Low-disruptive Dynamic Updating of Java Applications

Tianxiao Gu, Chun Cao*, Chang Xu, Xiaoxing Ma*, Linghao Zhang, Jian Lü

State Key Laboratory for Novel Software Technology at Nanjing University, Nanjing, China
Department of Computer Science and Technology at Nanjing University, Nanjing, China

Abstract

Context: In-use software systems are destined to change in order to fix bugs or add new features. Shutting down a running system before updating it is a normal practice, but the service unavailability can be annoying and sometimes unacceptable. Dynamic software updating (DSU) migrates a running software system to a new version without stopping it. State-of-the-art Java DSU systems are unsatisfactory as they may cause a non-negligible system pause during updating.
Objective: In this paper we present Javelus, a HotSpot VM-based Java DSU system with very short pausing time.
Method: Instead of updating everything at once when the running application is suspended, Javelus only updates the changed code during the suspension, and migrates stale objects on-demand after the application is resumed. With a careful design this lazy approach neither sacrifices the update flexibility nor introduces unnecessary object validity checks or access indirections.
Results: Evaluation experiments show that Javelus can reduce the updating pausing time by one to two orders of magnitude without introducing observable overheads before and after the dynamic updating.
Conclusion: Our experience with Javelus indicates that low-disruptive and type-safe dynamic updating of Java applications can be practically achieved with a lazy updating approach.

Keywords: Dynamic Software Updating; JVM; Lazy Updating; Low Disruption;

1. Introduction

Software is always evolving to new versions in order to eliminate defects and meet new requirements. Normally, software updating should follow the well-known shutdown-patch-restart scheme to get the new version working. For

*Corresponding authors
Email addresses: tianxiao.gu@gmail.com (Tianxiao Gu), caochun@nju.edu.cn (Chun Cao), changxu@nju.edu.cn (Chang Xu), xxm@nju.edu.cn (Xiaoxing Ma), zlh.nju@gmail.com (Linghao Zhang), lj@nju.edu.cn (Jian Lü)

mission critical systems with high availability requirements, such as those used in financial truncation processing, transportation management and medical life supporting, service downtime is unacceptable or prohibitively expensive. Even for those everyday software systems such as Microsoft Windows and Office, the disruption of software upgrading could be at least annoying. In addition, when debugging a complex software system, frequent restarts for code modification could be tedious and time-consuming. Dynamic Software Updating (DSU) can alleviate these problems by updating a running software system without having to stop and restart it.

In recent years, DSU has been extensively studied [16, 7, 12, 18, 30], although the idea of dynamically modifying running programs is not new – it can be traced back to dynamic linking, which was systematically supported in MULTICS [8]. Specifically, for updating Java programs at runtime, it is natural to extend the Java Virtual Machine (JVM) with DSU support [10, 31, 33]. In JVM a Java program at runtime consists of a set of classes, a set of heap objects instantiated from the classes and a set of thread stacks storing frames of active methods. To update a running program, one must replace the classes with their new versions and transform their objects accordingly with user provided or default *transformers*. To keep system consistency, DSU systems usually suspend the execution and apply updates when no method-to-update is active.

Most of existing proposals of DSU for Java take an eager approach. They locate *stale objects* (i.e., objects instantiated from old versioned classes) and update them all at once [31, 33]. Stale objects are detected by traversing the whole heap. Objects with increased size cannot be updated in-place, so all references to them, including stack variables and object fields, must be adjusted to point to the new locations. Generally, these tasks are fulfilled by exploiting the garbage collection facilities provided by the virtual machine. The garbage collector used here must be forced into the stop-the-world mode and make a full-heap collection, even if itself were a concurrent and incremental collector. The program is unable to proceed during this process, and the pausing time can be seconds long. Although this time is usually much shorter than that of doing shutdown-and-restart, it is still not acceptable for many applications that require very short response time, such as high-performance Web servers and interactive applications.

We take a lazy approach to DSU to cut down the pausing time. It is motivated by following two observations. First, for most updates, as only a small portion of all objects are affected, traversing the whole heap is unnecessary. Second, many updated objects would not be used immediately or even not used any more. Our lazy approach updates objects on-demand. Each stale object is updated on the first access to it. In this way, we eliminate expensive whole-heap traversing, avoid updating useless objects and disperse the effort of object transformation to later program execution.

However, lazy object updating is not without challenges. A naïve implementation would bring unwanted overheads at execution time because (1) access to objects would need to be trapped with validity checks and (2) references to objects whose size increased have to be made indirect. In addition, in a lazy

DSU, object transformers are invoked concurrently with application methods. So it could be extremely tricky to program these transformers correctly unless we provide a reasonable semantic guarantee on the updating process.

This paper presents the design and implementation of Javelus[1], a lazy DSU system for Java based on the OpenJDK Java HotSpot VM[2]. It supports arbitrary changes to Java classes and provides an easy but powerful model for programmers to write transformer. Its overhead during steady-state execution is negligible. In addition, Javelus guarantees that no stale code will run after the updating is triggered and no stale object will be accessed by any updated code. With a carefully optimized object validity checking strategy and a novel object allocation model, Javelus achieves very short pausing time without sacrificing update flexibility and system efficiency. In comparison with eager approaches, the pausing time can be tens to hundreds times shorter.

The main contributions of this work are an efficient lazy object updating mechanism and an implementation of this updating mechanism on an industry-strength Java HotSpot VM. The work was initially reported in [14], and the current paper is a rewritten that gives a more complete and clearer presentation of the work. Critical enabling techniques such as how we use versioned class hierarchies to ensure type safety and how we deal with type narrowing are presented for the first time. In addition, we improved our evaluation of Javelus by adding new subjects and making direct comparisons between lazy and eager approaches.

The rest of this paper is organized as follows. We first discuss the general requirements of and approaches to dynamic software updating in Section 2 and then give an overview of Javelus in Section 3. Next three sections are dedicated to how to prepare dynamic updates, how Javelus dynamically updates code and how it lazily update stale objects, respectively. In Section 7, we evaluate Javelus with a micro benchmark and also real version updates of Tomcat and H2 servers. We summarize related work in Section 8 before conclude the paper in Section 9.

2. Dynamic Updating: Eager versus Lazy

A Java program statically consists of a set of classes with inheritance and association relationships between them. Dynamically, in addition to the reified representation of these classes (called *metadata*), a runtime image of a running program also contains a stack (or stacks) of active method frames and a set of objects instantiated from the classes. To update a running program, one must not only refresh the class metadata to the new version but also rebuild the whole runtime image satisfying the following properties:

- *Type safety*: The program can continue to execute without type errors.

[1]The source code of Javelus and the applications used to evaluate it can be found at http://artemisprojects.org/projects/javelus

[2]http://openjdk.java.net

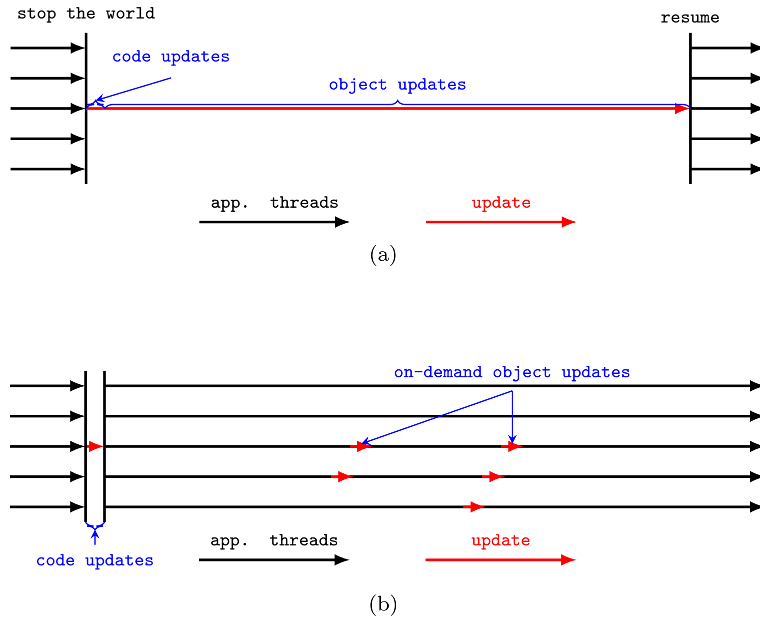

Figure 1: Eager versus lazy updating. (a)Time-line of eager updating; (b)Time-line of lazy updating.

