Dynamic Software Updates: A VM-centric Approach

Suriya Subramanian
The University of Texas at Austin
suriya@cs.utexas.edu

Michael Hicks
University of Maryland, College Park
mwh@cs.umd.edu

Kathryn S. McKinley
The University of Texas at Austin
mckinley@cs.utexas.edu

Abstract

Software evolves to fix bugs and add features. Stopping and restarting programs to apply changes is inconvenient and often costly. Dynamic software updating (DSU) addresses this problem by updating programs while they execute, but existing DSU systems for managed languages do not support many updates that occur in practice and are inefficient. This paper presents the design and implementation of JVOLVE, a DSU-enhanced Java VM. Updated programs may add, delete, and replace fields and methods anywhere within the class hierarchy. JVOLVE implements these updates by adding to and coordinating VM classloading, just-in-time compilation, scheduling, return barriers, on-stack replacement, and garbage collection. JVOLVE is *safe*: its use of bytecode verification and VM thread synchronization ensures that an update will always produce type-correct executions. JVOLVE is *flexible*: it can support 20 of 22 updates to three open-source programs—Jetty web server, JavaEmailServer, and CrossFTP server—based on actual releases occurring over 1 to 2 years. JVOLVE is *efficient*: performance experiments show that JVOLVE incurs *no overhead* during steady-state execution. These results demonstrate that this work is a significant step towards practical support for dynamic updates in virtual machines for managed languages.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Run-time environments

General Terms Languages, Experimentation, Reliability

Keywords dynamic software updating, virtual machine technology, garbage collection

1. Introduction

Software is imperfect. To fix bugs and adapt software to user demands, developers must modify deployed systems. However, halting a software system to apply updates creates new problems: safety concerns for mission-critical and transportation systems; substantial revenue losses for businesses [38, 34]; maintenance costs [44]; and at the least, inconvenience for users. These problems may translate into serious security risks if patches are not applied promptly [2, 3].

Dynamic software updating (DSU) is a general-purpose mechanism that solves these problems by updating programs while they run without a special software architecture or redundant hardware [19]. A practical DSU system must be safe, flexible, and efficient. *Safe* updates insure the program is as correct as deploying it from scratch. The update model would ideally be *flexible* enough to support all software changes, but DSU is still useful if the system supports most software updates. Since updates should be rare events, an *efficient* DSU system would ideally impose no runtime overhead during steady-state program execution.

Researchers have made significant strides toward making DSU practical for systems written in C or C++, supporting server feature upgrades [32, 13, 24], security patches [2], and operating systems upgrades [39, 5, 6, 25, 12, 3]. Enterprise systems and embedded systems—including safety-critical applications—are increasingly written in managed languages, such as Java, Ruby, and C#. Unfortunately, work on DSU for managed languages lags behind work for C and C++. For example, while the HotSpot JVM [27] and some .NET languages [14] support on-the-fly method body updates, this support is too inflexible for all but the simplest updates—limiting changes to method bodies would support less than half of the updates to our three Java benchmark programs. Academic approaches [37, 26, 35, 8] offer more flexibility, but have not been proven on realistic applications. Furthermore, they employ method and object indirection to make applications DSU capable, imposing substantial space and time overheads on steady-state execution.

This paper presents the design of JVOLVE, a DSU system for Java, which we implement in Jikes RVM, a Java Virtual Machine. JVOLVE's combination of flexibility, safety, and efficiency is a clear advance over prior approaches. The paper's key contribution is to show how to extend and integrate existing VM services to support DSU that is flexible enough to support a large class of updates, guarantees type-safety, and imposes no space or time overheads on steady-state execution.

JVOLVE DSU supports many common updates. Users can add, delete, and change existing classes. Changes may add or remove fields and methods, replace existing ones, and change type signatures. Changes may occur at any level of the class hierarchy. To initialize new fields and update existing ones, JVOLVE applies *class* and *object transformer* functions, the former for static fields and the latter for object instance fields. The system automatically generates default transformers, which initialize new and changed fields to default values and retain values of unchanged fields. If needed, programmers may customize the default transformers.

JVOLVE relies on bytecode verification to statically type-check updated classes. To avoid type errors due to the timing of an update [40, 32, 5], JVOLVE stops the executing threads at a *DSU safe point* and then applies the update. DSU safe points are a subset of VM *safe points*, where it is safe to perform garbage collection (GC) and thread scheduling. DSU safe points further restrict the methods that may be on each thread's stack, depending on the update. These methods include (1) updated methods, (2) methods that refer to updated classes (since their machine code may contain hard-coded offsets that the update changes), and (3) user-specified methods as needed for safety [21, 31]. JVOLVE installs return barriers [43]

PLDI'09, June 15–20, 2009, Dublin, Ireland.
Copyright © 2009 ACM 978-1-60558-392-1/09/06. . . $5.00

on these methods to inform the run-time system when a running method returns, to speed up reaching a safe point. JVOLVE also applies on-stack replacement (OSR) to recompile methods in the second category even as they run, as long as they do not contain inlined updated methods. This approach does not guarantee JVOLVE will reach a DSU safe point, but in our multithreaded benchmarks it does in nearly all cases. More sophisticated thread scheduling support could attain greater flexibility [30].

JVOLVE makes use of the garbage collector and JIT compiler to efficiently update the code and data associated with changed classes. JVOLVE initiates a whole-heap GC to find existing object instances of updated classes and initialize new versions using the provided object transformers. JVOLVE invalidates existing compiled code and installs new bytecode for all changed method implementations. When the application resumes execution these methods are JIT-compiled when they are next invoked. The adaptive compilation system naturally optimizes updated methods further if they execute frequently.

JVOLVE imposes no overhead during steady-state execution. During an update, it imposes overheads in classloading and garbage collection. After an update, the adaptive compilation system will incrementally optimize the updated code in its usual fashion. Eventually, the code is fully optimized and running with no additional overhead. The zero overhead in steady-state execution for a VM-based approach is in contrast to DSU techniques for C and C++. These approaches must use a compiler or dynamic rewriter to insert levels of indirection [32, 35] or trampolines [12, 13, 2, 3], which add a constant overhead during normal execution.

We assessed JVOLVE by applying updates corresponding to one to two years' worth of releases of three open-source multithreaded applications: Jetty web server, JavaEmailServer (an SMTP and POP server), and CrossFTP server. We found that JVOLVE can successfully apply 20 of the 22 updates—the two updates it does not support change a method within an infinite loop that is always on the stack. Microbenchmark results show that the pause time due to an update depends on the size of the heap and fraction of transformed objects. Experiments with Jetty show that applications updated by JVOLVE enjoy the same steady-state performance as if started from scratch.

In summary, this paper's main contributions are (1) techniques to extend and integrate standard virtual machine services for managed languages to support flexible, safe, and efficient dynamic software updating services, (2) the design, implementation, and evaluation of JVOLVE, a Java VM with support for dynamic software updating. JVOLVE is distinguished from prior work in its realism, flexibility, technical novelty, and high performance. We believe our demonstration is a significant step towards supporting flexible, efficient, and safe updates in managed code virtual machines.

2. Dynamic Updates in JVOLVE

This section overviews JVOLVE's approach, what changes it permits, and how the developer participates in the process.

2.1 System Overview

Figure 1 illustrates the dynamic update process. Assume that the VM is executing the current version of the program, whose code is depicted in the left top corner. Meanwhile, developers prepare a new version and fully test it using standard procedures. A developer then invokes JVOLVE's Update Preparation Tool (UPT) on the old and new versions. The UPT generates an update specification, which identifies new and updated classes, and a `JvolveTransformers.java` file that contains default *object* and *class transformer* methods.

Transformer methods take an object or class of the old version and initialize the corresponding object or class of the new version.

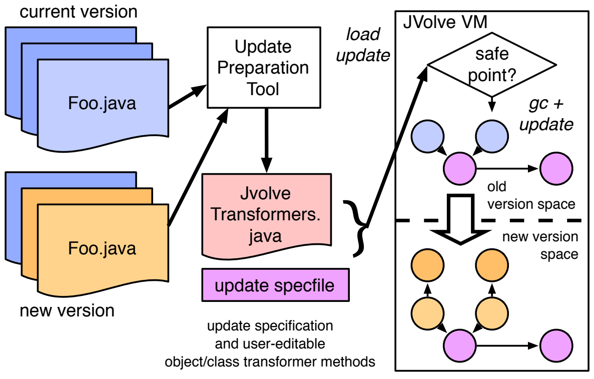

Figure 1. Dynamic Software Updating with JVOLVE

The default transformers assign default values, such as zero and `null`, to new instance, reference, and static fields, and copy values for unchanged fields. Developers may customize the default transformers as necessary. We present an example update and transformer in Section 2.3.

