2012 19th Asia-Pacific Software Engineering Conference

Javelus: A Low Disruptive Approach to Dynamic Software Updates

Tianxiao Gu*, Chun Cao†§, Chang Xu†, Xiaoxing Ma†, Linghao Zhang* and Jian Lu†
State Key Laboratory for Novel Software Technology, Nanjing University
Department of Computer Science and Technology, Nanjing University, Nanjing, China
*{tianxiao.gu, zlh.nju}@gmail.com, †{caochun, changxu, xxm, lj}@nju.edu.cn

***Abstract*—Practical software systems are subject to frequent updates for fixing their bugs or addressing new requirements. Updating a software system without stopping and restarting it is desired, as this helps reduce the redeployment cost as well as achieving the high availability. Existing techniques for dynamically updating Java programs may introduce noticeable pauses during which these programs are unable to function. We in this paper present Javelus, a dynamic Java update system with greatly reduced pausing time but without sacrificing update flexibility and system efficiency. Different from previous approaches, Javelus uses a lazy update mechanism with which an object-to-update will not be updated until it is really used. We implemented Javelus on top of an industry-strength OpenJDK HotSpot VM. We evaluated Javelus with real updates to Tomcat 7 and the same micro array benchmark used in evaluating Jvolve and DCE VM. The experiments report promising results that Javelus only incurred a pausing time two orders of magnitude smaller than those of Jvolve and DCE VM.**

***Keywords*-dynamic software updates; Java; virtual machine;**

I. Introduction

Computer software always needs updates to fix bugs and add features. However, the halt-and-restart update schema is unacceptable in many situations. Some critical systems, like financial service and traffic control system, must keep running all the time. Any downtime would give rise to an enormous loss. Dynamic software updating (DSU) can alleviate these problems by updating programs while they are running.

Java is a popular object-oriented programming language and been widely used for developing enterprise applications, which usually require limited service downtime. DSU support for Java has been extensively studies in the literature, and most of them are at the VM level [1]–[3]. Roughly speaking, in JVM a Java program at runtime consists of a set of classes (metadata), 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 apply updates when no method-to-update is active, and the program is paused during the update.

DSU itself takes time, most of which is caused by the update of objects [2], [3]. Although generally this time is much less than that of the halt-and-restart approach, it could be seconds long in some settings. This would introduce a significant disruption to the service of the application.

Most of existing proposals of DSU for Java take an eager method that finds out stale objects and updates them all at once [2], [3]. 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 VM. The garbage collector used here must be a stop-the-world one, so that all the entire update is performed when the program is paused.

Many realistic applications such as high-performance Web servers and interactive applications require very small response time. They often run on VMs tuned with concurrent and incremental garbage collectors [4]. However, to support eager DSU, these garbage collectors must fallback to do full heap collection. The long pausing time caused by tracing the whole heap, invoking transformer methods on each stale object and adjusting all outdated references, is annoying and sometimes unacceptable.

We take a lazy approach to DSU to reduce the pausing time. It is motivated by following two observations. First, for most updates only a small portion of all objects is affected, so tracing the whole heap may be unnecessary. Second, many updated objects would not be used immediately or even not used any longer. 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 tracing, avoid updating useless objects and distribute the labor of object transformation to later program execution.

However, lazy object updates are not without challenges. A naïve implementation would bring unwanted overheads at execution time because (1) access to objects had to be trapped with checks for validity and (2) references to objects with their size increased had to be indirected. In addition, in 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 update process.

This paper presents the design and implementation of

§Corresponding author: Chun Cao.

1530-1362/12 $26.00 © 2012 IEEE
DOI 10.1109/APSEC.2012.55

Javelus[1], a lazy DSU system for Java based on the OpenJDK HotSpot VM[2]. With a carefully optimized object validity checking strategy and a novel object implementation model, Javelus achieves very short pausing time without sacrificing update flexibility and system efficiency. Javelus supports arbitrary changes of Java classes. Its overhead during steady-state execution is neglectable. In addition, Javelus guarantee that no stale code will run after the update is triggered and no stale object will be accessed by updated code. Javelus also provides an easy but powerful model for programmers to write transformer methods.

The main contributions of this paper are:

1) An efficient lazy object update mechanism, and
2) An implementation of this update mechanism on an industry-strength HotSpot VM.

The rest of this paper is organized as follows. In Section II, we discuss general problems of dynamic updates and our lazy approach. After introducing some preliminaries of the HotSpot VM in Section III, we present the design and implementation of our Javelus in Section IV. In Section V, we evaluate Javelus with some real updates of Tomcat 7 and a micro benchmark. Section VI gives some further discussions on the benefits and limitations of Javelus. Before concluding the paper in Section VIII, we summarize related work in Section VII.

II. Dynamic Update: Eager versus Lazy

Statically a Java program consists of a set of classes connected to each other with inheritance and association relationships. 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 constructed 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 run time image satisfying following properties[3]:

- *Type safety*: The program can continue to execute without type error.
- *Semantic continuity*: The states in the current image are preserved as much as possible.

In addition, a practical DSU system should also fulfill such requirements [6] [2]:

- *Efficiency*: There should be no overhead before and after the update.
- *Flexibility*: It should allow most kinds of changes happened in real life software evolution.
- *Low disruption*: The interruption of service caused by the update should be minimized.
- *Timeliness*: The update should be applied as soon as possible.

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

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

[3]Generally it is impossible to automatically ensure the semantic correctness of a dynamically updated program [5]. It's DSU users' responsibility to maintain the system consistency on top of type safety and semantic continuity.

