References to Remote Mobile Objects in Thor[1]

Mark Day
Barbara Liskov
Umesh Maheshwari
Andrew C. Myers

Massachusetts Institute of Technology
Cambridge, MA 02139
USA

Abstract

Thor is a distributed object-oriented database where objects are stored persistently at highly-available servers called object repositories, or ORs. In a large Thor system, performance tuning and system reconfiguration dictate that objects must be able to migrate among ORs. The paper describes two schemes for object references that support object migration, one using location-independent names and the other, location-dependent names. The paper analyzes the performance of the two schemes and concludes that location-dependent names are the right choice for systems like Thor, where we want fast access to objects that have migrated.

Categories and Subject Descriptors: H.2.4 [Database Management]: Systems: *Distributed systems*; C.2.4 [Computer-Communication networks]: Distributed Systems — *Distributed databases*; D.4.7 [Operating Systems]: Organization and Design — *Distributed systems*; E.2 [Data Storage Representations]: *Linked representations*; H.2.2 [Database Management]: Physical Design;
General Terms: Design
Additional Key Words and Phrases: addressing, distributed object management, distributed object-oriented database, location-dependent reference, location-independent reference, naming, object migration, object mobility, object reference, scalability.

1 Introduction

In distributed systems of the future, we expect to find object-oriented databases whose objects reside at hundreds or thousands of nodes. An object residing at one of the nodes may contain references to objects at other nodes, and may have to migrate from one node to another. A useful system must provide efficient mechanisms for users to access objects of interest despite these complications.

An efficient access mechanism allows remote objects to be located quickly. This paper describes an implementation of object references that meets this criterion. It first describes an efficient technique for accessing objects when objects are not allowed to migrate. Then, it describes two ways of extending the scheme so that mobile objects can be accessed. Using location-dependent names as references, objects can be accessed with no additional time penalty and a modest space penalty; location-independent names consume even less space, but

[1] An earlier version of this paper titled "Naming and Locating Objects in Thor" was presented at the Workshop on Objects in Large Distributed Applications, Vancouver, BC, 17th October 1992. 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 and in part by the National Science Foundation under Grant CCR-8822158.

accessing an object that has moved requires additional communication. The schemes were developed within the context of the Thor object-oriented database system[11], but the results are applicable to other distributed object systems.

We begin in Section 2 with a brief overview of the architecture of Thor. Section 3 discusses the requirements for object references. Section 4 describes the schemes for implementing object references. We conclude with an analysis of the performance of the mechanisms.

2 Overview of Thor

Thor provides a universe of objects that are both persistent and highly available. That is, an object in the universe survives system failures with high probability and is accessible when needed. Each object has an encapsulated state that cannot be directly observed and a set of *methods* that users can call to interact with it. Its state contains data such as integers, booleans, and characters and also references to other objects; we estimate that the average object is on the order of 70 bytes and contains 5 references to other objects (similar to [1, 13, 14]). The size of an object can vary over its lifetime.

Although the Thor universe appears to clients as a single entity, it is distributed across servers called *object repositories*, or ORs, each of which stores some subset of the persistent objects. At any moment, an object resides at a single OR, but it can migrate from one OR to another. To achieve high availability, each OR is replicated at a number of server machines and its objects have copies stored at these servers.

Thor attempts to cluster objects both at ORs and on disk at a single OR. For example, if x refers to y, both objects are likely to be stored at the same OR, and at disk locations that are fairly close together. We assume that good clustering at ORs can be achieved, so references that cross OR boundaries are rare.

Thor provides a persistent root for the object universe. An object becomes persistent when it becomes reachable from the persistent root. If an object becomes unreachable from the root, its storage is reclaimed by a distributed garbage collector[12].

Client programs run at client workstations that are typically distinct from the servers that run ORs. For each client program, there is a *front end*, or FE, running on the client's machine. The FE provides an interface for the client to access Thor. To ensure consistency despite failures and concurrency, client calls take place within an atomic transaction[4]; the client program indicates when the current transaction should attempt to commit.

