Dynamic Software Updates: A VM-Centric Approach

Abstract

Software evolves to fix bugs and add features, but stopping and restarting existing programs to take advantage of these changes can be inconvenient and costly. Dynamic software updating (DSU) addresses these problems by updating programs while they run. This paper shows how virtual machine (VM) services can be extended and integrated to provide *flexible*, *safe*, and *efficient* DSU for managed languages. It presents the design and implementation of JVOLVE, a DSU-enhanced Java VM. JVOLVE adds to and coordinates VM classloading, just-in-time compilation, on-stack replacement, and garbage collection to support flexible updates that include additions, deletions, and replacements of fields and methods anywhere within the class hierarchy. JVOLVE uses bytecode verification and VM thread synchronization to check and schedule updates such that they never violate type-safety. With JVOLVE no overhead is incurred during normal execution. We test JVOLVE on three open-source programs, Jetty web server, JavaEmailServer, and CrossFTP server with updates matching actual releases that occurred over 1–2 years. We show that JVOLVE is flexible enough to support most (20 of 22) of these updates and produces programs that are as reliable and efficient as if started from scratch.

1. Introduction

Software is imperfect. To fix bugs and adapt software to meet 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 [29, 25]; maintenance costs [32]; and at the least, inconvenience for users, which can translate into serious security risks if patches are not applied promptly [2, 17]. Dynamic software updating (DSU) addresses these problems by updating programs while they run in a general-purpose manner, requiring no special software architecture and neither extra nor redundant hardware [27]. The challenge is to make DSU *safe*, such that updating a program is as correct as deploying it from scratch; *flexible*, supporting updates that occur in practice; and *efficient*.

Researchers have made significant strides toward making DSU practical for systems written in C or C++, supporting server feature upgrades [23, 8], security patches [2], and operating systems upgrades [30, 4, 19, 7, 17]. 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 [16] and several .NET languages [11] support on-the-fly method body updates, this support is too inflexible for all but the simplest updates—they support less than half of the changes in three Java benchmark programs we examine. Academic approaches [28, 20, 26] offer more flexibility, but have not been proven on realistic applications, and impose substantial *a priori* space and time overheads. The a priori overheads result because these approaches employ method and object indirection, which slows down normal execution in order to make an application DSU capable.

This paper presents the design and implementation of JVOLVE, a Java Virtual Machine (JVM) that implements DSU services in a manner that combines flexibility, safety, and efficiency, substantially improving on 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 while guaranteeing type-safety, and imposing no a priori space or time overheads.

JVOLVE DSU supports many common updates. Users can add, delete, or change existing classes. Changes may add new 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 a default value and retains values of unchanged fields. The programmer may provide custom transformers with the update instead.

JVOLVE relies on bytecode verification to ensure that updated classes are statically type-safe. To avoid type errors due to the timing of an update [23, 4], JVOLVE stops the executing threads at a *DSU safe point* and then applies the update. DSU safe points are a subset of garbage collection (GC) safe points that restrict which methods can be on each thread's stack. Methods in updated classes, modified methods, methods that refer to fields and offsets of updated classes, and additional methods specified by the user for safety reasons [14] cannot be active on stack. Though limiting in principle, these restrictions have not proven problematic in practice. We have implemented extensions to Jikes RVM's support for recompiling active methods (so called on-stack replacement) and find that it can remove some of these restrictions.

JVOLVE uses the class loader to incorporate new and updated classes into the running program. JVOLVE invalidates existing compiled code for all directly and transitively modified methods. JVOLVE then applies object transformers by piggybacking on a whole-heap GC to find and transform existing object instances of changed classes. The application then resumes normal execution and any invalidated methods are JIT-compiled when the application next invokes them.

JVOLVE imposes no overhead during normal execution. The modifications and overheads of class loading, recompilation, and garbage collection are only imposed during an update, which is a rare occurrence. The zero overhead for a VM-based approach is in contrast to DSU techniques for C and C++ that use a compiler or dynamic rewriter to insert levels of indirection [23, 26] or trampolines [7, 8, 2, 17], which degrade performance during normal execution.

We assess 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. JVOLVE successfully applies 20 of the 22 updates—the two updates it cannot apply 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 contribution is the design and implementation of JVOLVE, a Java VM with support for dynamic software updating, distinguished from prior work in its realism, flexibility, technical novelty, and high performance. We believe our demonstration is a significant step towards supporting highly flexible, efficient, and safe updates in managed code virtual machines.

2. 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 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 [16, 9], .NET Visual Studio for C# and C++ [11], and library-based support [10] for .NET applications all provide E&C. These systems typically support only code changes within method bodies. While this reduces safety concerns, and obviates the need for class and object transformers, it is inflexible: more than half of the updates discussed in Section 5 could not be supported.

DSU for managed languages without VM support. To avoid changing the VM to support DSU, researchers developed special-purpose classloaders and/or compiler support. The main drawbacks of these approaches are inflexibility and high overhead. Eisenbach and Barr [3] and Milazzo et al. [21] use custom classloaders to allow binary-compatible changes and component-level changes, respectively, but for example, cannot support class field additions.

