Technical Report RT/32/2011

DuST'M - Dynamic Upgrades using Software Transactional Memory

Luís Pina
INESC-ID / IST
luis.pina@gist.utl.pt

João Cachopo
INESC-ID / IST
joao.cachopo@ist.utl.pt

June 2011

Abstract

Upgrading a running program without stopping it is increasingly important in a world where users have grown to expect zero downtime from software services. And yet, surprisingly, no capable system exists that fulfills this promise: Developing a dynamic software upgrade system has proved to be a very challenging task.

In this paper we present the design and implementation of a practical dynamic software upgrade system for Java that tackles some of the most difficult challenges in the implementation of such a system. Namely, how to provide programmers with a simple programming model for specifying the program's data conversion logic without incurring into unacceptable pauses during an upgrade.

Our solution uses a multiversioned Software Transactional Memory to allow software upgrades that occur atomically, concurrently with the execution of the program, and that convert the program's data lazily, as data is progressively accessed by the execution of the upgraded program.

We show that our lazy approach to software upgrades is able to upgrade a system without affecting significantly the maximum response time of the system's operations, whereas an immediate approach shows values of maximum response time up to three orders of magnitude higher.

DuST'M - Dynamic Upgrades using Software Transactional Memory

Luís Pina   João Cachopo

Instituto Superior Técnico / INESC-ID

{luis.pina,joao.cachopo}@ist.utl.pt

Abstract

Upgrading a running program without stopping it is increasingly important in a world where users have grown to expect zero downtime from software services. And yet, surprisingly, no capable system exists that fulfills this promise: Developing a dynamic software upgrade system has proved to be a very challenging task.

In this paper we present the design and implementation of a practical dynamic software upgrade system for Java that tackles some of the most difficult challenges in the implementation of such a system. Namely, how to provide programmers with a simple programming model for specifying the program's data conversion logic without incurring into unacceptable pauses during an upgrade.

Our solution uses a multiversioned Software Transactional Memory to allow software upgrades that occur atomically, concurrently with the execution of the program, and that convert the program's data lazily, as data is progressively accessed by the execution of the upgraded program.

We show that our lazy approach to software upgrades is able to upgrade a system without affecting significantly the maximum response time of the system's operations, whereas an immediate approach shows values of maximum response time up to three orders of magnitude higher.

Categories and Subject Descriptors D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement; D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures

General Terms Languages, Reliability

Keywords dynamic software upgrades, software transactional memory, atomic upgrades, high availability, binary code refactoring

1. Introduction

Software is not static: It must evolve to fix bugs and incorporate new features. Software upgrades are the vehicle of such evolution, but they come with a cost. In their simplest form, software upgrades require stopping the outdated running program and restarting it in an updated version. This disruptive process is always undesired, ranging from inconvenient to unacceptable.

Dynamic software upgrades (DSU) tackle this problem by supporting the installation of a new version of a running program without restarting it. Making such a dynamic upgrade poses two complementary challenges.

The first challenge is how to upgrade the code of a running program. Solving this problem may range from trivial to extremely difficult, depending on the language used to implement the program. For instance, the Java programming language and runtime impose severe restrictions on the ability to modify dynamically a running program.

The second challenge is how to migrate the data of the running program to a new set of data that is compatible with the new version of the upgraded program, so that the program may continue to run in the same state that it was before. Solving this *state migration* problem is often the most difficult part, and where most systems for DSU fail to provide satisfactory solutions. Among other difficulties raised by the state migration problem, two major concerns stand out and are of particular interest to us in this paper.

First, how easy is it for programmers to specify the data conversion logic? In general, converting data between two versions of a program is domain dependent, which means that it must be made by conversion code written by the programmers. This task is easier if the DSU system provides a simpler programming model for the conversion code. For instance, if programmers only need to be concerned with the conversion of each data type, independently of the rest of the program, rather than having to be aware of the entire conversion process.

Second, how long is the pause imposed by the state migration to the rest of the system? The expected semantics for an upgrade is that, after the upgrade, new operations of the program run in the new version of the program, but this entails that those operations must see the converted state of the program. A simple way to convey this semantics is to assume that all of the program's data is converted when the upgrade is installed, but this naive approach may impose unacceptable delays in the response time of the system, effectively making it unavailable for extended periods of time. This problem is specially important for applications with very large heaps, which are becoming more common with the affordable availability of machines with hundreds of gigabytes of memory.

In this paper we present the design of DuST'M, a DSU system for Java that supports *atomic dynamic software upgrades*. DuST'M addresses the problems outlined above by using a *lazy* approach to the conversion of the program data, while requiring that programmers write only a `convert` method for each upgraded class. To write these methods, programmers just have to reason locally about the migration of each class from one program version to the following, making this approach highly modular. On the other hand, the lazy approach used to do the state conversion allows DuST'M to minimize the delay imposed on the maximum response time of each operation, making it a function of each operation working data, rather than the application's heap size.

DuST'M operates on the bytecode that results from compiling a new version of the application. It post-processes that bytecode to add an extra level of indirection that supports DSU and to generate helper classes that allow the developer to refer to old versions of the program. This enables the developer to use tools such as IDEs and debuggers before post-processing the program to install it as a new upgrade. The `convert` methods are similar to *transform functions* [23], which migrate objects by initializing an instance of the new class version using the existing instance of the current class version.

Migrating the state lazily, however, is often difficult, because it may break the intended atomic upgrade semantics. To solve this problem, DuST'M relies on the Java Versioned Software Transactional Memory (JVSTM) [7] to make `convert` methods able to access objects in the correct version of the program state. We claim that, using this approach, DuST'M provides a simpler programming model for migrating the program state, when compared to existing approaches.

As other DSU systems, DuST'M supports upgrades that change the class structure, such as adding, removing, and modifying methods and fields. DuST'M also supports refactoring the class hierarchy, with a few limitations.

All of this is accomplished by DuST'M without modifying the underlying Java Virtual Machine (JVM) in any way, nor relying on any particular features or implementation of a specific JVM or Java compiler.

In summary, this paper's main contributions are: (1) techniques to atomically upgrade a running system without stopping it, using a versioned STM to expose a simple upgrade semantics to the developer, and (2) the design, implementation, and evaluation of DuST'M, a pragmatic upgrade system that supports a wide range of modifications that upgrades can perform and that can atomically migrate the program state, keeping the objects' identity between upgrades and using modular conversion code.

The remainder of this paper is structured as follows: In Section 2 we present DuST'M's programming model. In Section 3 we describe the semantics of the program state conversion. In Section 4 we describe how we implemented DuST'M. In Section 5 we discuss the role that JVSTM plays in DuST'M. In Section 6 we describe some implementations that bridge the gap between DuST'M and the underlying JVSTM. In Section 7 we evaluate the overhead that DuST'M introduces. Section 8 situates DuST'M in relation to other work. Finally, we conclude in Section 9.

2. DuST'M Programming Model

One of the major design decisions made in the development of DuST'M was that it should provide a simple programming model: The system should not add much complexity to what developers have to do already to write their programs. To help in the discussion of the programming model of DuST'M, let us introduce a very simple application that manipulates geometric data, namely points and rectangles. The left part of Figure 1 shows a possible Java implementation of such application.

The right part of Figure 1 shows a possible evolution of the geometric application. This evolution introduces modifications to the internal structure of both classes (in `Point`, the representation changes from rectangular to polar coordinates; and in `Rectangle` we add a new vertex), and to the exported interface of class `Point` (method `dist` is now static and receives one more argument). Besides, the implementations of all methods change, as expected, to cope with these modifications.

2.1 Upgradable Types

DuST'M provides support for DSU on Java, which is an object-oriented programming (OOP) language. An application written in an OOP language may be described as a set of classes (types) whose instances interact with each other as the program runs. The application that the developer writes defines a subset of these types.

DuST'M provides support for upgrading only the types that the developer writes. We call these the *upgradable types*. All the other types are *non-upgradable types*. For instance, considering the example shown in Figure 1, classes Point and Rectangle are the only upgradable types. The non-upgradable types are, for instance, the types that the standard Java libraries define, such as `java.lang.String`.

DuST'M considers that applications are composed of two layers, as Figure 2 shows: The *execution platform*,

```
class Point {
  private double x, y;

  public Point(double x,double y) {
    this.x = x; this.y = y;
  }

  public double dist(Point p) {
    return
      sqrt(square(p.x-x)+square(p.y-y));
  }


  public String toString() {
    return "("+x+";"+y+")";
  }
}

class Rectangle {
  private Point topLeft, botRight;

  public Rectangle(Point tl,Point bt) {
    topleft = tl; botRight = br;
  }

  public double area() {
    Point tr = new Point(botRight.x,topLeft.y);
    return topLeft.dist(tr) * tr.dist(botRight);
  }


}
```

Version 1

```
class Point {
  private double rho,theta;

  public Point(double rho,double theta) {
    this.rho = rho; this.theta = theta;
  }

  public static double dist(Point a,Point b) {
    double ra = square(a.rho), rb = square(b.rho);
    return
      sqrt(ra+rb+2*a.rho*b.rho*cos(a.theta-b.theta));
  }

  public String toString() {
    return "("+rho+","+theta+")";
  }
}

class Rectangle {
  private Point topLeft, botRight, botLeft;

  public Rectangle(Point tl,Point bt,Point bl) {
    topleft = tl; botRight = br; botLeft = bl;
  }

  public double area() {
    return
      Point.dist(topleft,botLeft)
      *
      Point.dist(botRight,botLeft);
  }
}
```

Version 2

Figure 1. Geometric application example

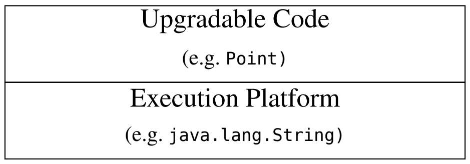

Figure 2. Layered view of an upgradable application.

which supports the execution of the application's code; and the *upgradable code*, which contains the application itself. DuST'M is part of the execution platform and, thus, is not upgradable. Given this architecture, none of the code in the execution platform is allowed to use upgradable code.