Client programs never obtain direct pointers to objects; instead, an FE issues *handles*. Handles can be used to identify objects in subsequent clients calls but are local to that client session. When starting, a client can request a handle for the persistent root object; it can then execute methods of this object (which is in fact a directory), obtaining handles for further objects, and so on. Since direct pointers to objects never move outside Thor, it is safe to destroy the storage of objects that are not reachable from the root or from a handle issued by an active FE.

To speed up method execution, the FE keeps copies of objects in an *object cache*. When a client program makes a call, no communication with any ORs will occur if all the objects needed to carry out the call already reside in the client cache. If there is a cache miss, however, the FE must *fetch* the needed object from its OR. When the FE fetches an object, the OR can supply additional objects that may be of use shortly. The FE caches these *prefetched* objects, in addition to the objects recently used by the client.

Objects in the FE cache may contain references to objects that are not in the cache; an

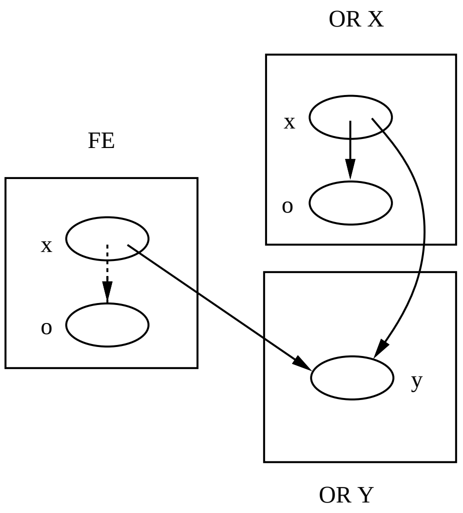

Figure 1: Object References

attempt to follow such a reference triggers a fetch request. For example, in Figure 1 object x (at X) contains references to objects o (at X) and y (at Y). The FE cache contains copies of objects x and o (perhaps o was prefetched when x was fetched); the copy of x at the FE points directly to the local copy of o. However, the FE does not have a copy of y. If an attempt is made to call a method of y (e.g., as part of executing a call on a method of x), the FE must determine which OR stores y and then fetch y from that OR.

3 Requirements

This paper is concerned with the form of references used at ORs and also at FEs to refer to remote objects (like y); we do not discuss the form of references at FEs (such as that from x to o). The form of references determines how quickly fetch requests can be carried out and when prefetching can be done. We believe that prefetching and fast fetches are critical to system performance. Prefetching determines how often method calls hit in the FE cache; we hope this happens most of the time. When a miss occurs, however, it is in the critical path for the user and has a direct effect on the latency of the call. Therefore, even if misses are rare, they heavily affect performance as perceived by users.

We want to provide good performance for clients as they invoke methods on objects. However, our implementation approach must be consistent with a number of constraints:

1. Thor is big. The object universe can be very large (e.g., 2^{40} or more objects), and there can be many servers and clients.

2. The size of the system may change over time. For example, as the load on the system grows, we may add additional ORs to store the persistent objects.

3. Objects may persist for a long time (years).

4. Objects may migrate from one OR to another.

Object mobility is needed in any system intended to be used on a larger scale than a single workgroup and for a long enough time that the system can outgrow its initial configuration. As the system grows or shrinks and as usage patterns shift, we may find that system performance would improve by moving some objects from their current OR to some other OR. For example, an entire group of related objects might be moved either to reduce load at the old OR; or, because the new OR is closer to the client nodes that usually are used to access the objects, an object might be moved to an OR that contains most of the references to it; or, if a system is shrinking so that it has fewer machines available, it may be desirable to eliminate an OR entirely, moving all its objects to other ORs. Individual objects probably do not move often and at any moment the number of objects that have moved "recently" (such as in the last hour) is probably quite small. However, because objects may persist for long times, the number of objects that have moved from their original "birth" OR may be very large. Our design is intended to deal with such a situation.

There are two main issues in designing an object reference scheme for a system with mobile objects. First, some way of locating objects is needed, i.e., determining which OR currently stores an object. Second, once an object has been located at an OR, we must find it in that OR's memory. Our goal is to minimize the costs of both of these activities: We would like almost always to determine the OR where the object resides without any communication, and we would like to determine its memory location at its OR without reading any addressing information from disk.