Orso et al. [26] support DSU via source-to-source translation by introducing a proxy class that indirects accesses to objects that could change. This approach requires updated classes to export the same public interface, forbidding new public methods and fields. A more general limitation of non VM-based approaches is that they are not *transparent*—they make changes to the class hierarchy, insert or rename classes, etc. This approach makes it essentially impossible to be robust in the face of code using reflection or native methods. Moreover, the required runtime support imposes a priori time and space overheads. Our VM approach naturally supports reflection, native methods (since these are updated as well), and is more expressive, e.g., supporting signature changes.

VM support for DSU in managed languages. JDrums [28] and Dynamic Virtual Machine (DVM) [20] 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 a priori overheads during normal execution, Both prior VMs 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 the pause due to an update can be amortized over subsequent execution. The main drawback is that the 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. In contrast, JVOLVE performs its update eagerly together with a full heap collection and therefore imposes

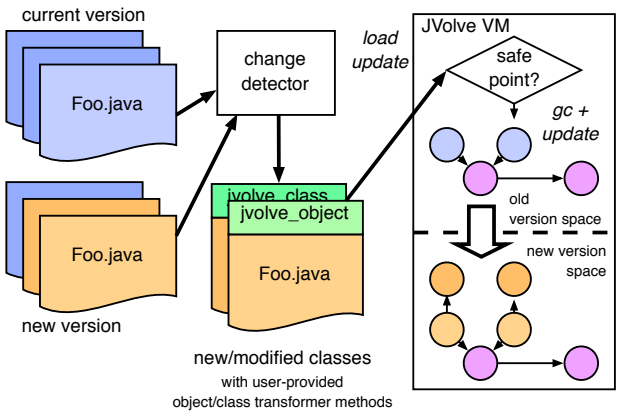

Figure 1. Dynamic Software Updating with JVOLVE

no overhead once an update is complete. JVOLVE also makes fewer restrictions on the JIT; e.g., inlining is fully supported.

PROSE performs run-time code patching with an API in the style of aspect-oriented programming [24]. PROSE aims to support short-term, run-time patches to code for logging, introspection, or performance adaptation, rather than for DSU. As such, PROSE only supports updates to method bodies, with no support for signature or state changes.

Gilmore et al. [13] propose DSU support for modules in ML programs using a similar, but more restrictive programming interface compared with JVOLVE. They formalized an abstract machine for implementing updates using a copying garbage collector, but omit update ordering constraints and did not implement it.

Boyapati et al. [6] support lazily upgrading objects in a *persistent object store* (POS). The programming interface is quite similar to JVOLVE: programmers provide object transformer functions for each class whose signature has changed. In their system, object transformers of some class A may access the state of old objects pointed to by A's fields, assuming these objects are fully encapsulated; i.e., they are only reachable through A. Encapsulation is ensured via extensions to the type system. By contrast, our transformers are more general and may dereference fields via old objects, but if these fields point to objects whose classes have been updated, they will see the *new* versions. We plan to explore the costs/benefits of these semantics in future work.

Dynamic Software Updating for C/C++ Several substantial systems for dynamically updating C and C++ programs have emerged that target server applications [15, 2, 23, 8] and operating systems components [30, 4, 7, 18, 19]. Although some of these systems are mature, the flexibility afforded by JVOLVE is comparable or superior; e.g., many existing systems do not support object-orientation of multithreading as JVOLVE does. Only Ginseng [23] imposes fewer restrictions on update timing than JVOLVE while retaining a comparable level of safety.

The lack of a VM is a significant disadvantage for C and C++ DSU. For example, because a VM-based JIT can compile and recompile replacement classes, it can update them with no persistent overhead. By contrast, C and C++ implementations must use either statically-inserted indirections [15, 23, 30, 4] or dynamically-inserted trampolines to redirect function calls [2, 7, 8, 17]. 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 [17], update them lazily [23, 8] or perform extra allocation and allocator book-

keeping to locate the objects at update-time [4]. These approaches limit update flexibility and/or impose time and space overheads on normal execution. Finally, the fact that C and C++ are not type-safe greatly complicates efforts to ensure that updates behave correctly.

3. Dynamic Updates in JVOLVE

This section overviews JVOLVE's approach and update model, and describes how the developer participates in the process.

3.1 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 that compiles correctly and is fully tested using standard procedures. A developer then passes a new version of the source tree to JVOLVE's *change detector*, which identifies new and changed classes and copies them into a separate directory.

The developer also may write some number of *object* and *class transformers*. Transformer methods take an object or class of the old version and produce an object or class of the new version. For example, if an update to class `Foo` adds a new instance field `x` and a new `static` field `y`, the programmer can write an object transformer (called `jvolve_object`) to initialize `x` for each updated object, and a class transformer (called `jvolve_class`) to initialize `y`. If the programmer chooses not to write one of these transformers, the system will dynamically produce a default one that simply initializes each new field to its default value and retains the values of any old fields that have not changed type. Transformer methods are the only portions of code where both views of the class definition are visible and they are only ever invoked at update time. This feature enables the programmer to develop the application largely oblivious to the fact that it will be dynamically updated.

