Safe Lazy Software Upgrades in Object-Oriented Databases

Barbara Liskov, Chuang-Hue Moh, Steven Richman,
Liuba Shrira, Yin Cheung, Chandrasekhar Boyapati
MIT Laboratory for Computer Science
{liskov,chmoh,richman,liuba,xyz,chandra}@lcs.mit.edu

ABSTRACT

Object-oriented databases allow objects that are manipulated by programs to be stored reliably so that they can be used again later and shared with other programs. Since objects in the OODB may live a long time, there may be a need to *upgrade* them: to change their code and storage representation. This paper describes a technique for upgrading objects in an OODB. The approach preserves the database state by transforming objects to their new classes while retaining their state and their identity. The approach is efficient: we do not interrupt application execution to run an upgrade, but instead run the upgrade incrementally, one transform at a time. Objects are transformed lazily, but just in time; applications never observe non-upgraded objects. Laziness can sometimes lead to problems for the code that transforms objects, however; e.g., a transform might observe broken invariants or interfaces unknown at the time it was written. We define precisely when these problems arise, and we also provide mechanisms for avoiding them. Ours is the first work to provide a full analysis of these problems and to allow safe lazy upgrades even when problems arise. We have implemented our approach on the Thor OODB and we present performance results that show that the overhead of our infrastructure is low.

1. INTRODUCTION

Object-oriented databases (OODBs) allow objects that are manipulated by programs to be stored reliably so that they can be used again later and shared with other programs. The database acts as an extension of an object-oriented programming language such as Java, allowing programs access to long-lived objects in a manner analogous to how they manipulate ordinary objects whose lifetime is determined by that of the program.

The research was supported in part by DARPA Contract F30602-98-1-0237, NSF Grant IIS-98-02066, and NTT.

The objects stored in the database may live a long time and as a result there may be a need to *upgrade* them: to change their code and storage representation. An upgrade can improve an object's implementation, to make it run faster or to correct an error; extend the object's interface, e.g., by providing it with additional methods; or even change the interface in an incompatible way, so that the object no longer behaves as it used to, e.g., by removing one of its methods or redefining what a method does. Incompatible upgrades are probably not common but they can be important in the face of changing application requirements.

This paper describes a mechanism for upgrading objects in an OODB. The mechanism allows object state and identity to be preserved across an upgrade. This preservation is cruicial: The whole point of the database is to store object state. When objects are upgraded, their state must survive, albeit in a modified form as needed in the new implementation. Furthermore, a great deal of object state is captured in the web of object relationships. This information is expressed by having objects refer to other objects. When an object is upgraded it is critical that it retain its identity so that all the objects that referred to it prior to the upgrade still refer to it.

Our approach makes it relatively easy for people to define upgrades. The upgrade mechanism is object-oriented: an upgrade definition describes what to do with each class that is changing, by providing a replacement class and a *transform function* that is used to initialize the new form of the object using the object's current state.

Executing an upgrade requires transforming every object belonging to a class that is being replaced, to change it into a member of the replacement class. The key issue in doing upgrades concerns how to carry this out. Most previous work [21, 2] takes a stop-the-world approach: application access to the OODB stops while the upgrade is performed. Such an approach can make the system unavailable to users for potentially long periods. The unavailability may not be a serious issue if the OODB is small, but if it is large (e.g., trillions of objects residing at thousands of servers), the time during which the system is unavailable to applications can be very long.

We avoid delaying applications by running the upgrade incrementally, one transform at a time. Each object is transformed just before an application accesses it, and therefore applications that run after the upgrade starts never see non-upgraded objects. Thus the work of doing the upgrade is interleaved with application accesses to the stored objects.

This approach is very efficient. It is also correct, i.e., it provides the desired behavior. Our approach to correctness is novel, and is a major contribution of this paper. The approach has three parts:

1. We define what behavior is expected when running an upgrade.
2. We define program properties that can be used to decide whether correctness problems could arise when running the upgrade incrementally.
3. We provide mechanisms that prevent problems from occuring when the properties don't hold.

Our approach is highly effective. The program properties are simple and easy to understand. Furthermore, most programs will satisfy the properties, meaning that no special effort is necessary to achieve correctness. Also, the mechanisms that prevent problems are lightweight.

We have implemented our approach within the Thor object-oriented database and used the implementation to run experiments. Our results show that our infrastructure has low cost, e.g., it has negligible impact on applications that don't use any objects requiring upgrades, which we expect to be the common case, since upgrades are likely to be rare (e.g., no more frequent than once a week or once a day). The results also show that the slowdown when upgrades are needed is small.

There has been quite a bit of earlier work on upgrades in OODBs [4, 18, 15, 23, 22]. This work has either avoided lazy upgrades entirely, or under much broader conditions than are necessary; or the systems allow lazy upgrades even when they are unsafe. We discuss related work in more detail in Section 5.

The paper is organized as follows. Section 2 describes our approach to defining and running upgrades; it describes how we analyze code to determine whether there could be a safety problem, and the mechanisms that can be used to ensure safety when problems arise. Section 3 describes how we implement upgrades in Thor. Section 4 describes our experiments. Section 5 discusses related work. We conclude with a summary and discussion of future work.

2. UPGRADES

We assume the object-oriented database contains conventional objects similar to what one might find in an object-oriented programming language; we will use Java in our examples. Objects can refer to one another and can interact by calling one another's methods. We assume the fields of objects are encapsulated: the only way for one object to access the fields of another is by calling its methods. This constraint does not lead to any loss of expressive power because access to a field can always be provided via two methods, one to read the field and the other to modify it.

The objects in the database belong to classes, which define their representation and methods. Each class implements a type. Types can be arranged in a hierarchy, so that one type can be a subtype of another. A class that implements a type implements all supertypes of that type.