- *Semantic continuity*: The states in the current image are preserved as much as possible.

Generally, it is impossible to automatically ensure the semantic correctness of a dynamically updated program [15]. It is DSU users' responsibility to maintain the semantic correctness in addition to type safety and semantic continuity.

A practical DSU system should also fulfill the following requirements [16, 31]:

- *Efficiency*: There should be no overhead before and after the updating.

- *Flexibility*: It should allow most kinds of changes happened in real-life software evolution.

- *Low disruption*: The interruption of service caused by the dynamic updating should be minimized.

- *Timeliness*: The update should be applied as soon as possible.

Dynamic updating can be done eagerly or lazily. As shown in Fig. 1, in an eager approach all updating work is carried out when the program is suspended at some safe point. In a lazy approach, the program is allowed to resume immediately after refreshing the class metadata and some stale objects in the heap are updated later in an on-demand way. Eager approaches were believed to be more efficient but, meanwhile, more disruptive than lazy ones. However,

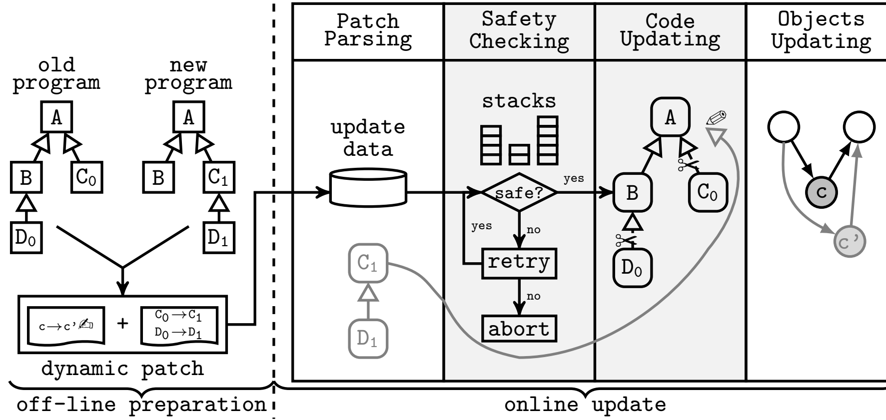

Figure 2: System Overview

lazy approaches can create overheads during program execution in that the program must be instrumented to trigger on-demand updating of stale objects and references. Besides, lazy approaches often use indirections to update objects with increased sizes, and require additional spaces to store *forward pointers*. Hence, the efficiency and space utilization will be degraded.

3. Javelus: Overview

In Javelus, the whole dynamic updating process goes through two phases: off-line preparation and online updating. Fig. 2 gives an overview of Javelus.

At the preparation phase, Javelus uses an off-line tool to compare the class files of the old and the new versions of a program and generates a *dynamic patch*. A dynamic patch consists of a summary of class changes as well as a set of *object transformers*. An object transformer upgrades the objects of an old version class to the corresponding new one[3]. Developers need to manually fill the transformation logic in the templates if the default logic is inappropriate.

After all transformers have been prepared, the dynamic patch is ready to be delivered to dynamic updating. Developers initiate a dynamic updating request by sending the patch to Javelus via a provided invoking interface. The whole online updating phase involves four steps: dynamic patch loading, safe point checking, code updating and objects updating.

After receiving an updating request, Javelus first loads in the dynamic patch and prepares metadata for the new version classes and corresponding object transformers. The new version classes are put into a *temporal dictionary* for later querying. The temporal dictionary is separated from the *system dictionary* where classes are loaded during normal program execution.

[3]Actually there also *class transformers* responsible for updating static fields of classes.

Next, Javelus waits for the program to be trapped into a VM safe point. A VM safe point is a point during program execution at which all application threads are suspended[4]. At the VM safe point, Javelus checks all stack frames to determine the safety of the following updating phases. If there is no active *restricted method*, i.e., the method whose frame is inconsistent with the new code, the VM safe point is indeed a DSU safe point [16] and it is safe to start updating code with no need to update stacks. Otherwise, Javelus waits for a certain time interval and retry, or just abort the updating process after a number of failed retries.

Then, Javelus updates all changed classes at the DSU safe point. All expired code of these classes, including those compiled and inlined elsewhere by the Just-In-Time (JIT) compiler, will be refreshed according to the new version of classes.

Finally, the program is resumed after all code has been updated. Javelus intercepts all access to potentially stale objects during the remaining execution. Once a stale object is detected, the thread will be trapped into an object updating routine. Any access from other threads to the object will be blocked until the routine has finished. To avoid costly pointer updating, Javelus first attempts to update the stale object in-place. If the size of the object is increased, Javelus will create a mixed object for it. A mixed object uses two physical objects to simulate one logical object and only the newly added fields have to be accessed indirectly. Eventually these two physical objects will be merged into one object after a compacting garbage collection. The routine will also initialize the new object with the state of the stale object by calling the corresponding object transformer.

4. Dynamic Patch Generation

Javelus uses the dynamic patch to specify what has changed and how to deploy them. A dynamic patch is composed of a sequence of *changed classes*. It may cover more changed classes than a static patch. For static updating, we only need consider one class's own implementation individually without considering those of its super-classes, as changes of super-classes will naturally propagate to sub-classes which have already been loaded during execution. However, in the case of dynamic updating, changes from super-classes will not propagate to sub-classes if we pay no attention to the latter. This is because sub-classes may have been loaded before changed super-classes are updated. Objects instantiated by sub-classes contain fields and methods inherited from super-classes. Therefore, we have to update such sub-classes as well and treat them as directly changed classes.

To generate the dynamic patch, Javelus first identifies a set of changed classes and determine their updating types at class-level and class member-level (meth-

[4]JVM now can exclusively carry out its internal work including, among others, critical GC operations such as moving objects around.

ods and fields). A changed class has an *old* version and a *new* version. A *deleted* class can be viewed as a changed class without a new version and an *added* class can be viewed as a changed class without an old version. Then, Javelus generates a set of transformer templates for *type changed classes* (this will be explained in section 4.2). After transformers have been prepared by DSU users, the dynamic patch can be delivered to dynamic updating.

4.1. Identifying Changed Classes

Javelus maps classes between two versions of programs by their identifiers. Every class has a unique identifier in the scope of a single version. At development time, the identifier is the qualified class name. At runtime, the identifier is composed of a class loader and the qualified name [20]. To map classes at runtime, Javelus defines a name space for a class loader in the dynamic patch. Changed classes loaded by the class loader are grouped into the corresponding name space. Javelus also has provided an interface for programmers to resolve the class loader of each name space at runtime. By comparing the identifiers, Javelus can easily figure out *deleted* and *added* classes. A class that has changed its identifier will result in a deleted class and an added class in the dynamic patch. For mapped classes, Javelus further figures out *changed* classes and determine their updating types.

More specific updating types of the classes are investigated against their members. A class member, i.e., a field or a method, is identified by its name and descriptor in the scope of a class [21]. A class member is marked as deleted if its *identifier* or *modifier* has changed. For mapped methods, Javelus further compares and analyzes their method bodies to determine their updating types. There are two method-updating types: body changed and indirectly changed. Body changed methods have changed their byte-code. Indirectly changed methods have not change their byte-code but have links to old class members.

Thus, Javelus classifies changed classes into five updating types in a dynamic patch, as shown in Table 1. They are,

- *Added*: According to whether the added classes are required by loading of other new classes, the added classes will be *defined* into the VM eagerly by the Javelus updating subsystem or lazily by executing code. In the Java HotSpot VM, *defining* a class means adding the class metadata into the system dictionary and then the class can be queried and used by other classes.

- *Deleted*: The deleted classes will be naturally collected by garbage collection once all objects instantiated by them are collected. Javelus *undefines* them and makes their methods obsolete and non-executable. *Undefining* a class means removing the class metadata from the system dictionary and then the class must not be used by any other class any more. More details will be discussed in the section 6.1.3.