JVOLVE translates the transformer functions to class files, which ensures (among other things) that they are type correct. The running VM then starts the updating process by loading the transformers and the new user class files. The update may proceed so long as all threads have reached a DSU *safe point* such that no thread's activation stack contains *restricted methods*. Restricted methods currently include all directly and *indirectly* updated methods. Users may also add restricted methods for safety reasons [14]. The modified thread scheduler suspends application threads as they reach safe points, and then applies the update. Section 4.2 describes extending Jikes RVM's on-stack replacement mechanism to reduce the number of restricted methods.

Then the VM adds any new entries to the method dispatch table, and invalidates any updated method implementations. Changed methods will be compiled the next time the program invokes them Finally, the VM initiates a full copying garbage collection to update the state of existing objects whose classes changed. When the collector encounters an object to update in the *from space*, it allocates a copy of this object and an object of the new class in the *to space*. At the end of the collection, JVOLVE runs object transformers on each of these pairs and then invokes the class transformers. At this point, the update is complete.

3.2 Update Model

We have designed a flexible, yet simple update model that supports updates that we believe are important in practice. JVOLVE classifies updates into the following three categories:

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

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

Indirect method updates: These are methods that are unchanged in the application, but must still be recompiled because they refer to offsets of fields and methods of updated classes.

Class updates may occur at any level of the class hierarchy. For example, an update that deletes a field from a parent class will propagate correctly to the class's descendants. JVOLVE does not support permutations of the class hierarchy, e.g., reversing a super-class relationship. While this update may be desirable in principle, in practice, it requires sophisticated transformers that enforce update ordering constraints. None of the program versions we observed made this type of change.

Method body updates are the simplest and most commonly supported change [16, 11, 10, 13, 26, 30, 15], because such changes can be applied at any time while preserving type safety (the next call to the method will be the new version). However, permitting only method body updates prevents many common changes [22]. Section 5 shows that over half the releases of Jetty, JavaEmailServer, and CrossFTP require supporting class signature changes.

Indirect method updates occur when unchanged methods refer to updated objects and methods. In this case, the generated code may now incorrectly dereference a field or invoke a method. These methods need to be recompiled.

JVOLVE's update model is quite flexible. Developers may change a method implementation to fix a bug. Developers may enhance functionality by adding and acting on a new parameter to a method, or by adding a new field and its access methods to a class (and its subclasses). JVOLVE's supported updates also match common refactorings, such as dividing a method into multiple methods, renaming a class or interface, changing types, and renaming fields [12].

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 two 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`. The class's `setForwardedAddresses` method is also altered to take an array of `EmailAddress`es instead of an array of `String`s, and code from `loadUser` accommodates this change as well.

3.3 Class and Object Transformers

For our example, the JVOLVE change detector identifies that the `User` and `ConfigurationManager` classes have changed. At this point, the programmer may elect to write object and class transformers or use the defaults. The user elects to write both a class and object transformer for the class `User`, as illustrated in Figure 3.

Object and class transformer methods are simply `static` methods that augment the new class. The class transformer method `jvolve_class` (body not shown) takes no arguments, while the object transformer method `jvolve_object` takes two reference arguments: `to`, the uninitialized new version of the object, and `from`, the old version of the object. For both methods, the old version of the changed class has its version number prepended to its name. In our example, the old version of `User` is redefined as class `v_1_3_1_User`, which is the type of the `from` argument to the `jvolve_object` method in the new `User` class. The

```
public class User {
  private String username, domain, password;
  private String[] forwardAddresses;
  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 String username, domain, password;
  private EmailAddress[] forwardAddresses;
  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

`v_1_3_1_User` class contains only field definitions from the original class, defined with access modifier `public` to allow them to be accessed from the `jvolve_object` method.

The code in transformer methods is essentially a kind of constructor: it should initialize all of the fields of the new class/object. Very often the best choice is to initialize a new field to its default value (e.g., `0` for integers or `null` for references) or to copy references to the old values. In the example, the first few lines simply copy `username`, `domain`, and other fields from their previous values. A more interesting case is the field type change to `forwardedAddresses`. The function allocates a new array of `EmailAddress`es initialized using the `String`s from the old array. Note that the default transformer function would instead copy the first three fields as shown, and initialize the `forwardedAddresses` field to `null` because it has changed type.

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 the object being updated and/or any objects reachable from it. JVOLVE presents a more limited interface (similar to past work [28, 20]). 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 `v_1_3_1_User` is defined in terms of the fields it contains; no methods are shown. As explained in Section 4.4, 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.

If a programmer does not write an object transformer, the VM initializes new or changed fields to their default values and copies the values from unchanged fields.