2.1 Defining Upgrades

Our approach to defining upgrades is object-oriented: an upgrader defines the upgrade by describing what should happen to the classes that need to be changed. The information for a class that is changing is captured in a *class-upgrade*. Each class-upgrade is a tuple

```
< old-class, new-class, TF >
```

The meaning of a class-upgrade is that all objects belonging to old-class will be transformed, through use of the *transform function*, TF, into objects of new-class. TF has the signature:

$$\texttt{TF: old-class} \longrightarrow \texttt{new-class}$$

i.e., it takes an old-class object as an argument and returns an object of new-class. At some point after TF returns (e.g., immediately) the upgrade infrastructure causes the new object to take over the identity of the old one, so that all objects that used to refer to the old one now refer to the new one.

When an upgrade is executed, the old-classes of its class upgrades disappear; they are no longer available for use in the system and their objects all "become" objects of the associated new-class. This means that any class that depends on the old-class must be upgraded as well. In particular, when there is an incompatible upgrade, the new class will not implement the same types as the old one. In this case, upgrades will be needed for subclasses of the old-class. Also, classes that use a type implemented by the old class but not by the new one may need upgrades – to use the type implemented by new-class. Such an upgrade can be avoided if the using class only depends on behavior that is not changing, e.g., the class only uses the *size* method, whose behavior is not affected by the upgrade.

An upgrade is a set of one or more class-upgrades. It should contain class-upgrades for all classes that need to change due to some class-upgrade it already contains.

An upgrade that includes all such classes is called a *complete upgrade* [13, 28]. Completeness can be checked using rules analogous to type checking. An upgrade is accepted only if it is complete. At this point we say the upgrade is *installed.*

Once an upgrade has been installed, it is ready to run. An upgrade is executed by running the transform functions on all affected objects, i.e., all objects belonging to the old classes.

For this to work correctly, TFs must be well-behaved:

> A TF is *well-behaved* if it is a pure observer: it does not modify any objects that existed before it started running, including its argument object and any objects reachable from it.

We require well-behaved transforms because the system determines this order, and therefore, any order it chooses must be correct. In particular, running any two transforms must produce the same result regardless of execution order, i.e., their executions must commute. TFs that are well behaved commute since they cannot affect the state of any object used by other transforms.

In addition, the TF must be correct: given any legitimate object of old-class, it must produce the equivalent object of new-class. Here of course we rely on the specifications of the two classes. By "legitimate" we mean any old-class object that satisfies the representation invariant [20] of old-class.

2.2 Running an Upgrade

We assume applications access objects in the database within atomic transactions, since this is necessary to ensure consistency for the stored objects; in addition transactions allow for concurrent access and they mask failures. An application transaction consists of calls on methods of persistent objects as well as local computation. The interface to the system can be very simple, e.g., an application thread can simply run one transaction after another. A transaction terminates by either committing or aborting. If the commit succeeds all changes to database objects become persistent. If instead the transaction aborts, none of its changes is installed in the database.

One could imagine running an upgrade as a single transaction that ran all the transforms in some order, and in fact this has been done in other work [2]. As mentioned, however, this approach has a serious problem: running such a transaction can make the system unavailable to users for potentially long periods.

We avoid delaying application transactions by running the upgrade incrementally and lazily. We run each transform as an individual transaction; these are interleaved with application transactions but are serialized with respect to the application transactions.

Our system runs as follows. When an application transaction, A, is about to use an object that is due to be upgraded, we interrupt A and run the transform at that point. The transform runs in its own transaction T. This transaction must be serialized *before* A (since A uses the transformed object produced by T). If T requires access to an old version of some object modified by A, we provide this access, taking advantage of the fact that A has not yet committed (and therefore the old version still exists). As soon as T finishes executing we commit it. Then we continue running A *unless* T modified some object that A read; this will not happen if TF is well behaved. If we cannot continue running A, we abort and rerun it.

While running T we might encounter an object that is too far in the past: it has a pending transform from an upgrade installed before T's upgrade. In this case, we interrupt T (just as we interrupted A) to run the pending transform.

In spite of running transforms lazily, we want to obtain the same behavior in a lazy system as in an eager one:

> Suppose a system with lazy upgrades has reached the point where all transforms for upgrade m have been performed. At that point some application transactions $A1$, $A2$, ..., Ak that are serialized after upgrade m's installation have also committed. The lazy system is *correct* if the system state at that point is the same as it would have been had we run upgrade m eagerly and then run the application transactions.

There are two problems that can arise in a lazy system that could undermine correctness. The first is that when a TF runs on some object x, it might attempt to use an object y that has already been transformed by a later upgrade; this would be a problem if that later upgrade changed y incompatibly. We could avoid this problem by guaranteeing that upgrade n completes before upgrade $n+1$ starts, but such a guarantee is undesirable since it could cause the same long delay to applications that we are trying to avoid.

The second problem is that when a TF runs on some object x, x's representation invariant might not hold because it concerns some subobject y that has been modified by an application that ran after the TF's upgrade was installed.

None of these problems can occur when objects are well-encapsulated:

> An object is *well-encapsulated* if there are no direct references from code outside it to objects it refers to.

Figure 1 shows an example of an object that is not well-

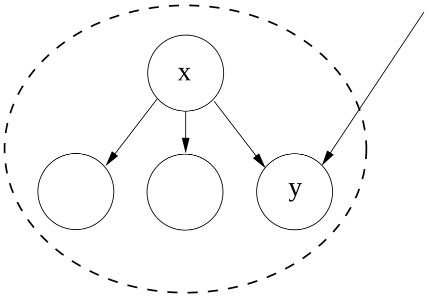

Figure 1: An object that is not well-encapsulated; object y is exposed.

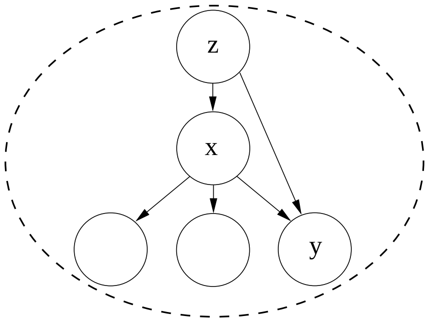

Figure 2: A containing object.

encapsulated. We refer to the subobjects that are accessible from outside of the object as *exposed subobjects.*

When an object x is well-encapsulated, its subobjects can be manipulated only by calling methods of x, which may then call methods of the subobjects. This means that x's methods will be called before methods of the subobjects. Therefore it will be transformed before any subobjects are transformed or used by application transaction.