- *Type Changed*: There are many reasons why a class is classified as type-changed, such as inheriting a type changed class, implementing a type

Table 1: Changed classes listed by updating types and updating actions.

Updating Type		**Updating Action**
added		*define*
deleted		*undefine*
changed	*type changed*	*redefine*
	method body changed	*swap*
	indirectly changed	*relink*

changed interface, adding or removing class members, adding or removing super types. Javelus will *redefine* the type changed classes, that is to *undefine* the old version and then *define* the new version again. As types have changed, objects instantiated by old versions must be updated.

- *Method Body Changed*: Method body changed classes only contain changes of method byte-code in class files. Javelus will load new versions and swap new methods of new versions into old versions. Redefining these classes is unnecessary. So, their types remain the same at runtime. Objects instantiated by these classes are assumed not to be updated. They may only have semantic changes, which can only be determined by programmers.

- *Indirectly Changed*: Indirectly changed classes have no changes in class files, nor are their super-classes type-changed. They just contains indirectly changed methods. Javelus will remove links to old class members and new links will be built again during execution.

4.2. Default Transformation and Custom Transformers

Once a class is identified as a type changed class and to be updated, all its objects become stale and need to be transformed to the new type. In Javelus the transformation of a staled object is triggered by the first access to it after the class updating. Such transformation includes two phrases: first, reallocating the space for the new object; and second, assigning valid values for fields in the new object. The first phrase is done internally by Javelus. Although some deliberated rearrangements of the layout of fields may happen here (see Section 6.3 for details), these rearrangements are completely transparent to application code. For the second phrase, Javelus performs default transformation, and invokes custom transformers if there are any, to convert the state of the stale object into a proper state of the new object, so that the application can continue correctly in the new version.

Javelus' default transformation logic just copies values of unchanged fields from the stale object to the new object and assign a default value to each newly added field. Here, the default value is based on the type of the field, e.g., `0` for an integer field and `null` for a reference field. Fig. 3(c) uses a piece of Java code

to show what the default transformer actually does, where `v0` refers to a stale object and `v1` refers to the corresponding new object.

By providing default transformation, Javelus rids programmers of writing tedious routine transformation code. In our experiences, default transformation is often able to successfully update real applications. However, sometimes default transformation is insufficient to establish a correct state for the new object. For the example shown in Fig. 3(d), the new field `forwardAddresses` must be assigned with an array of `EmailAddress` if the old `forwardAddresses` in the stale object is not null. This kind of custom transformation logic is application dependent and must be provided by users with *custom transformers*. Javelus automatically generates *transformer templates* for type changed classed to facilitate the development of custom transformers.

```
1 class User{
2   String userName;
3   String password;
4   String[] forwardAddresses;
5 }
```

(a) `v0`

```
1 class User{
2   String userName;
3   String password;
4   EmailAddress[] forwardAddresses;
5 }
```

(b) `v1`

```
1 void defaultTransformer(User_v0 v0, User v1){
2     v1.username = v0.username;
3     v1.password = v0.password;
4     v1.forwardAddresses = null;
5 }
```

(c) `a transformer template`

```
1 class User{
2   String userName;
3   String password;
4   EmailAddress[] forwardAddresses;
5   public void updateObject(
6     @OldField(name="forwardAddresses",desc="[String;",clazz="User")
7     String[] p1){
8     final int length = p1.length;
9     forwardAddresses = new EmailAddress[length];
10    for(int i=0;i<length;i++){
11      forwardAddresses[i] = new EmailAddress(p1[i]);
12    }
13  }
14  public static void updateClass(){}
15 }
```

(d) `a transformer template`

Figure 3: An example of transformer templates.

A transformer template of a type changed class contains four essential parts described in the following.

Declared fields A template class re-declares all fields of the new version, so that its object transformer method can access them freely. Deleted fields are passed to the transformer method as annotated parameters.

Object transformation method The object transformation logic is specified in a method named as `updateObject`. Programming the object transfor-

mation logic is quite similar to programming a constructor. In the body of this method, `this` is referred to the to-be-transformed new object, and the value of each deleted field is provided by Javelus through an annotated method parameter.

Class transformation method The class transformation method, named as `updateClass`, is similar to the object transformation method, except that it works on static fields. Again, the value of each deleted static filed is provided as an annotated method parameter.

Super Types When the super-class of the current type changed class is also type-changed, the corresponding transformer class for the super-class needs to be inherited as a super-class. If there is an object transformation method defined for the super-class, Javelus automatically calls it at the beginning of the object transformation method for the current class.

Transformer developers are quite free in writing the transformation code in Java language. However, they must be aware that class updating can happen at any DSU safe points and object transformation is triggered on-demand after class updating. To ensure semantic correctness, a deep understanding of the application logic and the difference between the two versions are necessary. Fortunately, in most cases, the difference is quite small (otherwise it not meaningful to update *dynamically*) and the transformation logic is straightforward. For the example shown in Fig 3(d), the semantics of the class is not changed but the representation of the field `forwardAddresses` is changed from an array of strings to an array of wrapper classes.

Programming transformers could be tricky if the application logic is significant changed. Two rules of thumb are (1) never use locks in transformation code to avoid potential deadlocks, and (2) do not access values other than the local state of the current object unless you know what you are doing.

An effective strategy to simplify the transformers is to restrict *when* the DSU happens to those periods during which the behavior of the old and the new versions "converges". To restrict the time points of DSU, one can manually include some methods in the set of *restricted methods* (see the next Section) so that DSU will not happen when these methods are active. However, one should be aware that this restriction could reduce the timeliness of dynamic updating.

5. Updating Code

5.1. Reaching DSU Safe Point

The running program needs to reach a DSU safe point to update. DSU safe points are stricter VM safe points where none of the application thread stacks contains any frame of active restricted methods. There are four types of restricted methods: 1) deleted and body changed methods; 2) methods pointing to an object instantiated by a type narrowed class (see Section 6.1.2); 3) methods that have been overridden and their receivers are stale objects as a new method

will be dispatched after the receivers have been updated; 4) methods that have been blacklisted by DSU users.

Note that for those methods of changed classes whose byte-code is unchanged, Javelus does not include them as restricted methods but automatically repair corresponding method frames in stacks. In this way Javelus can achieve DSU safe point despite the activeness of these methods. This technique greatly improves the timeliness of dynamic updating.

If no restricted method is found, the VM safe point is indeed a DSU safe point and Javelus continues to update classes. Otherwise, Javelus inserts a *return barrier* [31] in the frame of the deepest restricted method for each thread and interrupts the updating process. Once the restricted method has returned, the updating process is resumed by the return barrier. After a certain number of retries fail, the updating process is eventually aborted.

5.2. Updating Classes

Javelus reads the dynamic patch and loads all new classes and interfaces in an order that supertypes are loaded first. For method body changed classes, their types remain unchanged and their objects are valid in new version. Only the method body in corresponding class metadata must be updated. Note that we cannot simply remove metadata objects of old classes and create corresponding new metadata objects. This is because the address of a classs metadata object has been used in every object of this class as its type identifier, and the identifier must be kept the same before and after the updating to avoid unnecessary object updating. Javelus uses a technique that swaps the contents between old and new metadata objects to keep such type identifiers the same. For type changed classes, old versions are undefined to keep type consistency and new versions are defined. Values of unchanged static fields are copied from old versions to the corresponding new versions by a default transformation.

For new defined classes, the class initialization method is replaced by the class transformer. Javelus leaves new classes in the uninitialized state and class transformers are executed for initialization at the first access to the class during the remaining execution.

Javelus repairs frames of active unchanged methods in the old classes. All compiled methods are forced to be recompiled. For active compiled methods, the Java HotSpot VM interprets them after de-optimization. After code updating, the VM turns into running in the interpreted mode for a moment.

6. Updating Objects

The program is resumed immediately once the class updating is finished. Stale objects will be updated in an on-demand way during the remaining execution. Javelus must guarantee that

- no stale object would be used as an instance of the new version of the class without being upgraded first, and upgraded objects should inherit

their original roles[5],

- no old code would be executed anymore, and
- no type error would be caused by dynamic updating.