```
public class v_1_3_1_User {
  public String username, domain, password;
  public String[] forwardAddresses;
}
public class User {
 ...
 public static void jvolve_class() { ... }
 public static void
  jvolve_object(User to, v_1_3_1_User from) {
    to.username = from.username;
    to.domain = from.domain;
    to.password = from.password;
    // default transformer would have:
    //   to.forwardAddresses = null
    int l = from.forwardAddresses.length;
    to.forwardAddresses = new EmailAddress[l];
    for (int i = 0; i < l; i++) {
      to.forwardAddresses[i] =
       new EmailAddress(from.forwardAddresses[i]);
}}}
```

Figure 3. Example `User` object transformer

4. VM Support for DSU

This section describes how we support DSU in JVOLVE by extending common virtual machine services. JVOLVE is built on the Jikes RVM (SVN r15125) [1], a high-performance [31] Java-in-Java Research VM. Our current JVOLVE implementation uses Jikes RVM's dynamic classloader, JIT compiler, thread scheduler, copying garbage collector (GC), and on-stack replacement features. Both *class updates* and *method body updates* require VM classloading, JIT compilation, and thread scheduling support. *Class updates* may additionally require GC and OSR support.

After the user prepares and tests modifications and makes them available for the update, the update process in JVOLVE proceeds in four steps. First, a standalone tool prepares the update. The user then signals JVOLVE, which waits until it is safe to apply the update. At this point JVOLVE stops running threads, loads the updated classes (although it would be straightforward to perform this step asynchronously), and installs the modified methods and classes. Finally, if needed, it performs a modified garbage collection that implements class signature updates by transforming object instances from the old to the new class definitions.

4.1 Update Preparation Tool (UPT)

To determine the changed and transitively-affected classes for a given release, we wrote a simple Update Preparation Tool (UPT) that examines differences between the old and new classes provided by the user. UPT is built using jclasslib.[1] UPT first finds *class updates*—classes whose signatures have changed due to field or method additions or deletions, or due to method signature changes. UPT then finds methods whose bodies have changed and classifies them as *method body updates*. Finally, UPT determines *indirect*

[1] `http://www.ej-technologies.com/products/jclasslib`

method updates, whose source code is unchanged but refers to a field or method in an updated class. After an update is loaded, the VM also adds methods to this list similarly affected by its JIT inlining choices. JVOLVE does not load or compile indirectly-updated methods, but rather invalidates the old compiled versions and the JIT later recompiles them on demand. UPT also can provide templates for user object and class transformer functions, which the user may modify if necessary. The user then presents the list of updated classes and user transformers to JVOLVE.

4.2 DSU safe points

To preserve type safety, JVOLVE ensures that the update to the new version is atomic. No code from the new version must run before the update completes, and no code from the old version must run afterward. JVOLVE requires the running system to reach a *DSU safe point* before applying updates. DSU safe points occur at *VM safe points*, but also restrict the methods referenced by running threads' stacks. To safely perform VM services such as thread scheduling, garbage collection, and JIT compilation, Jikes RVM (like most production VMs) inserts yield points at all method entry and exit points, 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 an update is available, it sets the yield flag. Once application threads on all processors have reached VM safe points, a DSU thread checks all thread stacks. If none refers to a restricted method, as defined below, JVOLVE applies the update. Otherwise, the user may try again.

Similar to most other DSU systems [28, 20, 2, 10, 16, 11, 8, 30], JVOLVE's restricted methods include those belonging to classes that are being updated. To see why this restriction is important, consider the update from Figure 2 and assume the thread is stopped at the beginning of the `ConfigurationManager.loadUser` method. If the update takes effect at this point, the 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 `setForwardedAddresses` with an array of `String`s, resulting in a type error.

Preventing an update until updated methods are no longer on the stack ensures type safety because when the new version of the program resumes it will be internally type correct. If a programmer changes the type of a method `m`, for the program to have compiled properly, the programmer must also change any methods that call `m` to the new type. 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`.

However, restricting methods belonging to updated classes is not enough—we must also restrict those methods updated indirectly, because these methods depend on the particular field and method offsets of the classes to which they refer, and must be recompiled. Similarly, we must restrict those methods into which updated methods—whether directly or indirectly—have been inlined. For example, if an updated method `m` is inlined into another method `n`, then `n` must also be considered restricted. JVOLVE keeps track of the compiler's inlining decisions and, when an update is available, computes a transitive closure of all methods that have inlined methods appearing in the update method set. JVOLVE adds these additional methods to the set of restricted methods used to determine a safe point.

Finally, we allow users to manually add methods to the restricted set if they wish. Gupta [14] has shown that such restrictions are sometimes necessary to ensure correct behavior (though we have yet to find them necessary in our experience).

On-stack replacement. The number of restricted methods can be reduced by taking advantage of VM support for on-stack replacement (OSR). OSR is normally used to replace a baseline-compiled version of an active method with an optimized version. We have made preliminary extensions to JVOLVE so that it employs OSR to reduce the restricted methods set.