One possibility is to use virtual memory addresses as references. The idea is to partition the address space among the ORs; each OR stores a portion of the address space and its objects have addresses in this portion. However, the fixed, global addresses mean that ORs do not have control of their local memory layout and thus cannot relocate objects. Object relocation may be needed if objects grow or shrink, or if formerly persistent objects cease to be persistent and are garbage collected.

Therefore, we conclude that we must use names to refer to objects. The next section describes two naming schemes, location-independent names and location-dependent names. Location-independent names do not change even when an object migrates from one OR to another, whereas location-dependent names change when this happens.

4 Forms for Object References

This section describes two forms for object references. We begin by assuming that objects don't move and describe an efficient way of implementing references; our technique is somewhat like that in Mneme[13]. Then we discuss how to handle mobile objects. The two approaches differ in whether an object's name changes when it moves.

4.1 The Basic Scheme

When an object becomes persistent, an OR is selected for it. This *birth site* OR assigns it an entity called an *xref*, which will be used to name it. An xref is a pair containing the OR-id of this OR and an *oref*, which is a name for the object within the OR. This structure makes it easy for an OR to issue xrefs for newly persistent objects; it just selects an oref that is not used locally. In addition, since the xref contains the OR-id, an FE always knows the OR of an object (given our assumption that objects don't move) — so fetching an object requires just one message round trip. An OR can easily determine which references in a fetched object are to local objects. They can then be prefetched and returned to the FE in the same message as the fetched object.

An OR stores objects in *segments*. A segment is stored in a contiguous region of memory and is read and written to disk as a unit[5]. It contains a group of objects that are related to one another, i.e., objects are clustered in segments.

An oref is structured so that an OR can efficiently find the corresponding object. The oref is divided into a *segment id* and an *object number*. A table at the OR maps segment ids to disk addresses. At the beginning of a segment is a header that maps the object numbers of its objects to their disk locations; this header is stored with the segment and read/written when the segment is read/written.

Since we want to read the segment as a unit, it should not be too big; a size of about 64K bytes seems likely to use disk throughput effectively[2]. (The limit on segment size means that storing large objects requires another scheme, which is not discussed in this paper.) If objects average around 70 bytes, a segment can hold about 900 objects.

The segment table is reasonably small. If an oref were 32 bits long and 10 bits were allocated to indexing objects in a segment, the number of segments that could be indexed would be 4M. An OR could hold around 3.6 G objects, and we would need a table size of 64M bytes, assuming each entry holds an 8 byte disk address. This table might well fit in the primary memory of machines of the near future.

To find an object, an OR looks up its segment id in the segment table. Then it reads the segment from disk if necessary; the read may not be necessary because segments are cached in the OR's primary memory. Because of object clustering in segments, we expect that often there will be no disk I/O either to access the segment table entry or the segment.

This scheme has excellent performance but does not allow objects to migrate. Below we show how to extend it to handle mobile objects. The first extension continues to reference the object using the xref assigned to it when it became persistent. The second gives the object a new xref each time it moves, and propagates the information about the new name so that references use the new name instead of the old one.

4.2 Location-Independent Names

Location-independent names, also called *object identifiers* or *oids*, are attractive because they stay the same even when an object moves from one OR to another. Therefore, all references to the object continue to be accurate in spite of the move. However, finding the OR that stores an object becomes more difficult because the reference does not encode its location. Some entity, the *locator*, must record information about the current locations of objects.

The xref assigned to an object at the birth site becomes its oid and continues to be used to refer to it even if it moves. If an object moves, its new OR assigns it a new xref containing the id of the new OR as its first part, and tells the locator about the new xref. The oref in the new xref is selected in the same way that orefs are assigned to newly-created objects: the OR places the object in a local segment close to related objects and assigns it an oref in this segment.

When the FE fetches an object, it determines the current xref of the object as discussed below. Then it sends the fetch request to the OR identified in the current xref; the new xref is included in the fetch request. When the fetch request arrives at the new OR, it can be handled in the usual way by locating the object's segment using the segment table and then reading the segment from disk if necessary.