2.2 Conceptual Model

Figure 3 shows the main components of an upgradable application. The execution environment runs the application by executing the *program* — a sequence of instructions that defines the behaviour of the application. It receives a *stimulus* from the exterior and reacts accordingly, producing the *response* — the result of the program execution.

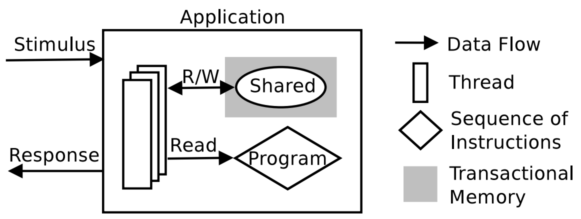

Figure 3. Software architecture of an application using an STM to synchronize the shared program state.

The execution environment keeps the *program state* — the current state of the application. To process each stimulus, a thread accesses the program state to compute the response. All threads may access the *shared program state*.

We assume the existence of a Software Transactional Memory (STM) to synchronize the concurrent manipulation of the program state. Using it, the developer groups several operations into a unit of work — a *transaction*. Transactions are atomic: Either they complete successfully and commit, thus modifying the program state, or they conflict with other transactions and abort, leaving the program state unchanged.

2.3 Development Process

We implemented DuST'M as a post-processor and a runtime library. The post-processor operates on the bytecode that the Java compiler generates. It modifies the original bytecode and generates a set of runtime support classes and a set of old classes. Old classes allow the conversion code of the next version, which refers to the current version, to compile correctly. Figure 4 shows how DuST'M integrates with the development process.

This development process allows the developer to use source and bytecode tools such as debuggers, Integrated Development Environments (IDEs), and profilers. Moreover, the developer uses the standard version of the Java compiler. When a new program version is mature enough to be installed as a new upgrade, the developer uses the post-processor to generate the DSU enabled version.

The old classes, that we shall define in the next section, are not meant to be executed. Their sole purpose is to provide symbolic integration, allowing the developer to compile the conversion code that refers to the old program state. The post-processor replaces the references to the dummy generated old classes by references to the real old version of the classes.

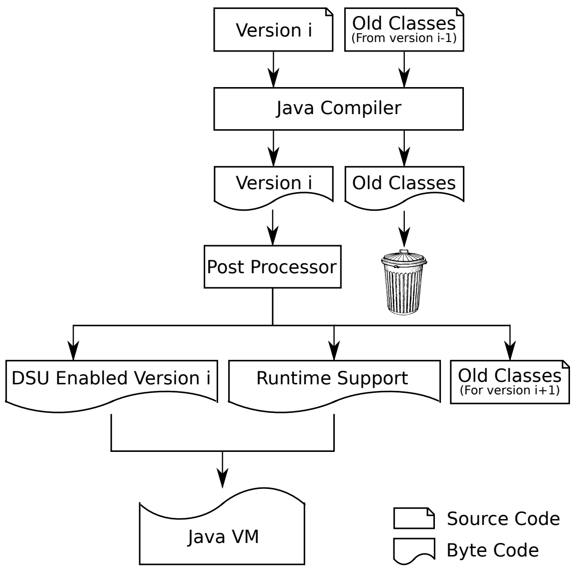

Figure 4. Development process of an upgradable application, using DuST'M

2.4 `convert` Methods and Old Classes

Using DuST'M, a developer specifies how to migrate instances between program versions using `convert` methods. Each class of the new program version must have a `convert` method that receives two arguments: (1) an instance of the old version of the class, and (2) an instance of the new version of the class. The purpose of the `convert` method is to initialize the new instance, based on the old instance. Figure 5 shows possible `convert` methods for converting the instances from version 1 to version 2 of the geometric application shown in Figure 1.

```
class old.Point {
  public double x, y;

  double dist(old.Point p) {return 0.0;}

  String toString() { return  ""; }

  public Point convert() {
    return null;
  }
}

class old.Rectangle {
  public old.Point topLeft, botRight;

  double area() { return  0.0; }

  public Rectangle convert() {
    return null;
  }
}

class Point {
  private double rho,theta;
  ...
  static void convert(old.Point o,Point n) {
    n.rho = sqrt( square(o.x) + square(o.x) );
    n.theta = arctan( o.y / o.x );
  }
}

class Rectangle {
  static void convert(old.Rectangle o,Rectangle n) {
    n.topLeft  = o.topLeft.convert();
    n.botRight = o.botRight.convert();
    old.Point botLeft =
      new old.Point( o.topLeft.x, o.botRight.y);
    n.botLeft = botLeft.convert();
  }

  ...

}
```

Figure 5. Example of `convert` methods and old classes.

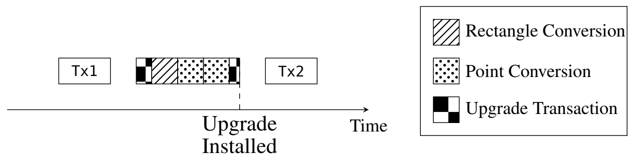

Figure 6. Immediate Upgrade Semantics.

The developer defines the `convert` methods in the new program version. However, to convert the instances from the previous program version, he must have access to that version. To enable the developer to refer to the old version of the program, DuST'M generates the *old classes*.

The old classes have the same fields as the real program version, but DuST'M makes them publicly visible. Thus, the developer is able to access the internal state of the old program version, even if it was private in the original class.

The old classes also have the same methods as the real program version, but these are dummy methods that return 0 or null, according to the return type. Likewise, methods belonging to the old classes are publicly visible, for the same reason as fields. Besides the existing methods, all old classes have a method named `convert`. This method allows the developer to specify that a field, in the new version, must keep the same object that it had on the previous version. Please note that the developer cannot write `n.topLeft = o.topLeft` because `n.topLeft` has type `Rectangle` and `o.topLeft` has type `old.Rectangle`, which are incompatible. Figure 5 shows the usage of such methods on the method `Rectangle.convert`, when converting fields `topLeft` and `botRight`.

Besides allowing the compilation of the conversion code, the old classes also provide symbolic integration. The Java compiler enforces the types, keeping the type safety of the conversion code.

3. Atomic Upgrades

In the previous section we described the DuST'M's programming model. Now, we describe the semantics of the program state conversion.

3.1 Immediate Upgrades

A DSU that supports program state migration may perform such migration eagerly, when the new upgrade is installed. This behaviour is what we call *immediate upgrades*.

To illustrate immediate upgrade semantics, consider the following example. Version 1 of the geometric application, shown in Figure 1, starts executing. Then, a transaction `Tx1` creates a rectangle (and the respective two points). After that, the application is upgraded, thus installing version 2. Figure 6 illustrates this sequence of events.

Using immediate upgrade semantics, transaction `Tx2` expects to find instances of version 2 of the program only. Given that this upgrade semantics is quite simple to use and to understand, we designed DuST'M so that it provides this simple upgrade semantics to the developer. However, a naïve eager immediate conversion of the program state is very costly and may introduce a long pause in the program execution while the conversion is taking place.

3.2 Lazy Upgrades

Another approach to convert the program state after installing an upgrade is to do it lazily. In this case, an instance needs to be converted only when the new version of the program attempts to access it. So, the upgrade system does not need to perform such conversion sooner. This is a key observation: The upgrade system is free to delay the conversion of any instance to the last possible moment.

Let us consider the same sequence of events described in the previous section. Figure 7 shows those events on an upgrade system that converts the program state lazily. Each object is converted when transaction `Tx2` first manipulates it.

Lazy upgrades do not need to convert all the program state when installing an upgrade. However, after installing the upgrade, there may be a burst in object conversions. Such burst may pause the application's execution, just as immediate upgrades do, but for a smaller period.

3.3 Atomic Lazy Upgrades

The lazy upgrade semantics is more complex than the immediate upgrade semantics. After installing an upgrade, the system is now running a mixture of both the old and new versions. This may lead to odd behaviour. For instance, consider the *conversion ordering problem*: A point is converted before the rectangle. The conversion code of the rectangle, shown in Figure 5, accesses both points in the old version. But each point is on a different program version.

There are several approaches to solve this problem: Provide support for accessing any given program version [2]; allow that the conversion code accesses the new program version only [20]; or require *backward conversion code* that migrates from version $i+1$ to version i [15]. These approaches either increase the complexity of the system or limit the expressiveness of the conversion code.