Jikes RVM's OSR functionality can recompile the topmost active method. OSR takes effect when the thread running the to-be-recompiled method reaches a yield point. Having compiled an optimized version, the VM needs to change the thread's current PC to switch to the equivalent location in the new implementation, and adjust the stack to have appropriate values. In this case, when the thread yields, Jikes RVM examines its active frame and extracts the values of local variables. It generates a special prologue to the optimized method that sets up a stack frame with these extracted values. The last instruction in the prologue jumps to the new PC location. The VM overwrites the return address of the yield point function to jump to the prologue.

We extend OSR to support updating the following types of (active) methods: indirectly updated methods that need to be recompiled with new offsets for fields and methods; methods with changed bodies that resume execution at the same PC location, for instance, simple changes like using a different string or operator; and finally, unmodified methods of classes that have changed signature. Jikes RVM only supports replacing the topmost method on stack. We have also extended Jikes RVM to support multiple stack activation records, instead of just the topmost stack frame. In DSU, a changed method's callers are on stack and often need to be replaced as well.

We currently support OSR on methods whose original bytecode has not changed, or has changed in a trivial manner. For methods whose bodies contain more substantial changes, we add them to the restricted set or rely on assistance from the programmer. In particular, the programmer may specify a mapping between equivalent DSU safe points in the old and new version, along with transformer functions to convert the stack as expected by the new version.

Our current implementation supports updating several simple programs, but bugs unrelated to OSR in Jikes RVM's baseline version prevented us from updating our larger benchmarks. We will demonstrate OSR on our applications in the final paper.

4.3 Loading Modified Classes

Once the program reaches a safe point, JVOLVE initiates the update. There are two main steps in this process. First, JVOLVE loads the changed classes by adding metadata for new classes and updating existing ones. Second, the garbage collector transforms existing object instances to refer to the new metadata and, in the case of class signature changes, to use the new object layout, initialized by the default or user-provided object transformer function.

In Jikes RVM, a class has several data structures. Each class has a corresponding `RVMClass` meta-object that describes the class. It points to other meta-objects that describe the class's methods and fields, which describe the field or method's type, and its offset in an object instance. The compiler uses offset information to generate field and method access code, while the garbage collector uses it to identify and trace object references. `RVMClass` also stores a *type information block* (TIB) for each class, which maps a method's offset to its actual implementation. Jikes RVM chooses to always compile a method directly to machine code on-demand, when the method is first called. Each object instance contains a pointer to its TIB, to support dynamic dispatch. When the program invokes a

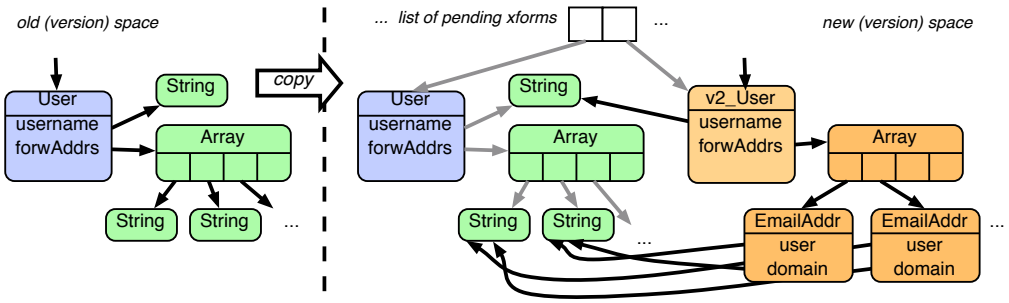

Figure 4. Running object transformers following GC

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 update, the class's number, type, and order of fields or methods may have changed, which in turn impacts an object's layout and its TIB. To effect these changes requires two steps, the first properly modifies the class datastructures, and the second modifies objects of changed classes. For the first step, JVOLVE renames the old metadata for the class (to eliminate name conflicts during object updates), and 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. Finally, the VM invalidates the TIB entries for each updated method; just as with a newly-loaded method, the JIT will compile the updated method when it is first invoked following the update.

4.4 Applying Transformers

Changed objects must be made to point to their new TIB, and for those whose class signature has changed, must be transformed to conform to it. We modify the Jikes RVM semi-space copying collector [5] to update changed objects as part of the collection, which is safe because DSU safe points are a subset of GC safe points. The VM ensures that the stack maps are correct at every thread yield point. Stack maps enumerate all registers, and stack variables that contain root references in to the heap and are required to identify reachable objects.

A semi-space copying collector normally works by traversing the pointer graph in the 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*). Once the collector copies an object, it overwrites its header with a forwarding pointer to the new copy in to-space. If the collector encounters the old object later during the traversal 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 a 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 copy of the old version. After the collection completes, JVOLVE goes through the update log and invokes the relevant object transformer (either the default one, or the user provided `jvolve_object`), passing the old and new object pairs as arguments. Once all pairs have been processed, the log is deleted, making the duplicate old versions unreachable. Finally, JVOLVE executes the class transformers.

Figure 4 illustrates a part of the heap at the end of the GC phase while applying the update from Figure 3 (forwarding pointers are not shown to avoid clutter). 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 have been 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 constructor 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 `jvolve_object` 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. Note that we take care that `jvolve_object` functions invoked recursively to transform old objects 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. If we discover that some object o must be transformed while running p's transformer method, we could scan the remainder of the update log to find o and then pass it and its uninitialized new version to the `jvolve_object` method to update it. To avoid multiple scans of the update log, we instead cache a pointer to the old version from its new version when performing the GC. We must take care when a transformation function dereferences an old field. It should be straightforward to determine this case automatically, through a read barrier or a simple analysis of the `jvolve_object` bytecode. In our current implementation, the programmer must use a special VM function to perform the dereference correctly.