The user signals the running VM to apply the update, and the VM loads the new class files and transformers and schedules the update. The VM scheduler signals an interrupt, which stops all threads at VM safe points, where it is safe to perform thread scheduling and garbage collection. JVOLVE then checks if the VM is at a *DSU safe point*. DSU safe points require that no thread's activation stack contains a *restricted* method. Restricted methods are of three categories: (1) methods changed by the update, (2) methods whose bytecode is unchanged but whose compiled representation may change, and (3) methods specified by the user. If restricted methods are on stack, the VM installs return-barriers for (1) and (3), and performs on-stack-replacement for (2) to reach a DSU safe point. Section 3.2 describes this process in detail.

Once all application threads have synchronized at DSU safe points, JVOLVE applies the update. It first invalidates the compiled versions of all changed methods. These methods are recompiled as needed—the adaptive JIT compiler will generate code the next time the program invokes an invalidated method, and may optimize it further, if the program executes it frequently. The VM then invokes the class transformers. Finally, the VM initiates a full copying garbage collection. It piggybacks on the garbage collector to detect all existing objects whose classes change. It allocates objects that conform to the new type declarations, and performs object transformations to populate the new objects with valid state. At this point, the update is complete.

2.2 Programmer Update Model

We have designed a flexible, yet simple update model that supports updates that we believe are important in practice.

Programmers may change method bodies. Method body updates are the simplest and most commonly supported change [27, 14, 17, 20, 35, 39, 22], because DSU systems can preserve type safety by simply invoking the new method the next time the program executes the method. However, restricting updates to method bodies prevents many common changes [29]. Section 4 shows that over half the releases of Jetty, JavaEmailServer, and CrossFTP change more than just the method bodies.

Programmers may also change class signatures in various ways. They may change method signatures, e.g., by changing the type or number of method arguments. They may add or delete instance and static field members and change the types or access modifiers

```
public class User {
  private final String username, domain, password;
  private String[] forwardAddresses;
  public User(...) {...}
  public String[] getForwardedAddresses() {...}
  public void setForwardedAddresses(String[] f) {...}
}
public class ConfigurationManager {
  private User loadUser(...) {
     ...
     User user = new User(...);
     String[] f = ...;
     user.setForwardedAddresses(f);
     return user;
  }
}
```

(a) Version 1.3.1

```
public class User {
  private final String username, domain, password;
  private EmailAddress[] forwardAddresses;
  public User(...) {...}
  public EmailAddress[] getForwardedAddresses() {...}
  public void setForwardedAddresses(EmailAddress[] f) {...}
}
public class ConfigurationManager {
  private User loadUser(...) {
     ...
     User user = new User(...);
     EmailAddress[] f = ...;
     user.setForwardedAddresses(f);
     return user;
  }
}
```

(b) Version 1.3.2

Figure 2. Example changes to JavaEmailServer `User` and `ConfigurationManager` classes

of existing members. These changes may occur at any level of the class hierarchy. For example, programmers may delete a field from a parent class and this change will propagate correctly to the class's descendants. We rely on the bytecode compiler to ensure that the resulting program is type-safe, e.g., there are no more accesses to the deleted field in the program. JVOLVE does not support permutations of the class hierarchy, e.g., reversing a super-class relationship. While this change may be desirable in principle, in practice, it requires sophisticated transformers that enforce update ordering constraints. None of the program versions we examined make this type of change.

Example. Consider the following update from JavaEmailServer, a simple SMTP and POP e-mail server. Figure 2 illustrates a pair of classes that change between versions 1.3.1 and 1.3.2. These changes are fully supported by JVOLVE. JavaEmailServer uses the class `User` to maintain information about e-mail user accounts in the server. Moving from version 1.3.1 to 1.3.2, there are three differences. First, the method `loadUser` fixes some problems with the loading of forwarded addresses from a configuration file (details not shown). This change is a simple method update. Second, the array of forwarded addresses in the new version contains instances of a new class, `EmailAddress`, rather than `String`. This change modifies the class signature of `User` since it modifies the type of `forwardedAddresses`. Finally, the class's `setForwardedAddresses` method is also altered to take an array of `EmailAddresses` instead of an array of `Strings`, and code from `loadUser` accommodates this change as well.

2.3 Class and Object Transformers

For classes whose signatures have changed, an object transformer method initializes a new version of the object based on the old version. For example, consider a class `List` with field `next` of type `List` and an update that adds a new `int` field `x` to `List`. The object transformer's job is to modify each object instance of type `List` to conform to its new class definition. Class transformers serve a similar purpose and update static fields, rather than instance fields. The UPT generates default class and object transformers automatically, retaining unchanged fields and initializing new or changed ones. The default object transformer for our changed `List` copies the `next` field from an old object to a transformed object and initializes `x` to zero, i.e, `transformed.next = old.next` and `transformed.x = 0`.

For our running example, the UPT identifies that the `User` and `ConfigurationManager` classes have changed, and produces default object transformers. The programmer elects to modify the object transformer for the class `User`, as illustrated in Figure 3.

```
public class v131_User {
  private final String username, domain, password;
  private String[] forwardAddresses;
}
public class JvolveTransformers {
 ...
 public static void jvolveClass(User unused) {}
 public static void
  jvolveObject(User to, v131_User from) {
    to.username = from.username;
    to.domain = from.domain;
    to.password = from.password;
    // default transformer would have:
    //   to.forwardAddresses = null
    int len = from.forwardAddresses.length;
    to.forwardAddresses = new EmailAddress[len];
    for (int i = 0; i < len; i++) {
      String[] parts =
        from.forwardAddresses[i].split("@", 2);
      to.forwardAddresses[i] =
        new EmailAddress(parts[0], parts[1]);
}}}
```

Figure 3. Example `User` object transformer

Object and class transformer methods are simply `static` methods in the class `JvolveTransformers`. The class transformer method `jvolveClass` takes an instance of the new class as a dummy argument; standard overloading in Java distinguishes the `jvolveClass` methods for different classes. (In our example, `jvolveClass` does nothing.) The object transformer method `jvolveObject` takes two reference arguments: `to`, the uninitialized new version of the object, and `from`, the old version of the object. We prepend a version number to the names of old classes to distinguish them from the new versions. Based on the UPT specification, but before the VM loads the `JvolveTransformers` class, the VM renames the old class in all its internal data structures. This renaming makes the class name space and the `JvolveTransformers` class type-correct. In our example, the VM renames the old version of `User` to class `v131_User`, which is the type of the `from` argument to the `jvolveObject` method in the new `User` class. The `v131_User` class contains only field definitions from the original class; all methods have been removed since the updated program may not call them, as discussed below.

A typical transformer initializes a new field to its default value (e.g., `0` for integers or `null` for references) and copies references to the old values. In the example, the first three lines simply copy the previous values of `username`, `domain`, and password. A more interesting case is the field type change to forwardedAddresses. The default transformer function would initialize the forwardedAddresses field to null because of the type change. Here, the programmer has customized the function to instead allocate a new array of EmailAddresses and initialize them to the Strings from the old array.

Because the transformer class is separate from the old and new object classes, the Java type system would normally forbid the transformer access to their private fields. There is no obvious solution to this problem that conforms to the Java type system. We could define object transformers as methods of the old changed classes, which would grant access to the old fields, but not the new ones. Defining transformers as methods of the new changed class has the reverse problem. Also, the Java type system would disallow writes to final fields from within the transformer functions. To avoid these problems, we compile our separate transformation class with the JastAdd Java-to-bytecode extensible compiler [18] using a simple extension we wrote that ignores access modifiers (e.g., private and protected) and allows methods to assign to final fields. Bytecode that ignores these modifiers would not normally verify, so we have to modify the VM to allow it in this special circumstance.[1] The VM executes these Java functions normally, because they are otherwise standard Java. Since the transformation class is only active and available during the update, the VM may delete it after transformation. Separating transformers from updated classes avoids cluttered class files at run-time, and makes DSU more transparent to developers.