Some prefetching is still possible with this scheme, since the OR can recognize references to local objects. An object may refer to objects in the same segment as itself; if so, when it is

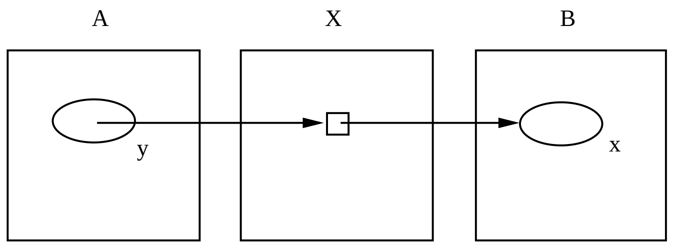

Figure 2: Tracking a Mobile Object using Location-Independent Names.

fetched the OR will be able to prefetch objects to which it refers. But if the object moves to an OR containing objects that refer to *it*, prefetching will not work even if it is stored in the same segment as these objects — the other objects will refer to it using its oid, which appears to be a reference to an object at a different OR.

There are two ways the FE can find out the current xref of an object. The first possibility is that some OR keeps track of where the object is. Since many objects do not move once created, the obvious locator OR is the birth site. To do a fetch, an FE sends a fetch request to the OR listed in the object's oid; the OR either responds with the object (and some other prefetched objects) or it tells the FE the current xref of the object.[2] Because the birth site is a locator, it must store the locations of all objects born there until they are no longer referenced. When an object moves from its birth site, its new location is stored in a small *stub*.

Figure 2 illustrates this scheme. OR A contains an object y, which refers to an object x that has moved from its birth site OR X to OR B. Object y refers to x using its oid; this oid actually refers to a stub at OR X that contains x's new xref.

A problem with this scheme is that two message round trips will be needed to locate an object that has moved; this cost will be incurred for many objects since we expect that a sizable fraction of objects will eventually migrate. In addition, an OR that happens to act as the locator for many heavily-used objects could become a bottleneck. Even if the objects move elsewhere, we cannot move the location workload away because we cannot change which OR is encoded into the objects' oids. Also, the locator must logically exist as long as any of its objects exist; to remove an OR requires assigning its locator role to some other OR.

Another way to locate an object is to make use of a name service that maintains the mapping from oids to xrefs. The name service must be highly-available. Otherwise, an object might be unavailable because of a crash of the name service — even though its current OR is available[3].

A name service requires a lot of replicated storage. Worse, two round trips are needed to fetch an object: one to a name server, and a second to the storing OR. Since an object is likely to be at its birth site, it may be a good idea to check there before consulting the name service. But then three round trips are needed to fetch an object that has moved.

Both the birth-site and name-service schemes can be speeded up by storing a hint in a reference, making it a $\langle$*hint, xref*$\rangle$ pair. The hint contains information about the current location of the object, unless the object moved very recently. The scheme won't work well

[2]A birth-site locator scheme was used in R*[10]. R* sites were not highly available, so the crash of a locator could make an migrated object unavailable even though the object's site was available. This problem does not exist in Thor because ORs are highly available.

[3]Mechanisms for implementing such highly-available services are described in [6] and [9].

unless the hint is usually accurate. This can be accomplished with the tracking mechanism described in Section 4.3.1.

Using the birth site as the locator seems preferable to using the name service because it is simpler and has at least as good performance. Therefore we consider only this scheme in the rest of the paper.

One final point: if an object moves from an OR that isn't its birth site, the OR must mark its former xref as unused — for example, by marking the entry for the object's oref in the segment header. The mark must be retained until the xref is no longer in use anywhere, which can be determined with a tracking mechanism like that in Section 4.3.1.

4.3 Location-Dependent Names

This section describes the alternative to location-independent names: location-*dependent* names.

When an object moves, its new OR places it in a local segment that contains related objects and then gives it a new, local xref. The object's previous storage at the old OR is turned into a *surrogate* that contains its new xref. (Surrogates are like forwarders in Mneme[13] and leaves in LOOM[8].) Thus, instead of a predefined locator OR, the OR that last held an object has up-to-date information about its location.