Boyapati et al. [4] propose another solution. The system converts each object in a separate *conversion transaction*. However, convert transactions are special: The upgrade system must serialize a conversion transaction T before serializing any application transaction that uses the object that T converts.

Figure 8 shows the atomic upgrade semantics. Although the system converts the program state lazily, the conversion code executes at the same logical moment as the upgrade occurred. Therefore, the upgrade system exposes immediate upgrade semantics to the developer. This atomic semantics solves the conversion ordering problem because it ensures that the conversion code of the rectangle always sees the old versions of the points.

In their implementation, Boyapati et al. rely on object encapsulation to generate a correct order in which the conver-

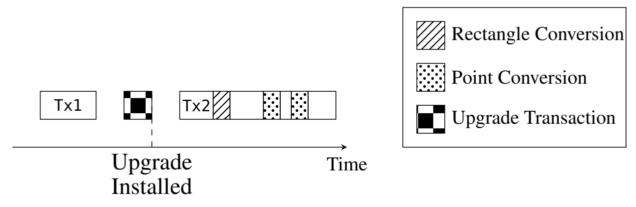

Figure 7. Lazy upgrade semantics.

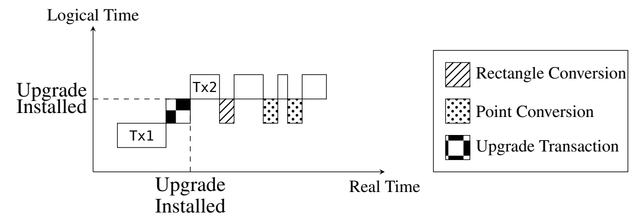

Figure 8. Atomic upgrade semantics

sion code executes. If the rectangle's points are encapsulated by the rectangle, i.e. accessible only by using the rectangle, the application naturally converts the rectangle before converting the points.

However, they identify some corner-cases where the encapsulation mechanism fails. They solve such cases by falling back to versioning: They keep the old version of the points after converting them. Using the old version of the points, the rectangle's conversion code can access them in the expected program version.

DuST'M solves this problem by using JVSTM [7], an STM whose transactional locations (VBoxes) keep a history of successfully committed values. As JVSTM already supports object versioning, DuST'M uses it to provide atomic upgrade semantics to the developer.

4. Implementation

We implemented DuST'M as a post-processor and a runtime library. The post-processor takes as input the bytecode that the Java compiler generates and performs some binary refactoring, enhancing the bytecode to support dynamic software upgrades. The post-processor uses ASM [6] to perform the binary refactoring. This section describes the modifications that the post-processor performs on the original bytecode.

4.1 Handles as Transactional Proxies

Object oriented applications work by manipulating *objects*. The developer reasons about objects and their interaction when he is writing the application. But when the application is executing, those objects are mapped to instances. An *instance* is a runtime manifestation of a class.

To install a new version of an upgradable type, DuST'M does not modify the existing class loaded by the JVM. Instead, it considers the evolved type as a new class, totally unrelated to the existing one. Consider the geometric application that Figure 1 describes. To install version 1, DuST'M renames the classes `Point` and `Rectangle` to `Point$1` and `Rectangle$1`, respectively. Thus, version 2 is composed of classes `Point$2` and `Rectangle$2`.

Typically, the mapping between objects and instances is simple: An object maps to a single instance. But, if we introduce a new Java class per class upgrade, an object is now represented by different instances across program versions.

Consider that object `rectangle` refers to object `point` of class `Point`. At runtime, instance `rectangle` refers to an instance `point$1` of class `Point$1`. When the upgrade system installs a new program version, it installs the new version of class `Point`: Class `Point$2`. In this new version, the same object `point` is now represented by an instance `point$2` of class `Point$2`.

Instances `point$1` and `point$2` represent the same object, in different versions of class `Point`. Thus, the identity of object `point` is kept by a different instance at each version of class `Point`. But instance `rectangle` has a reference to instance `point$1`, even after the new version of class `Point` is installed.

DuST'M solves this problem using *handles*. An handle is an extra level of indirection, responsible for keeping the identity of an object across program versions. Handles maintain the non-upgradable state of an object and keep a reference to an instance that maintains the upgradable state of the object at a certain program version. So, with each new program version, handles replace that instance with a new instance that represents the object in the new version. DuST'M ensures that all references to an upgradable object are, in fact, references to its handle, and never to any of the instances that keep the state of the object at a certain version of the program.

Using handles, instance `rectangle` has a reference to handle `h`, which in turn has a reference to instance `point$1` in version 1, and to instance `point$2` in version 2. Therefore, handle `h` is responsible for keeping the identity of object `point` across program versions.

Handles are transactional memory locations. They are implemented using VBoxes [7], as Figure 9 shows. VBoxes are transactional memory locations that keep an history of values. In this case, each value in the history corresponds to a version of the object in a program version. Thus, handles always return the correct instance: The one that represents the object in the current program version.

Let us recall the conversion ordering problem. This scenario, introduced in Section 3.3, considers that the upgrade system converts a point before converting the rectangle that refers to it. Such situation may be problematic because the code that converts the rectangle, shown in Figure 5, expects both points in the old version of the program, and one of these points has already been converted.

Handles, as transactional locations, solve this problem naturally. Figure 10 shows why. The natural program flow

```
class Handle {
  static VBox<Integer> systemVersion;
  VBox<Pair> pair;

  static Object getObject(Handle h) {
    Pair p = h.pair.get();

    if (systemVersion.get() > p.version) {
      h.convert();
      p = h.pair.get();
    }

    return p.object;
  }

  void convert() {
    //Creates an instance in the new program version
    //Creates a new transaction in the past
    //Invokes the custom convert method
    //Puts the migrated instance in the VBox pair
  }
}

class Pair { Object object; int version; }
```

Figure 9. Outline of class `Handle`

leads to converting a point in real time instant 1. Then, transaction $Tx1$ commits, generating logical time instant B.

Soon after that, transaction $Tx2$ starts and attempts to manipulate the rectangle, which needs to be converted, at real time instant 2. DuST'M launches the conversion transaction in the logical time instant A. At this logical time instant, both handles return the points in the old program version.

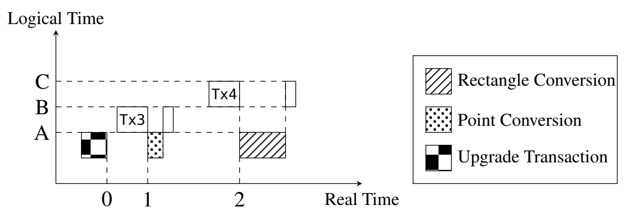

Figure 10. Transactional solution to the conversion ordering problem.

When transaction $Tx2$ finishes, creating logical time instant C, there is still a point that needs conversion. DuST'M will convert that point when the natural program flow attempts to use it.

Putting together Figure 9 and Figure 10, we can see how DuST'M internally uses JVSTM to enforce the atomic upgrade semantics. Let us consider that the operation of installing the new program version, which ends at real time instant 0, increments the value of `Handle.systemVersion` from 1 to 2. Application transactions $Tx1$ and $Tx2$ see that value as 2, but the conversion transactions see that value as 1. The behaviour of the involved `Handle.pair` boxes is similar.

4.2 Handles and Inheritance

DuST'M's post processor replaces all upgradable types from fields, local variables, and method arguments with handles. Therefore, the program always manipulates objects that belong to upgradable types using their handle.

A possible generic handle would extend all non-upgradable classes that are supertypes of upgradable ones, and implement all non-upgradable interfaces that upgradable classes do. However, in Java it is not possible to create such generic handle. Consider the example that the upper part of side of Figure 11 shows. In this simple case, a generic handle could inherit from the non-upgradable class `Rectangle` and implement interface `Comparable`. Such generic handle has two problems: (1) class `Square` does not implement interface `Comparable` but the generic handle used to represent its instances does, and (2) if another upgradable type inherited from an non-upgradable type different from `Rectangle`, the generic handle would be impossible to implement in Java, which does not support multiple inheritance.

To deal with the inheritance problems, DuST'M generates specific handles according to the upgradable types. For each *root upgradable type* (an upgradable type that inherits directly from a non-upgradable type) DuST'M generates two classes: a *primary handle* and a *handle helper*. For each remaining upgradable type that implements another non-upgradable interface, DuST'M generates a *secondary handle* that implements that non-upgradable interface.

The lower part of Figure 11 shows the classes after post processing. Primary handles and handle helpers (`Rectangle$Handle` and `$Rectangle`) override all the methods defined and inherited by the non-upgradable type. Secondary handles (`RectangleComparable$Handle`) implement all the methods defined by their non-upgradable interfaces.

Handles override all the methods defined and inherited by the non-upgradable type. These *downward methods* delegate on the upgradable instance that the handle keeps.

To implement each method, the handle's code cannot cast the upgradable instance that it keeps to the non-upgradable super type because the post-processor broke that relationship. Note that, in the lower part of Figure 11, class `Square$1` does not inherit from class `Rectangle`. Besides, handles cannot have direct references to upgradable types. Handle helper classes bridge that gap, providing the same interface as the upgradable types. Thus, the primary handle can delegate on the upgradable instance by casting it to the handle helper class, `$Rectangle` in this case.