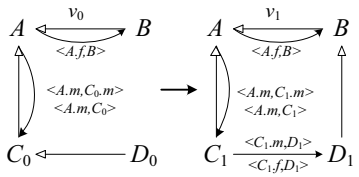

Fig. 1. An update

To refresh class metadata one must also take the relationships between them into account. For example, Fig. 1 shows two versions of a program consisting of 4 classes named A, B, C and D respectively. In the original version, B and C_0 inherit directly from A while D_0 inherits from C. Here, we use subscript to distinguish different versions of the same class. A method m of class A contains a local variable that is a reference to an object of class C_0 (denoted as $\langle A.m, C_0\rangle$); and it can call a method m of class C_0 (denoted as $\langle A.m, C_1.m\rangle$). In class A there is also a field f refers to an object of class B (denoted as $\langle A.f, B\rangle$). Suppose we need to update class C's methods and add to this class a field referring to an object of class D. Class D also changes its super class from class B to class C. In this case, we cannot simply swap the definitions of C and D with their new versions. References to old class metadata must be adjusted accordingly.

To avoid the trouble of updating frame stacks of active methods, most Java DSU approaches defer updates to a *DSU safe point* when no method-to-update is currently active [2], [6], [7]. This strategy also helps ensure that no stale code would be executed after update.

The trickiest part of a dynamic update is the transformation of existing objects of updated classes to the new version. States of old objects must be mapped to their new versions with user-provided transformers [2], [6] or default transformers. All references to old objects, no matter they are local variables or object fields, must be (re-)directed to corresponding new objects. This is crucial to preserve type safety and semantic continuity. Due to the potentially large number of objects and free distribution of the references, this part of work can be very costly.

Dynamic updates can be done eagerly or lazily. In an eager update all above work are carried out when the program is suspended at a safe point, while in a lazy update the program is allowed to resume early and some objects are updated later in an on-demand way. Eager updates are believed to be more efficient but also more disruptive than lazy updates. For our example, as shown in Fig. 2, suppose there is an object c of class C_0, which is referred by a local variable in method $A_1.m$. A DSU safe point occurs when $C_0.m$ returns. If we updated the program eagerly, after updating the metadata, we would transfer c from C_0 to C_1 with a new field typed with D_1 and update the local variable in $A_1.m$ accordingly before resuming the program.

Lazy updates are less disruptive than eager ones. A lazy update only updates class metadata when the program is

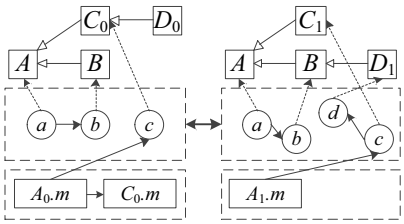

Fig. 2. The runtime after the eager update

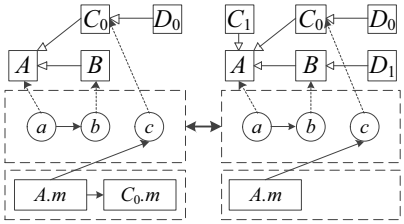

Fig. 3. The runtime after the first phase of lazy updates

suspended at a safe point. The stale objects in the heap are left untouched in this step. Then the execution of the program is resumed immediately after this step finishes. As shown in Fig. 3, if we took a lazy approach, the object c and the local variable of method $A_1.m$ would not be updated at this stage. They will be updated when the program eventually use them.

However, lazy updates can bring overheads to program execution because the program must be instrumented to trigger on-demand update of stale objects and references. Codes of *validity checks* should be inserted in codes in which a stale object may be wrongly used. These checks may bring significant overheads during execution. In addition, lazy updates often use indirections when objects increase their sizes. Besides, all objects should be allocated with sufficient spaces to store the *forward pointer*.

Our Javelus is an efficient lazy approach. As Java is a statically typed language, by analyzing class hierarchies at runtime, Javelus can find all validity check points at a fine-grained level. A stale object, such as object *c*, may be wrongly used in the context in which an object of the new class, such as class C_1, could be used. If a class removes a super class in the new application, a stale object of the sub class may also be wrongly used in the context in which an object of the removed super class could be used, e.g., the local variable represented by $\langle A.m, C_0\rangle$, which must be updated to point to an object of class C_1, may in fact refer to an object of class D_0, which is updated to an object of class D_1. However, class D_1 is not a sub class of class C_1.

Javelus has many techniques to eliminate *explicit* validity checks (i.e., checks on both stale and fresh objects). Javelus also exploits many dynamic features of the VM to implement *implicit* checks. We have observed that, in most cases, stale objects could be used as objects of a *common super class (CSC)* of the old and new classes. For instance, as shown in Fig. 1, class A is a CSC of class D_0 and D_1. Thus there is no need to check whether an object of class D is stale (indeed an object of class D_0) before reading a field declared in class A. Problems arise when the sub (invalid) class has overridden a method of a CSC. However, Javelus can implicitly check stale objects at this situation. Therefore, validity checks do not affect the CSCs and other sub classes of CSCs.

Javelus equips the *MixObjects* object model, which is an extension of the object model of the HotSpot VM, to support increasing spaces of objects. Only new added fields are indirected, and other fields can be accessed directly. We observed that many updates are incremental. So the *MixObjects* object model may have well performance for real updates. We reuse the existing space defined by the original object model in each object to store the forward pointer. Thus objects can be allocated with normal sizes.

Javelus performs updates only at the DSU safe point, whose semantics is derived from that of Jvolve [2]. The DSU safe point requires that no *restricted method* is active in all stacks. Restricted methods are often methods that have changed their bytecodes. Moreover, some unchanged methods are also prohibited in the stack when updating. For example, an unchanged method that might cause semantics inconsistency must also be restricted and this could only be done manually.