To this end, Javelus has to insert additional checks for all potential violations of these properties. At the same time, the amount of these checks must be kept minimal to reduce overhead. Javelus first makes a safe estimation of the scope of references to stale objects by analyzing their types affected by the update (Section 6.1). This analysis is done at the offline preparation phase. At the online updating phase Javelus also optimizes the object validity checking in various aspects (Section 6.2).

To reduce the performance penalty caused by size-increased objects, Javelus uses a novel object allocation model that realizes in-place upgrading where access indirections are kept minimal (Section 6.3). The model is fully transparent to program code and is compatible to the Java HotSpot VM.

6.1. Affected types

Objects in the Java HotSpot VM are accessed directly through references. A stale object would not cause any problem unless it is referenced. A reference has a static type declared in the code, and a dynamic type determined by the runtime object it points to. If the dynamic type is *not* a subtype[6] of the static type, dereferencing will cause a type error.

In Javelus, the static type of a reference is updated when the corresponding class is refreshed, but the dynamic type could be invalid if the object it refers were a stale one whose defining class is not a subtype of the refreshed class anymore. To avoid blindly checking access to all objects, Javelus identifies a set of *affected types* and only intercepts dereferencing of references statically typed with those affected types. A type is affected if a reference of this type points to a stale object and we call this reference *mismatched*.

6.1.1. Versioned Class Hierarchy

A versioned class hierarchy is a combination of class hierarchies of different versions of programs[7]. Fig. 4(c) shows a versioned class hierarchy that contains classes from two versions, namely `v0` (see Fig. 4(a)) and `v1` (see Fig. 4(b)). Every class in the class hierarchy is annotated with the version it was born in (i.e., defined) and the version it died (i.e., be undefined). For example, B_0^1 is defined in version `v0`, which is indicated by the subscript, and undefined in version `v1`, which is indicated by the superscript.

---

[5]This means that *all* references to a stale object should be redirected to the same upgraded object.

[6]Note that the subtype relation is reflexive.

[7]We omit interfaces in the versioned class hierarchy, as in our implementation, interface types never violate the type-safety (see Section 6.2.1).

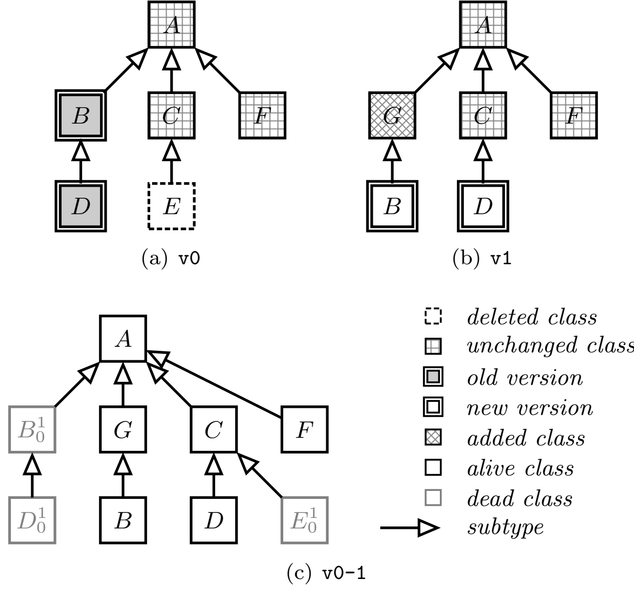

Figure 4: (a) Class hierarchy of `v0`; (b) Class hierarchy of `v1`; (c) Versioned class hierarchy from `v0` to `v1`.

When a class is said to be *alive* or *dead*, it is with respect to the *current* version after possibly a series of updating. Since dead classes are not used anymore in the updated code, affected types must be alive classes.

A newly updated class must be counted as affected, together with all its super-classes in the new version. In the above example, B, G and A (because of B_0^1), and D, C and A (because of D_0^1) are affected. On the other hand, alive super-classes of the old version must also be counted in. For example, A (because of B_0^1 and D_0^1) is affected.

To mark that a stale object may be pointed to by references of the new version, we introduce the *super-class-by-updating* relation between the old and new version. In short, we name this relation as α, e.g., $B_0^1 \xrightarrow{\alpha} B$. Note that the tail of an α edge must be an old version. Super-classes reachable by following a path contains a α edge are called *induced super-classes*. For example, as shown in Fig. 5, G is an induced super-class of B_0^1.

All alive classes reachable from the old version following any subtype and α edge are marked as affected. In the example in Fig. 5, only F is unreachable from an old version and is not affected. Any other class is affected.

6.1.2. Type Narrowing

Type errors could still occur even if every stale object had been updated. For example, suppose that there is a reference statically typed in B. It may point to a stale object of D_0^1 before updating and then point to an object of D after updating. However, D is not a subtype of B in the current version. Besides, a

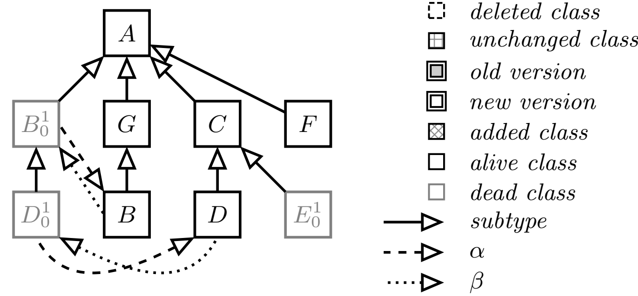

Figure 5: Versioned class hierarchy with α and β edges.

reference value typed in B can be passed to a reference typed in G without any type checking. This makes class G possibly point to an object instantiated by D, although a reference typed in G cannot point to an object instantiated by D before updating.

In the aforementioned example, the most notable distinction is that not all the supertypes (i.e., super-classes and implemented interfaces) of D_0^1, including those induced by α edges, is part of supertypes of D. We say D is type narrowed, using the definition in [34]: *a type is narrowed if one of its original supertypes is no longer a supertype in the new version of the type.*

We have to amend the definition in [34] as Javelus, as we have pointed out, uses a lazy approach. First, we introduce the *sub-class-by-updating* relation between the new and old versions. In short, we name this relation as β, e.g., $D \xrightarrow{\beta} D_0^1$. Note that the tail of the β edge must be a new version. A new versioned class is *narrowed* in Javelus if there exists an alive type which can be reached from the new version *only* in a path beginning with a β edge. References statically typed with these alive types are subject to *type narrowing checking*.

6.1.3. Deleted Classes

Stale objects instantiated by deleted classes should not be used any more after the deleted classes have been undefined. However, they can still be accessed via references typed in alive super-classes and methods of deleted classes may be executed even after code have been updated. For example, an object instantiated by E_0^1 may be pointed to by a reference typed in C.

Currently, Javelus treats such stale objects as they were instantiated by alive super-classes. Conceptually, one can think of these objects being updated to new versioned classes identical to the alive super-classes. We believe this is a reasonable choice to preserve semantic continuity. For example, the object instantiated by E_0^1 is used as an instance of C. Users can change this behavior by defining specific object transformers for deleted classes. For method invocations dynamically dispatched to the method defined in deleted classes, Javelus would redirect to the corresponding method in super-classes.

```
1 class Main{// main class
2   void main(){
3     Shape shape = new Circle();
4     /* dynamic updating happened */
5     shape.draw();}}
6 abstract class Shape{// super class
7   abstract void draw();
8   abstract void draw(Canvas canvas)
        ;}
9 class Circle{// old version
10   void draw(){
11     draw(Canvas.getDefault());}
12   void draw(Canvas canvas){
13   /*code draw a circle*/}}
14 class Circle{// new version
15   Canvas canvas;
16   void draw(){draw(this.canvas);}
17   void draw(Canvas canvas){
18   /*code draw a circle*/}}
```

Figure 6: An example for common super-classes.