Supported in its full generality, a transformer method may reference any object reachable from the global (static) namespace of both the old and new classes, and read or write fields or call methods on the old version of an updated object and/or any objects reachable from it. JVOLVE presents a more limited interface (similar to past work [37, 26]). In particular, the only access to the new class namespace is via the to pointer, whose fields are uninitialized. The old class namespace is accessible, with two caveats. First, fields of old objects may be dereferenced, but only if the update has not changed the object's class, or if it has, after the referenced objects are transformed to conform to the new class definition. Second, no methods may be called on any object whose class was updated. In Figure 3 class v131_User is defined in terms of the fields it contains; no methods are shown. As explained in Section 3.4, these limitations stem from the goal of keeping our garbage collector-based traversal safe and relatively simple. This interface is sufficient to handle all of the updates we tested.

An alternative programming model would be that transformers could dereference from object fields and see the *old* objects, rather than the transformed ones. Boyapati et al. [11] implement this model, as described in Section 5. Our experience and that of others [5, 32, 30, 24] indicate that our model expresses many updates well. We leave to future work a detailed investigation of the semantics and expressiveness of both models.

3. JVOLVE DSU Implementation

This section describes how we support DSU in JVOLVE by extending common virtual machine services. JVOLVE is built on the Jikes RVM, a high-performance Java-in-Java Research VM [1, 42]. JVOLVE integrates and extends the Jikes RVM's dynamic classloader, JIT compiler, thread scheduler, copying garbage collector (GC), and support for return barriers and on-stack replacement to implement DSU.

[1] Jikes RVM, on which JVOLVE is built, does not implement a bytecode verifier. Aside from this exceptional case, JVOLVE classes are compiled normally and should pass verification.

After the user prepares and tests a program's modifications, the update process in JVOLVE proceeds in five steps. (1) Our UPT generates an update specification. (2) The user signals JVOLVE. (3) JVOLVE stops running threads at a DSU safe point. (4) It loads the updated classes, the transformer functions, and installs the modified methods and classes. (5) JVOLVE then applies object and class transformers following a modified GC.

3.1 Preparing the update

To determine the changed and transitively-affected classes for a given release, we wrote a simple Update Preparation Tool (UPT)[2] that examines differences between the old and new classes provided by the user. UPT groups changes into three categories, and lists them in the update specification file:

Class updates: These updates change the class signature by adding, removing, or changing the types of fields and methods.

Method body updates: These updates change only the internal implementation of a method.

Indirect method updates: These are methods whose bytecode is unchanged, but the VM recompiles them because they refer to fields and methods of updated classes. The compiled code uses hard-coded field offsets, and the update may change these offsets.

UPT generates default object and class transformer functions for all class updates, which the programmer may optionally modify. After compiling the transformers with our custom JastAdd compiler (described in Section 2.3), the user initiates the update by signaling the JVOLVE VM and providing the new version of the application, the update specification file, and the transformers class file.

3.2 DSU safe points

JVOLVE requires the running system to reach a *DSU safe point* before it applies updates. DSU safe points occur at *VM safe points* but further restrict the methods on the threads' stacks. These restrictions provide sensible update semantics: no code from the new version executes before the update completes, and no code from the old version executes afterward. As mentioned in Section 2.1, we divide restricted methods into three categories: (1) methods whose bytecode has changed, due to a class update or a method body update; (2) methods whose bytecode has not changed but that access an updated class; and (3) methods the user blacklists.

This subsection next discusses why these restrictions improve the safety and semantics of updates, and then describes the actions JVOLVE takes to reach a DSU safe point.

Semantics of DSU safe points. Our choice of restricted methods is similar to other DSU systems [37, 26, 2, 17, 27, 14, 13, 39]. To understand why category (1) methods are restricted, consider the update from Figure 2. Assume the thread is stopped at the beginning of the ConfigurationManager.loadUser method. If the update takes effect at this point, the new implementation of User.setForwardedAddresses will take an object of type EmailAddress[] as its argument. However, if the old version of loadUser were to resume, it would still call setForwarded-Addresses with an array of Strings, resulting in a type error.

Preventing an update until changed methods are no longer on the stack ensures type safety because when the new version of the program resumes it will be self consistent. If a programmer changes the type signature of a method m, for the program to compile properly, the programmer must also change any methods that call m.

[2] UPT is built using jclasslib: http://www.ej-technologies.com/products/jclasslib.

In our example, the fact that `setForwardedAddresses` changed type necessitated changing the function `loadUser` to call it with the new type. With this safety condition, there is no possibility that the signature of method `m` could change and some old caller could call it—the update must also include all updated callers of `m`.

Category (2) methods are more subtle. Suppose some method `getStatus` calls method `getForwardedAddresses` from our example, but `getStatus` source code and bytecode has not changed from versions 1.3.1 to 1.3.2. Nevertheless, `getStatus`'s *machine code*, produced by the JIT compiler, may need to be recompiled. For example, if the new compiled version of `getForwardedAddresses` is at a different offset than before, then the VM must recompile `getStatus` to correctly refer to the new offset. An update may also change field offsets in modified classes, which requires recompiling any class that accesses them as well. Ginseng [32] and POLUS [13], two DSU systems for C, likewise consider functions as changed if their source code is the same but they access data types whose (compiled) representation is different. Note that the VM would not need to restrict category (2) methods if it used an interpreter that looked up offsets at each access.

Even if a method has not changed, a user may need to manually blacklist it. For example, suppose a method `handle` calls methods `process` and `cleanup`, and the method `cleanup` initializes a field that it uses. Now suppose we update this program to move the initialization statement into `process`, because `process` needs to use the field as well. In both versions, the field is properly initialized when the program runs from scratch. However, suppose that JVOLVE applies the update and the thread running `handle` yields in between the calls to `process` and `cleanup`. In this case, `handle`'s bytecode has not been changed (`process` and `cleanup` are method body changes, not class updates), so we could go ahead with the update. But if we did, then the program would have called the old `process` method, which did not perform any initialization, and then would call the new `cleanup` method, which performs no initialization either, since it the new version `process` does it, leading to incorrect semantics. To avoid such *version consistency* problems [31] the programmer can include `handle` in the restricted set. Our benchmarks did not require manual restrictions.

Finally, note that when the VM JIT compiler uses inlining we may need to increase the number of restricted methods to include those into which restricted methods are inlined. In particular, if a category (1), (2), or (3) method `m` is inlined into method `n`, we should also restrict `n` (and recompile it, after the update) to prevent the old `m` from running after the update. Jikes RVM initially compiles a method with its *base*-compiler, which generates machine code but does not apply sophisticated optimizations. Based on run-time profiling information, the VM may recompile the same method later using its *opt*-compiler, which performs standard optimizations, including inlining. It performs inlining of small, frequently used methods; cost-based inlining for larger methods; and may inline multiple levels down a hot call chain. As a consequence, JVOLVE restricts inlined callers of restricted methods.

Reaching a DSU safe point. To safely perform VM services such as thread scheduling, garbage collection, and JIT compilation, Jikes RVM (like most production VMs) inserts yield points on all method entries, method exits, and loop back edges. If the VM wants to perform a garbage collection or schedule a higher priority thread, it sets a yield flag, and the threads stop at the next VM safe point. JVOLVE piggybacks on this functionality. When JVOLVE is informed that an update is available, it sets the yield flag. Once application threads on all processors have reached VM safe points, JVOLVE checks the paused threads' stacks. If no stack refers to a restricted method, JVOLVE applies the update.

If any thread is running a restricted method, JVOLVE defers the update and installs a *return barrier* [43] on the topmost restricted method of each thread. A generic return barrier replaces the regular method return branch back to the next instruction in the calling method with a branch to *bridge code*, which performs some special action and then executes the return branch. We added this generic return barrier functionality to Jikes RVM, but this technology is standard in other VMs. Our bridge code restarts the update process. Therefore, when a restricted method returns, the thread will block and JVOLVE will restart the update process, which will either reach a DSU safe point, or the VM will insert more return barriers. If JVOLVE does not reach a safe point within 15 seconds, it aborts the update (the length of timeout is arbitrary, and can be configured by the user). However, it turns out we can sometimes proceed with an update despite category (2) methods on-stack, as described next.