III. The Java HotSpot VM

Java is a statically typed object-oriented programming language. The basic unit of a program is the class (file). At runtime, class files are loaded into the VM dynamically by a *class loader*, which is a Java object in the VM. The entire class loading has three phases, loading, linking and initializing [8]. In the linking phase, the HotSpot VM parses the class file to create some *VM internal objects*, such as the *class metadata object (CMO)*, and objects holding bytecodes of methods. Such a design of internal objects facilitates the dynamic update, e.g., there is no need to allocate a new CMO if we only make changes to method bodies and thus we could avoid updating references to the CMO.

The CMO contains static fields of the class, the virtual method table (VMT) and the interface method table etc. Therefore, the size of the CMO is different among classes. The class is defined by adding its CMO to the *system dictionary*. At runtime, the loaded class is identified by the combination of the class name and class loader [8]. The last phase is to invoke the class initialization method.

Java methods are compiled into bytecodes before being executed in the VM. Bytecodes are executed by the interpreter first. Performance sensitive methods will be compiled into machine codes by the just-in-time compiler. The running applications are executed in *mix mode*, i.e., some methods are compiled and some methods are interpreted. In the HotSpot VM, each active method has a corresponding data structure, called *frame*, for different execution modes of methods in the stack. The frame stores the local state of an active method, such as the return address, local variables. The compiled method has a different *calling convention* with the interpreted

method [9]. The HotSpot VM uses adapters at the method entry to transfer data, such as parameters, between compiled and interpreted frames. In addition, interpreted methods have common entries for setting up and removing the frame. The HotSpot VM can transfer a compiled frame to an interpreted frame by the *de-optimization* mechanism. In contrast, the HotSpot VM can also transfer an interpreted frame to a compiled frame by the *on-stack-replacement (OSR)* mechanism.

The HotSpot VM may modify the state of the runtime, e.g., garbage collection. These operations are executed by a VM thread at the *VM safe point* while any other thread is suspended. The HotSpot VM can reach the VM safe point in bounded time. Dynamic updates in the VM often only happen at the DSU safe point. The DSU safe point is indeed a VM safe point with restrictions on stacks.

Each VM has an object model to describe the semantics and layout of objects. Although the object model is implementation dependent, it should define some common abstract fields (i.e., fields that defined by the VM) for each object [10], such as *default hash code*. Each object in the HotSpot VM must contain two words at least. One word is used to refer the CMO. The other *mark* word is shared by other abstract fields.

The Java HotSpot VM has several garbage collectors [4]. Compacting garbage collectors will move objects to collect space fragments. All references should be updated when moving objects, since objects are accessed directly in the HotSpot VM. Non-compacting garbage collectors cannot move objects but might fallback to perform compacting if space fragments are too many.

IV. Implementation of Javelus

This section describes how we implement Javelus on top of the HotSpot VM.

A. System Overview

Javelus is designed aiming at reducing the disruption of the dynamic update. Most of update work has been done before pausing the program. At first, programmers prepare two versions of programs, which are both correct, and use an off-line analysis tool provided by Javelus to compare class files of these two programs. A dynamic patch file and some transformer template files will be generated. Programmers fill in method bodies of transformer methods if necessary. The update is requested by sending the dynamic patch file to Javelus.

Javelus parses the patch file and prepares data used by the following update, such as new CMOs, transformer methods, before entering the DSU safe point. To keep type consistency during preparing data, Javelus use a *temporal dictionary* to store the new created CMOs. The temporal dictionary is used only by the dynamic update subsystem to store and look up new class metadata. These new class metadata will be moved into the system dictionary during the update.

Javelus will checks the states of all stacks at the next VM safe point. If it is a DSU safe point, class metadata and stacks are updated and then the paused program will continue to execute. However, stale objects are still in the heap. Javelus checks these objects before accessing to them. When a stale object is detected, the thread that is going to access it falls into an *object update routine*. Javelus creates a pair of *MixObjects* for the stale object that cannot be reallocated in-place. We will explain *MixObjects* in details in the following section. Transformer methods are invoked if needed. The thread continues to execute after updating object.

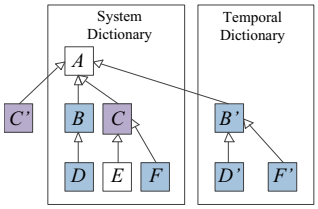

Fig. 4. system dictionary before update

B. Off-line Preparation

Currently, Javelus uses a static analysis tool to generate the dynamic patch file. The dynamic patch file specifies how to update each class. A class (CMO) can be *swapped, redefined or relinked*. If a class only changes bytecodes of some methods, then the class will be *swapped* with new method metadata. If a class has any other changes, e.g., adds a field, then the class and all its sub classes will be *redefined*. Javelus will load the new class metadata and replace the old metadata in the system dictionary. Otherwise, if a class has no changes but its CMO has referred to old VM internal objects, such as old CMOs, then the class will be *relinked* to new internal objects. Since the system dictionary is changing while the application is running, comparing all loaded classes should be performed at the DSU safe point, and this will cause a large disruption. Therefore, we choose to figure out these affected classes off-line.

A disadvantage of off-line analysis is that the change set generated statically is imprecise. We plan to provide an incremental online method to solve this problem. Another disadvantage is that we cannot specify the runtime identifier of a class, since the class loader is just an object. Currently, we group classes that may be loaded by the same class loader in the dynamic patch file. We specify a test method for each group. At updating time, Javelus will use the test method to resolve class loader.