6.1.4. Common Super Classes

Intercepting all dereferencing of references of affected types could still be expensive, and should be avoided unless it is necessary. We have observed that old and new versions of a class often have unchanged common super-classes. For example, the `Object` class in Java is the *common super-class* (CSC) of all other classes. Accessing stale objects through references typed in common super-classes never causes type errors. Fields in CSC are assumed to be valid by default unless users explicitly include them in object transformers. So we can avoid intercepting accesses to these fields even though the object is stale.

However, semantic problems may arise when a virtual method of the common super-classes is called. For example, as shown in Fig. 6, suppose that the dynamic updating happens at line 4. Then, `shape` at line 5 is pointing to a stale object. Calling the `draw` on it will lead to the old method started at line 10, whereas the valid result should be the new method started at line 16. Javelus solves this problem by implementing implicit checks at the beginning of old methods. Calling such virtual methods on stale objects will trigger the validity checks but not on new or updated objects. After stale objects have been updated, Javelus forces the caller to call the method again. Therefore, explicit checks for CSC are not necessary.

6.2. Optimizing Object Validity Checks

6.2.1. Optimistic Checks

Valid checks can be done pessimistically or optimistically. One may pessimistically check the target object before every dereferencing, or optimistically assume that the target objects is valid and rely on some roll-back mechanism to fix the illegal accessing. Apparently, the optimistic strategy can be more efficient if illegal accessing is rare. In well-designed Java programs many references are statically typed with interfaces[8]. Javelus piggybacks the mechanism of interface method dispatching to implement an optimistic check strategy.

[8]A principle of reusable object-oriented design is "Program to an interface, not an implementation" [11].

In the Java HostSpot VM, dispatching an interface method requires two steps. The VM first checks whether the receiver implements the interface (an exception will be thrown if it fails). Next, the method with the given method symbol is looked up. Suppose that Javelus dispatches a method of an interface I to a stale object. I must be implemented by at least one of the old and new versions of the class of this object.

If I is implemented by the old version, the situation is very similar to that of common super-classes. The updating will be triggered by the code inserted at the beginning of old methods. If the new version class implements I, the method invocation will be re-dispatched to the new version. Otherwise an exception will be thrown due to type narrowing.

If I is only implemented by the new version, an exception will be raised immediately. Instead of throwing the exception to program code, Javelus will intercept the exception and check the receiver, update it if possible, and then dispatches the method again.

6.2.2. Eliminating Redundant Checks

Javelus can eliminate redundant validity checks, i.e., multiple checks on a same reference for successive dereferences. We implemented this optimization by piggybacking on the null pointer elimination algorithm of the JIT compiler. During the analysis of the JIT compiler, Javelus adds information to each reference to indicate whether it may be mismatched. Hence, validity checks on any known matched reference can be eliminated. Note that access via mismatched references to valid members of common super-classes can also be eliminated.

The above optimization will only work on DSU-free code blocks, as dynamic updating will kill all known matched references. Note that new compiled methods after classes have been updated are indeed DSU-free, as Javelus will recompile all methods and de-optimize the active compiled methods. The only case that dynamic updating can happen inside a compiled method is the on-stack-replacement of a long running loop. On-stack-replacement of the Java HotSpot VM ensures that an interpreted frame that contains a long running loop can be transformed into a compiled frame on-the-fly. In this case, the loop body is DSU-free and the optimization is still effective.

6.2.3. Other Optimizations

We have also optimized validity checks in the interpreter. In the Java HotSpot VM, programs are executed in a mixed-mode: some methods are interpreted directly by the interpreter while other methods are compiled to native code by the JIT compiler. There are also interpreted frames and compiled frames co-existing in thread stacks. Passing control from interpreted callers to interpreted callees is performed by common *method entries* while to compiled callees is by *adapters*. Javelus creates another set of method entries and adapters that contain validity checks. Consequently, checks before method call are unnecessary unless the method is virtual. Checks before virtual method dispatching is necessary because the new virtual method table could be larger

than the old one and dispatching a method not in the old virtual method table would cause an error.

6.3. Mixed Object Model

One major challenge in implementing a lazy approach is to update objects in-place even though their sizes are increased, because otherwise we have to update all references targeting at a object moved to somewhere else. Note that in the Java HotSpot VM objects are directly pointed to by references.

Javelus uses an object allocation model called *Mixed Object* to solve this problem. A mixed object is a logic object implemented with two physical objects, viz. *in-place object* and *phantom object* respectively. The in-place object takes the space of the stale object, and the phantom object is allocated somewhere else to hold fields that cannot fit into the space of the in-place object. A phantom object is only referenced by its corresponding in-place object. Any access to a mixed object must take its in-place object as the entry point.

To implement this model, Javelus leverages the *abstract fields* of objects. In JVM, every object is allocated with some abstract fields to facilitate VM functions such as locking, default hash code and garbage collection. In the 32-bit Java HotSpot VM, 64 bits are used for these fields [28]. Javelus defines a new pattern of the abstract fields to indicate that the current object is a mixed object and encode the address of its phantom object. The pattern takes the space of the original abstract fields, which are moved to the phantom object. Note that mixed objects are only used for transforming objects whose size is increased. Objects directly instantiated from the new versioned class are allocated normally. To support the transparent access to fields in phantom objects, Javelus intercepts all accesses to these fields and makes redirection when the target object is actually a mixed object.

In principle Javelus does not depend on any specific garbage collection algorithm. However, it can leverage the garbage collector to convert mixed objects to normal objects, so that the overhead caused by access indirections and loss of memory locality is eventually eliminated.

Figure 7 shows how a mixed object is merged by a mark-compact garbage collector. There are three objects, x, y and z, in the initial heap. After updating, a mixed object is created for y. During garbage collecting, y's in-place object is merged with its phantom object and a normal object is formed. Then the in-place object can be freed. References to y (e.g. the one in x) are eventually updated to y's new location when objects are compacted.

6.4. Transforming Objects

Javelus generates a set of transformer templates for type-changed classes. As Javelus performs a *default* transformation first, users only need to write transformers for classes that really require custom transformation. The default transformation will update the type (identifier) of an object, retain values for unchanged fields, and initiate newly added fields with default values.

Once a stale object is detected, the current thread will be trapped into an object updating routine. If the object is being updated by another thread, the

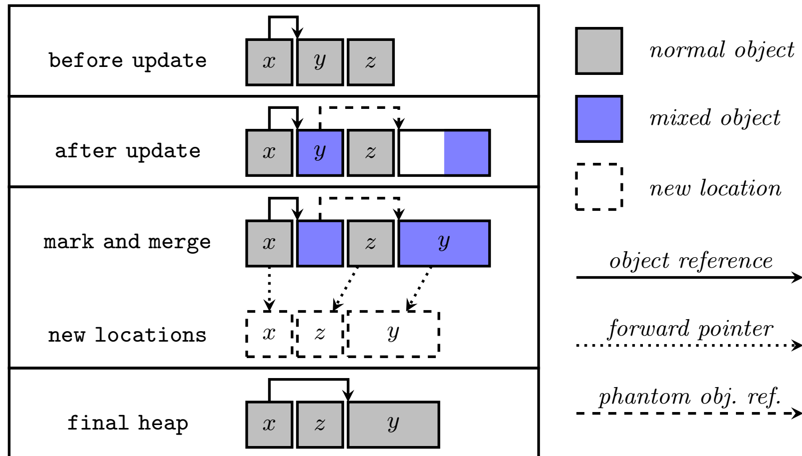

Figure 7: Integrate mix object model with a mark-compact garbage collector.

current thread will be blocked until the updating is finished. Currently, Javelus uses a global lock to schedule all updating routines. Although this policy might cause bottlenecks for multi-threaded applications, it provides an easy sequential programming model for object transformers.

The process of object transformation is as follows. At the beginning, a stale object is typed in the old version. Javelus first backups values of deleted fields to a special data structure and copies values of unchanged fields to a prototype object. Next, Javelus allocates the new object in-place, using the mixed object model if the object size is increased. Now the object is typed in the new version of the class. Then Javelus copies all unchanged fields back from the prototype object. Finally Javelus invokes the object transformer to complete the transformation.