For each downward method in an handle, the handle helper class has a method with the same name and signature. Such methods execute if the handle delegates a downward method on an upgradable type A that does not override that method, and neither do any of A's upgradable parents. For instance, consider invoking `toString` on a `Rect-`

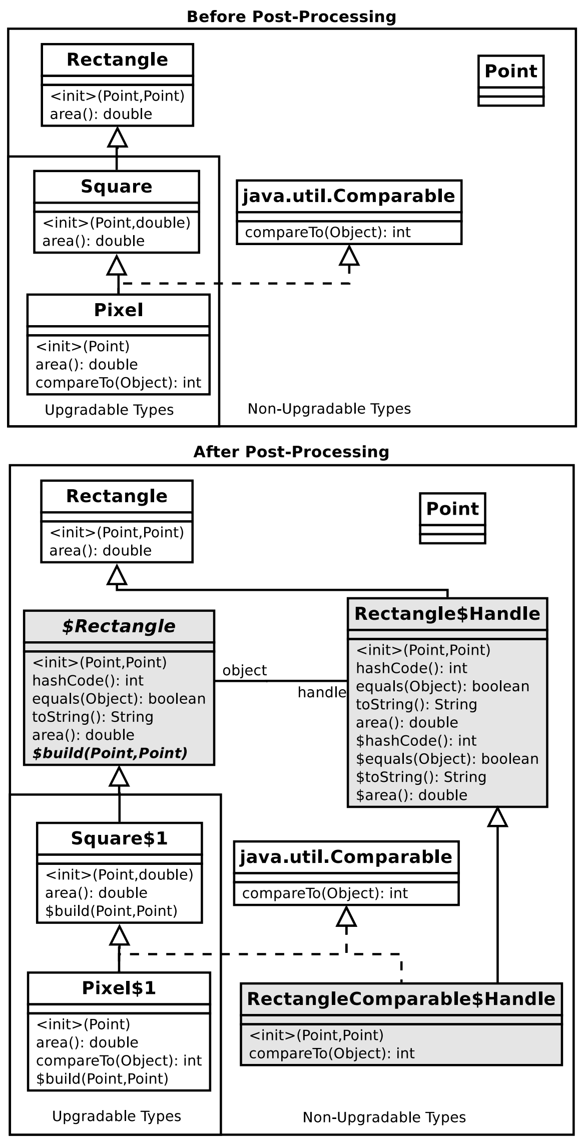

Figure 11. Upgradable and non-upgradable types, before and after post-processing. Please note that, unlike in Figure 1, class `Rectangle` is non-upgradable.

`angleComparable$Handle`. The handle keeps an instance of `Pixel$1` that does not override method `toString`. As none of its upgradable super types override the method `toString`, the handle `RectangleComparable$Handle` ends up invoking the method `toString` on the handle helper class `$Rectangle`.

The handle defines an *upward method* for each downward method, which delegates on the non-upgradable super type. Upward methods have the same signature as the original downward methods. Upward methods' names are the result of prepending the character "$" to the original downward method name. In the example we have been following, method `toString` on class `$Rectangle` delegates on upward method `$toString` on class `Rectangle$Handle`, which in turn invokes method `toString` on the non-upgradable super class `Rectangle`.

Finally, the handle overrides all constructors existing on the non-upgradable super type. These constructors just relay the arguments to the respective constructor on the non-upgradable super type.

4.3 Instantiating Upgradable Types

So far, we showed how DuST'M uses handles to keep the identity of upgradable objects across program versions. Handles are tightly coupled with the instances that they represent. They keep the non-upgradable state of the upgradable object, whilst the upgradable instance itself keeps the upgradable state in each program version.

As a consequence, when the developer creates an upgradable object, DuST'M must create two instances: The handle and the upgradable instance. For instance, when the developer instantiates class `Pixel`, shown at the upper part of Figure 11, DuST'M must create an instance of class `Pixel$1` and another instance of class `RectangleComparable$Handle`, shown at the lower part of Figure 11.

The developer expects that the constructors of `Pixel` execute in the following order: (1) `Rectangle`, (2) `Square`, and (3) `Pixel`. Thus, to keep the original program semantics, DuST'M must create the handle inside the constructor of the handle helper class `$Rectangle`. To create the correct handle type (`Rectangle$Handle` for `Square$1`, `RectangleComparable$Handle` for `Pixel$1`), the constructor of the handle helper class starts by invoking the abstract method `$build`. This method, which DuST'M injects in every upgradable type, is responsible for instantiating the correct handle type. The resulting order in which the constructors are invoked, for class `Pixel$1`, is the following: (1) `Rectangle`, (2) `Rectangle$Handle`, (3) `RectangleComparable$Handle`, (4) `$Rectangle`, (5) `Square$1`, and (6) `Pixel$1`.

5. Discussion

The major goal of our upgrade system is to provide a natural semantics to an upgrade (similar to an immediate upgrade) but being able to do it lazily. Moreover, as an important part of an upgrade is the conversion of the program state, we want the code that performs the conversion (which must be written by the application programmer) to be as simple as possible. We argue that our approach satisfies both of these goals, unlike previous work on this area.

The multi-versioned STM (JVSTM [7]) plays a major role in our system because it allows us to convert objects lazily, as they are needed by the application code, but still guarantee that the conversion code will get a consistent view of the program state at the logical time of the update, regardless of how long ago it took. This is possible because whenever there is an upgrade of the system, it is as if all of the objects were replaced by new versions of them, keeping all of the old versions unmodifiable and available for referral at any later time (needed to ensure that the conversion code may run lazily at any time). The key elements of our proposal are the introduction of versioned, transactional handles that wrap the objects, and ensuring that all of the application's code goes through those handles.

This would not be possible with other STMs that do not support multiple versions of the same transactional location, and this is one of the things that distinguishes our proposal from others. Whereas some of the previous work on upgrade systems provide similar semantics, they either restrict too much the upgrades that can be done or they restrict the type of conversion code and how it must be written. For instance, the work that most approximates ours (the work of Boyapati et al [4]) uses transactions to do the conversion, but they require the developer to write the code using ownership types to enforce encapsulation. Encapsulation ensures that the conversion code accesses the consistent view of the program state that is needed to allow a modular specification of the conversion code.

A current limitation of our work, however, is that we are assuming that the original program is already using transactions (STM transactions), and we do not explain what to do if that is not the case, but that work is out of the scope of this paper. This is important because transaction boundaries are safepoints for upgrades to occur. So, if we do not have transactions, we cannot guarantee the desired consistency of the upgrades unless we identify somehow the safepoints for the upgrades. But this is an orthogonal issue. If there is some mechanism that identifies safepoints for upgrades, then our system will still work in a program that is not transactional in the first place, provided that the upgrade operation does not occur concurrently with any other code of the program, which is not a major limitation given that the upgrade operation in itself is quite short. After the upgrade operation, the conversion code will be able to run lazily and safely within the transactions that are introduced by the upgrade system.

In short, to make a program upgradable using DuST'M, the developer has to delimit transactions and define the upgradable objects. If any conversion function needs to access non-upgradable objects, those objects must be transactional to enable DuST'M to provide immediate upgrade semantics.

Please note that just adding JVSTM transactions to an existing application does not add any extra overhead to its execution: That overhead would come if, besides adding transactions, the developer also adds VBoxes to identify the shared state (e.g., by making non-upgradable objects transactional). The overhead in JVSTM comes from keeping the shared state consistent inside a transaction, and not from managing the transaction itself.

Still, assuming that we already have a transactional program, using our upgrade system does not interfere with the shared state of that program.

The upgrade system that we propose introduces new mutable state in the form of the handles. These mutable locations were not present in the original program and, therefore, are not accessed by the application code. They are accessed only by the runtime of the upgrade system when it is converting objects (which happens when the application code needs to access an object that was not converted yet).

One important consequence of this approach is that the conversion code runs within the scope of the original application threads. So, if the original program was sequential, so will be the modified program after the transformation made by our upgrade system.

This is important to understand whether the upgrade system introduces new shared state or not (it does not). The upgrade system itself never changes the original program's state. Instead, it just changes the handles. Likewise, the original program's code never changes the handles directly.

In what concerns sharing of handles, we have two cases:

1. A handle is wrapping an object that was not shared in the original program. In this case, the handle itself will not be shared either, and the transaction that is converting the object will never conflict with another transaction because this transaction will change only the handle, which is not shared.
2. A handle is wrapping a shared object. In this case, the handle will be shared among various threads as well, and they may try to access the same object concurrently. In this case, they may try to convert the object concurrently, storing in the handle the new version of the object. But as this is done within transactions, the STM system will detect a conflict and it will restart all but one of the transactions. The restarting transactions, however, will see that the object is already converted and will not try to convert it again.

So, to summarize, we claim that one of the major contributions of our work is to show how a multi-versioned STM may be used to implement an immediate upgrade semantics with lazy conversion of the program state. This is, to the best of our knowledge, a novel contribution in this area.

Another contribution of our paper is the description of an implementation of this idea on top of an unmodified JVM. This is not, of course, the only way that we envision that this could be implemented, but it is one approach that has a series of advantages, as well as some drawbacks.

One of the potential problems of our implementation strategy is the amount of overheads introduced in the original program. The experimental evaluation that we present in Section 7 discusses what are these overheads.