C. Apply Changes to Runtime Data

Javelus provides an interface for programmers to invoke the DSU by passing the dynamic patch file. The dynamic patch file is parsed before the application is suspended. New VM internal objects are created in topological order, from super classes to sub classes. Javelus creates a temporal dictionary to hold new classes that will be redefined. The temporal dictionary is used for managing dependences between classes that will be redefined. As shown in Fig. 4, class B, D and F will be redefined. Here we use an apostrophe to indicate that the

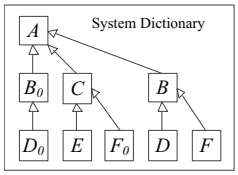

Fig. 5. system dictionary after update

class(CMO) is created for updating. Class C will be swapped with new definitions. During parsing class D', Javelus will first try to find its super class in the temporal dictionary and class B' will be returned. Class F' uses class C' as its super class. Class E does not have a new class. After swapping class C with class C', Javelus will update class E and F' to retrieve new definitions inherited from class C'.

Javelus then tries to update the application at the next VM safe point. The update will be deferred until the VM reaches a DSU safe point, or just be aborted since it tries too many times. Javelus also has been implemented with the *return barrier* [2].

When reaching the DSU safe point, Javelus discards all compiled methods to remove possible dependence to stale compiled methods. All active compiled methods will be de-optimized.

Javelus then updates all interfaces and classes in the topological order defined by the new class hierarchy. Old classes must be preserved until there are no stale objects. Javelus renames these classes to keep type consistency and these classes are not visible to the new program then. We do not use a separate universe as DCE VM [3] since a runtime symbol can only be resolved to one runtime entity. We also do not use a new class loader as Javeleon [11]. Custom class loaders are popularly used in enterprise applications. Changing class loaders may induce unexpected behavior of the original application.

After classes have been updated, stacks may contain active old methods whose bytecodes have not changed. All compiled frames of such methods have been de-optimized, and new methods will be executed by the interpreter automatically. However, interpreted frames of such methods still contain references to old internal objects, such as objects holding bytecodes. Javelus takes a walk on each stack to repair such references. After this repair, Javelus leaves the DSU safe point and continues the execution.

D. Lazy Object Updates

Javelus ensures such consistency semantics of lazy object updates: *new definitions of the class will always be used in the context where they are required.* The entire lazy object update is transparent to the running program. That means it is unnecessary to mind stale objects at the time of developing. Javelus marks those instance fields and methods, which are defined in classes under the least CSC,as *invalid members*. We assume instance fields and methods inherited from CSCs are valid members (see below).

Javelus checks the target object when accessing invalid members. JNI[4] and reflection API are also hacked with checks. Javelus inserts check points dynamically. To check at all points will give rise to significant overheads. We have many techniques to eliminate validity checks.

When invoking invalid methods, validity checks could be taken either at the caller or the invalid callee. For the interpreter, Javelus prefers to check at the invalid callee, since the codes of the interpreter are shared. Javelus creates individual entries and adapters for interpreted invalid methods. These entries and adapters will check the receiver before interpreting the bytecodes. Thus valid methods could be interpreted directly without any check. If the receiver is a stale object, Javelus will update it and resolve the callee again. The ongoing method invocation will be switched to the new callee. Javelus checks stale objects at the caller explicitly for interpreter only when dispatching virtual methods, since the index of the new method in the new VMT may be out of the old VMT.

For the compiler, Javelus checks stale objects explicitly at the caller. During the compilation of a method that accesses invalid members, Javelus uses an intra-procedure data flow analysis, which is similar with the *null pointer elimination* of the JIT compiler, to eliminate the multiple validity check. The analysis works well in a consistent code region, in which no dynamic update has happened, since a dynamic update kills all eliminated check points. Usually the region is just the method body, since all active compiled method will be de-optimized after update.

However, Javelus may transfer an active interpreted frame to a compiled frame on the fly. This scenario happens when there is a long running loop that triggers an OSR. In this situation, the consistent code region is the loop body.

Javelus assumes that a stale object could be used as a valid object typed in a CSC if the transformer method does not depend on the fields inherited from the CSCs. The update of a stale object can be deferred further. Static binding of members declared in the CSC can always be used correctly, since they are the same to both the old and new classes. However, the dynamic binding (dispatch) has some troubles, since the dynamic dispatch may result in a method declared in the old class. Therefore, Javelus creates a phantom method, which is always checking stale objects, in the old class for each dynamic dispatched method of the CSC. The dynamic dispatching on stale objects will result in the phantom method. On the other hand, the dynamic dispatching on valid objects will result in the right method.

Javelus also checks stale objects implicitly at the *failure path*, i.e., the execution path in which an exception has been created. For example, if a class has been implemented with a new interface, the VM will throw an *IncompatibleClassChangeError* if we invoke an interface method that declared in the new interface on the stale object. Before throwing the exception, Javelus checks the receiver and try to dispatch the method again.

[4]Java Native Interface

Javelus can only update a stale object to an object of the new class. If an old class has removed a super class, any reference typed in the removed super class cannot be updated to point to an object of the new class. Thus we could only update these references at the container, which may be objects or frames. The container method is marked as a restricted method, since Javelus does not support custom transformers for stacks.

E. The MixObjects Object Model

We use a partial delegation method to support adding fields to classes. We extend the object model of the HotSpot VM to add one more abstract field, the *MixNewObject* pointer. *MixNewObject* is a phantom object we create to hold fields that cannot be put in the space of the original old object. These fields are called *MixNewField*s. We call the old object *MixOldObject*, fields in it as *MixOldFiled*s and the two objects *MixObjects* if we refer them together. Only *MixNewField*s require a *MixObjects* check preceding any access to them and may result in an indirection. *MixOldFiled* can be accessed directly. We do not create a *MixNewObject* for objects allocated after update.

The *MixNewObject* pointer shares the space of other abstract fields. Therefore, we do not need to allocate one more word for each object initially. Since most updates are incremental, we could adjust the field layout of the VM to put hot fields in the *MixOldObject* and put less used fields in *MixNewObject*. Our partial delegation method does not have the *self*-problem [12]. *MixNewObject* is only visible to its *MixOldObject*. Any other object or method can only access *MixObjects* through the *MixOldObject*. As the *mark* word is shared by different abstract fields, any write to it during execution must be taken in a compare-and-swap operation. *MixObjects* pattern is a stable value for the *mark* word. Any other abstract field, including lock state, cannot overwrite it during execution. Other abstract fields will use the *mark* word in the *MixNewObject*. Therefore, these abstract fields are indeed *MixNewField*s.

