus our disk management at the server is very effective [Ghe95].

7 Conclusions

In this paper, we described the interface and implementation of Thor, a new object-oriented database developed for use in heterogeneous distributed systems. We presented two novel aspects of Thor: its support for type-safe heterogeneous sharing, and its client cache management. Type-safe sharing is important but places several requirements on a database. We described techniques for reducing the performance costs of safe sharing.

Maintaining individual objects in the client cache can improve cache utilization. We presented a new, adaptive prefetching algorithm that fetches only a few objects when clustering is poor and many objects when clustering is good. When clustering is poor, page-based systems waste space or network capacity transferring unused objects.

Finally, we presented performance results for several OO7 benchmark traversals running on Thor0, a prototype that we plan to make available. The results show that adaptive prefetching can be very effective; it outperforms prefetching of fixed-size pages on both well- and poorly-clustered workloads. These results provide a strong argument for decoupling the choice of fetch size from the unit of disk transfer, and for using object-based caching at clients. Our experiments also show that when the code transfer technique is used, the cost of safe sharing is negligible; thus it is possible to have both safety and high performance.

Acknowledgements

We would like to thank the other members of the Thor group and the referees for their helpful comments.

References

[AGLM95] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 23–34, San Jose, CA, May 1995.

[Bak78] Henry Baker. List processing in real time on a serial computer. *CACM*, 21(4):280–294, April 1978.

[BL94] Philip Bogle and Barbara Liskov. Reducing cross-domain call overhead using batched futures. In *OOPSLA '94*, Portland, OR, October 1994.

[BOS91] P. Butterworth, A. Otis, and J. Stein. The GemStone database management system. *CACM*, 34(10), October 1991.

[C+89] M. Carey et al. Storage management for objects in EXODUS. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*. Addison-Wesley, 1989.

[Cat94] Richard G.G. Cattell, editor. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, 1994.

[CDF+94] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring up persistent applications. In *ACM SIGMOD Int. Conf. on Management of Data*, Minneapolis, MN, May 1994.

[CDN93] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 benchmark. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 12–21, Washington, DC, May 1993. WWW users: see URL ftp://ftp.cs.wisc.edu/OO7.

[CDN94] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 benchmark. Technical Report; Revised Version dated 7/21/1994 1140, University of

Wisconsin-Madison, 1994. WWW users: see URL ftp://ftp.cs.wisc.edu/OO7.

[D+90] O Deux et al. The story of O2. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):91–108, March 1990.

[Day95] Mark Day. *Client cache management in a distributed object database*. PhD thesis, Massachusetts Institute of Technology, February 1995.

[DGLM95] Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *OOPSLA '95*, Austin, TX, October 1995.

[DLMM94] Mark Day, Barbara Liskov, Umesh Maheshwari, and Andrew C. Myers. References to remote mobile objects in Thor. *ACM Letters on Programming Languages and Systems*, March 1994.

[Ghe95] S. Ghemawat. *The Modified Object Buffer: a Storage Manamement Technique for Object-Oriented Databases*. PhD thesis, Massachusetts Institute of Technology, 1995.

[HM93] Antony L. Hosking and J. Eliot B. Moss. Protection traps and alteratives for memory management of an object-oriented language. In *14th ACM Symp. on Operating System Principles*, pages 106–119, Asheville, NC, December 1993.

[Hwa94] Deborah J. Hwang. *Function-Based Indexing for Object-Oriented Databases*. PhD thesis, Massachusetts Institute of Technology, February 1994.

[KK94] Alfons Kemper and Donald Kossmann. Dual-buffer strategies in object bases. In *20th Int. Conf. on Very Large Data Bases (VLDB)*, Santiago, Chile, 1994.

[LACZ] B. Liskov, A. Adya, M. Castro, and Q. Zondervan. Type-safe heterogeneous sharing can be fast. Submitted for publication.

[LCD+94] Barbara Liskov, Dorothy Curtis, Mark Day, Sanjay Ghemawhat, Robert Gruber, Paul Johnson, and Andrew C. Myers. Theta reference manual. Programming Methodology Group Memo 88, MIT Lab. for Computer Science, February 1994. Also available at http://www.pmg.lcs.mit.edu/papers/thetaref/.

[LGG+91] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp file system. In *13th ACM Symp. on Operating System Principles*, pages 226–238, Pacific Grove, CA, October 1991.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *CACM*, 34(10), October 1991.

[LSAS77] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction mechanisms in CLU. *CACM*, 20(8):564–576, August 1977.

[ML94] U. Maheshwari and B. Liskov. Fault-tolerant distributed garbage collection in a client-server, object-oriented database. In *Third PDIS Conference*, pages 239–248, Austin, TX, September 1994.

[ML95] Umesh Maheshwari and Barbara Liskov. Collecting cyclic distributed garbage by controlled migration. In *14th ACM Symp. on Principles of Dist. Computing*, pages 57–63, Ottawa, Canada, August 1995.

[Mos92] J. Eliot B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Trans. on Software Engineering*, 18(3), August 1992.

[OMG91] OMG. *The Common Object Request Broker: Architecture and Specification*, December 1991. OMG TC Document Number 91.12.1, Revision 1.1.

[Ont92] Ontos. Inc. Ontos reference manual, 1992.

[OS94] James O'Toole and Liuba Shrira. Opportunistic Log: Efficient Installation Reads in a Reliable Object Server. In *Proceedings of OSDI*, 1994.

[RCS93] J. Richardson, M. Carey, and D. Schuh. The design of the E programming language. *ACM Trans. on Programming Languages and Systems*, 15(3), July 1993.

[SKW92] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: An efficient, portable persistent store. In *5th Int'l Workshop on Persistent Object Systems*, San Miniato, Italy, September 1992.

[SQL92] American National Standards Institute, New York, NY. *Database Language SQL*, ANSI X3.135-1992 edition, 1992.

[Sun95] Sun Microsystems. *The Java Virtual Machine Specification*, release 1.0 beta edition, August 1995. http://ftp.javasoft.com/docs/ javaspec.ps.tar.Z.

[TN92] Manolis M. Tsangaris and Jeffrey F. Naughton. On the performance of object clustering techniques. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 144–153, California, June 1992.

[WD92] S. White and D. DeWitt. A performance study of alternative object faulting and pointer swizzling strategies. In *18th Int. Conf. on Very Large Data Bases (VLDB)*, Vancouver, British Columbia, August 1992.

[WD94] Seth J. White and David J. DeWitt. QuickStore: A high performance mapped object store. In *ACM SIGMOD Int. Conf. on Management of Data*, Minneapolis, MN, May 1994.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *14th ACM Symp. on Operating System Principles*, pages 203–216, Asheville, NC, December 1993.

[Zon95] Quinton Zondervan. Increasing cross-domain call batching using promises and batched control structures. Master's thesis, Massachusetts Institute of Technology, June 1995.

A couple of useful WWW URL's:

OO7 papers and technical reports:
`ftp://ftp.cs.wisc.edu/OO7`

Other Thor papers:
`http://www.pmg.lcs.mit.edu/Thor-papers.html`