The information about the new xref for a moved object is propagated (in a manner discussed below) to objects that refer to it. Once propagation is complete, no references to the old location will exist, and the surrogate can be garbage-collected.

When the FE fetches an object, it sends the fetch request to the OR named in the xref. This OR will hold the object unless the object has moved recently. The object can be located within an OR using just the segment table, and prefetching works just as when objects didn't move.

Figure 3 illustrates location-dependent names. In part *i*, an object o has recently moved from OR B to OR A. Object p at OR B and object q at OR C both refer to o. Object p refers to o using its new xref, but q uses the old xref; this old reference gets it to a surrogate at OR B that contains the new xref. In part *ii*, information about the new xref has propagated throughout the system. Now q refers to o using the new xref, and the superfluous surrogate at OR B has been deleted.

Location-dependent names also work well if an object moves to a different segment at the same OR. With location-independent names, both the old and new segments of such a moved object must be read when it is fetched; with location-dependent names, usually only the new segment needs to be read.

4.3.1 Tracking Objects

Location-dependent names depends on timely propagation of xref information for moved objects. This section describes the tracking mechanism.

For distributed garbage collection, each OR maintains an *inlist* for each other OR, listing the local objects referred to by objects at that other OR. This table acts as an additional root during garbage collection, preventing externally referenced objects from being collected even though they may not be reachable locally. (A similar table keeping information about references exists at FEs.)

ORs also maintain a *location table* that stores $\langle$*old xref, new xref*$\rangle$ pairs. Whenever an object moves away from an OR, in addition to turning the object's storage into a surrogate, the OR sends messages to any other ORs that have references to that object, informing them of the

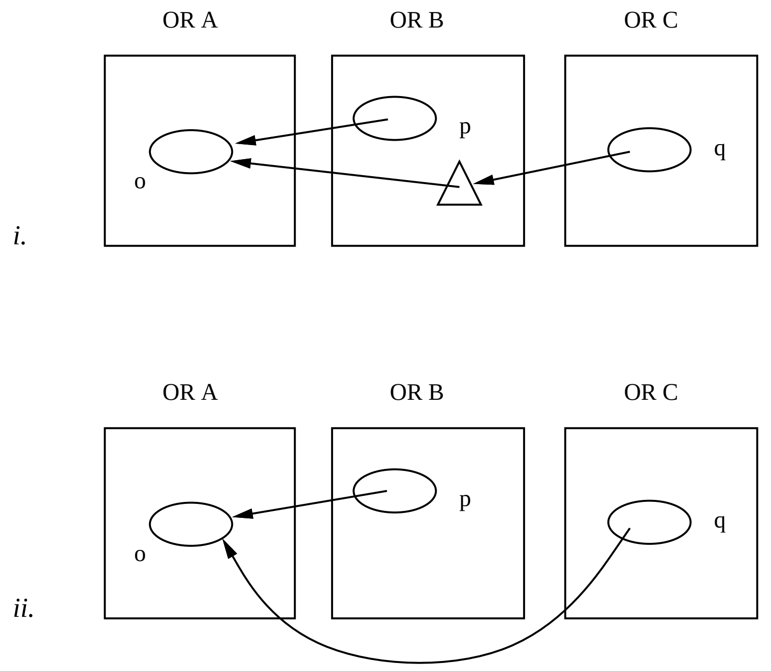

Figure 3: Tracking a Mobile Object using Location-Dependent Names.

new xref. It uses the inlists to determine which ORs to contact. When these ORs learn of a new location for an object to which its objects refer, they add an entry to the location table.

The garbage collector uses the location table to substitute new xrefs. Each xref contained in an object is looked up in the table and replaced by the new xref if a match is found. Information about new xrefs can also be propagated when objects are fetched and prefetched. The speed with which information about the new xref propagates depends on the frequency of garbage collection; we assume that garbage collection happens reasonably often.

Propagation of the new xref means that surrogates and location table entries are eventually unneeded. The garbage collector can recognize this and discard surrogates and table entries.

Our use of forwarding addresses is similar to Fowler's work[3, 7]. Frequently-moved objects may generate chains of surrogates. In Fowler's scheme, clients perform path compression by sending updates to members of a chain of forwarding addresses. In Thor, path compression is performed by the ORs instead.