F. Transform Object Concurrently

Once a stale object is detected, the thread will fall into an object update routine. If the stale object is already being updated, the thread will be blocked. Currently, we use one lock to schedule all update routine just like they are performed by a single thread. This policy induces bottlenecks for multi-threaded applications, but circular dependence can be easily prevented, i.e., two objects are updating at the same time by two threads and depend on each other. In the future work, we plan to conduct an analysis of the transformer method to equip a more flexible synchronization mechanism.

A stale object need pass three phases before turning into a valid object. At first, the stale object is an object typed in the old class. We preserve values of deleted fields in a particular data structure and copy values of matched fields to a prototype object. Then we try to reallocate the new object in-place. If the object size has decreased, the residual space is filled with dead objects. If the object size has increased, a *MixNewObject* is created. After reallocation, the stale object is typed in the new class, but other threads that try to access it will still be blocked. We first apply default transformation, which copies all matched fields back from the prototype object. Then we invoke the transformer method on the object. After all, the object is a totally new valid object and all threads can use it freely since then.

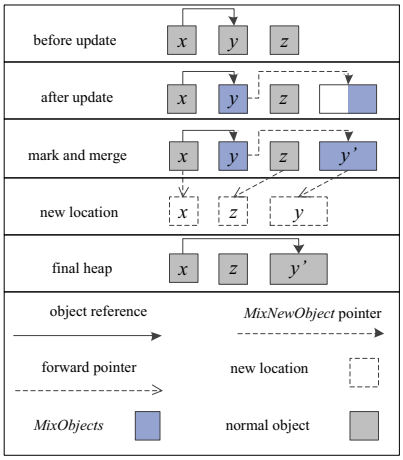

Fig. 6. Integrate *MixObjects* with a mark-compact garbage collector.

We provide the following semantics for programmers to write the transformer method: *all methods, including transformer methods, never see a stale object*. Deleted fields could not be accessed directly. Values of these fields are passed to the transformer method by annotated parameters. Therefore, only the transformer method of the object being transformed can access these fields. Problems arise when the transformer method uses objects other than the object being transformed. This is the *complex conversion* problem stated in [13]. First, the transformer method may depend on the old state of another stale object. We cannot control the order of lazy object updates. Many existing eager approaches such as Jvolve or DCE VM do not provide the mechanism either. Second, the transformer method may mutate another object, which should acquire a lock. Deadlock may arise since application methods are written separately with transformer methods. Currently, the deadlock problem could only be prevented by transformer writers.

G. Integrate MixObjects with Garbage Collection

Javelus does not depend on a specific garbage collector. However, the *MixObjects* object model has many drawbacks; it takes two objects to represent one object and accesses *MixNewField*s indirectly. We believe that these drawbacks could be removed by combining with compacting garbage collection algorithms. *MixObjects* can be merged to free the space of *MixOldObject*. Indirections and the loss of the locality of memory are removed at the same time. Fig. 6 shows that *MixObjects* can be integrated with a mark-compact garbage collector. There are three objects, x, y and z, in the initial heap. After updating, Javelus creates a *MixNewObject* for object

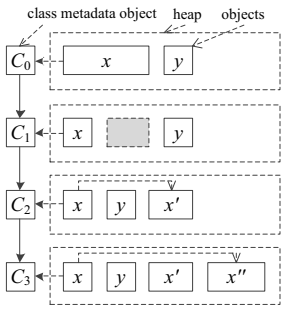

Fig. 7. Update object x continuously.

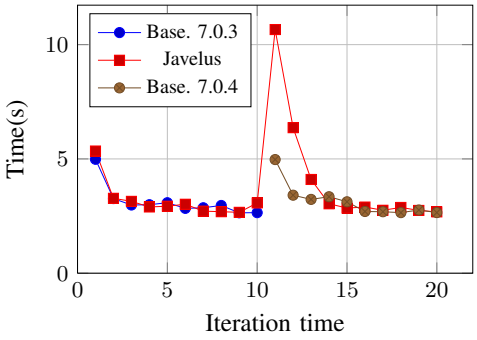

Fig. 8. 20 runs of the Tomcat benchmark.

y. When garbage collection happens, the *MixOldObject* y is merged into its *MixNewObject*. A valid new object y' then is formed. The *MixOldObject* can be freed. Object x contains a reference to object y. This reference is updated by two steps of forwarding, from x to y' followed with y' to its new location. We do not use the rescue buffer in DCE VM [3]. All objects can be slid to new locations from right to left as shown in Fig. 6.

H. Continuous Update

Javelus supports immediate continual updates, which means the next DSU can be performed just after the previous one. This feature is very useful when debugging. Class metadata can be updated eagerly. However, objects can only be updated lazily, which means objects in different versions may coexist in the heap. When detecting a stale object, Javelus updates the object continuously until the object is in the newest version.

At this time, *MixNewField*s are decided by comparing their offsets with the *minimal object size* among all versions of classes. The space of the stale object may be reallocated, and residual spaces can be padded with dead objects. As shown in Fig. 7, object x has a large space initially. In the first update, the space is shrank. In the second update, a *MixNewObject* x' is created for object x. The dead space cannot be reused at the next update, since both GC and DSU may happen during each update. In the third update, a new *MixNewObject* is created then. Finally, object x is typed in class C_3. A *MixOldObject* can change or remove its *MixNewObject*. However, *MixNewObject* pointer can only be modified at the VM safe point.