In Javelus, transformers can directly access fields defined in the new version of classes. Fields defined by old classes are passed to transformers as parameters. Normally a transformer only reads from and writes to the fields of the current object and will not cause any data race with program code. However, sometimes a transformer needs to access other objects, and users should be aware that this could occur concurrently with the execution of other threads of the program.

6.5. Continuous Updates

Javelus supports continuous updates, i.e., the next updating can be performed immediately following previous one. Since Javelus uses a lazy updating strategy, continuous updates implies that objects in different versions can coexist in the heap. Nevertheless, at any time, only the latest version of program can be executed in Javelus, and stale objects are upgraded before use.

Our mixed object model also facilitates continuous updates. In this case mixed objects can also be stale, but the in-place object are always allocated at

the same position no mater how many updates happened[9]. Fields in phantom objects are decided by comparing their offsets with the *smallest* object size among all related versions.

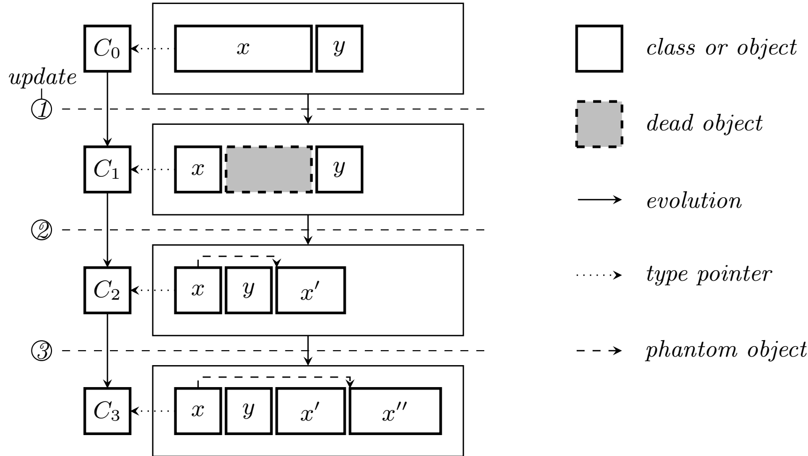

Figure 8: Mixed objects in continuous updates.

As shown in Fig. 8, object x has a large space initially. In the first update, the space is contracted. In the second update, a phantom object x' is created for object x. Note that we cannot assume the unused space after x is still there because garbage collection may occur at any time. For example, as shown in Fig. 8, the unused space has been collected by the garbage collection that occurs before the second update. In the third update, a new phantom object is created and replaces the old phantom object. Finally, object x is typed in class C_3.

7. Evaluation

We have evaluated Javelus by measuring the updating disruption and the performance of steady-state execution. For practical reasons[10], we did not directly compare Javelus with other Java dynamic updating systems, but instead we implemented an eager updating mode in Javelus for the purpose of comparison.

[9]Unless they are moved by the garbage collector.

[10]Most Java DSU systems, including Jvolve [31], JDrums [2] and DVM [24], are built on research or early JVMs, which are much slower than modern HotSpot VMs, so it would be meaningless to compare with them. DCE VM could be a suitable rival for Javelus since both build on HotSpot JVM. Unfortunately, DCE VM has to run in debugging mode to support dynamic updating, and this makes its performance seriously degraded. For a fair comparison, we tried but failed to extend DEC VM to support dynamic updating in normal mode. According to the updating mechanism and the performance data reported in [33], it is not very different from Javelus in eager mode.

To perform eager updating, after all classes have been updated, Javelus scans the heap to collect all stale objects, transforms all stale objects, and then forces a full heap compacting garbage collection immediately. Note that all check related flags are not set in this mode and thus there will be no updating relevant check in later execution.

All experiments were conducted on an Intel Core i7-2600 CPU (@ 3.40 GHz) with 4GB RAM. The operating system is Arch Linux with kernel version 3.9, and the JRE used is 1.6.0_45-b06 build.

7.1. Experiments with Micro Benchmarks

We first experimented with the same micro benchmarks used to evaluate Jvolve [31] and DCE VM [33]. Table 2 lists the relevant classes used in the benchmark. The program creates 4,000,000 objects, of which 0% to 100% are to be updated in different runs.

Table 2: Micro Benchmarks

Base	Increase	Reorder	Decrease

class C{
int i1;
int i2;
int i3;
Object o1;
Object o2;
Object o3;
void touch
(){
o2=o1;
o3=o2;
o1=o3;
i1++;
i2++;
i3++;
}
}
	class C{
	int i1;
	int i2;
	int i3;
	int i4;
	Object o1;
	Object o2;
	Object o3;
	Object o4;
	void touch
	(){
	o2=o1;
	o3=o2;
	o4=o3;
	o1=o4;
	i1++;
	i2++;
	i3++;
	i4++;
	}
	}
		class C{
		int i3;
		int i1;
		int i2;
		Object o3;
		Object o1;
		Object o2;
		void touch
		(){
		o2=o1;
		o3=o2;
		o1=o3;
		i1++;
		i2++;
		i3++;
		}
		}
			class C{
			int i1;
			int i2;
			Object o1;
			Object o2;
			void touch
			(){
			o2=o1;
			o1=o2;
			i1++;
			i2++;
			}
			}

Fig. 9 shows the mean pausing time of 20 repeated runs. The disruption of lazy updating (about 0.04 ms for each configuration) is 4 order of magnitude less than that of eager updating (ranging from 298 to 1825 ms). Note that according to [31], Jvolve's pausing time ranges from 618.7 ms to 2627.9 ms, and according to [33], DCE VM takes more than 400 ms, although their configurations (OS and machine) of experiments are different from ours.

Fig. 9 also confirms that the pausing time of eager updating is proportional to the number of affected objects, while that of lazy updating remains stable when the number of stale objects increases.

To measure the overhead of post-update execution, we used a `touch` method that read and wrote each field of the object once and counted the execution time of this method. All objects were touched four times after updating. The first touching triggered the lazy object updating. After the second touching a

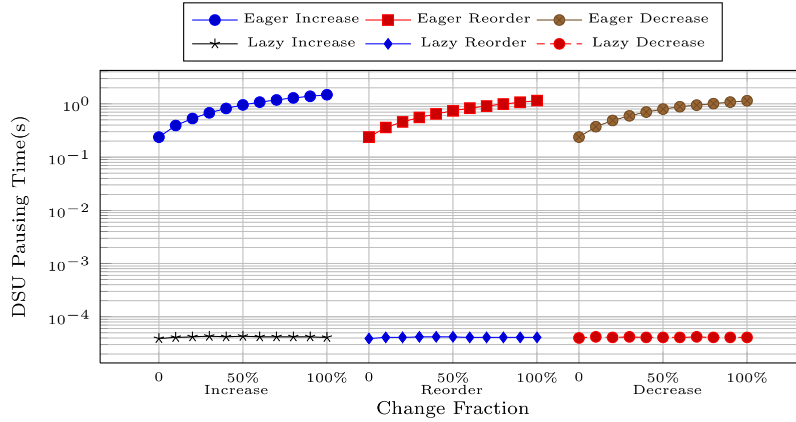

Figure 9: DSU pausing time of micro benchmarks.

garbage collection was explicitly invoked. The second and the third touching were used to measure the performance before and after mixed objects have been merged.

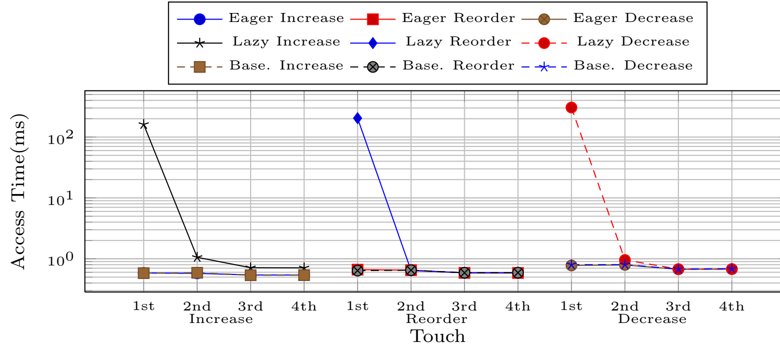

Figure 10: Average access time.