Lifting category (2) restrictions. JVOLVE reduces the number of restricted methods in category (2) by leveraging VM support for on-stack replacement (OSR). Jikes RVM normally uses OSR to replace a *base*-compiled version of an active method with an optimized version. We observe that for category (2) restricted methods, the situation is much the same: an unchanged, on-stack method requires recompilation, in our case to fix any changed offsets. If the stack only contains category (2) methods, JVOLVE first performs OSR, and then starts the update. As of this writing, we only support OSR for *base*-compiled category (2) methods, which do not contain any inlined calls, though we plan to support OSR on *opt*-compiled methods as well.

Jikes RVM's standard OSR functionality works as follows. After reaching a yield point, OSR recompiles the topmost method on a thread's stack. The VM then modifies the thread's current PC to switch to the equivalent location in the new implementation, and adjusts the stack to reflect the recompiled version. Jikes RVM examines the active stack frame and extracts the values of local variables. It generates a special prologue to the recompiled method that sets up a stack frame with these extracted values. The last instruction in this prologue jumps to the new PC location. The VM then overwrites the return address of the yield point function to jump to the prologue.

We extend Jikes RVM's OSR facilities to support multiple stack activation records, and multiple stack frames on the same stack. This later addition makes it more likely to reach a DSU safe point when a string of category (2) methods precede a changed method on the stack. Given this support, JVOLVE ignores *base*-compiled category (2) methods when testing for a safe point. If any *base*-compiled category (2) methods are on stack at an otherwise DSU-safe point, JVOLVE uses OSR to replace them. The exact timing of OSR for DSU requires the VM to first load modified classes, as explained next.

3.3 Installing modified classes

Once the program reaches a safe point, JVOLVE begins the update by loading and installing the changed classes, and updating relevant metadata in the existing versions.

Jikes RVM represents classes with several internal data structures. Each class has an `RVMClass` meta-object that describes the class. It points to other meta-objects that describe the class's method and field types and offsets in an object instance. The compiler and garbage collector query this metadata. Often the compiler can statically determine the type of the object reference. In this case, it queries the meta-object and hard-codes constant offsets into the machine code to generate efficient field and method access code. The garbage collector uses meta-objects to identify object reference fields and trace the referent objects. `RVMClass` also stores a *type information block* (TIB) for each class, which maps a method's offset to its actual implementation. Jikes RVM always compiles a method directly to machine code when the method is first invoked. Each object instance contains a pointer to its TIB, to

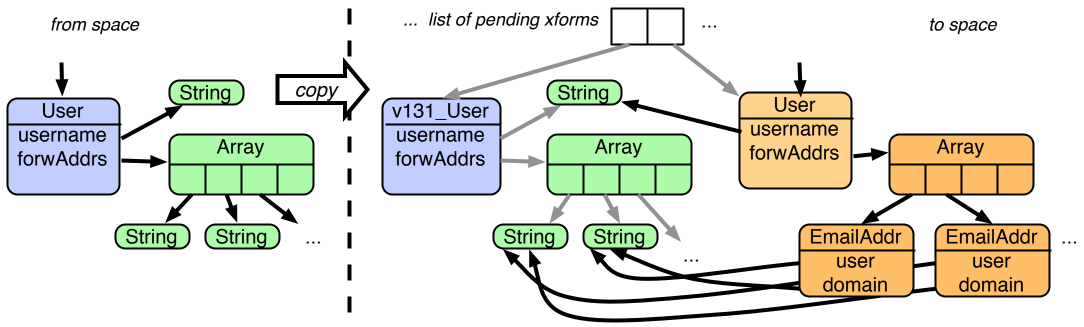

Figure 4. Running object transformers following GC

support dynamic dispatch. When the program invokes a method on an object, the generated code indexes the object's TIB at the correct offset and jumps to the machine code.

For a class with only method body updates, all of the class's metadata is the same in both the old and new versions. Therefore, JVOLVE invalidates the TIB entries for each replaced method, reads in the new method body implementations, and modifies the existing class metadata to refer to the replacement methods' bytecode. The JIT will compile the updated method when the program next invokes it, after the update.

For a class update, the class's number, type, and order of fields or methods may have changed, which in turn impacts the class's metadata, including its TIB. JVOLVE modifies existing class metadata as follows. First, it changes the old class's metadata to use a modified class name, e.g., metadata for class `User` is renamed to `v131_User` in our example update from Figure 2 and 3. Next, it installs the new `RVMClass` and corresponding metadata for the new version. Then the VM updates several Jikes RVM data structures (e.g., the Java Table of Contents for `static` methods and fields) to indicate that the newly-loaded class is now the up-to-date version. Note that all TIB entries for the newly-installed class are invalid, so all methods in the class will be compiled on demand. JVOLVE invalidates the TIB entries and other data structures for the old class so that they can be garbage-collected.

Once method and class updates are installed, if category (2) methods are active, the VM initiates OSR for these methods.

Invalidating changed methods will impose overhead on the execution just following the update when these methods are first *base*-compiled and then when they are progressively optimized at higher levels, if they execute frequently. We could reduce this overhead somewhat by optimizing new versions directly to their prior level of optimization. Updates to method bodies however invalidate execution profiles and without branch and call frequencies, code quality would degrade. Thus, we believe it is better to let the adaptive compiler work as it was intended. In any case, since dynamic updates are relatively rare events, any added overhead due to recompilation will be short-lived.

3.4 Applying Transformers

We modify the Jikes RVM semi-space copying collector [9] to update changed objects as part of a collection. The collector transforms old objects of an updated classes to conform to their new class signature and point to their new TIB. A semi-space copying collector normally works by traversing the pointer graph in the old heap (called *from-space*) starting at the *roots* and performing a transitive closure over the object graph, copying all objects it encounters to a new heap (called *to-space*). The roots include statics, stack-allocated local variables, and references in registers. The compiler generates a *stack map* at every VM safe point (a superset of DSU safe points). The stack map enumerates all register and local variables on the stack that reference heap objects. When the collector first encounters an object, it copies it to to-space and then overwrites its header with a forwarding pointer to the new copy. If the collector encounters a forwarded object later via another reference, it uses the forwarding pointer to redirect the reference to the new object.

Our modified collector works in much the same way, but differs in how it handles objects whose class signature has changed. In this case, it allocates a copy of the old object *and* a new object of the new class, which may have a different size compared to the old one. The collector initializes the new object to point to the TIB of the new type, and installs the forwarding pointer in from-space to this new version. Next, the collector stores a pair of pointers in its *update log*, one to the copy of the old object and one to the new object. The collector continues scanning the old copy.

After the collection completes, JVOLVE first executes transformers for all classes and then for all objects. JVOLVE goes through the update log and invokes the object transformer, passing the old and new object pair as arguments. Once it processes all pairs, the log is deleted, making the duplicate old versions unreachable. Since they are unreachable, the next garbage collection will naturally reclaim them. If we put them in a special space, we could reclaim them immediately.

Example. Figure 4 illustrates a part of the heap at the end of the GC phase while applying the update from Figure 3 (forwarding pointers not shown). On the left is a depiction of part of the heap prior to the update. It shows a `User` object whose fields point to various other elided objects. After the copying phase, all of the old reachable objects are duplicated in to-space. The transformation log points to the new version of `User` (which is initially empty) and the duplicate of the old version, both of which are in to-space. The transformer function can safely copy fields of the `from` object. The figure shows that after running the transformer function, the new version of the object points to the same `username` field as before, and it points to a new array which points to new `EmailAddress` objects. The `EmailAddress` constructor called within the transformer function initialized these objects by referring to the old e-mail `String` values and assigning fields to point to substrings of the given `String`.

In our example, the `jvolveObject` function only copies the contents of the old `User` object's fields. More generally, our update model allows old object fields to be dereferenced in transformer functions so long as the fields point to transformed objects. If some object o is dereferenced while running p's transformer method, but o has not yet been transformed, we must find o and pass it and its uninitialized new version to the `jvolveObject` method to initialize it. Since p points to the new version of o, we could scan the remainder of the update log to find the old version. To avoid this cost, we instead cache a pointer to the old version in the new version during the collection. We take care that `jvolveObject` functions invoked recursively in this manner do not loop infinitely, which would constitute one or more ill-defined transformer functions. We detect cycles with a simple check, and abort the update. In our current implementation, the programmer uses a special VM function to force a field's referenced object to be transformed. We should be able to handle this case automatically, through a read barrier or a simple analysis of the `jvolveObject` bytecode.