Therefore problems can arise only when an object has exposed subobjects. But even then there isn't a problem unless in addition that object is being transformed by a TF that uses the exposed objects, i.e., calls their methods, because it doesn't matter whether the exposed object is transformed or modified before the TF runs on the containing object if the TF of the containing object won't observe the change.

Even when these conditions hold, however, there may not be a real problem. In the case where the TF encounters an object that has been transformed as part of a later upgrade, there is a problem only when that transform changed the object in an incompatible way and this affects the TF, i.e., methods it uses have changed their interface or behavior. In the case of an application modification, there is a problem only when the modification violates the representation invariant of the object being transformed.

For example, consider the situation in Figure 1. If x.TF only calls the *size* method on y, and this method has the same interface and meaning in the new class as in the old class, there is no problem even though other things have changed; however if x.TF calls the *insert* method on y, and this method no longer exists, there is a problem. Also, if the representation invariant of x assumes that y is non-empty, and an application transaction has modified y to become empty, there is a problem; however, there wouldn't be a problem if the representation invariant of x made no assumption about the size of y.

So to summarize, a problem really exists when

- An object has exposed subobjects that are used by its TF, and
  - A subobject has been upgraded incompatibly and TF uses methods of that subobject that have disappeared or changed their behavior, or
  - A TF's object's representation invariant no longer holds.

2.2.1 The Solution

We assume that a potential problem can be detected by static analysis, e.g., [26, 11, 5, 6, 24, 12]. The analysis might be overly conservative: it might detect a problem when there isn't one, e.g., by not being able to figure out that the object used by the TF isn't the exposed one. But we can rely on the analysis to bring the potential problem to the attention of the upgrader, who can then decide whether it is really a problem, and if so, how to avoid it.

If there is really a problem, there are two ways to fix it. The first (preferred) approach can be used when the encapsulation problem is "contained" as illustrated in Figure 2. This figure shows a well-encapsulated object z that contains both x and its exposed subobject y. In such a case we can take advantage of the containing object to control the transform order explicitly: the upgrader adds a *trigger* to z's class, C_z, indicating that when a C_z object is used, x should be transformed before the C_z method starts to run. Containment might be more complex than what is shown in the figure. For example, C_z objects might share C_x objects in some arbitrary pattern so that no single C_z object contains both x and y; however if as a group the C_z objects contain all C_x objects and their exposed objects, we can still use this approach.

If there is no containing object (or objects), we fall back on the second method of providing versions for C_y objects: we make a snapshot of the object when it is transformed, or the first time it is modified after an upgrade is installed.

When there is an incompatible upgrade for C_y, the class of the exposed objects, the class upgrades for C_x and C_y might be in the same upgrade; in this case, the user

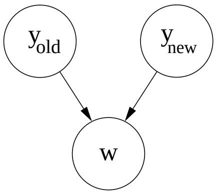

Figure 3: Shared subobjects.

decides as part of defining that upgrade whether there is a problem and if there is, how to fix it (via a trigger or a version). Or, C_x can be in an earlier upgrade than C_y. In this case, there is no problem when the upgrade for C_x is defined, but there might be one later, if an incompatible upgrade for C_y is defined in a later upgrade. Therefore we carry the information forward and bring it to the attention of the upgrader should an incompatible upgrade for C_y be defined. At that point the upgrader decides what to do. In either of these two cases, if the upgrader decides to handle the problem (using either a trigger or a version), we no longer have to consider C_x's TF in future upgrades, e.g., if D_y is upgraded to incompatible E_y. Otherwise, we continue to carry the information forward to future upgrades.

When there is a violated representation invariant, the situation is similar. For example, consider the situation in Figure 2. Suppose that z ensures some property of y that x depends on, e.g., y is a set, but x assumes y is non-empty. Now suppose C_x is upgraded to no longer make this assumption, and C_z is upgraded to not support it; in this case C_x and C_z are in the same upgrade. Or, C_x might be changed in upgrade n and C_z changed later in upgrade m; in this case we note the problem when upgrade n is defined, but only do something about it later, when upgrade m is defined.

2.2.2 *Nested Versions*

Some care is needed in making versions. For example consider the situation illustrated in Figure 3. Here the old and new versions of y share w.

Sharing between versions is a problem if methods of y_{new} modify the shared object in a way that causes the representation invariant for y_{old} to no longer hold. E.g., perhaps y_{old} stores the size of w and its implementation depends on the size being accurate. If a method on y_{new} adds an element to w, the invariant of y_{old} will no longer hold. Now suppose a TF from an early upgrade runs and calls a method on y; this call goes to y_{old}, and the method call fails because the representation invariant doesn't hold.

To avoid this problem, we make a version for the subobject (and its mutable subobjects) as part of making the version for the containing object. This is done only if the upgrader tells us to, i.e., there really is a problem.

2.2.3 *Correctness*

Our system provides correct executions in the presence of lazy transforms provided the user has used triggers and versions as needed to avoid problems. In this case, TFs will only transform objects whose representation invariant holds, and they will never encounter objects with incompatible interfaces. These conditions mean the TFs will always be able to transform their object properly – assuming they are correct.

Of course, in a lazy system a TF can encounter an object that has already been modified by an application transaction A. But this is acceptable assuming the TF itself is implemented correctly. A correct TF produces the object of new-class that corresponds to its input. It doesn't matter what the state of its input object is when it does this, i.e., the effect will be the same regardless of whether we run the TF followed by A, or we run A followed by the TF.