6. Optimizations

To minimize the steady state overhead that DuST'M introduces, we implemented some optimizations that provide a better integration between the handles and the JVSTM. We describe those optimizations in this section.

To introduce the needed extra level of indirection, DuST'M injects handles on every reference to upgradable instances. Then, DuST'M uses the `getObject` method, shown in Figure 9, to dereference each instance. This method, therefore, represents an important source of overhead.

Taking a closer look, we can see that this method contains two `VBox.get` invocations (i.e. opens two VBoxes): (a) the object-version pair, and (b) the system version. This is an expensive method, as opposed to simply read a field, because it performs several tasks: (1) checks if the VBox was previously written in the context of the current transaction, returning the written value if so, (2) adds the VBox to the transaction's read-set, and (3) navigates through the versions that the VBox keeps to find the most recent version that the current transaction may read.

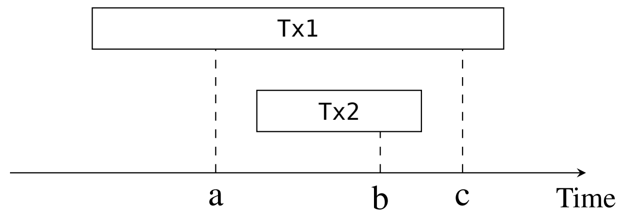

Figure 12. Overlapping transactions that write/read the same VBox

Figure 12 provides explanation for case (3). Consider that transaction `Tx1` reads a VBox at instants *a* and *c*, and transaction `Tx2` writes that VBox at instant *b*. To keep its consistency, JVSTM ensures that transaction `Tx1` reads the same value on instants *a* and *c* [11].

There is a key observation that Figure 12 highlights: Reading the same VBox time and time again inside the same transaction that does not write to that VBox always returns the same result. This is exactly what happens with the VBox that keeps the system version. With this knowledge, we implemented a new type of transaction that starts by reading the system version and then stores that value on a thread local variable. [1]

As for the VBox that holds the object-version pair, we can simplify the operation of opening it. Task (1) is redundant because handles are written only in conversion transactions, thus we can skip it. As for task (2), it ensures that, at commit time, this transaction conflicts with any other that converted any handle that the committing transaction read. Handles already perform this validation, as Figure 9 shows in the `if` statement. Moreover, the validation that handles perform does not involve restarting the transaction. Therefore, task (2) is also redundant and we can skip it.

[1] In JVSTM, there is a one-to-one correspondence between transactions and threads.

These two optimizations significantly lower the overhead that DuST'M imposes on steady state.

7. Performance Evaluation

As stated before, our main goal in the design of DuST'M was to provide a simple programming model, while still permitting a wide range of upgrades. But, of course, the design of a pragmatic DSU system must take into consideration also the overheads that it entails. In fact, even though performance was not our main target during this work, our design decisions were carefully balanced to reduce the overheads of DuST'M.

To assess these costs, we evaluated the overhead that DuST'M introduces on STMBench7 [12]. STMBench7 is a benchmark for evaluating STM implementations. It builds a set of graphs that simulate how real life applications structure their objects. Then, it traverses the graphs using several operations that model a wide range of object traversal patterns that resemble how real world applications navigate through their objects.

We executed the benchmark on a quadcore system, with an Intel Core i5 750 processor (4 cores) and 8GB RAM, running a 64-bit Linux 2.6.36, and Java SE version 1.6.0_24 (Java HotSpot 64-Bit Server VM, build 19.1-b02, mixed mode).

7.1 Benchmark Configuration

STMBench7 is highly configurable: Users can choose, among other things, a workload, the number of threads, and the benchmark length. It is also easily extendable to use different STM implementations as backend. In all the results that we present in this section, we used STMBench7 with a JVSTM backend and configured it to run the read-write workload with 4 threads for 180 seconds.

In STMBench7, it is possible to reach every instance starting from a single global static reference. To compare DuST'M results against immediate upgrade systems, we used this feature to implement an immediate upgrade mode. In this mode, DuST'M suspends all threads after installing a new program version and then, it touches all STMBench7 instances, converting them as it progresses. When it has visited all instances, DuST'M resumes all threads and STMBench7 continues executing.

When running STMBench7, we disabled two types of operations: (1) long traversals and (2) structural modifications. Our JVSTM implementation of STMBench7 builds huge read-sets on long traversals, which stress the garbage collection mechanism, thus generating unacceptable amounts of noise on the final results. We choose to disable structural modifications because they delete a significant part of the object graphs, resulting in non significant object conversion times when running DuST'M in immediate conversion mode.

As we explain in Section 2, DuST'M considers as upgradable only a part of the application. We configured it to use the JVSTM implementation of STMBench7's backend as the upgradable types. Semantically, this means that it would not be possible to perform modifications to, for instance, how a specific operation performs its traversal. However, it would be possible to replace the current JVSTM backend by a more efficient one, for instance replacing a set implemented using a list by another set implemented using a tree.

To get results of DuST'M's behaviour when performing upgrades, we produced a series of program upgrades. Each upgrade does not introduce any code modification, but it still must convert the program state by copying the existing fields from the old instances to the new instances.

7.2 Maximum Latency

STMBench7 measures the latency (or response time) of each operation, i.e. the time that passes since an operation is issued until that operation finishes. When the benchmark finishes, it reports the maximum latency that it observed for each operation.

DuST'M converts the program state lazily when it installs an upgrade, as the natural flow of the application touches that state. With this approach, and unlike immediate program state conversion, DuST'M significantly reduces the overhead added to the maximum latency of each operation during the benchmark, as shown in Figure 13. It shows, in terms of maximum latency per operation, how STMBench7 post-processed by DuST'M behaves in the absence of upgrades and, in the presence of an upgrade, how it behaves when using immediate and lazy state conversion.

As expected, immediate upgrades greatly increase the maximum latency of each operation. Lazy upgrades, on the other hand, keep the maximum latency within the same order of magnitude than the maximum latency measured on the benchmark that does not install any upgrade.

However, DuST'M introduces a new level of indirection on the original application. To understand how this extra indirection affects the maximum latency when no upgrades exist, we ran two versions of STMBench7: One unmodified and another post-processed by DuST'M, but without making any upgrade to the system. Figure 14 shows the obtained results.

Naturally, using DuST'M increases the maximum latency for every operation when no upgrades exist, which is the price that we have to pay to be able to do upgrades lazily with a reduced effect on that same maximum latency.

7.3 Throughput

So far, we presented how DuST'M changes the maximum latency of STMBench7. As important as the maximum latency is for an application, there are still other important performance metrics to consider, such as average latency and throughput. We dedicate this section to show the throughput results that we obtained for the STMBench7 benchmark.

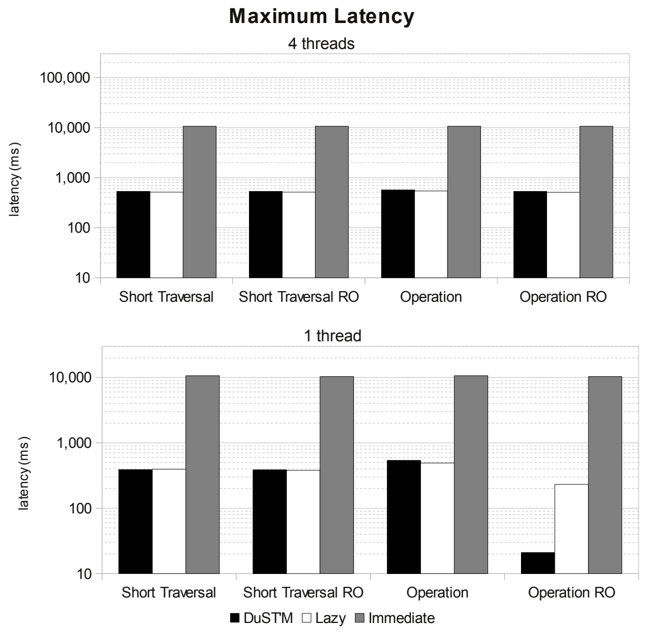

Figure 13. Maximum latency for each STMBench7 operation. To generate this chart, we executed the benchmark 15 times and registered the maximum observed latency for each operation. The column *original* refers to the benchmark without installing any upgrade, columns *immediate* and *lazy* refer to installing one upgrade (at 90 seconds) and performing the state conversion immediately or lazily, respectively.

Besides the maximum latency, the STMBench7 benchmark measures the throughput, i.e., how many operations are completed per second. When the benchmark finishes, it reports the overall average throughput. We modified it to report at each second, and while running the benchmark, how many operations completed during the past second.

Similarly to the results for the maximum latency, the lazy state conversion that DuST'M performs imposes a lighter penalty on the average throughput than an alternative immediate upgrade. Figure 15 shows those results.

Figure 15 does not show the original results because they show a high degree of variation from second to second. This would render the plotting unreadable. Furthermore, computing the average at each second from several benchmark executions did not smooth the line enough. Thus, to make each line readable, we computed, at each second, the average of the current throughput value and the value of the past 4 seconds. This average of a sliding window of 5 seconds makes the plotting readable without compromising the integrity of the data.