3.5 Discussion

Our implementation of object transformers uses an extra copy of all updated objects and adds temporary memory pressure. We could instead copy the old versions to a special block of memory and reclaim it when the collection completes. We could attempt to avoid extra copies altogether by invoking object transformer functions during collection. This approach is more complicated because our transformer model requires recursively invoking the collector from the transformer if a dereferenced field has not yet been processed. We also would need to use a GC-time read barrier to follow forwarding pointers before dereferencing an object in order to determine whether an object has been transformed.

We use a stop-the-world garbage collection-based approach that requires the application to pause for the duration of a full heap GC. This pause time could be mitigated by piggybacking on top of a concurrent collector. We could also consider applying object and class transformers lazily, as they are needed [37, 26, 11, 32, 13]. The main drawback here is efficiency. The VM would need to insert code to check, at each dereference, whether the object is up-to-date, imposing extra overhead on steady-state execution. Moreover, stateful actions by the program after an update may invalidate assumptions made by object transformer functions. It is possible that a hybrid solution could be adopted, similar to Chen et al. [13], which removes checking code once the system updates all objects. We leave exploration of these ideas to future work.

Finally, our OSR support is currently limited to on-stack methods whose bytecode has not changed. We plan to further extend OSR to support *changed* methods on the stack, similar to what is provided by UpStare, a DSU system for C [24]. For changed methods the user wishes to update while they run, she must additionally provide a mapping between the yield points in the old method to similar points in the new method. For example, a common change is to modify the contents of an event handling loop. The user would map the yield point at the end of the old loop to the yield point at the end of the new loop. The user would also have to provide the analogue of an object transformer for initializing the contents of the new method's stack frame, given the old stack frame contents. As with object transformers, this update model poses a question: should the stack frame transformer be allowed to dereference objects in the old stack frame if they too have changed? We leave exploration of updating active methods to interesting future work.

4. Experience

To evaluate JVOLVE, we used it to update three open-source servers written in Java: the Jetty webserver[3], JavaEmailServer,[4] an SMTP and POP e-mail server, and CrossFTP server.[5] These programs belong to a class that should benefit from DSU because they typically run continuously. DSU would enable deployments to incorporate bug fixes or add new features without having to halt currently-running sessions.

Config.	Throughput (MB/s)		Latency (ms)	
	Median	Quartiles	Median	Quartiles
Jikes RVM	122.437	121.44–123.32	0.442	0.394–0.496
JVOLVE	121.308	121.12–121.41	0.349	0.341–0.351
JVOLVE upd	121.242	121.09–121.29	0.345	0.341–0.349

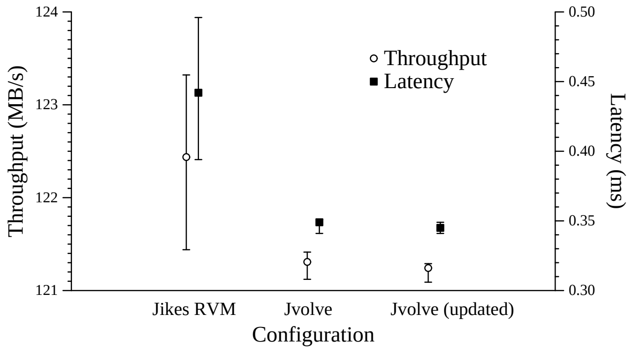

Figure 5. Throughput and latency measurements of Jetty webserver v5.1.6

We explored updates corresponding to releases made over roughly one to two years of each program's lifetime. Of the 22 updates we considered, JVOLVE could support 20 of them—the two updates we could not apply changed classes with infinitely-running methods, and thus no safe point could be reached. To our knowledge, no existing DSU system for Java could support all these updates; indeed, previous systems with simple support for updating method bodies would be able to handle only 9 of the 22 updates. Although JVOLVE cannot support every update, it is the first DSU system for Java that has been shown to support changes to realistic programs as they occur in practice over a long period of time.

In the rest of this section, we first examine the performance impact of JVOLVE, and then look at updates to each of the three applications in detail.

4.1 Performance

The main performance impact of JVOLVE is the cost of applying an update; once updated, the application eventually runs without further overhead. To confirm this claim, we measured the throughput and latency of two Jetty versions while running on stock Jikes RVM and on JVOLVE after dynamically updating to the next version. The performance of these configurations is essentially identical.

The cost of applying an update is the time to load any new classes, invoke a full heap garbage collection, and to apply the transformation methods on objects belonging to updated classes. Roughly, the time to suspend threads and check that the application is in a safe-point is less than a millisecond, and classloading time is usually less than 20ms. Therefore the update disruption time is primarily due to the GC and object transformers, and is proportional to the size of the heap and the fraction of objects being transformed. We wrote a simple microbenchmark to measure these overheads. This experiment shows that object transformation is the dominant cost.

We conducted all our experiments on an Intel Core 2 Quad machine running at 2.4 GHz machine with 2 GB of RAM. The ma-

[3] `http://www.mortbay.org`

[4] `http://www.ericdaugherty.com/java/mailserver/`

[5] `http://www.crossftp.com/`

# objects	Heap size	Fraction of updated objects 0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
Garbage collection time (ms)												
280000	160 MB	78.2	81.3	83.1	89.3	99.0	103.2	108.3	113.2	113.3	120.3	120.0
770000	320 MB	148.9	165.0	181.9	195.8	213.2	223.2	237.0	249.0	262.0	269.5	278.6
1760000	640 MB	313.3	347.7	382.9	416.0	449.8	478.9	506.8	534.0	558.8	583.7	601.5
3670000	1280 MB	615.4	694.6	763.0	833.6	900.1	965.9	1019.0	1076.4	1129.9	1181.2	1217.5
Running transformation functions (ms)												
280000	160 MB	0.1	13.0	23.2	34.6	43.9	54.0	62.7	74.5	84.1	93.9	104.2
770000	320 MB	0.1	33.7	63.1	91.2	116.8	145.4	173.9	201.0	231.3	262.0	292.6
1760000	640 MB	0.1	77.9	143.9	207.7	269.5	333.7	397.6	464.0	534.6	604.5	674.9
3670000	1280 MB	0.1	160.8	299.2	429.4	560.2	693.8	827.3	975.0	1119.6	1263.7	1405.4
Total DSU pause time (ms)												
280000	160 MB	82.8	99.0	109.5	128.0	147.6	161.2	174.5	192.8	202.5	218.8	228.1
770000	320 MB	153.6	202.9	249.0	291.4	334.5	372.6	414.8	455.4	498.1	535.3	576.8
1760000	640 MB	316.6	429.5	530.5	627.2	723.4	816.0	908.6	1002.6	1097.5	1191.5	1281.2
3670000	1280 MB	618.7	859.0	1065.9	1269.9	1466.1	1663.6	1850.8	2054.2	2253.1	2448.5	2627.9

Table 1. Microbenchmark results: JVOLVE update pause time (in ms) for various heap sizes

chine ran Ubuntu 7.10 on Linux kernel version 2.6.22. We implemented JVOLVE on top of Jikes RVM (SVN r15532).

Jetty performance. To see the effect of updating on application performance, we measured Jetty under various configurations using httperf, a webserver benchmarking tool.[6] We used httperf to issue roughly 800 new connection requests/second, which we observed to be Jetty's saturation rate. Each connection makes 5 serial requests for a 40 Kbyte file. Httperf reports average throughput and average per-request latency over a 60 second period. We ran this experiment 21 times and report the median and quartiles of the throughput and latency reports. With 21 runs, the range between the quartiles serves as a 98% condence interval [36]. In order to eliminate network traffic effects, we ran the server on two cores of a quad-core machine and the client on another core.

Figure 5 shows our results in tabular form and plotted graphically. The second and third columns of the table report the median throughput and the range between the two quartiles. The third column and fourth column report the median latency and the inter-quartile range. The first and seconds rows illustrate the performance of Jetty version 5.1.6 running on stock Jikes RVM and JVOLVE, respectively. The third row shows the performance on JVOLVE of Jetty 5.1.6 dynamically updated from version 5.1.5 prior to the start of the experiment. The performance of the two JVOLVE configurations is essentially identical: the two configurations' corresponding inter-quartile ranges largely overlap. The performance of JVOLVE is also quite similar to the performance of stock Jikes RVM. There are many small differences between JVOLVE and the stock implementation that change VM code size, code layout, and garbage collection behavior. These differences may impact performance directly and they may indirectly affect other elements of the VM, such as the timing of garbage collections and JIT optimizations (such indirect effects make VMs notoriously difficult to benchmark [10]).