Correctness depends on the upgrader making the right decisions once informed about a potential problem. We believe it is appropriate to rely on the upgrader to do this work; after all, we rely on the upgrader to define the transform function and the new class correctly! Note also that the upgrader can always fall back on requiring versions; this will always be correct, albeit at the cost of reduced performance.

2.3 Transform Functions and Triggers

In this section we briefly discuss transform functions and triggers.

Transform functions can be thought of as constructors of the new class:

```
C_new TF (C_old x) { ... }
```

The one thing that is different about them is that they may need to do type-incorrect assignments. For example, consider the situation in Figure 1, and suppose that after the upgrade the new version of x is supposed to refer to the new version of y. Then in the transform function for x, we would like to assign a pointer to y to a field of the new object. However, the type of that field is Dy, whereas the object obtained from x is of type C_y. Our approach is to allow such assignments only in a special part of the TF definition; a pre-processor can check whether the assignments are type-correct given the upgrade. For example, we might have:

```
Dx TF (Cx x) { ... } [els: x.els]
```

Here *els* has type Dy and *x.els* has type C_y; the assignment is type correct given the upgrade because the object referred to by *els* will become a Dy object before it is used.

It may seem that there is a better way to handle this problem: convert *x.els* immediately. But this can lead to a problem when there is a cycle [16], e.g., if y points to x. In this case if we try to transform y before the transform of x terminates, we may encounter a similar assignment in y's transform. At this point we would be stuck: we can't finish y's transform until x's tranform completes, and so on.

Triggers are functions that produce objects needing upgrades. For example, consider the situation in Figure 2 and suppose we want to attach a trigger to z. This can be written as:

```
Cx trigger(Cz z) { return z.myX; }
```

Note that the trigger is written in terms of the current class of z. Effectively it is an extra method of C_z defined as part of the upgrade. (A trigger can return an array if it needs to return more than one object.)

When an object with an attached trigger is encountered, we run the trigger immediately and keep track of its result. Then we transform the object if necessary. The trigger and the transform run as a single transaction.

After this transaction commits we transform each object identified by the trigger, but only if the pending transform is "current" with respect to the trigger: the pending transform comes from an upgrade no later than the upgrade that caused the running of the trigger. The objects are transformed in trigger order (i.e., in the order they are placed in the array).

Triggers are also supposed to be well-behaved: they should be pure observers.

3. IMPLEMENTATION

This section describes how we implement upgrades within the Thor object-oriented database. More information about Thor and its implementation can be found in [19, 10, 1, 7].

Thor is a client-server system. Persistent objects reside at servers; each persistent object resides at a particular server. Application transactions run at client machines on cached copies of the persistent objects.

Thor uses optimistic concurrency control [1]. Client machines fetch objects into the cache as needed. They track all objects used and modified by a transaction. When the transaction attempts to commit, the client machine sends a commit request containing information about used objects and states of new and modified objects to one of the servers. The server decides whether that transaction can commit (using two-phase commit if the transaction used objects at more than one server) and informs the client machine of its decision.

3.1 Basic Strategy

Our base approach is to interrupt application transactions and transform transactions when we encounter objects due to be upgraded and also when we encounter objects with triggers attached.

1. Each time an application transaction, AT, or a transform transaction, TT, uses an object, we check whether that object needs to be transformed or has an attached trigger. If so, we interrupt AT or TT and start a transaction to run the code on that object.
2. We run transaction T. Note that T must be serialized before all the interrupted transactions. Therefore as it runs we check whether it is using objects already modified by an interrupted transaction; any such object is reverted to its prior state.
3. When T is ready to commit, we check whether it modified any objects already used by interrupted transactions. If there are such object, we abort all the interrupted transactions including AT. Then we commit T and any other TTs that have already completed. Then we run the AT over again. Note that we will never need to abort if the transform (and triggers) are well-behaved.
4. If T is a transform we create a version for it if that is indicated.
5. If T has triggered some other transforms we run them provided they are defined by upgrades no later than the upgrade that causes T to run. Note that T is finished executing at this point, i.e., we don't interrupt it to run these additional transforms.
6. When there are no triggered transforms left to run, we continue running the AT or TT that was interrupted to run T.
7. When the AT is ready to commit, we commit it along with all the TTs that ran on account of it. If this fails, we commit as many of the TTs as possible; we abort the rest and then rerun the AT.

The key to making all of this run efficiently is implementing step 1 properly, since this test is required on every method call. Our approach is described below; it is based on the assumption that most objects used by application transactions are not due to be upgraded and do not have attached triggers.

3.2 Some Details

Here we describe a few details of our implementation; more information is contained in [27] and the appendix.

Objects in Thor refer to one another using *orefs* [10]. These are references particular to one of the servers: they identify a page at that server and an object number within that page.

Since these references would be expensive to use when running transactions, Thor client machines *swizzle* pointers when they are first used, so that they can be followed efficiently to locate the object being referred to in the client cache. Swizzling is done using an indirection table called the ROT (resident object table). A swizzled pointer points to an entry in the ROT. That entry either points to the object in the client cache, or it is empty.

We get an efficient test for whether an object is due to be upgraded or has an attached trigger by maintaining the following invariant while an application transaction is running:

> While an application transaction is running, all non-empty entries in the ROT are for up-to-date objects without attached triggers.

This invariant means that while we are running the application transaction, we discover upgrades and triggers when we fill ROT entries. ROT entries are filled less often than they are used; therefore we avoid the need to test for upgrades and triggers in the normal case of running method calls on objects that are entered in the ROT.

The invariant is not useful while running transforms, however, because objects in the ROT may be too recent for the transform (i.e., already transformed due to a later upgrade) and if the object is versioned the transform needs to find the appropriate earlier version to use. Therefore, we need to test on every method call whether we are running an application transaction or a transform transaction. This test is very fast: it involves looking at a boolean variable that, because it is used so frequently, ends up in a register or the fastest hardware cache.