4.3.2 Xrefs vs. Orefs as references

With location-dependent names we have a choice: we can either use xrefs or orefs as references. Above we described how the system would work using xrefs as references. This is what we plan to do in our implementation. However, using orefs is attractive because orefs are smaller than xrefs. For example, if we allow a maximum of 4G objects per OR, we could get by with 32-bit orefs. By contrast, xrefs are likely to be 64 bits.

We plan to use xrefs as references for three reasons. First, we expect many client machines will be 64-bit machines. If we use orefs as references at ORs, when objects are copied to 64-bit FEs we will need to expand them, replacing the 32-bit orefs with 64-bit virtual memory addresses at the FE. This will require an extra copy of the object, which can be avoided if we use xrefs as references at ORs.

Second, using xrefs as references is simpler than using orefs. If we use orefs as references, objects cannot hold references to objects at other ORs. Instead, we would need surrogates for all cross-OR references. To refer to a remote object, an object would contain the oref of a local surrogate, and the surrogate would contain the xref of the remote object. We would want to prefetch surrogates when we fetch objects; otherwise, when the FE follows such a cross-OR reference, it would have to fetch the surrogate from the surrogate's OR before it could fetch the object. By contrast, with xrefs as references, prefetching of surrogates is unnecessary. A usually-correct xref for the referenced object will always be fetched to the FE.

A third reason for using xrefs is that using orefs imposes an upper bound on the number of objects at a single OR. Limiting an OR in this way may not be so bad; since orefs can be reused, it only constrains the number of objects in existence at a particular moment in time. Furthermore, in Thor it is easy to add another OR if more objects are needed, since Thor hides the existence of multiple ORs from client programs. Nevertheless, if a single application had a data set so large that it led to many cross-OR references, its performance would be poor. With the xref scheme, such an application could be supported by a larger OR; the oref scheme lacks this flexibility.

4.3.3 The Need for Unique Identifiers

With location-dependent names, two different xrefs may refer to the same object. Such a situation is shown in Figure 3*i*. Here object p refers to o using an xref of the form $\langle \mathrm{OR}_A, \alpha \rangle$ while q contains an xref of the form $\langle \mathrm{OR}_B, \beta \rangle$ — since OR C has not yet heard about the move. Despite the difference in form, p and q refer to the same object.

To implement object identity correctly, each object contains a unique identifier field (uid) that is not visible to client code. Determinations of object identity use uids instead of pointers. When an FE fetches an object, it compares its uid with those of other objects already present (using a uid table) and discards the object if it is a duplicate. An identity test is performed as follows. If both operands are present at the FE, we do the obvious thing: the two operands are identical if and only if they have the same virtual memory address. Also, if both operands have the same xref they are identical. However, if the operand xrefs differ, operands not present at the FE must be fetched before the test can be carried out. With location-independent names, this fetch wouldn't be necessary.

5 Discussion

We have discussed how to implement references in a distributed, object-oriented database system. We first presented a base mechanism suitable for any system in which objects are not allowed to move from one server to another. Then we described two ways that mobile objects can be supported by building on top of the base mechanism. One scheme uses location-independent names as references, and the other, location-dependent names.

The base mechanism allows objects to be accessed in one communication and at most two disk reads, assuming the segment table must be paged. However, because of the way objects are clustered in segments, we expect that often no disk reads will be needed. The performance of the schemes for mobile objects is summarized in Figure 4. The two schemes are referred to as IND (for location *independent* names) and DEP (for location *dependent* names); in addition, we analyze scheme IND-hint, which uses location-independent names as references but augments

Scheme	Messages	Disk I/O	Space
IND	1 round trip	1 read	10 bytes/moved object
IND-hint	none	none	40 bytes/object + 10 bytes/moved object
DEP	none	none	8 bytes/object

Figure 4: Additional Cost of Schemes in Space and Time.

them with a hint that contains the current xref of the object (unless the object has moved recently).