Fig. 10 shows the average access time to updated objects. Note that in the figure "Base." means access time measured on the baseline VM without dynamic updating. For lazy updating, access to stale objects at the first time is expensive, but the cost quickly decreases in the following accesses. There are three factors contribute to the cost of first access: 1) the VM has to warm up again; 2) Javelus has little optimization for lazy updating in the interpreter; 3) all updating actions are triggered by optimistic checks and the recovery from illegal states takes time.

Access to *Increased* objects at the second touch is slower than that of *Reorder* and *Decrease*. This is because *Increase* objects require mixed object redirections. With mix objects being merged, this overhead decreases gradu-

ally. Nevertheless, as mixed object checks still exist, the access time can be a little bit longer than that of eager updating, but this extra overhead is generally unobservable in practice.

7.2. Experiments with Tomcat and H2

We also evaluated Javelus with real updates of the Tomcat application server[11] (from 7.0.3 to 7.0.4) and the H2 database management system[12] (from 1.2.125 to 1.2.126). The dynamic patch generated for Tomcat contains 318 changed classes, and the patch for H2 contains 267 classes. We used the Dacapo[5] benchmark to drive the two servers in our experiment. Since DaCapo verified responses, these experiments also tested the correctness of Javelus.

Fig. 11 shows the pausing time for updating Tomcat and H2. The data is the mean of 20 runs. The eager approach is 20 (for Tomcat) to 300 times (for H2) disruptive than the lazy approach. The difference is smaller than that in the micro benchmarks because in this experiment the changed objects is only a few hundreds.

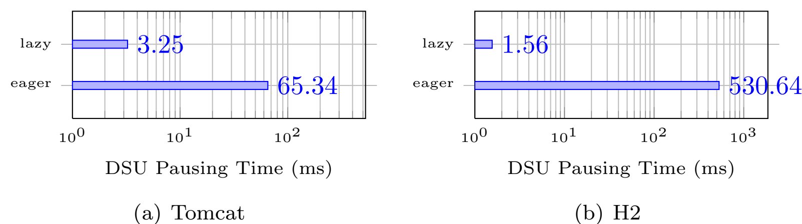

Figure 11: DSU pausing time for DaCapo benchmark.

To investigate dynamic updating's impact on application performance, we used JMeter[13] to measure Tomcat's response time when the server is dynamically updated from 7.0.3 to 7.0.4. In the experiment JMeter is configured with 8 concurrent threads. Fig. 12 shows the response time of each request sent during the timespan from 500 to 4000 ms (the first 500 ms is omitted as it is the warm-up phase). A dynamic updating request is issued at 2000 ms.

With the eager approach, there are a few requests with response time more than 80 ms, which is about 10 to 20 times larger than normal. On the contrary, with the lazy approach, the largest response time is about 9 ms, as the cost of dynamic updating is distributed more evenly. We calculate the maximum value, the minimum value, the mean and the variance in eager (with and without outliers), lazy, and baseline settings. The results have been shown in Table 3.

[11]http://tomcat.apache.org/
[12]http://www.h2database.com/
[13]http://jmeter.apache.org/

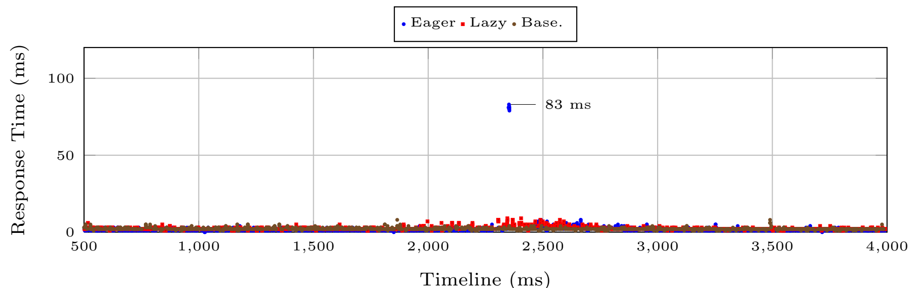

Figure 12: Tomcat response time

Table 3: Statistics of Tomcat response time

	Max (ms)	Min (ms)	Mean (ms)	Var. (ms^2)
Eager	83	0	2.07	16.28
Eager (without outliers)	8	0	1.87	0.61
Lazy	9	1	2.43	1.01
Base.	8	1	2.21	0.49

8. Related Work

8.1. Dynamic Updating Systems for Java

Dynamic updating systems for Java can be classified into two categories: VM-based systems and transformation-based ones. VM-based systems have their advantages in flexibility and efficiency, as changes are made to the runtime state in the VM directly. However, they often heavily depend on the specific implementation of the VMs, and tightly coupled with VMs' facilities such as dynamic class loading, garbage collection and just-in-time (JIT) compilation. On the other hand, in theory transformation-based approaches can be VM-independent, although in practice many of them such as those proposed in [29, 30] depend on the HotSwap mechanism, which may not present in some VMs. Transformation-based systems have to introduce wrappers to trigger updating as well as indirections to bring new objects into effect. Gregersen concluded that eventually supporting dynamic updating in VM is a better choice [13].

8.1.1. VM-based Systems

JDrums [2] and DVM [24] are implemented on early VMs. Both of them update objects lazily but are built on top of classic JVMs, in which program runs in interpretation mode only, and objects are referred indirectly through handles. Therefore, their techniques can hardly be applied to modern JVMs where JIT compilation is enabled, objects are directly accessed, and various types of garbage collectors are used.

Dmitriev [10] introduced HotSwap as part of the debugger interfaces of the Java HotSpot VM. HotSwap can only support swapping method bodies, but

Dmitriev has presented many possible proposals on how to support more flexible changes [9]. Subramaniam et al. implemented Jvolve [31] on top of the JikesRVM[14]. Jvolve is more flexible than HotSwap: it can support all changes except changing super-classes or interfaces. Würthinger et al. [33] extended HotSwap and developed DCE VM, which supports arbitrary changes of classes. Both Jvolve and DCE VM require that the VM be configured with a simple modified garbage collector in that objects can be updated eagerly through a stop-the-world garbage collection.

8.1.2. Transformation-based Systems

PROSE [27] aimed at replacing method code at runtime. In other words, it does not support class schema changes. PROSE updates code in a controlled and systematic way and the control logic is also written in Java. DUSC [29] creates several auxiliary classes to simulate one original class at load time. DUSC requires more spaces and cannot change public interfaces. JavAdaptor [30] is based on HotSwap and can update class schemas by class renaming and caller updating. JRebel [17, 18] is a relatively flexible and efficient transformation-based approach. It can be well integrated with existing Web servers, and it helps to prove that a less flexible DSU system is also useful. JRebel can be adapted to three JVM implementations at least [18]. DUCS [3] was first presented by Bialek et al. as a framework for updating component based systems and was extended for updating class-based Java programs [4]. DUCS first partitions class-based programs into larger modules. These modules are used as the basic updating unit. Hence, classes within the same module can be accessed directly.

8.2. Dynamic Updating Systems for Other Programming Languages

Neamtiu et al. developed Ginseng for dynamic updating C programs [26, 25]. In order to add new fields, heap data must be allocated with additional spaces. Every access to fields must be trapped with a version check in order to trigger updating. Besides, Ginseng requires that update points must be inserted into the source code first and the program should be compiled by a special compiler.

Chen et al. presented POLUS for C-like programs [6, 7]. By using a *relaxed consistency model*, changed code can be updated even when they are active. In POLUS, different versions of programs can be executed simultaneously. To ensure system consistency, POLUS requires programmers to write methods to convert states bi-directionally between old and new code. However, POLUS can not update heap data. Mariks et al. [23] implemented UpStare to support immediate updating by reconstructing stacks. Nevertheless, programmers have to write valid transformers for stacks, which is a non-trivial task.