When a client machine learns of an upgrade, it clears all ROT entries for objects of old-classes of the upgrade and also aborts its currently running transaction if it used any objects of these classes. This ensures that the ROT invariant holds. Processing a new upgrade at a client machine is relatively expensive. However, we believe that upgrades are not installed very often, e.g., no more than once a week or once a day. Therefore it is not worthwhile to optimize this processing.

3.2.1 Versions

When a transform transaction commits, the system determines whether a version is needed. If not, it stores the new object in the storage of the old one if there is room. The new object can actually be larger than the old one; all that is required is room in the object's page, because orefs are logical, not physical, and because the ROT allows the client machine to move objects around in the cache. If the new version is too big, the original object is changed to a surrogate that points to the new object; the structure is illustrated in Figure 4a. If a version is needed, the original object is changed to a proxy object that points to both the old and new versions, as illustrated in Figure 4b; the new version is placed in the object's page if possible.

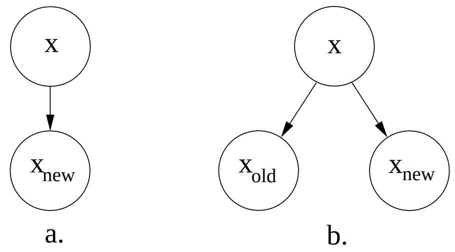

Figure 4: Surrogates and proxy objects.

3.2.2 Committing

When an application transaction commits, we need to send the servers information about all objects that were transformed during its processing. In the case where no version was needed and the new object fit in the page, we just send the new state of the transformed object. If a surrogate is needed, we send two objects (the surrogate and the new object); if a proxy object is needed, we send a special proxy-record containing the orefs of the old and new versions, and the new version. (Only the versions are sent and not their containing pages because Thor uses object-shipping [10]).

4. PERFORMANCE

To evaluate the performance of our techniques we extended Thor to support upgrades. The main goal of our performance study is to evaluate the impact of upgrade system infrastructure on application performance.

Before presenting our results we describe our experimental setup. We use two systems in our experiments: *ThorOld* is the Thor system without support for upgrades; *ThorUp* is Thor extended to support lazy upgrades. The prototype includes the components of the system required to evaluate the upgrade system's impact on the performance of an application running on a client but does not include upgrade installation at the server and subsequent notification to the client. Instead, we configure the client with a pre-canned sequence of "dormant" upgrades and activate upgrades while running applications.

Our application workloads are based on the single-user OO7 Benchmark [9]; this benchmark is intended to capture the characteristics of many different CAD applications, but does not model any specific application. We use OO7 because it is a standard benchmark for measuring object storage system performance. 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 bidirectional *connection* objects, reachable from a single *root* atomic part; each atomic part has three connections. We use the small OO7 Benchmark database configuration, where each composite part contains 20 atomic parts.

We consider both read-only and read-write transaction workloads in our analysis, since upgrades have a different commit cost in workloads with and without modifications. We use the read-only T1 dense traversal, which performs a depth-first traversal of the entire composite part graph (touching every atomic part); we run read-write traversals T2a, T2b, and T2c, which perform a T1 traversal, with T2b modifying all atomic parts, T2a modifying only root atomic parts and T2c modifying each atomic part four times.

We run a single client and a single server, running on the same machine. The test machine has a 600MHz Intel Pentium III processor and 512MB of memory, and runs Linux 2.2.16.

Our experiments focus on showing the impact of the upgrade infrastructure. Section 4.1 shows the overhead of the infrastructure on application transactions that do not require upgrades. Section 4.2 shows the infrastructure cost when transforms must be performed.

4.1 Baseline Experiments

In this section we consider the performance cost imposed by the upgrade infrastructure when the system does not encounter any objects that need to be upgraded. Since we assume that upgrades are rare, our design tries to optimize this baseline case.

Our baseline experiments evaluate two types of application accesses: fast access to an object already installed in the ROT and the slower access to an object that needs to be installed in the ROT. Our upgrade infrastructure introduces an extra cost, *PenaltyResident*, for an access of an object resident in the ROT, which includes a check of the global flag that indicates whether the current transaction is an application or transform transaction. For an access of an object that isn't resident in the ROT, our upgrade code introduces an extra cost, *PenaltyNonresident*, which includes the expense of checking if a trigger or transform needs to be run for the object.

To evaluate *PenaltyResident*, we compare application execution times for ThorOld and ThorUp with a fully-populated ROT. To evaluate *PenaltyNonresident*, we repeat the experiment, but with an initially empty ROT.

Figure 5 shows the execution times for these experiments. All experiments use a hot cache, since otherwise the cost of fetching objects into the cache would dominate execution time.

The full ROT comparison is the expected case for most application executions. Conversely, the empty ROT comparison represents a worst-case for baseline performance in ThorUp: the maximum amount of work must be performed for each nonresident object access. This is a case that we would not expect to occur normally, as typically only a few empty ROT entrees are encountered when a transaction runs. Both in empty ROT execution, and execution with ROT installation, our upgrade infrastructure introduces an overhead below 1%.

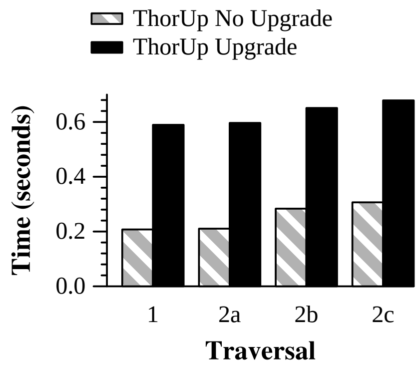

Figure 6: Upgrade traversal cost. 9,880 objects are transformed.

4.2 Upgrade Experiments

Next we measure the cost of executing upgrades and running transforms. We install a schema upgrade and run a database traversal that encounters old objects that need to be transformed.

The schema change we use in these experiments upgrades the atomic part class. Because our goal is to measure the basic upgrade processing overhead in our infrastructure, we use a null upgrade that minimizes application-specific costs (like running transforms defined by programmers). The upgraded atomic part class has the same methods and fields as the old class, and the transform function just copies the fields from the old object to the new. As before, all our experiments use a hot cache.