Discussion. Our approach of requiring an extra copy of all updated objects 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 copying altogether by invoking object transformer functions during collection. This approach is more complicated because it requires recursively invoking the collector from the transformer if a dereferenced field has not yet been processed. We also would need to insert an extra GC-time read barrier which follows forwarding pointers before dereferencing an object, and which determines whether an object has been transformed yet.

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 [28, 20, 23, 8]. The main drawback here is that code must be inserted to check, at each dereference, whether the object is up-to-date, imposing extra overhead on normal 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. [8], which removes checking code once all objects have been updated. We leave exploration of these ideas to future work.

5. Experience

To evaluate JVOLVE, we used it to update three open-source servers written in Java: the Jetty webserver[2], JavaEmailServer,[3] an SMTP and POP e-mail server, and CrossFTP server.[4] 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.

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, and furthermore previous systems would support only 9 of 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.

5.1 Performance

The main performance impact of JVOLVE is the cost of applying an update; once updated, the application runs without further overhead. To confirm this, we measured the throughput of Jetty when started from scratch and following an update and found them to be 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 around 1ms; the classloading time is usually less than 20ms; the time to resume execution is usually less than 20ms. Therefore the update disruption time is primarily due to the GC and object transformers, which is proportional to the size of the heap and the fraction of objects being transformed. We wrote a simple microbenchmark to measure these overheads. We report our results, which show object transformation to be the dominant cost.

We conducted all our experiments on a dual P4@3GHz machine with 2 GB of RAM. The machine ran Ubuntu 6.06, Linux kernel version 2.6.19.1. We implemented JVOLVE on top of Jikes RVM (SVN r15125).

Jetty performance. To see the effect of updating on application performance, we measured Jetty under various configurations using

Config.	Req. rate (/s)	Resp. time (ms)
5.1.5 (JVOLVE)	361.3 +/- 33.2	19.2
5.1.6 (Jikes RVM)	352.8 +/- 28.5	17.4
5.1.6 (JVOLVE)	366.2 +/- 26.0	15.9
5.1.6 (upd. idle)	357.4 +/- 34.9	15.2
5.1.6 (upd. midway)	357.5 +/- 41.6	17.5

Table 1. Throughput measurements for Jetty webserver

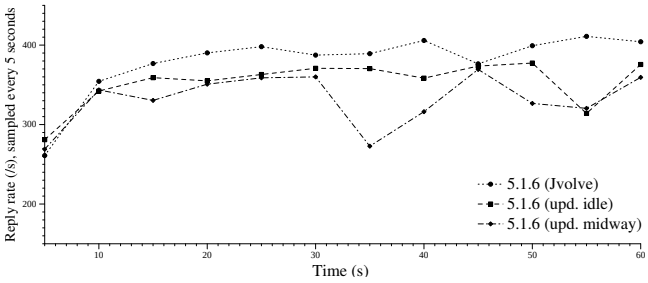

Figure 5. Webserver request rate over time

httperf,[5] a webserver benchmarking tool. We used httperf to issue roughly 400 new connection requests/second, which we observed to be Jetty's saturation rate. Each connection makes 5 serial requests to a 10Kbyte file. The tests were carried out for 60 seconds. The client and server were run on different processors in the same machine. Thus, these experiments do not take into account network traffic, but do exercise our multithreaded capabilities.

Table 1 shows our results. The second column reports the average rate at which requests were handled, measured every five seconds over the sixty second run, along with the standard deviation. The third column is the average response time per request. The first row illustrates the performance of Jetty version 5.1.5 using the JVOLVE VM, while the remaining measurements consider Jetty version 5.1.6 under various configurations. The second and third lines measure the performance of 5.1.6 started from scratch, under the Jikes RVM and JVOLVE VMs, while the fourth line measures the performance of 5.1.6 updated from 5.1.5 before the benchmark starts. The performance of these three configurations is essentially the same (all within the margin of error), illustrating that neither the JVOLVE VM nor the updated program is impacted relative to the stock Jikes RVM.

The last line of the table measures the performance of Jetty version 5.1.6 updated from version 5.1.5 midway through the benchmarking run. This causes the system to pause during the measurement, which correspondingly affects the response rate. We can see this pictorially in Figure 5, which plots throughput over time for the last three configurations in Table 1. Two details are worth mentioning. First, the throughput during the run is quite variable in general. Second, when the update occurs at time 30, the throughput dips quite noticeably shortly thereafter. We measured this pause at about 1.36 seconds total, where roughly 99% of the time is due to the garbage collector, and less than 0.1% is due to transformer execution. When updating Jetty when it is idle, the total pause is about 0.76 seconds.

Microbenchmarks. The two 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 costs of each,

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

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

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

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

# objects	Fraction of `Change` objects										
	0.00	0.10	0.20	0.30	0.40	0.50	0.60	0.70	0.80	0.90	1.00
Total pause (ms)											
1000	381.21	387.66	389.61	390.28	392.14	391.21	394.40	395.90	397.41	399.47	400.77
10000	382.62	433.23	436.40	433.09	474.41	461.82	479.59	496.99	510.60	528.99	544.46
50000	382.73	463.11	541.53	619.35	702.17	779.67	886.48	1559.58	1802.33	1990.18	2152.32
100000	383.64	541.03	698.73	852.36	1067.21	1276.26	3347.12	3968.23	4738.28	6712.72	7903.21
Running transformation functions (ms)											