Microbenchmarks. The two dominant factors that determine JVOLVE update time are the time to perform a GC, determined by the number of objects, and the time to run object transformers, determined by the fraction of objects being updated. To measure the cost of each, we devised a simple microbenchmark that creates an array of objects and transforms a specified fraction of these objects when a JVOLVE update is triggered. The microbenchmark has two simple classes, `Change` and `NoChange`. Both contain three integer fields, and three reference fields that are always `null`. The

[6] `http://www.hpl.hp.com/research/linux/httperf`

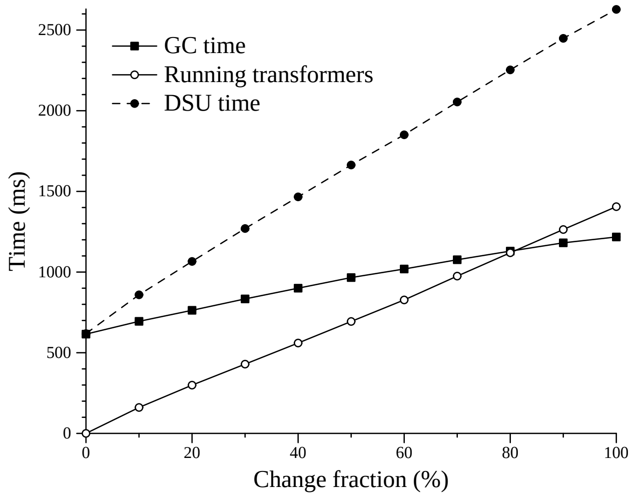

Figure 6. Microbenchmark pause times with a heap size of 1280 MB containing 3.67 million objects

update adds an integer field to `Change`. The user-provided object transformation function copies the existing fields and initializes the new field to zero. We measure the cost of performing an update while varying the total number of objects and the fraction of objects of each type. The number of objects is the maximum that can fit in heap sizes 32, 64, 128 and 256 MB. Note that Jikes RVM's heap includes VM data structures as well. We measure the running time in a generous heap, five times the minimum required size, such that the only collections are those DSU triggers. We report the median of 21 runs.

Table 1 shows the elapsed time while varying the number of total objects and the fraction of the objects that are updated. The variance was insignificant, so we do not report it. The first group of rows reports garbage collection time, the second group reports the time to transform all updated objects, and the final group reports the total update time, which includes the sum of the GC and transformation time, the time to load and install the updated classes, synchronize running threads, and find a DSU safe point. The first column of the table shows the number of objects in the test, and the second column the heap size. Columns 3 though 13 show pause times for varying fractions (from 0% to 100%) of updated objects.

Ver.	# classes added	# changed classes	methods add	methods del	methods chg	fields add	fields del
5.1.1	0	14	4	1	38/0	0	0
5.1.2	1	5	0	0	12/1	0	0
5.1.3*	3	15	19	2	59/0	10	1
5.1.4	0	6	0	4	9/6	0	2
5.1.5	0	54	21	4	112/8	5	0
5.1.6	0	4	0	0	20/0	5	6
5.1.7	0	7	8	0	11/2	9	3
5.1.8	0	1	0	0	1/0	0	0
5.1.9	0	1	0	0	1/0	0	0
5.1.10	0	4	0	0	4/0	0	0

Table 2. Summary of updates to Jetty

To shed light on the results in the table, Figure 6 plots collection time, transformer time and total update time for the microbenchmark with 3.67 million objects in a 1280 MB heap. The figure shows that the costs of garbage collection and transformation increase as a function of the number of changed objects. The slope of the "GC time" line illustrates the cost to deal with an increasing number of transformed objects. This cost includes creating an additional copy of each transformed object; creating the update log entry with a pointer to the old and new copy; and caching a pointer to the old copy from the new copy. The slope of the "Running transformers" line illustrates the added cost of iterating over the update log and actually running the transformers. This extra processing to handle transforming objects increases the total pause time with all objects updated by roughly four times compared to the pause time with no object updated. The "Running Transformers" line is steeper than the "GC time" line, revealing that the cost of running transformers is higher than the extra copying cost incurred during GC.

Transformations are more expensive than standard copying GC. The GC uses `memcopy`, which is highly optimized, whereas our transformer functions use reflection to look up `jvolveObject`, and this function copies one field at a time. One optimization would be to eliminate the log by copying the old and new objects to their own space and walking through and transforming each object. The cost of reflection could be reduced by caching the lookup, but even then a naïvely compiled field-by-field copy is much slower than the collector's highly-optimized copying loop.

4.2 Jetty webserver

Jetty is a popular webserver written in Java. It supports static and dynamic content and can be embedded within other Java applications. Jikes RVM, and thus JVOLVE, is not able to run the most recent versions of Jetty (6.x). Therefore we considered 11 versions, consisting of 5.1.0 through 5.1.10 (the last version prior to version 6). Version 5.1.10 contains 317 classes and about 45,000 lines of code. Table 2 shows a summary of the changes in each update. Each row tabulates the changes relative to the prior version. For the column listing changed methods, the notation x/y indicates that $x + y$ methods were changed, where x changed in body only, and y changed their type signature as well. For dynamic updating systems that only support changes to method bodies, only the first and last three of the ten updates could be supported, since the rest either change method signatures and/or add or delete fields.

Reaching a safe point in Jetty. We successfully wrote dynamic updates to all versions of Jetty that we examined. For each version starting at 5.1.0, we ran Jetty under full load. After 30 seconds we tried to apply the update to the next version. We did this five times per version. Other than the update to 5.1.3, all versions immediately reached a safe point every time, with no need of return barriers.

We could not apply the update to version 5.1.3 (denoted with an asterisk in the table) because JVOLVE was never able to reach a safe point. The update modified `ThreadedServer.acceptSocket()`, a method that waits for a connection from the client, and this method is nearly always on stack. We installed a return barrier that is triggered when `acceptSocket` returns, but this barrier is not sufficient to perform the update since the main methods of several threads were themselves modified. For example, we also install a return barrier on `PoolThread.run()`, but this barrier is never triggered because this method never becomes inactive.

Ver.	# classes add	# classes del	# changed classes	methods add	methods del	methods chg	fields add	fields del
1.2.2	0	0	3	0	0	3/0	0	0
1.2.3	0	0	7	0	0	14/2	12	0
1.2.4	0	0	2	0	0	4/0	0	0
1.3*	4	9	2	11	3	6/9	12	5
1.3.1	0	0	2	0	0	4/0	0	0
1.3.2	0	0	8	4	2	4/2	3	1
1.3.3	0	0	4	0	0	3/0	0	0
1.3.4	0	0	6	2	0	6/0	2	0
1.4	0	0	7	6	1	4/1	6	0

Table 3. Summary of updates to JavaEmailServer

Ver.	# classes add	# classes del	# changed classes	methods add	methods del	methods chg	fields add	fields del
1.06	4	1	1	0	0	3/0	1	0
1.07	0	0	3	4	0	14/0	5	0
1.08	0	1	3	2	0	10/0	0	2

Table 4. Summary of updates to CrossFTP server

4.3 JavaEmailServer

For JavaEmailServer, we considered 10 versions—1.2.1 through 1.4—spanning a duration of about two years. Version 1.4 consists of 20 classes and about 5000 lines of code. Table 3 shows the summary of changes for each new version. Approaches that only support updates to method bodies will be able to handle only four of these updates. We could successfully construct updates for all versions we examined, and we could successfully apply all of them except the update to version 1.3. This update reworks the configuration framework of the server, among other things removing a GUI tool for user administration and adding several new classes that implement a file-based configuration system. As a result, many of the classes are modified to point to a new configuration object. Among these classes are threads with infinite processing loops (e.g., to accept POP and SMTP requests). Because these threads are always active, the safety condition can never be met and thus the update cannot be applied.

The update from 1.3.1 to 1.3.2 indirectly changes the `SMTPSender.run()` and `Pop3Processor.run()` methods. These methods contain processing loops run by several threads. Though these methods are always running, JVOLVE applies OSR and the update succeeds. JVOLVE also uses OSR for the update from 1.3.2 to 1.3.3.