Figure 6 compares the traversal execution times of ThorUp without any upgrades and ThorUp with an atomic part upgrade installed just before the traversal. The extra cost in each upgrade experiment is proportional to the number of distinct atomic parts visited by the traversal. All traversals visit each atomic part multiple times; the version check and transform cost is only incurred on the first visit. Note that these experiments are an atypically bad case for ThorUp, since a single application transaction visits all objects that need to be upgraded. In the expected case, each application transaction would encounter few or no objects that need to be upgraded, so the total upgrade cost presented here would be amortized over many application transactions.

By counting the number of objects transformed in these traversals, we are able to compute the average cost of running a transform; this is 38.7 microseconds. Note that this is an overly conservative estimate, because it includes the cost of running the TF as part of the infrastructure overhead.

The data in Figure 6 does not include the cost of committing the transaction. When a transaction that caused transforms commits there is an additional cost proportional to the number of objects that were transformed but not modified by that transaction, since each such

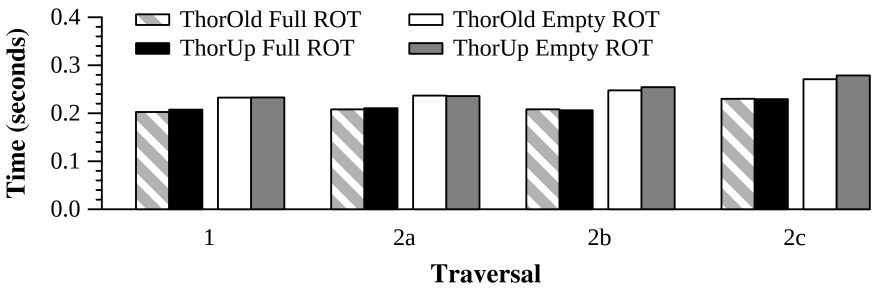

Figure 5: Baseline (no-upgrade) traversal times.

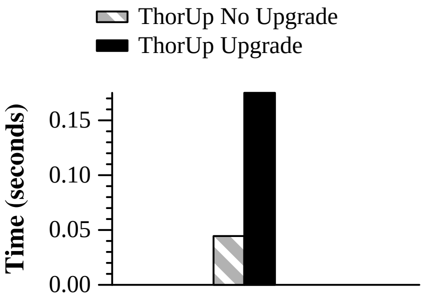

Figure 7: Upgrade commit costs for Traversal 1. 9,880 objects are transformed.

transformed object must be sent to the server. Figure 7 compares the commit times for T1 in ThorUp with and without upgrades. The per object local commit cost for an upgraded object is 13.1 microseconds. In the upgrade case, each of the 9,880 transformed atomic parts must be sent to the server.

If an application transaction that triggers a transform subsequently modifies that object, there is no additional commit cost (assuming the new object fits and does not require a version), since the application's modifications would have to be shipped to the server anyway. T2b, for instance, has identical commit costs with and without upgrades.

The atomic part upgrade in our experiments doesn't require versioning, and a transformed object is the same size as its old object (and can therefore overwrite the old object). If we needed a version or if a new object didn't fit on the page of its old object, we would have to commit two objects instead of one, roughly doubling the commit cost per such object.

5. RELATED WORK

The two different approaches to modifying the conceptual structure of an object database, broadly categorized, following [14], are schema evolution and schema (or class) versioning. In the evolution approach, the database has one logical schema to which modifications of class definitions are applied. Instances are converted (eagerly or lazily, but once and forever) to conform to the latest schema. In the schema or class versioning approach, multiple versions of a schema or class can coexist. Instances can be represented as if they belong to a specific version of their class, but how this is done (e.g. by creating a separate image of instance or by keeping one version-specific copy and dynamically converting it as needed) depends on the concrete system.

Our discussion focusses on the schema evolution approach because it is most relevant to our work; a problem with the versioning approach is the huge amount of storage it requires. The scheme evolution approach is used in the commercial systems O2 [15, 3], GemStone [23, 8], Objectivity/DB [22], and Versant [25], and in the research systems Orion [4], OTGen [18] and PJama [2, 14], and is the only approach available in commercial RDBMS. Very few of these systems support general transforms and lazy conversion: Gemstone and Orion do not support user-defined or complex transform functions; ObjectStore supports a limited form of eager conversion and no lazy conversion; Versant supports lazy conversion for default transforms but user-defined and complex transforms require eager conversion; PJama supports eager conversion for user-defined and complex transforms but has no support for lazy conversion. (A *complex* transform is one that calls methods of subobjects.)

The OTgen [18] system supports lazy conversion for a restricted form of complex transform that copies data and object references from the old object version to the new one but does not run arbitrary methods; this avoids the reordering problem caused by deferred execution of general complex transforms.

The Objectivity/DB [22] system supports complex transforms and provides lazy, eager, and on-demand conversion that can trigger previously defined lazy conversions to happen eagerly on selected subsets of evolving objects. Lazy conversion with complex transforms is not supported or is at least discouraged. The supported lazy conversion is restricted to setting only primitive object fields in transforms; this avoids the problem discussed in Section 2.2.2 of versions sharing mutable subobjects.

The O2 [15] system supports lazy conversion and complex transforms. This work introduces the upgrade correctness condition based on the equivalence of lazy and eager conversion, and is the first to identify problems posed by deferred complex transforms and incompatible upgrades [16]. Like our system, it requires well-behaved transforms. O2 insures type safety for deferred complex transforms using a "screening" approach similar to versioning. It retains all versions of transformed objects and performs method dependability analysis to avoid retaining fields inaccessible to deferred transforms. Unlike our approach, however, O2 analysis [16] does not take encapsulation into account. Whenever an incompatible upgrade occurs after a complex transform is installed, it either activates stop-the-world conversion, or runs with all versions. This approach is unnecesarily conservative (e.g., the switch to eager transforms occurs even when the complex transform doesn't use the class with the incompatible upgrade). Also, O2 does not deal with the problem of deferred access to objects modified by applications, which is unsafe as discussed in Section 2.2.