8.3. Dynamic Updating for Component-based Systems

DSU can also be carried out in a granularity coarser than classes. To update a component in a (distributed) component-based software system, normally one

[14]http://www.jikesrvm.org

would not trouble himself dealing with the internal logic of the component, but drives the component into a steady state before upgrading it. The problem is how to ensure the correctness of ongoing activities on other components depending on the to-be-updated component. To this end in a seminal paper Kramer and Magee proposed the *quiescence* criterion for safe updating [19]. However, the approach could be very disruptive. Vandewoude et al. relaxed the condition with the *tranquility* criterion to reduce disruption [32], but it cannot guarantee consistency. Ma et al. used *version consistency* to achieve low-disruptive updating without sacrificing consistency [22]. Ajmani et al. [1] introduced a model that allows different versions of components to serve at the same time so that the updating can be gradually done.

9. Conclusion

This paper presents Javelus, a DSU system which is implemented on top of the industry-strength Java HotSpot VM. Javelus supports arbitrary changes of Java classes and can achieve low updating disruption without sacrificing efficiency. We also have conducted experiments on real updates of software widely used by industry to evaluate Javelus. The experiment results have shown that Javelus is low disruptive yet efficient.

Although the implementation of Javelus depends on facilities provided by the underlying JVM, we believe that many of the techniques discussed in this paper, such as the mixed object model and the techniques on reducing validity checks, can be adapted to other JVM implementations.

On the other hand, better integration with the JVM may further improve the performance. For example, we plan to investigate how to reduce the disruption caused by the re-compilation of updated code, which requires a deeper involvement of the HotSpot technology.

Acknowledgement

We thank the anonymous reviewers for their detailed and helpful comments and suggestions. This research was partially funded by the 863 Program (2013AA01A213) and National Nature Science Foundation (61100038, 91318301, 61321491, 61361120097) of China. Chang Xu was also partially supported by Program for New Century Excellent Talents in University, China (NCET-10-0486).

References

[1] Ajmani, S., Liskov, B., Shrira, L., 2006. Modular software upgrades for distributed systems. In: Proceedings of the European Conference on Object-Oriented Programming. ECOOP'06. Springer-Verlag, Berlin, Heidelberg, pp. 452–476.

[2] Andersson, J., Ritzau, T., 2000. Dynamic code update in JDRUMS. In: Proceedings of the ICSE Workshop on Software Engineering for Wearable and Pervasive Computing. pp. 1–9.

[3] Bialek, R., Jul, E., 2004. A framework for evolutionary, dynamically updatable, component-based systems. In: Proceedings of the 24th International Conference on Distributed Computing Systems Workshops - W7: EC (ICDCSW'04) - Volume 7. ICDCSW '04. IEEE Computer Society, Washington, DC, USA, pp. 326–331.

[4] Bialek, R. P., Jul, E., Schneider, J.-G., Jin, Y., 2004. Partitioning of Java applications to support dynamic updates. In: Proceedings of the Asia-Pacific Software Engineering Conference. pp. 616–623.

[5] Blackburn, S. M., Garner, R., Hoffman, C., Khan, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B., Oct. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In: OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications. ACM Press, New York, NY, USA, pp. 169–190.

[6] Chen, H., Yu, J., Chen, R., Zang, B., Yew, P.-C., 2007. POLUS: A powerful live updating system. In: Proceedings of the International Conference on Software Engineering. ICSE '07. IEEE Computer Society, Washington, DC, USA, pp. 271–281.

[7] Chen, H., Yu, J., Hang, C., Zang, Yew, P.-C., Oct. 2011. Dynamic software updating using a relaxed consistency model. IEEE Transactions on Software Engineering 37 (5), 679–694.

[8] Daley, R. C., Dennis, J. B., May 1968. Virtual memory, processes, and sharing in multics. Communications of the ACM 11 (5), 306–312.

[9] Dmitriev, M., Mar. 2001. Safe class and data evolution in large and long-lived Java applications. Ph.D. thesis, Department of Computing Science,University of Glasgow.

[10] Dmitriev, M., 2001. Towards flexible and safe technology for runtime evolution of Java language applications. In: Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution.

[11] Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[12] Gregersen, A. R., Jørgensen, B. N., Mar. 2009. Dynamic update of Java applications - balancing change flexibility vs programming transparency.

Journal of Software Maintenance and Evolution: Research and Practice 21 (2), 81–112.

[13] Gregersen, A. R., Simon, D., Jørgensen, B. N., 2009. Towards a dynamic-update-enabled JVM. In: Proceedings of the Workshop on AOP and Meta-Data for Software Evolution. RAM-SE '09. ACM, New York, NY, USA, pp. 2:1–2:7.

[14] Gu, T., Cao, C., Xu, C., Ma, X., Zhang, L., Lu, J., 2012. Javelus: A low disruptive approach to dynamic software updates. In: Proceedings of the 2012 19th Asia-Pacific Software Engineering Conference - Volume 01. APSEC '12. IEEE Computer Society, Washington, DC, USA, pp. 527–536.

[15] Gupta, D., Jalote, P., Barua, G., 1996. A formal framework for on-line software version change. IEEE Transactions on Software Engineering 22 (2), 120–131.

[16] Hicks, M., Nettles, S., Nov 2005. Dynamic software updating. ACM Transactions on Programming Languages and Systems 27 (6), 1049–1096.

[17] Kabanov, J., 2011. JRebel tool demo. Electronic Notes in Theoretical Computer Science 264 (4), 51–57.

[18] Kabanov, J., Vene, V., 2012. A thousand years of productivity: the JRebel story. Software: Practice and Experience.

[19] Kramer, J., Magee, J., 1990. The evolving philosophers problem: dynamic change management. IEEE Transactions on Software Engineering 16 (11), 1293–1306.

[20] Liang, S., Bracha, G., 1998. Dynamic class loading in the Java virtual machine. In: Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. OOPSLA '98. ACM, New York, NY, USA, pp. 36–44.

[21] Lindholm, T., Yellin, F., 1999. Java Virtual Machine Specification, 2nd Edition. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[22] Ma, X., Baresi, L., Ghezzi, C., Panzica La Manna, V., Lu, J., 2011. Version-consistent dynamic reconfiguration of component-based distributed systems. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. ESEC/FSE '11. ACM, New York, NY, USA, pp. 245–255.

[23] Makris, K., Bazzi, R. A., 2009. Immediate multi-threaded dynamic software updates using stack reconstruction. In: Proceedings of the Conference on USENIX Annual Technical Conference.

[24] Malabarba, S., Pandey, R., Gragg, J., Barr, E., Barnes, J. F., 2000. Runtime support for type-safe dynamic Java classes. In: Proceedings of the European Conference on Object-Oriented Programming. pp. 337–361.

[25] Neamtiu, I., Hicks, M., 2009. Safe and timely updates to multi-threaded programs. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '09. ACM, New York, NY, USA, pp. 13–24.

[26] Neamtiu, I., Hicks, M., Stoyle, G., Oriol, M., 2006. Practical dynamic software updating for C. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '06. pp. 72–83.

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

[28] Oracle, 2006. The Java HotSpot performance engine architecture. http://www.oracle.com/technetwork/java/whitepaper-135217.html, Accessed: 2013-06-1.

[29] Orso, A., Rao, A., Harrold, M. J., 2002. A technique for dynamic updating of Java software. In: Proceedings of the IEEE International Conference on Software Maintenance. pp. 649–658.

[30] Pukall, M., Kästner, C., Cazzola, W., Götz, S., Grebhahn, A., Schröter, R., Saake, G., 2013. JavAdaptor—Flexible runtime updates of Java applications. Software: Practice and Experience 43 (2), 153–185.

[31] Subramanian, S., Hicks, M., McKinley, K. S., 2009. Dynamic software updates: a VM-centric approach. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 1–12.

[32] Vandewoude, Y., Ebraert, P., Berbers, Y., D'Hondt, T., Dec. 2007. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. IEEE Transactions on Software Engineering 33 (12), 856 –868.

[33] Würthinger, T., Wimmer, C., Stadler, L., 2010. Dynamic code evolution for Java. In: Proceedings of the International Conference on the Principles and Practice of Programming in Java. pp. 10–19.

[34] Würthinger, T., Wimmer, C., Stadler, L., Jul. 2013. Unrestricted and safe dynamic code evolution for Java. Science of Computer Programming 78 (5), 481–498.