V. Experiments

We evaluated Javelus on the update disruption and the performance of steady-state execution, especially on lazy object updates and the *MixObjects* object model. All results of Javelus have been compared with those of the unmodified baseline.

A. Tomcat

We evaluated Javelus based on real updates of Tomcat[5] 7.0.3 to 7.0.4 with Dacapo [14] benchmark. We design three configurations: a dynamic update from 7.0.3 to 7.0.4 on Javelus, 7.0.3 on the baseline and 7.0.4 on the baseline. For each configuration, we iterated 20 times and recorded the time. All our experiments were conducted on an Intel Core 2 Quad CPU with 2.83 GHz per core and 4 GB of RAM. By using our off-line analysis tool, we found that the update contains 72 classes that should be reloaded. Among them, 35 classes might be swapped, and 37 class might be redefined. The overall disruption caused by update is about 2 ms. The update includes a redefining of class `ApplicationContext`, which is used by each request. Results have been shown in Fig. 8. The dynamic update is invoked after the 10th run. To eliminate the influence of the JIT compiler, first ten runs on the Javelus are compared with first ten runs of the baseline that running Tomcat 7.0.3 and last ten runs of the Javelus are compared with first ten runs of the baseline that running Tomcat 7.0.4.

As shown in Fig. 8, the performance peak is mainly caused by lazy object updates and the warm-up. The `ApplicationContext` object is shared by all threads. First access to the object will result in a race. Other threads must wait until the object is updated. For the lacking of system programming on the product quality HotSpot VM, the object update routine is only a basic implementation. We just reuse most codes of the method resolution routine of the HotSpot VM. We plan to refactor codes of object update routine with further study on the HotSpot VM in the future. Another reason of the performance peak is that all compiled methods have been discarded. The VM should warm up again.

We also used JMeter[6] to trace the peak of response time, which happened at the 11th run. We used 16 threads to request a same URL. The peak of response time is 125 ms, which is comparable with the time of the first response, 110 ms. So the peak is mainly caused by warming up of the application just after updating, not the updating disruption.

B. Micro Benchmarks

We used the micro benchmarks described in [2], [3] to measure the performance of the *MixObjects* object model and lazy object updates. We created a total of 4,000,000 objects,

[5]http://tomcat.apache.org/

[6]http://jmeter.apache.org/

TABLE I
MICRO BENCHMARK

Base	Decrease	Reorder	Increase

class C{
int i1;
int i2;
int i3;
Object o1;
Object o2;
Object o3;
}
	class C'{
	int i1;
	int i2;
	int i3;
	Object o1;
	}
		class C'{
		int i3;
		int i1;
		int i2;
		Object o3;
		Object o1;
		Object o2;
		}
			class C'{
			int i1;
			int i2;
			int i3;
			int i3;
			Object o1;
			Object o2;
			Object o3;
			Object o4;
			}

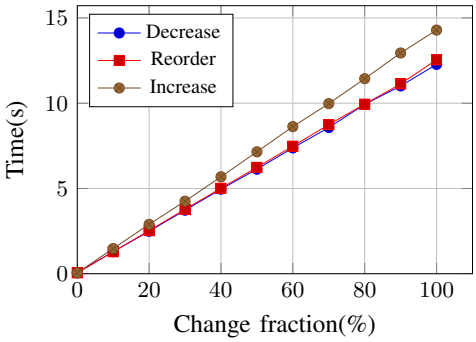

Fig. 9. Performance of lazy object updates.

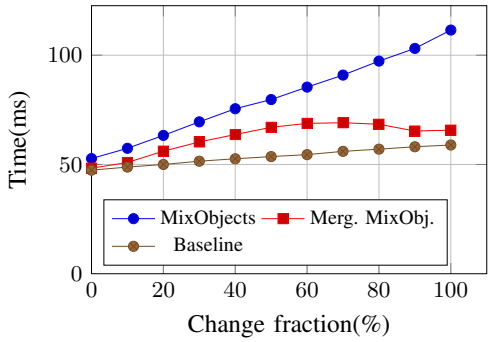

Fig. 10. Touch time of increase.

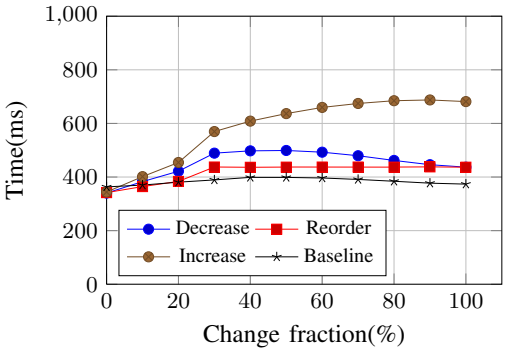

Fig. 11. Garbage collection time.

in which old objects have fractions between 0% and 100%. We repeated each configuration 10 times and report the mean.

To trigger lazy object updates, we created a `touch` method that read and writes each field once. All objects were touched three times. The first touching triggered the lazy object update. As shown in Fig. 9, the overall time of lazy object updates increases largely. The disruption is less than 0.5 ms for each configuration, while that of Jvolve ranges from 618.7 ms to 2627.9 ms [2], and that of DCE VM is more than 400 ms at least [3]. Although there are differences on the configuration (OS and machine) of the experiments, the time of the stop-the-world garbage collection has the same order of magnitude.

As the `touch` method is a virtual method, after update, the new `touch` method should be resolved and invoked again. The resolution takes more time than a direct virtual method dispatch and makes the first touching much longer than the last two. The second touching is used for measuring the performance of accessing *MixNewField*s in *MixObjects* and the third touching is used for measuring performance of accessing *MixNewField*s after merging *MixObjects*. The entire access time increased with the fraction of *MixObjects*. For increase benchmark, there are two *MixNewField*s (25% of total fields).