Detailed information about implementations for commercial systems supporting lazy conversion with complex transforms is generally not available. We found limited information for O2 [15], e.g., we found no information about the mechanisms supporting the atomicity of individual transforms, or about the performance impact of upgrade support on normal case operation. The O2 screening approach co-locates the versions of upgraded objects physically near the new version of the object [15]. This requires database reorganization when versions are created. In contrast, our system does not require co-location of object versions; this allows us to preserve clustering of non-upgraded objects without database reorganization and furthermore, we are often able to preserve clustering for upgraded objects as well. Preserving clustering is very important for system performance because of its impact on disk access [17].

Some of the implementation issues caused by complex user-defined transforms that need to be addressed by lazy conversion system arise also in the implementation of eager conversion. E.g., an eager system has to support arbitrary order of transforms and access to potentially incompatible transformed objects. The Pjama eager conversion system [2, 14] keeps old and new versions to solve this problem. To support large databases, it performs incremental partitioned conversion that creates partitions with old and new versions, and at the end of conversion deletes the old copies by copying the converted partitions. It uses write-ahead logging to to support atomicity and recoverability.

6. CONCLUSIONS

This paper has described a technique for upgrading objects in an OODB. The approach preserves the database state by transforming objects to their new classes while retaining their state and their identity. The approach is also efficient: we do not interrupt application transactions to run an upgrade, but instead run the upgrade incrementally, one transform at a time. Objects are transformed lazily, but just in time; application transactions never observe non-upgraded objects. Laziness can sometimes lead to problems, e.g., a transform function might observe broken invariants or interfaces unknown at the time it was written. We define precisely when these problems arise, and we also provide mechanisms for avoiding them (triggers and versions). Ours is the first work to provide a full analysis of these problems and to allow safe lazy transforms even when problems arise.

We have implemented our approach on the Thor OODB and we present performance results that show that the overhead of our infrastructure is low. These results are especially interesting because the Thor implementation was developed without considering upgrades.

We rely on a combination of static analysis and help from the upgrader; we notify the upgrader of potential problems and expect him or her to do the semantic analysis to determine whether there really is a problem and if so how to solve it. We believe it is appropriate to rely on the upgrader to do this work; after all, we rely on the upgrader to define the transform function and the new class correctly!

There are several interesting directions for future work. One is garbage collecting old versions; we want to get rid of them once we are sure they will never again be needed to run a transform. Another is doing upgrades eagerly, e.g., upgrading all objects on a page while garbage collecting the page, without violating the ordering imposed by triggers.

7. APPENDIX

This appendix contains more details about how the implementation works.

7.1 Installing Upgrades

An upgrade consists of a set of class-upgrades and a set of class-triggers. Class-upgrades have the following form:

```
< old-class, new-class, TF,
  version-flag, trigger >
```

Here the version-flag is on if versions are required for this upgrade, and the trigger is a (possibly empty) pointer to the trigger function that should run when an object of old-class is encountered. Class-triggers are used when a class has an attached trigger but is not being upgraded; they have the form:

```
< old-class, trigger >
```

There is no more than one entry in the upgrade for any old-class.

When an upgrade is completely defined (and is complete!) its description is entered at the upgrade server; this is one of the Thor servers that is designated to do this work. At this point the upgrade is serialized with respect to any other upgrade installations, and it is assigned an upgrade number that is one greater than that of the previous most recent upgrade.

Objects in Thor point to their class object. Upgrades are processed by modifying class objects of the old-classes, and creating class objects for the new-classes. A class object contains a pointer to the dispatch vector, a unique name for the class, and an upgrade number, plus some additional fields as discussed below.

The server processes the installation as follows. In the case of an class-upgrade, it creates a class-object for new-class; this class-object contains a pointer to the dispatch vector for the class, the unique name for the class, and the upgrade number assigned to this upgrade. The server also modifies the class-object of old-class: the class-object points to the class-object for new-class, it points to the TF, it contains the version flag, it points to the trigger function, and it contains a work-needed flag, which is set to on.

Thus our class objects contain some extra fields, to store the additional information. But since they are shared by many objects, this doesn't impose much storage overhead.

In the case of a class-trigger, the server also creates a new class object; this object has the upgrade number of the current upgrade, but the same class id and dispatch vector as the old class. Then it modifies the class-object of the old-class: the class-objects points to the new class and to the trigger function, and its work-needed flag is on; its other fields are null.

The server maintains an upgrade number U, which is always set to the number of the highest installed upgrade. When the processing of the upgrade definition is complete, the server sets U to the new upgrade number. U is sent on every communication to client machines (and to other servers); in this way the client machines learn about new upgrades.

Class definitions can be added to the system between upgrades; these class objects are assigned the current upgrade number, U.

7.2 Running Transactions

While an application transaction is running, the transform flag will always be off. However, before filling a null ROT entry, the client machine will check the class-object of the object. If the work-needed field of the class-object is on, this means a transform or trigger must be performed. Note that this test is also lightweight (especially compared to other work that goes on at this point, including sometimes having to fetch a page into the cache).

The client machine maintains information about the currently running transaction in the form of a ROS (the set of objects used by the transaction), MOS (the set of objects modified by the transaction), UNDO (an undo-log storing the pre-states of all modified objects), UNUM (the number of the upgrade currently in force – this will be the one greater than the highest numbered upgrade in the case of an application transaction), and TRIG (the list of objects identified by running the trigger).

When an object's class has the work-needed flag on, the current transaction will be interrupted to run it if the UNUM in its new-class is less than that of the current UNUM. This test will always hold when the application transaction is running, but may fail when a transform transaction is running because the pending transform is in the future relative to the transform transaction running at this point.

When the transaction is interrupted, its ROS, MOS, UNDO, and TRIG are saved, and a new transaction starts with an empty ROS, MOS, UNDO, and TRIG. Its UNUM is the number stored in the object's class. Also the transform bit is turned on.