4.4 CrossFTP server

CrossFTP server is an easily configurable, security-enabled FTP server. CrossFTP allows simple configuration through a GUI and more advanced customization using configuration files. We did not use the GUI interface and therefore do not consider changes to that part of the program. We looked at 4 versions— 1.05 through 1.08, details shown in Table 4—spanning a duration of more than a year. Version 1.08 contains about 18,000 lines of code spread across 161 classes. JVOLVE successfully applies all three updates to this application. Note that since all updates either add or delete fields, simple method body updating support on its own would be insufficient.

JVOLVE could only apply the update from version 1.07 to 1.08 when the server was relatively idle. The server runs a new `RequestHandler` thread to process each FTP session, and the `RequestHandler.run()` method was changed by the update. JVOLVE installs a return barrier on this method, but with many active sessions, this method is essentially always on stack, preventing the update. Future work could address this problem using scheduler support for coordinating updates among active threads [30].

5. Related Work

We compare our VM-centric approach to DSU with related work on implementing DSU for managed languages, C, and C++.

Edit and continue. Debuggers and IDEs have long provided *edit and continue* (E&C) functionality that permits limited modifications to program state to avoid stopping and restarting during debugging. For example, Sun's HotSwap VM [27, 15], .NET Visual Studio for C# and C++ [14], and library-based support [17] for .NET applications all provide E&C. These systems restrict updates to code changes within method bodies. While this restriction reduces safety concerns and obviates the need for class and object transformers, the resulting systems are inflexible. They cannot perform more than half of the updates discussed in Section 4.

DSU for managed languages without VM support. To avoid changing the VM to support DSU, researchers have developed special-purpose classloaders and/or compiler support. The main drawbacks of these approaches are inflexibility and high overhead. For example, Eisenbach and Barr [4] and Milazzo et al. [28] use custom classloaders for binary-compatible and component-level changes respectively, but cannot perform class field additions.

Orso et al. [35] use source-to-source translation for DSU by introducing a proxy object that indirectly accesses an object that may change. This approach requires updated classes to export the same public interface, forbidding new public methods and fields. Non VM-based approaches are in general limiting because they are not *transparent*—they make visible changes to the class hierarchy, and insert or rename classes. This approach makes it essentially impossible to be robust in the face of code using reflection or native methods. Moreover, the indirection imposes time and space overheads on steady-state execution. Our VM approach naturally supports reflection and native methods (these are updated as well), and is more expressive, e.g., it supports signature changes.

VM support for DSU in managed languages. The PROSE system performs short-term, run-time patches to code for logging, introspection, and performance adaptation, rather performing general updates [33]. An Eclipse plug-in performs run-time bytecode instrumentation and a modified JIT performs method code replacement, using an API in the style of aspect-oriented programming. PROSE has the same update model as the E&C systems: it supports updates to method bodies but not class or method signature changes that require changes to object state.

JDrums [37] and Dynamic Virtual Machine (DVM) [26] both implement DSU for Java within the VM, providing a programming interface similar to JVOLVE, but are lacking in two ways. First, neither JDrums nor DVM have ever been demonstrated to support updates from real-world applications. Second, their implementations impose overheads during steady-state execution. They both update *lazily* and use an extra level of indirection (the *handle space*). Indirection conveniently supports object updates, but adds extra overhead. For example, JDrums traps all object pointer dereferences to apply VM object transformer function(s) when the object's class changes. Lazy updating has the advantage that it amortizes pauses due to an update over subsequent execution. The main drawback is that its overhead persists during normal execution, even though updates are relatively rare. DVM works only with the interpreter. Relative to this interpreter, which is already slow, the extra traps result in roughly 10% overhead.

Compared to these two, JVOLVE performs updates eagerly by employing a full heap collection at update-time. This stop-the-world approach imposes a longer pause at update time, but eliminates overhead during steady-state execution. Likewise, by invalidating updated methods, JVOLVE's performance is slowed just after the update as these methods are being recompiled. However, compared to running with an interpreter, steady-state execution is much improved, since methods will be much better optimized.

Boyapati et al. [11] support dynamic updates to classes kept in a *persistent object store* (POS). While the setting is different, their basic update model, and in particular their notion of object transformer function, is similar to ours. In their system, programmers manually write an object transformer that they view as a method on the old version of the updated class, i.e., the transformer method is type-safe with respect to the old class. In JVOLVE, object transformers may access the *new* versions of objects pointed to from the old class. Instead, Boyapati et al.'s transformers may access the *old* versions. To implement this model, they rely on *encapsulation* based on *ownership types*: if an object a of class A has an "owned" field pointing to an object b of class B, then only a can point to (and access) b. Encapsulation thus ensures the system will always transform a before b, which makes the transformation algorithm more efficient. They rely on the programmer to enforce encapsulation, and describe how the compiler could automate language support for encapsulation in a non-standard type system. JVOLVE takes the opposite tack of forcing old object fields to point to up-to-date objects, and thus requires no special language support. Moreover, JVOLVE's model follows that of earlier work [5, 32, 30, 24] which has proven its effectiveness on a half-dozen realistic applications across several years' worth of releases. However, further research to understand the costs and benefits of the two updating models would be useful.

Boyapati et al. also differs from JVOLVE in that, like JDRUMS and DVM, updates are applied incrementally as objects are accessed following an update rather than all at once using a stop-the-world GC. This incremental cost is more natural in a POS since indirection is already required to access external objects. The POS model also permits programmers to specify ACID transaction boundaries, which can help ensure that updates are applied consistently and safely. In contrast, our work focuses on supporting dynamic upgrades in a high-performance VM for Java, and thus many of the issues we consider—reaching a safe point via return barriers and OSR, and coexisting with the JIT compiler—are the unique contributions of our work.

Dynamic Software Updating for C/C++ There are several substantial systems for dynamically updating C and C++ programs that target server applications [22, 2, 32, 13, 24, 30] and operating systems components [39, 5, 12, 23, 25]. Although some of these systems are mature, the flexibility afforded by JVOLVE is comparable or superior. JVOLVE's timing restrictions and Java's type safety also provide comparable or superior safety; the fact that C and C++ programs often circumvent the languages' weak type systems greatly complicates efforts to ensure that updates behave correctly. Some prior systems [30, 24, 13] have focused on means to reach DSU safe points quickly, and we plan similar efforts as future work. In particular, we plan to extend our support for OSR to apply to running methods whose bytecode has changed, allowing the user to map an active method's PC and stack frame and those of its new version, similar to support provided by UpStare [24].

The lack of a VM is a disadvantage for C and C++ DSU. For example, because a VM-based JIT can compile and recompile replacement classes, it imposes no steady-state execution overhead. By contrast, C and C++ implementations must use either statically-inserted indirections [22, 32, 39, 5, 24] or dynamically-inserted trampolines to redirect function calls [2, 12, 13, 3]. Both cases impose persistent overhead on normal execution and inhibit optimization. Likewise, because these systems lack a garbage collector, they either do not update object instances at all [3], update them lazily [32, 13] or perform extra allocation and bookkeeping to locate the objects at update-time [5]. Finally, because these systems lack support for on-stack replacement, they must pre-compile potentially long-running methods specially, so that they can be updated while they run. These techniques impose time and space overheads on steady-state execution, and in some cases limit update flexibility.

Other proposals Gilmore et al. [20] propose DSU support for modules in ML programs using a similar, but more restrictive programming interface compared with JVOLVE. They formalize an abstract machine for implementing updates using a copying garbage collector. Duggan [16] also proposes dynamic updates to ML programs, focusing on lazy updates to data type definitions. Neither approach was ever implemented.

UpgradeJ [8] is an extension to the Java language design supporting class upgrades, in two flavors: *revision upgrades*, which may modify method bodies, and *evolution upgrades*, which may add new methods and fields. Programmers control the effects of upgrades using *version annotations*, introduced by Bierman et al. [7] in earlier work. For example, the programmer may write `o = new Button[1=]()` to force `o` to always use version 1 methods, while writing `p = new Button[1+]()` or `p = new Button[1++]()` allows `p` to be revised or evolved, respectively. UpgradeJ's update model is easier to implement than JVOLVE's because it need not change existing object instances. Of course, the downside is a loss of flexibility. Many of the updates to our benchmark applications change field contents and layout. UpgradeJ does not support these updates. On the other hand, evolution upgrades add power over simple method body updates, and consequently enable more real-world updates to be supported [41]. There is no implementation of UpgradeJ.