When converting the state immediately, there is a period following the installing of the upgrade during which the throughput drops to zero. We measured this period and it averaged 10 seconds. This value agrees with the maximum latency of around 10 seconds that Figure 13 shows for immediate state conversion. Lazy state conversion, on the other

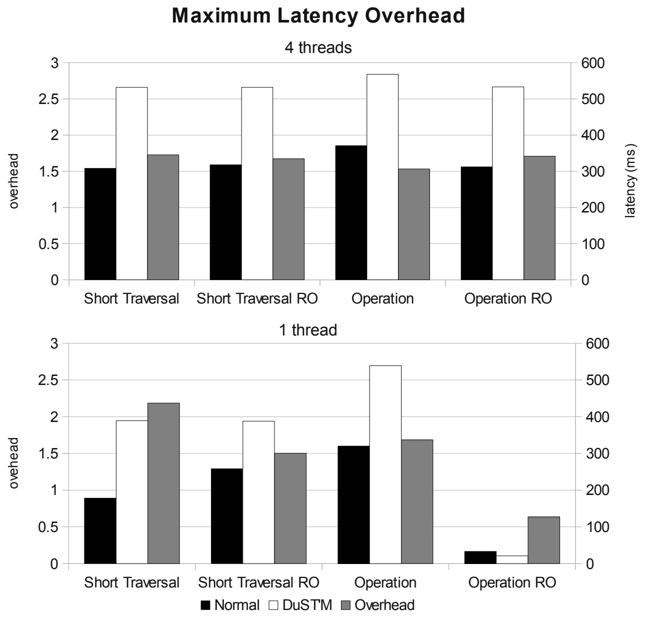

Figure 14. DuST'M overhead in terms of maximum latency. To generate this chart, we executed the benchmark 15 times and registered the maximum observed latency for each operation. The column *normal* refers to the benchmark without being post-processed, *DuST'M* refers to the benchmark after being post-processed. The chart shows a derived column *overhead*, computed as $(DuST'M/original)$. Both *normal* and *DuST'M* columns use the right y axis, whereas the *overhead* column uses the left y axis.

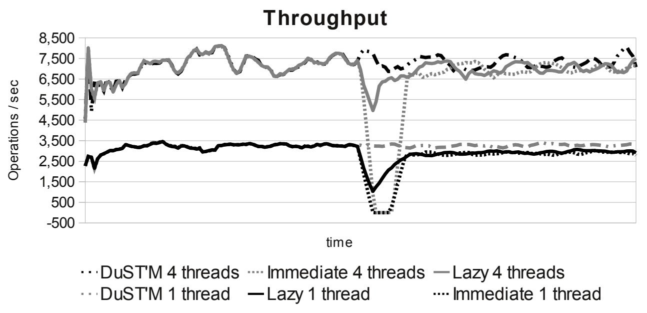

Figure 15. Throughput tracing at each second. To generate this chart, we executed the benchmark 15 times and registered the number of finished operations at each second. This figure plots the measured average for each second. The line *DuST'M* refers to the benchmark without installing any upgrade, lines *lazy* and *immediate* refer to installing one upgrade (at 90 seconds) and performing the state conversion lazily or immediately, respectively.

hand, does not impose such a large throughput penalty. We can still notice a milder throughput drop after installing the upgrade that lasts for a smaller period.

As for the maximum latency, we ran two versions of STMBench7—one unmodified and another post-processed by DuST'M—to assess what is the overhead on the throughput caused by the extra level of indirection of DuST'M. Figure 16 shows the result.

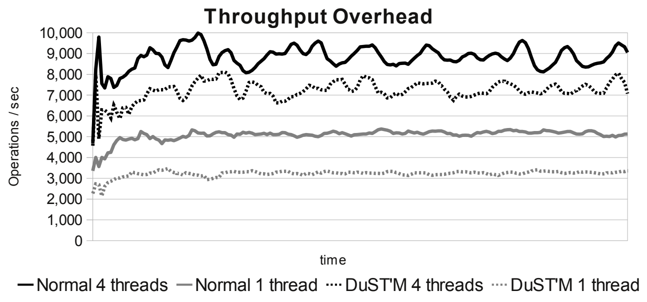

Figure 16. Throughput tracing at each second. To generate this chart, we executed the benchmark 15 times and registered the number of finished operations at each second. The column *normal* refers to the benchmark without being post-processed, *DuST'M* refers to the benchmark after being post-processed. With 4 threads, DuST'M's version is, on average, 18% slower than the original version. With 1 thread, this overhead value is 37%.

7.4 Converted Instances

Up to now we compared lazy state conversion with immediate state conversion in terms of maximum latency and throughput. The results meet our expectations that a lazy upgrade should perform better because we do not need to convert all objects to execute a new operation. To validate this assumption, we measured the number of converted instances with each upgrade approach. We show those results in Figure 17.

Converting the program state immediately means that there is a single thread that converts the same instances by the same order in every execution while the other threads wait for the conversion to finish. As a result, all lines of immediate upgrades, that Figure 17 shows, match closely regardless of the number of threads.

On the other hand, the number of converted instances per second for lazy upgrades, shown at the top plotting of Figure 17, reveals significant differences related to the number of threads. This happens because lazy upgrades convert instances in parallel. Thus, the results for 4 threads show the highest peak. Conversely, performing lazy conversion with 1 thread yields the lowest peak of object conversion because that thread is interleaving object conversion and program execution whereas immediate conversion, which also uses 1 thread when converting, is dedicated to object conversion.

The bottom plotting of Figure 17 shows that lazy state conversion does not convert as many instances as immediate state conversion. This happens because lazy state conversion naturally identifies the minimal working set of instances that the program uses. After converting that set, it converts instances that are more rarely used as the benchmark continues, and at a much lower rate.

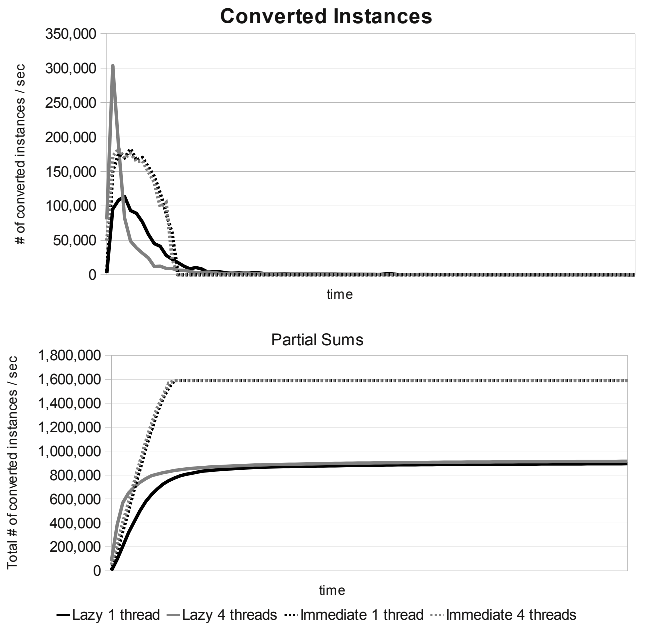

Figure 17. Instances converted after installing an upgrade using immediate and lazy program state conversion. This plotting refers to the last 90 seconds of a 180 seconds benchmark run that installs an upgrade at 90 seconds. To generate this chart, we executed the benchmark 15 times and registered the average number of converted instances at each second. The top plotting shows the instances converted at each second, the bottom plotting shows at each second the total number of converted instances until that second.

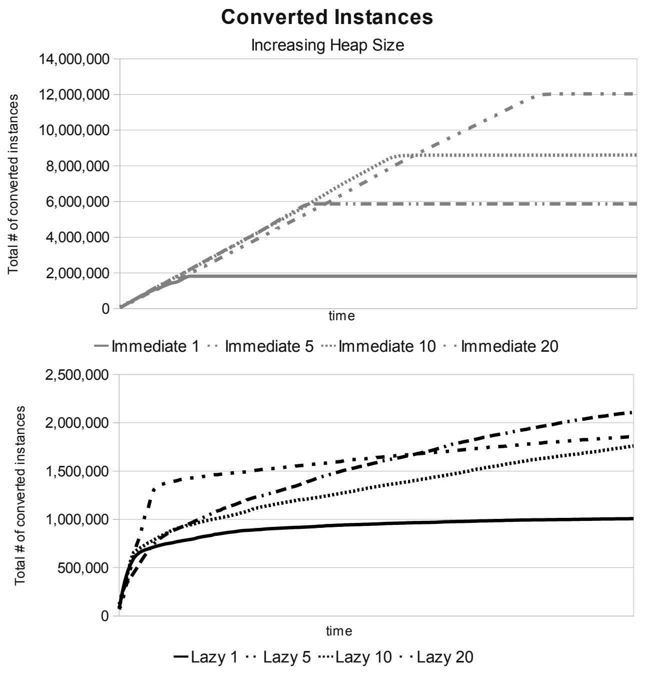

Figure 18. Instances converted after installing an upgrade using immediate and lazy program state conversion and increasing the total number of instances. This plotting refers to the last 180 seconds of a 270 seconds benchmark run that installs an upgrade at 90 seconds. To generate this chart, we executed the benchmark 10 times and registered the average number of converted instances at each second. The number at each legend line is the factor by which the total amount of a common type of instance was multiplied. The top plotting shows the total number instances converted using immediate state conversion at each second, the bottom plotting the same information for lazy state conversion.