As shown in Fig. 6, *MixObjects* have a significant impact on the time of touching. After *MixObjects* are merged, the entire touching time decreases and has the largest relative overhead about 26% (the change fraction is 60%). Although the overhead is significant, we believe that many realistic applications evolve incrementally. We can adjust the field layout and put hot fields into *MixOldObject*. On the other hand, delegation could be well optimized. We can allocate one more local variable to cache the reference to the *MixNewObject*. All accesses to *MixNewField*s can share the same reference.

Javelus also has an impact on the compacting garbage collector. As shown in Fig. 11, the time of garbage collection increases with the change fraction. This is because Javelus merges *MixObjects* that are live objects during the collection. The baseline does not copy objects. Another reason is that the *MixObjects* object model destroys an optimization technique of garbage collection in the HotSpot VM. The HotSpot VM can jump a large free space by writing the end address at the beginning of the free space. The head of *MixOldObject* must be preserved. Thus even if there may be continuous *MixOldObject*s, which could be collected after merging, Javelus has to jump each *MixOldObject* one by one. Javelus does not depend on specific GC algorithms. We could use a concurrent garbage collector. Thus *MixObjects* may be collected before merged by a compacting collector.

VI. DISCUSSION

Javelus is low disruptive, safe, flexible and efficient. By using lazy object updates, Javelus reduces the update disruption largely. Javelus has many techniques to eliminate the overhead caused by validity checks. The empirical study on Tomcat 7 has shown that Javelus has neglectable overheads during steady-state execution. Javelus creates a *MixNewObject* when the object increases its size. Too many *MixObjects* may cause

memory peak and overheads. We can modify the compiler to use a local variable to cache the *MixNewObject* for common accesses. *MixObjects* can also be merged by a compacting garbage collector, and the space occupied by *MixOldObject* can be freed. However, merging *MixObjects* has a performance impact on compacting garbage collection. We believe that Javelus will have well performance if changed objects have short lifetimes.

We also believe that the more dynamic the applications are, the more efficient Javelus will be.This is based on the observation that most statically bound calls (e.g., private methods) on a stale object are inside a dynamically bound call (e.g., public methods) on the same stale object. Here checks at such statically bound call can be eliminated, since there is a check at the preceding dynamic bound call. Besides, most checks at dynamic bound calls can be performed implicitly.

Javelus is a DSU safe point approach. However, this safe criterion has many drawbacks. The update will be deferred until there is no restricted method. Programmers also have to participate in finding restricted methods. In the future, we plan to extend Javelus to support more consistency models, e.g., allow old and new codes to coexist. If so, stacks could be left unchanged and the update can be performed any time. However, programmers should manually write lots of synchronization methods to keep consistency between the old and new applications during execution. We believe that the lazy update has an inherent advantage to implement such consistency models.

VII. Related Work

We compare our Javelus with related work on DSU in this section.

A. Java

Dynamic update systems for Java can be divided into two main categories: VM modification approaches and program transformation approaches .

VM based approaches are often more flexible and efficient, but they often heavily depends on the VM implementation. Existing approaches proposed many ideas on how to enable a VM to support dynamic updates. These ideas inspired our Javelus a lot. However, due to the complexity of VM technologies, many approaches, such as JDrums [15] and DVM [16], can only be implemented on a classic old VM, which makes the DSU system unpractical. JDrums and DVM both can update objects lazily, however, their technologies cannot be easily adapted to modern VMs. In comparison with them, our Javelus is implemented on a modern industrial-strength VM, and we have conducted experiments on real updates to evaluate the performance.

Dmitriev [1] carried out the HotSwap mechanism on the HotSpot VM. Their implementation can only support swapping method bodies. Subramaniam et al. [2] and Wurthinger et al. [3] implemented the Jvolve and DCE VM respectively. Both of them update programs eagerly and have no overheads during execution. However, they are more disruptive than our Javelus.Our Javelus is efficient, low disruptive and also flexible. DCE VM and our Javelus support arbitrary changes of a Java class. Jvolve is less flexible: it can not change super classes or interfaces.

Program transformation approaches are often less flexible and efficient. They have no control of the overall program. The running program itself should check update requests periodically. They use wrappers to simulate interactions between different parts of the program at a high level. The simulation is often incomplete and cannot reflect all changes due to trade-off between flexibility and efficiency.

DUSC, which proposed by Orso et al. [17], creates several auxiliary classes to serve as one original class when the program starts. However, this approach cannot support changing public interfaces. To improve efficiency, Bialek et al. [18] extend the DUSC and partition the original program in larger units, such as components. Classes within the same component could be accessed directly. JRebel [19] is a relatively efficient and flexible program transformation based approach. JRebel can be well-integrated with current Web servers, and it helps to show that a less flexible DSU system is also useful, especially when it well understands the running application.

An advantage of program transformation approaches is that they do not depend on a specific VM. However, Gregersen et al. [20] conclude that eventually a dynamic-update-enabled VM is better for dynamic updates of Java applications.

Although our Javelus is based on some implementations of the HotSpot VM, we believe that many ideas of Javelus, such as the *MixObjects* object model and techniques on the elimination of validity checks, could be adapted to other VM implementations.

B. Procedural Languages

Many dynamic update approaches for procedural languages have been proposed. Neamtiu et al. carried out Ginseng for C programs [7], [21]. In order to add new fields, data must be allocated with larger space initially. Type is wrapped and every access to fields must be trapped with a version check. Ginseng is also a DSU safe point approach. They extract the loop to a single function to allow updating changed long running loops. Programmers should specify some update points in the original program first. Then Ginseng finds more induced update points that ensure *version consistency (VC)*. Since Java has polymorphism, ensuring VC, which should be performed dynamically, is thus more challenging. Chen et al. [22], [23] presented POLUS by using the debugging interfaces provided by the underlying OS. POLUS uses a *relaxed consistency model* to ensure update safety. Programmers should write many bi-directional synchronization methods to keep consistency between the old and new applications. Mariks et al. [24] implemented UpStare to support immediate update of multi-threaded programs by reconstructing stacks. However, programmer should write valid transformer for stacks.