6. Conclusions

This paper presents JVOLVE, a Java virtual machine with support for dynamic software updating. JVOLVE is the most full-featured, best-performing implementation of DSU for Java published to date. We demonstrate its flexibility and safety by successfully applying updates for one to two years worth of releases for three programs: Jetty webserver, JavaEmailServer, and CrossFTP server. JVOLVE imposes no overhead during a program's steady-state execution. JVOLVE's DSU support builds naturally on top of existing VM services, including dynamic class loading, thread synchronization, return barriers, on-stack replacement, JIT compilation, and garbage collection. It is probably optimistic to believe that DSU will be able to support every update. Nevertheless, our results demonstrate that dynamic software updating support can be naturally incorporated into modern VMs, and that doing so has the potential to significantly improve software availability by reducing downtime.

Acknowledgments

We thank Iulian Neamtiu and the anonymous reviewers for helpful comments on drafts of this paper. We thank the Jikes RVM community, Mike Bond, David Grove, and Filip Pizlo for helpful advice, particularly toward understanding Jikes RVM. This work is supported by NSF CNS-0346989, CCF-0811524, NSF CNS-0719966, NSF CCF-0429859, Intel, IBM, CISCO, and Microsoft. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

References

[1] B. Alpern, D. Attanasio, J J. Barton, A. Cocchi, S. Flynn Hummel, D. Lieber, M. Mergen, T. Ngo, J Shepherd, and S. Smith. Implementing Jalapeño in Java. In *Proc. OOPSLA*, 1999.

[2] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. OPUS: Online patches and updates for security. In *Proc. USENIX Security*, 2005.

[3] Jeff Arnold and Frans Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proc. EuroSys*, 2009.

[4] Miles Barr and Susan Eisenbach. Safe upgrading without restarting. In *Proc. ICSM*, 2003.

[5] A. Baumann, J. Appavoo, D. Da Silva, J. Kerr, O. Krieger, and R. W. Wisniewski. Providing dynamic update in an operating system. In *Proc. USENIX Annual Technical Conference*, 2005.

[6] Andrew Baumann, Jonathan Appavoo, Robert W. Wisniewski, Dilma Da Silva, et al. Reboots are for hardware: challenges and solutions to updating an operating system on the fly. In *Proc. USENIX Annual Technical Conference*, 2007.

[7] Gavin Bierman, Michael Hicks, Peter Sewell, and Gareth Stoyle. Formalizing dynamic software updating. In *Proc. Second International Workshop on Unanticipated Software Evolution*, 2003.

[8] Gavin Bierman, Matthew Parkinson, and James Noble. UpgradeJ: Incremental typechecking for class upgrades. In *Proc. ECOOP*, pages 235–259, 2008.

[9] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *Proc. ICSE*, 2004.

[10] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffman, A. M. Khan, 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. Wake up and smell the coffee: Evaluation methodology for the 21st century. *Communications of the ACM*, 51(8):83–89, August 2008.

[11] Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira, Chuang-Hue Moh, and Steven Richman. Lazy modular upgrades in persistent object stores. In *Proc. OOPSLA*, 2003.

[12] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. Live updating operating systems using virtualization. In *Proc. VEE*, June 2006.

[13] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. POLUS: A POwerful Live Updating System. In *Proc. ICSE*, 2007.

[14] Microsoft Corporation. Edit and continue. `http://msdn2.microsoft.com/en-us/library/bcew296c.aspx`, 2008.

[15] M. Dmitriev. Towards flexible and safe technology for runtime evolution of java language applications. In *Proc. Workshop on Engineering Complex Object-Oriented Systems for Evolution*, 2001.

[16] Dominic Duggan. Type-based hot swapping of running modules. *Acta Inf.*, 41(4-5):181–220, 2005.

[17] Marc Eaddy and Steven Feiner. Multi-language edit-and-continue for the masses. Technical Report CUCS-015-05, Columbia University Department of Computer Science, April 2005.

[18] Torbjörn Ekman and Görel Hedin. The Jastadd extensible Java compiler. In *Proc. OOPSLA*, 2007.

[19] Slashdot forum. Patch the kernel without reboots. http://tech.slashdot.org/article.pl?sid=08/04/24/1334234, April 2008. Consists of a lively technical debate about the benefits and drawbacks of in-place dynamic updates vs. using redundant hardware.

[20] Stephen Gilmore, Dilsun Kirli, and Chris Walton. Dynamic ML without dynamic types. Technical Report ECS-LFCS-97-378, LFCS, University of Edinburgh, 1997.

[21] Deepak Gupta, Pankaj Jalote, and Gautam Barua. A formal framework for on-line software version change. *IEEE Trans. Softw. Eng.*, 22(2):120–131, 1996.

[22] G. Hjálmtýsson and R. Gray. Dynamic C++ classes, a lightweight mechanism to update code in a running program. In *Proc. USENIX Annual Technical Conference*, 1998.

[23] Yueh-Feng Lee and Ruei-Chuan Chang. Hotswapping linux kernel modules. *J. Syst. Softw.*, 79(2):163–175, 2006.

[24] Kristis Makris and Rida Bazzi. Multi-threaded dynamic software updates using stack reconstruction. In *Proc. USENIX Annual Technical Conference*, 2009.

[25] Kristis Makris and Kyung Dong Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *Proc. EuroSys*, March 2007.

[26] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz Barnes. Runtime support for type-safe dynamic Java classes. In *Proc. ECOOP*, 2000.

[27] Sun Microsystems. Java Platform Debugger Architecture, 2004. This supports class replacement. See http://java.sun.com/javase/6/docs/technotes/guides/jpda/.

[28] Marco Milazzo, Giuseppe Pappalardo, Emiliano Tramontana, and Giuseppe Ursino. Handling run-time updates in distributed applications. In *Proc. SAC*, 2005.

[29] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proc. International Workshop on Mining Software Repositories*, 2005.

[30] Iulian Neamtiu and Michael Hicks. Safe and timely dynamic updates for multi-threaded programs. In *Proc. PLDI*, 2009.

[31] Iulian Neamtiu, Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *Proc. POPL*, 2008.

[32] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for C. In *Proc. PLDI*, 2006.

[33] Angela Nicoara, Gustavo Alonso, and Timothy Roscoe. Controlled, systematic, and efficient code replacement for running Java programs. In *Proc. Eurosys*, 2008.

[34] D. Oppenheimer, A. Brown, J. Beck, D. Hettena, J. Kuroda, N. Treuhaft, D. A. Patterson, and K. Yelick. Roc-1: Hardware support for recovery-oriented computing. *IEEE Trans. Comput.*, 51(2):100–107, 2002.

[35] Alessandro Orso, Anup Rao, and Mary Jean Harrold. A technique for dynamic updating of Java software. In *Proc. ICSM*, 2002.

[36] J. W. Pratt and J. D. Gibbons. *Concepts of Nonparametric Theory*. Springer-Verlag, 1981.

[37] Tobias Ritzau and Jesper Andersson. Dynamic deployment of Java applications. In *Proc. Java for Embedded Systems Workshop*, 2000.

[38] D. Scott. *Assessing the Costs of Application Downtime*. Gartner Group, 1998.

[39] C. Soules, J. Appavoo, K. Hui, D. Da Silva, G. Ganger, O. Krieger, M. Stumm, R. Wisniewski, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis. System support for online reconfiguration. In *Proc. USENIX Annual Technical Conference*, June 2003.

[40] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. *Mutatis Mutandis*: Safe and flexible dynamic software updating (full version). *TOPLAS*, 29(4):22, August 2007.

[41] Ewan Tempero, Gavin Bierman, James Noble, and Matthew Parkinson. From Java to UpgradeJ: an empirical study. In *Proc. International Workshop on Hot Topics in Software Upgrades (HotSWUp)*, 2008.

[42] The Jikes RVM Core Team. VM performance comparisons, 2007. http://dacapo.anu.edu.au/regression/perf/head.html.

[43] Taiichi Yuasa. Design and implementation of Kyoto Common Lisp. *Journal of Information Processing*, 13(3):284–295, 1990.

[44] Benjamin Zorn. Personal communication, based on experience with Microsoft Windows customers, August 2005.