The figure shows the additional cost of supporting mobile objects. For example, locating an object that has moved using IND requires a message round trip (to the object's birth site) in addition to the communication to the object's actual OR. It may also require a disk access at the birth site (if the object's original segment is not in primary memory).

The space calculations assume that 64 bits are sufficient to hold xrefs. Scheme IND requires additional space proportional to the number of objects that have moved from their birth site: 10 bytes per moved object, since the stub must contain the new xref plus 2 bytes in the segment header to store the index of the stub. Scheme IND-hint requires 8 additional bytes per reference: with five references per object on average, a cost of 40 bytes per object — for all objects in the system, not just moved objects. Finally, scheme DEP requires 8 bytes per object to store the object uid. We do not count the space used to store surrogates since they are short-lived.

We see in the figure that scheme IND-hint consumes more space than DEP without achieving better performance. (This would be true even if we had shorter hints, e.g., that just identified an object's new OR.) Therefore we eliminate IND-hint from further consideration. The comparison of IND and DEP is simple: DEP has better time performance than IND but consumes more space. Note that the figure underrepresents the time cost of IND, because IND is unable to prefetch an object that has migrated.

We conclude that the choice of scheme depends on how important it is to fetch moved objects quickly and how important prefetching is for them. Good performance for moved objects is an important goal for Thor; we want to allow objects to migrate freely, and we expect that a large proportion of the total objects in the system will move. Therefore, we plan to use location-dependent names as references in our implementation.

6 Acknowledgements

The authors gratefully acknowledge the assistance of Sanjay Ghemawat and the anonymous referees.

References

[1] Carey, M. J., DeWitt, D. J., and Naughton, J. F. The OO7 benchmark. In *Proceedings of the 1993 ACM SIGMOD* (Washington, DC, May 1993), pp. 12–21.

[2] Carson, S., and Setia, S. Optimal write batch size in log-structured file systems. In *Proceedings of the 1992* Usenix *File Systems Workshop* (1992), pp. 79–91.

[3] Fowler, R. J. Decentralized object finding using forwarding addresses. Tech. Rep. 85-12-1, Department of Computer Science, University of Washington, December 1985.

[4] Gray, J., and Reuter, A. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, San Mateo, California, 1993.

[5] Hornick, M. F., and Zdonik, S. B. A shared, segmented memory system for an object-oriented database. *ACM Transactions on Office Information Systems 5*, 1 (January 1987), 70–95.

[6] Hwang, D. J.-H. Constructing a highly-available location service for a distributed environment. Tech. Rep. MIT/LCS/TR-410, MIT Laboratory for Computer Science, January 1988.

[7] Jul, E., Levy, H., Hutchinson, N., and Black, A. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems 6*, 1 (February 1988), 109–133.

[8] Kaehler, T., and Krasner, G. LOOM – Large object-oriented memory for Smalltalk-80 systems. In *Readings in Object-Oriented Database Systems*, S. B. Zdonik and D. Maier, Eds. Morgan Kaufmann, 1990, pp. 298–307.

[9] Ladin, R., Liskov, B., Shrira, L., and Ghemawat, S. Lazy replication: Exploiting the semantics of distributed services. Tech. Rep. MIT/LCS/TR-484, MIT Laboratory for Computer Science, July 1990.

[10] Lindsay, B. Object naming and catalog management for a distributed database manager. In *Proceedings of the 2nd International Conference on Distributed Computing Systems* (Paris, 1981), pp. 31–40.

[11] Liskov, B., Day, M., and Shrira, L. Distributed object management in Thor. In *Distributed Object Management*, M. T. Özsu, U. Dayal, and P. Valduriez, Eds. Morgan Kaufmann, San Mateo, California, 1993.

[12] Maheshwari, U. Distributed garbage collection in a client-server, transactional, persistent object system. Tech. Rep. MIT/LCS/TR-574, Massachusetts Institute of Technology, 1993.

[13] Moss, J. E. B. Design of the Mneme persistent object store. *ACM Transactions on Information Systems 8*, 2 (April 1990), 103–139.

[14] Stamos, J. W. A large object-oriented virtual memory: Grouping strategies, measurements, and performance. Tech. Rep. SCG-82-2, Xerox PARC, May 1982.