1000	0.22	1.20	2.05	2.89	3.77	4.62	5.43	6.29	7.21	8.03	8.89
10000	0.22	9.55	17.36	25.80	36.32	44.05	51.40	61.74	68.36	78.98	85.39
50000	0.22	42.92	86.01	128.87	176.29	215.94	267.37	928.25	1132.23	1288.18	1423.34
100000	0.22	86.97	170.81	256.17	392.32	511.02	2539.37	3088.71	3789.88	5693.90	6809.91
Garbage collection time (ms)											
1000	376.83	382.29	383.45	383.19	384.27	382.47	384.73	385.40	386.06	387.27	387.78
10000	378.25	419.11	414.94	403.02	433.65	413.64	422.63	431.04	438.11	445.85	454.95
50000	378.36	416.04	451.41	486.34	521.74	559.62	614.63	627.00	663.25	697.86	723.43
100000	379.29	449.92	522.38	591.98	673.23	756.01	803.67	875.33	944.38	1014.69	1089.16

Table 2. Microbenchmark results: JVOLVE update pause time (in ms) for various heap sizes

we devised a simple microbenchmark that creates 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 start with a single integer field. The update adds another integer field to `Change`. The user-provided object transformation function copies the first field and initializes the new field to zero. The benchmark contains two arrays, one for `Change` objects and one for `NoChange` objects. We measure the cost of performing an update while varying the total number of objects and the fraction of objects of each type.

Table 2 shows the JVOLVE pause time for 1000 to 100000 objects (the rows) while varying the fraction of the objects that are of type `Change` (the columns). The first group of rows measures the total pause time, the second group measures the portion of this time due to running transformer functions, and the final group measures the portion of this time due to running the garbage collector. The first column of the table shows that there is large fixed cost of performing a whole heap collection even for a small number of objects. This time includes the time to stop the running threads and perform other operations. As we move right in the table, we can see that the cost of object transformation can outweigh the cost of the garbage collection by quite a bit. Also, with more objects to be transformed, the time to run object transformation functions increases non-linearly, because of caching effects.

The highly optimized base copying sequence does a `memcopy`, whereas our transformer functions use reflection and copy one field at a time. For each transformed object, JVOLVE looks up and invokes an object's `jvolve_object` function using reflection, and then copies each of the fields one by one. The cost of reflection could be reduced by caching the lookup, but a naïvely compiled field-by-field copy is much slower than the collector's highly-optimized copying loop. Note however that the number of transformed objects in our actual benchmarks was usually very low, less than 25 objects in the applications we considered, as illustrated by Jetty pause times reported above.

5.2 Jetty webserver

Jetty is a widely-used webserver written in Java. It supports static and dynamic content and can be embedded within other Java applications. Jikes RVM on top of which JVOLVE is built 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 one prior to version 6). Version 5.1.10 contains 317 classes and about 45000 lines of code. Table 3 shows a summary of the changes in each update. Each row tabulates the changes relative to the prior version.

Ver.	# classes added	# changed classes	# changed methods			# changed fields	
			add	del	chg	add	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 3. Summary of updates to Jetty

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 other 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.

With JVOLVE we were able to successfully write dynamic updates to all versions of Jetty we examined. However, we could not apply an update to version 5.1.3 because JVOLVE was never able to reach a safe point. 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; if a safe point could not be immediately reached, we deemed the attempt as failed. The results are presented in Table 4. Column 2 shows the number of times out of five such runs where the application reached a safe point. The methods whose presence on a thread stack precluded the application from reaching a safe point are mentioned below the table. For the update to 5.1.3, the offending method was always active because it contained an infinite loop. The other updates either always succeeded, or did after a small number of retries.

Column 3 contains the total number of methods in the program at runtime, where the number in parentheses is the number of those which the compiler inlined when using aggressive optimization. This provides an upper bound on the effect of inlining in reaching a safe point. The next group of columns contains the restricted method set. Each column in the group specifies the number of methods loaded at run time by the VM, followed by the total number of methods in that category in the program. The first column in this group is the number of methods in classes involved in

Upd. to ver.	Reached safe point?	Number of methods at runtime	# methods not allowed on stack, due to *class updates*	*method body updates*	*indirect method updates*	Total	Number of restricted methods	Number of restricted methods (with OSR)
5.1.1	always	1378 (376)	26/49	7/12	20/29	53/90 (17)	67	10
5.1.2	4/5†	1374 (375)	25/25	3/5	35/43	63/73 (35)	67	4
5.1.3	0/5*	1374 (375)	326/382	4/6	42/45	370/433 (97)	373	23
5.1.4	always	1384 (374)	82/82	5/6	15/16	101/104 (24)	101	10
5.1.5	always	1380 (372)	14/80	39/60	13/15	62/155 (17)	62	60
5.1.6	3/5†	1394 (378)	203/219	3/3	16/19	222/241 (40)	223	20
5.1.7	always	1394 (380)	186/187	1/2	53/69	239/258 (74)	243	12
5.1.8	always	1402 (379)	0/0	1/1	0/0	1/1 (1)	1	1
5.1.9	always	1402 (379)	0/0	0/1	0/0	0/1 (0)	0	0
5.1.10	always	1402 (379)	0/0	4/5	0/0	4/5 (2)	6	4

†Restricted method `HttpConnection.handleNext()` was active
*Restricted method `ThreadedServer.acceptSocket()` was always active

Table 4. Impact of safe point restrictions on updates to Jetty