Immediate upgrades impose a pause that is proportional to the size of the application's working set. To find how the size of the working set scales with the total amount of used instances, we modified STMBench7's setup parameters so that it uses more instances by multiplying the total number of the a common instance type in the benchmark by a factor of 5, 10, and 20. To avoid running the benchmark at the physical limit of available memory, and thus avoiding pauses induced by JVM garbage collection and system memory swapping mechanisms, we ran this test on a different machine with 4 AMD Opteron 6168 (12-Core) CPUs and 128GB of memory, running a 64-bit Linux 2.6.32, and Java SE version 1.6.0_22 (Java HotSpot 64-Bit Server VM, build 17.1-b03). Figure 18 shows the results we obtained for total number of converted instances.

Figure 18 shows that the size of STMBench7's working set grows at a much smaller rate than the total number of live instances that the benchmark uses. This means that immediate upgrades take increasingly more time to convert the whole program state after installing an upgrade. We can already see this behaviour looking at the point at which the lines of the top plotting stop increasing. To make this result clear, we also registered the maximum latency when running the benchmark.

STMBench7 executes several operations that traverse a portion of all the existing objects. Increasing the number of instances that STMBench7 uses modifies the behaviour of such operations. To correctly assess the pause that immediate conversion imposes, we registered the maximum latency of operations whose behaviour remains constant despite increasing the total number of instances that the benchmark uses. Figure 19 shows those results.

As expected, the maximum latency that Figure 19 shows for immediate state conversion shows a proportional increase on the maximum operation latency proportional to the total number of instances. On the other hand, lazy state conversion remains nearly constant.

7.5 Discussion

DuST'M does not impose noticeable pauses on the application's execution to install an upgrade and perform the required program state conversion. The results that we showed in this section for maximum latency and throughput support

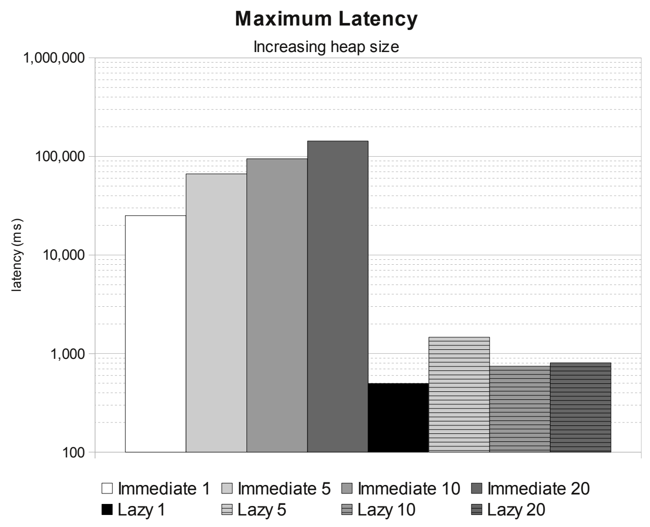

Figure 19. Maximum latency observed for any STMBench7 operation. To generate this chart, we executed the benchmark 10 times for 270 seconds, installing an upgrade at 90 seconds, and registered the maximum observed latency. The number at each legend line is the factor by which the total amount of a common type of instance was multiplied.

this claim, especially when comparing DuST'M's lazy conversion with immediate conversion.

However, DuST'M introduces a steady state overhead when compared to the original application execution. This overhead doubles the maximum observed latency and reduces the average throughput by a maximum of 38%.

In this section, we did not present values for the average latency. Although such results complement the maximum latency analysis, we choose not to perform such analysis for two reasons: (1) STMBench7 does not collect average latency values and (2) the average latency would yield similar results to the throughput, whose values STMBench7 already collects.

When performing program state conversion, we dedicated one thread to perform that conversion even in the case when the application ran with 4 threads. On our implementation, making the immediate state conversion multi-threaded is not trivial because each thread would need to know if a certain object is currently being converted by another thread to skip it. Given that JVSTM isolates all transactions, such information is difficult and expensive to access. A naive implementation could result in all 4 threads trying to convert the same instance, which would make 3 abort and 1 commit.

Furthermore, immediate upgrades blindly convert all program state, regardless of whether it belongs to the application's working set or not. As the number of live instances that an application uses grows, so does the time of immediate state conversion. Lazy state conversion, on the other hand, naturally starts by converting the important working set of the application. Our results show that if we increase the total number of live instances, the maximum operation latency using DuST'M remains nearly constant despite the increasing amount of live instances.

Moreover, this is not a question of implementing a more efficient immediate conversion mechanism: Even if we consider an hypothetical immediate state conversion technique that uses multiple threads and yields linear speedups, our results still display a nearly constant maximum latency.

8. Related Work

Since DSU's first approach [9], researchers have developed and implemented several techniques for supporting it. For instance, techniques such as *rolling upgrades* [5] take advantage of redundant hardware, already used for fault tolerance and load balancing, to enable the dynamic upgrade of a distributed system by restarting one node at a time. This approach, however, is not generally applicable and does not address any of the challenges of implementing a DSU system (in particular, it does not address the migration of the state between program versions).

So, in the following, we limit our discussion to the work that has more similarities to the approach that we describe in this paper, explaining the key differences. We start with approaches that address only the first challenge of upgrading a running Java program, and then move on to approaches that address also the state migration problem.

8.1 Upgrading the behaviour

The JVM itself provides support for HotSwap [1], a limited type of DSU that allows to modify method bodies only. Development time tools, such as IDEs and debuggers, use this feature to edit code that is running and show the new behaviour immediately. Kim et al. [13] introduce a binary refactoring technique that can be used with HotSwap to allow structural changes to a running program. They bypass the JVM's dispatch method to invoke newly added methods. Our work also uses binary code refactoring, but it strives to keep the original program semantics by delegating method dispatching to the JVM.

PROSE [16] uses HotSwap to perform runtime method patches to running Java programs without stopping them. It borrows its terms from Aspect Oriented Programming (AOP), and enables dynamically adding and removing aspects from a running program. Experimental results show that PROSE imposes low overhead on the original application.

Besides HotSwap, the JVM also provides support for multiple classloaders [14]. In theory, a system can use separate classloaders to isolate each version of the program. JavaRebel is a development time tool that operates at this level to load detected changes in the compiled code automatically. But, as JavaRebel is a proprietary upgrade system, developed and sold by Zero Turnaround[2], there is not much detail available on JavaRebel internals.

Würthinger et al. [22] present a modification to the JVM that provides support for installing arbitrary modifications to loaded classes. They attempt to apply the modifications at safe points (code positions when threads can be suspended safely to perform garbage-collection and rescheduling). In their upgrade system, upgrades may fail. For instance, when an upgrade deletes a method and a thread is suspended when it is executing a method that shall invoke the deleted method. After installing the upgrade, the invocation is performed and results in a runtime error, thus stopping the program. Nevertheless, such upgrade system is useful for debugging purposes. They install the new program version with overheads similar to a full-heap garbage collection and do not impose any noticeable overhead on steady-state execution.

8.2 Migrating the program state

All the systems mentioned so far support behaviour upgrading only. Furthermore, they do not provide support for migrating the program state among program versions. On the other hand, object data stores focus on migrating the program state. Dimitriev developed a tool [8] that provides support for PJama to migrate object stores offline, when no program is using them. He uses transform functions to migrate the objects between versions. O2 [10] is another object data store that supports lazy state migration. They do not use versions as we do. Instead, they require the developer to write forward and backward conversion code for each version.

Similarly to DuST'M, other upgrade systems also use an extra level of indirection The Java Distributed Runtime Update Management System [18] (JDRUMS) is an example. It provides support for adding new versions of existing classes and migrating the internal state of objects. It migrates objects lazily as the program control flow touches them. The handles JDrums uses are not transactional, and the authors do not specify how to solve the upgrade ordering problem (e.g., A refers to B, B is upgraded before A, when A is upgraded it reads an unexpected structure of B). JDrums uses a modified JVM that currently works in interpreted mode only, leading to very poor performance. Orso et al. [17] also introduce such extra levels of indirection, but forbid new fields and methods when upgrading a class.

Subramanian et al. [20] present JVolve, a JVM enhanced for supporting dynamic software upgrades. Programs upgraded using JVolve can perform a wide range of modifications, including adding or removing fields and methods, replacing existing ones, and changing type signatures. Like Würthinger et al. [22], JVolve stops threads at upgrade safe points before attempting to install the upgrade. An upgrade may fail if JVolve does not succeed at stopping threads at upgrade safe points during a reasonable time period. If an upgrade fails, JVolve does not install it and the program keeps executing in the old version. JVolve does not support atomic software upgrades. Developers need to blacklist methods that have data dependences with other methods to avoid installing the upgrade in between the two method invocations. JVolve supports custom transformer methods to convert the objects between versions, but such conversion code can access the new version of the program only. JVolve piggybacks on a forced full heap GC to convert the program state immediately, converting each live object. The authors provide no information on whether the immediate conversion is multithreaded or not. It imposes no overhead in steady state execution.