Now the transform transaction T runs. First the trigger function (if any) is executed and its result is stored in TRIG. Then the TF runs.

Because the transform bit is set, entries in the ROT are examined before they are used and if an entry points to a proxy object the code will follow the pointers to find the right version. It is able to figure out what the right version is because of the version numbers in the class objects, together with the UNUM: the version whose class object has the highest number that is less than UNUM is the correct one. While T is executing, the system checks whether objects it uses have been modified by interrupted transactions, and if so it uses the UNDO list of the transaction that modified the object to revert the object to its previous state. When T completes, the system compares its MOS with the ROS's of interrupted transactions to see whether these need to be aborted because of a conflict.

If the current transaction can commit, and if it contained a transform, the system stores the new version as discussed in Section 3.2.1, and it sends the new information to the servers when it commits the application transaction as discussed in Section 3.2.2. Objects that ran triggers and that weren't transformed are also modified, to contain a pointer to their new class, and they are also sent to the servers in the commit request.

After T commits we run upgrades for the objects in its TRIG; if TRIG is an array, we process the objects in the array order. An object x in TRIG is upgraded if it has a pending upgrade and the UNUM of its new class is less than or equal to the current UNUM.

When all this work is over, we restore the ROS, MOS, UNDO, TRIG, and UNUM for the interrupted transaction and continue running it.

8. REFERENCES

[1] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient Optimistic Concurrency Control using Loosely Synchronized Clocks. In *Proceedings of ACM SIGMOD*, 1995.

[2] M. P. Atkinson, M. Dmitriev, C. Hamilton, and T. Printezis. Scalable and Recoverable Implementation of Object Evolution for the PJama 1 Platform. In *Proceedings of the 9th International Workshop on Persistent Object System*, 2000.

[3] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object-Oriented Database System - The Story of O2*. Morgan Kaufmann Publishers, 1992.

[4] J. Banerjee, H. Chou, H. Kim, and H. Korth. Semantics and implementation of schema evolution in object-oriented databases, 1987.

[5] B. Blanchet. Escape Analysis for Object-Oriented Languages. Application to Java. In *Proceedings of the 14th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1999.

[6] J. Bogda and U. Holzle. Removing Unnecessary Synchronization in Java. In *Proceedings of the 14th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1999.

[7] C. Boyapati. JPS: A Distributed Persistent Java System. Master's thesis, Massachusetts Institute of Technology, Cambridge, September 1998.

[8] R. Bretl, D. Maier, A. Otis, D. J. Penney, B. Schuchardt, J. Stein, E. H. Williams, and M. Williams. The GemStone Data Management System. In *Object-Oriented Concepts, Databases, and Applications*, pages 283–308. ACM Press and Addison-Wesley, 1989.

[9] M. J. Carey, D.J. Dewitt, C. Kant, and J. F. Naughton. A Status Report on the OO7 OODBMS Benchmarking Effort. In *Proceedings of the 9th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 1994.

[10] M. Castro, A. Adya, B. Liskov, and A. C. Myers. HAC: Hybrid Adaptive Caching for Distributed Storage Systems. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP)*, 1997.

[11] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Escape Analysis for Java. In *Proceedings of the 14th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 1–19, 1999.

[12] D. G. Clarke, J. M. Potter, and J. Noble. Ownership Types for Flexible Alias Protection. In *Proceedings of the 13th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1998.

[13] C. Delcourt and R. Zicari. The design of an integrity consistency checker (ICC) for an object-oriented database system. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1991.

[14] M. Dmitriev. Safe Class and Data Evolution in Large and Long-Lived Java Applications. Technical Report TR-2001-98, Sun Microsystems, 2001. http://research.sun.com/techrep/2001/-smli_tr-2001-98.pdf.

[15] F. Ferrandina and G. Ferran. Schema and database evolution in the O2 Object Database System. In *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB)*, 1995.

[16] F. Ferrandina, T. Meyer, and R. Zicari. Correctness of Lazy Database Updates for Object Database Systems. In *Proceedings of the 6th International Workshop on Persistent Object Systems*, 1994.

[17] F. Ferrandina, T. Meyer, and R. Zicari. Measuring the Performance of Immediate and Deferred Updates in Object Database Systems. In *Proceedings of the OOPSLA Workshop on Object Database Behavior, Benchmarks and Performance*, 1995.

[18] B. S. Lerner and A. N. Habermann. Beyond Schema Evolution to Database Reorganization. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1990.

[19] B. Liskov, M. Castro, L. Shrira, and A. Adya. Providing Persistent Objects in Distributed Systems. In *Proceedings of the 13th European Conference for Object-Oriented Programming (ECOOP)*, 1999.

[20] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2001.

[21] Object Design Inc. *ObjectStore Advanced C++ API User Guide Release 5.1*, 1997. http://support.odi.com/i/documentation/doc/-objectstore/r51/ostore/doc/refcoll/index1.htm.

[22] Objectivity Inc. *Objectivity Technical Overview, Version 6.0*, 2001. http://www.objectivity.com/DevCentral/-Products/TechDocs/pdfs/techOverview6.pdf.

[23] D.J. Penny and J. Stein. Class Modification in the GemStone Object-Oriented DBMS. In *Proceedings*

of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 1987.

[24] A. Salcianu and M. Rinard. Pointer and Escape Analysis for Multithreaded Programs. In *Proceedings of the 8th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2001.

[25] Versant Object Technology, Menlo Park, CA. *Versant User Manual*, 1992. http://www.versant.com/products/vds/.

[26] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 1999.

[27] Y. Zhang. Lazy Schema Evolution in Object-Oriented Databases. Master's thesis, Massachusetts Institute of Technology, Cambridge, September 2001.

[28] R. Zicari. A framework for schema updates in an object-oriented database system. In *Proceedings of the 7th International Conference on Data Engineering (ICDE)*, 1991.