a class update. Recall that when a class is updated, say by adding a field, all its methods are considered restricted. The second column in this group is the number of methods whose bodies are updated, the third is the number of methods indirectly updated, and the fourth sums these, with the number of methods that were inlined written in parentheses. The final two columns list the total number of methods in the restricted set; they differ from the first number in the fourth column by the number of (transitively) inlined callers of the restricted methods that were not already restricted. The final column uses JVOLVE's OSR analysis to determine the number of restricted methods although OSR itself was not applied.

The table shows that both indirect method calls and inlining significantly add to the size of the restricted set. Inlining though, is small by comparison, because all callers of an updated class's methods are *already* included in the indirect set. Therefore, inlining these methods adds no further restriction. In most cases OSR support would dramatically reduce the number of restricted methods and increase the likelihood of reaching a DSU safe point. Interestingly, having a greater number of restricted methods overall does not necessarily reduce the likelihood that an update will take effect; rather, it depends on the frequency with which methods in this set are on the stack.

5.3 JavaEmailServer

For JavaEmailServer we looked at 10 versions—1.2.1 through 1.4—spanning a duration of about two years. The last version we considered consists of 20 classes and about 5000 lines of code. Table 5 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 the nine updates we considered. We could successfully construct updates for all versions we examined, and we could successfully apply all of them but the update to version 1.3 (denoted with an asterisk in the table). This update reworked the configuration framework of the server, among other things removing a GUI tool for user administration and added several new classes to implement a file-based system. As a result, many of the classes were modified to point to a new configuration object. Among these classes were 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.

In addition, for the update from 1.3 to 1.3.1, the processing loop of one class was *indirectly* updated, and this initially precluded the update from taking place. To remedy this problem, we manually extracted the body of the loop and made it a separate function, essentially a manual application of the "loop extraction" transformation used in other work [23]. However, even in this case the update would only take effect if the server was idle; a similar situation occurred for the update to version 1.3.3. We could avoid this transformation and the need for idleness by using a limited form of on-stack replacement (OSR) as mentioned in Section 4.2. Note that loop extraction would not help for our other problematic updates because it was the `run` methods themselves whose code was changed, and not some method that they call.

5.4 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 6—spanning a duration of more than a year. Version 1.08 contains about 18000 lines of code spread across 161 classes. We could successfully handle all three updates to this application, though one update would apply only rarely under

Ver.	# classes		# changed					
	add	del	classes	methods			fields	
				add	del	chg	add	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 5. Summary of updates to JavaEmailServer

Ver.	# classes		# changed					
	add	del	classes	methods			fields	
				add	del	chg	add	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 6. Summary of updates to CrossFTP server

load. Note that since all updates either add or delete fields, simple method body updating support on its own would be insufficient.

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 normal execution—the only overhead occurs at the time of the update. JVOLVE's DSU support builds naturally on top of existing VM services, including dynamic class loading, thread synchronization, 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.

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 *OOPSLA*, Denver, CO, November 1999.

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

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

[4] 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 ATC*, 2005.

[5] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. pages 137–146, Scotland, UK, May 2004.

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

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

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

[9] 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, in association with OOPSLA 2001*, October 2001.

[10] 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.

[11] Edit and continue. `http://msdn2.microsoft.com/en-us/library/bcew296c.aspx`.

[12] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

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

[14] 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.

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

[16] Java platform debugger architecture. This supports class replacement. See `http://java.sun.com/j2se/1.4.2/docs/guide/jpda/`.

[17] Ksplice: Rebootless Linux kernel security updates, 2008. `http://web.mit.edu/ksplice/`.

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

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

[20] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz Barnes. Runtime Support for Type-Safe Dynamic Java Classes. In *Proc. ECOOP*, 2000.

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

[22] 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 (MSR)*, pages 1–5, May 2005.

[23] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for C. In *Proc. PLDI*, pages 72–83, 2006.

[24] Angela Nicoara, Gustavo Alonso, and Timothy Roscoe. Controlled, systematic, and efficient code replacement for running Java programs. In *Proceedings of the ACM EuroSys 2008 Conference (EuroSys 2008)*, April 2008.

[25] 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.

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

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

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

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

[30] 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 ATC*, June 2003.

[31] The Jikes RVM Core Team. VM performance comparisons, 2007. `http://jikesrvm.anu.edu.au/~dacapo/index.php?category=release`.

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