8.3 Atomic Dynamic Software Upgrades

The work that resembles DuST'M more closely is the one that Boyapati et al. present in [4]. They introduce the atomic lazy upgrade semantics that DuST'M uses, which we explain in Section 3. However, their system targets dynamic upgrades on a persistent object data store, whereas DuST'M targets dynamic upgrades on a running program.

Despite having similar upgrade semantics, there are key differences between the two upgrade systems. In their implementation, they rely on object encapsulation to generate a correct order in which the conversion code executes. If an object A encapsulates another object B, then all accesses that the application makes to B must first pass through A. Thus, the conversion code of A knows that it can access B in its old version because it is impossible to touch B first (and trigger its conversion) circumventing A. Using this knowledge, they avoid keeping versions for object B.

The developer must write the persistent objects using ownership types [3] to ensure sound object encapsulation, and thus avoid to keep versions for all objects. However, there are corner-cases where the encapsulation mechanism fails. Furthermore, their system does not strictly require the developer to write the persistent objects using ownership types. To solve such cases, they fall back to versioning. They keep old instances of persistent objects after converting them so that the conversion code of other persistent objects can still access the (correct) old version. This is very similar to what DuST'M does: DuST'M uses VBoxes to keep versions for every instance belonging to an upgradable type.

Indeed, their approach appears more efficient than ours because it avoids versioning unless absolutely necessary. However, there is a key difference on the underlying upgrade model that separates DuST'M from Boyapati et al's work: Their system targets persistent object data stores, whereas DuST'M targets a Java program running on an unmodified JVM. As a consequence, their work does not support non-upgradable objects.

In a persistent object store, it makes sense to support the upgrade of all stored objects. However, it is impossible to replace all object references inside the JVM by handles without modifying the JVM itself. Furthermore, a persistent object data store only keeps objects written by the developer.

[2] http://www.zeroturnaround.com/javarebel/

Thus, the developer knows how to write conversion code for all persistent objects. The same does not apply to the JVM because it uses several object that the developer is not interested in writting conversion code for them (he may not even be aware of their existence). This drives the need for DuST'M's upgrade model to support non-upgradable objects.

Therefore DuST'M differs from Boyapati et al's work in the sense that it supports non-upgradable objects and provides semantics that describe what happens when a conversion transaction that runs in the logical past accesses a non-upgradable object that was modified since the upgrade was installed.

8.4 Other approaches

There are yet other approaches to dynamic software upgrades. CLOS [19] uses a meta-object protocol to redefine classes, migrating the program state using custom code. The moment when such conversion takes place (immediately or lazily) is left implementation dependent. Bierman et al. [2] present UpgradeJ, an extension to Java that provides language support for dynamically upgrading an application. Using UpgradeJ, developers manipulate classes using explicit version numbers. Although such an approach is considered expressive enough for most types of upgrades [21], manipulating explicit version numbers is an heavy burden for the developer writing the application. Besides, there is no implementation of UpgradeJ to this date.

9. Conclusion

We presented DuST'M, a system that allows making upgrades to a running Java application without stopping it. Using DuST'M, the developer is able to specify how to migrate the program state between versions using type-safe and modular code. DuST'M uses this custom conversion code to migrate each instance from one version to the following one. Although it performs such conversion lazily, DuST'M relies on a versioned STM, JVSTM, to expose simple atomic immediate upgrade semantics to the developers.

DuST'M is implemented as a post-processor and runtime library. Thus, it does not modify the underlying JVM, nor does it depend on any specific implementation of any JVM. The post-processor takes as input the bytecode that the Java compiler generates, and performs binary code refactoring introducing an extra level of indirection. Developers can use IDEs, profilers, and debuggers to write a mature program version before turning it into a new program upgrade, using the post-processor.

Using an extra level of indirection, DuST'M supports a wide range of program modifications between program versions. Such modifications include, with few restrictions, changing the position of a class in the hierarchy.

Experimental results show that converting the program state lazily as the natural flow of the program touches it has important advantages over converting the program state immediately when the upgrade is installed: Smaller pause on the application's execution and faster steady-state throughput recovery after installing an upgrade.

The good properties of lazy program state conversion happens because, as opposed to immediate state conversion, it does not convert all program state blindly when the upgrade is installed. It starts by converting the application's working set and then moves on to parts of the state that are used infrequently. Meanwhile, the upgraded program is running and performing useful work.

This allows lazy upgrades to scale to applications with very large heaps, unlike immediate upgrades, assuming that the size of each operation's working set is much smaller than the size of the full application's heap. If we consider applications whose sheer number of instances makes immediate program state conversion prohibitive, we expect DuST'M to be able to provide the same behaviour that we report in our experimental evaluation.

When compared to the unmodified application, DuST'M doubled the maximum latency per operation and introduced a throughput overhead ranging from 18% to 37%.

Regardless of whether the presented implementation is acceptable in terms of performance, we believe that the idea of relying on a multi-versioned STM to support dynamic software upgrades is worth pursuing and developing. We hope that our initial work spurs further research in this area.

9.1 Open Issues and Future Work

The post-processor performs several modifications to the bytecode that may break applications that use reflection. Although such modifications do not change the semantics of the application, they change the class structure by, for instance, adding parameters to methods and changing the names of methods and classes. This is a known issue related with systems that perform binary code refactoring.

Current JVMs are free to load classes to a memory space that is not garbage collected (`permgen` in Oracle's HotSpot JVM). Such a scenario limits the number of upgrades that DuST'M can perform because DuST'M considers each class upgrade as a new class. As a result, the system may run out of memory after installing some upgrades.

This paper suggests a new way for the developer to specify how to convert the program state between program versions. We did not, however, test the expressiveness of the conversion code. We plan to explore this path in future work, as well as ways to decrease the overheads introduced.

In its current implementation, DuST'M does not provide a general solution to garbage collect old values from the handles. However, please note that the benchmark that we presented fully loads the CPU, which is uncommon in typical applications that interact with users. We plan to create background threads, which run with less priority, to "push" the world forward, converting instances from older program versions. When all instances that belong to an old program version have been converted, DuST'M can safely consider them as garbage.

References

[1] Java(tm) platform debugger architecture. http://java.sun.com/javase/6/docs/technotes/guides/jpda/.

[2] G. Bierman, M. Parkinson, and J. Noble. Upgradej: Incremental typechecking for class upgrades. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 235–259. Springer-Verlag, 2008.

[3] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '03, pages 213–223, New York, NY, USA, 2003. ACM.

[4] R. Boyapati, B. Liskov, L. Shrira, C. hue Moh, and S. Richman. Lazy modular upgrades in persistent object stores. In *In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA*, pages 403–417, 2003.

[5] E. A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, 2001.

[6] E. Bruneton, R. Lenglet, and T. Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[7] J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 63(2):172–185, 2006.

[8] M. Dimitriev and M. P. Atkinson. Evolutionary data conversion in the pjama persistent language. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 25–36, London, UK, 1999. Springer-Verlag.

[9] R. S. Fabry. How to design a system in which modules can be changed on the fly. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 470–476, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[10] F. Ferrandina, T. Meyer, R. Zicari, G. Ferran, and J. Madec. Schema and database evolution in the o2 object database system. In *VLDB '95: Proceedings of the 21th International Conference on Very Large Data Bases*, pages 170–181, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[11] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 175–184, New York, NY, USA, 2008. ACM.

[12] R. Guerraoui, M. Kapalka, and J. Vitek. Stmbench7: a benchmark for software transactional memory. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 315–324. ACM, 2007.

[13] D. K. Kim and E. Tilevich. Overcoming jvm hotswap constraints via binary rewriting. In *In First ACM Workshop on Hot Topics in Software Upgrades*, 2008.

[14] S. Liang and G. Bracha. Dynamic class loading in the java tm virtual machine. In *In Proc. 13th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98), volume 33, number 10 of ACM SIGPLAN Notices*, pages 36–44. ACM Press, 1998.

[15] S. Monk and I. Sommerville. Schema evolution in oodbs using class versioning. *SIGMOD Rec.*, 22(3):16–22, 1993.

[16] A. Nicoara, G. Alonso, and T. Roscoe. Controlled, systematic, and efficient code replacement for running java programs. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 233–246, New York, NY, USA, 2008. ACM.

[17] A. Orso, A. Rao, and M. J. Harrold. A technique for dynamic updating of java software. In *ICSM '02: Proceedings of the International Conference on Software Maintenance (ICSM'02)*, page 649, Washington, DC, USA, 2002. IEEE Computer Society.

[18] T. Ritzau and J. Andersson. Dynamic deployment of Java applications. In *Java for Embedded Systems Workshop*, London, May 2000.

[19] G. Steele. *Common Lisp the Language*. Digital Press, 2nd edition, June 1990.

[20] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic software updates: a vm-centric approach. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 1–12. ACM, 2009.

[21] E. Tempero, G. Bierman, J. Noble, and M. Parkinson. From java to upgradej: an empirical study. In *HotSWUp '08: Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades*, pages 1–5, New York, NY, USA, 2008. ACM.

[22] T. Würthinger, C. Wimmer, and L. Stadler. Dynamic code evolution for java. In *PPPJ '10: Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, pages 10–19, New York, NY, USA, 2010. ACM.

[23] R. Zicari. A framework for schema updates in an object-oriented database system. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 2–13, Washington, DC, USA, 1991. IEEE Computer Society.