C. DSU for Component-based Applications

Many ideas of DSU systems for component-based applications could also be adapted to Javelus. Gregerse et al.

present Javeleon on Netbeans platform [11], [25]. They create a separate class loader for classes of new components. Components are updated lazily. Different versions of components are interacted through a *version barrier*. Ajmani [26] proposed a model for allowing different versions of components to service at the same time. Components should have the ability to simulate the past and future nodes. To implement this idea in class-based systems, we can use proxy classes in the Java reflection API to wrapper each class for outside interactions. Besides, we build a list of objects in different versions. When an outside object sends a message to the wrapper object, one of objects in the list will be picked to response to the message.

VIII. Conclusion and Future Work

This paper presents the Javelus, a DSU system which is implemented on the industry-strength HotSpot VM. We implement a lazy object update method for Javelus. The experiment results have shown that Javelus is low disruptive yet efficient. Javelus is flexible, and it supports arbitrary changes of a Java class. Javelus uses the DSU safe point to assure the safety of update. DSU safe point approaches often have less timeliness but stronger consistency than those of none-DSU-safe-point approaches. In the future work, we plan to extend Javelus to support multiple consistency models to enhance timeliness.

Acknowledgment

This research was partially funded by the 973 Program (2009CB320702), 863 Program (2011AA010103) and National Nature Science Foundation (61100038, 61021062 and 60973044) of China. Chang Xu was also partially supported by Program for New Century Excellent Talents in University, China (NCET-10-0486).

References

[1] M. Dmitriev, "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*, 2001.

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

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

[4] Sun Microsystems, Inc, "Memory management in the Java HotSpot virtual machine," 2006. [Online]. Available: http://java.sun.com/j2se/reference/whitepapers/memorymanagement_whitepaper.pdf

[5] D. Gupta, P. Jalote, and G. Barua, "A formal framework for on-line software version change," *IEEE Transactions on Software Engineering*, vol. 22, no. 2, pp. 120 –131, feb 1996.

[6] M. Hicks and S. Nettles, "Dynamic software updating," *ACM Transactions on Programming Languages and Systems*, vol. 27, no. 6, pp. 1049–1096, Nov. 2005.

[7] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol, "Practical dynamic software updating for c," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006, pp. 72–83.

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

[9] M. Paleczny, C. Vick, and C. Click, "The Java HotSpot™ server compiler," in *Proceedings of the Java Virtual Machine Research and Technology Symposium*, 2001, pp. 1–12.

[10] D. F. Bacon, S. J. Fink, and D. Grove, "Space- and time-efficient implementation of the Java object model," in *Proceedings of the European Conference on Object-Oriented Programming*. London, UK: Springer-Verlag, 2002, pp. 111–132.

[11] A. R. Gregersen and B. N. Jørgensen, "Dynamic update of Java applications - balancing change flexibility vs programming transparency," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 21, no. 2, pp. 81–112, Mar. 2009.

[12] G. Kniesel, "Type-safe delegation for run-time component adaptation," in *Proceedings of the European Conference on Object-Oriented Programming*. Springer, 1999, pp. 351–366.

[13] F. Ferrandina, T. Meyer, R. Zicari, and G. Ferran, "Schema and database evolution in the O2 object database system," in *Proceedings of the International Conference on Very Large Data Bases*, 1995, pp. 170–181.

[14] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *Proceedings of the annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. New York, NY, USA: ACM Press, Oct. 2006, pp. 169–190.

[15] J. Andersson and T. Ritzau, "Dynamic code update in JDRUMS," in *Proceedings of the ICSE Workshop on Software Engineering for Wearable and Pervasive Computing*, 2000.

[16] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes, "Runtime support for type-safe dynamic java classes," in *Proceedings of the European Conference on Object-Oriented Programming*, 2000, pp. 337–361.

[17] A. Orso, A. Rao, and M. J. Harrold, "A technique for dynamic updating of Java software," in *Proceedings of the IEEE International Conference on Software Maintenance*, 2002, pp. 649–658.

[18] R. P. Bialek, E. Jul, J. Schneider, and Y. Jin, "Partitioning of Java applications to support dynamic updates," in *Proceedings of the Asia-Pacific Software Engineering Conference*, 2004, pp. 616–623.

[19] J. Kabanov, "JRebel tool demo," *Electron. Notes Theor. Comput. Sci.*, vol. 264, no. 4, pp. 51–57, 2011.

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

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

[22] H. Chen, J. Yu, R. Chen, B. Zang, and P. Yew, "POLUS: a powerful live updating system," in *Proceedings of the International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 271–281.

[23] H. Chen, J. Yu, C. Hang, B. Zang, and P. Yew, "Dynamic software updating using a relaxed consistency model," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 679–694, Oct. 2011.

[24] K. Makris and R. A. Bazzi, "Immediate multi-threaded dynamic software updates using stack reconstruction," in *Proceedings of the Conference on USENIX Annual Technical Conference*, 2009.

[25] A. R. Gregersen and B. N. Jørgensen, "Run-time phenomena in dynamic software updating: causes and effects," in *Proceedings of the International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*. New York, NY, USA: ACM, 2011, pp. 6–15.

[26] S. Ajmani, B. Liskov, and L. Shrira, "Modular software upgrades for distributed systems," in *Proceedings of the European Conference on Object-Oriented Programming*. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 452–476.