Dynamic Software Updates for Java: A VM-Centric Approach

Suriya Subramanian* and Michael Hicks† and Kathryn S McKinley*
*The University of Texas at Austin †University of Maryland

Abstract

Software evolves to fix bugs and add new 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 execute. The challenge is to develop DSU infrastructure that is *flexible*, *safe*, and *efficient*—DSU should enable updates that are likely to occur in practice, and updated programs should be as reliable and as efficient as those started from scratch.

This paper presents a new approach to supporting DSU for Java. The paper's key contribution is the design and implementation of a JVM enhanced with DSU support we call JVOLVE. JVOLVE is flexible, supporting additions and replacements of fields and methods anywhere within the class hierarchy, and in a manner that may alter class signatures. JVOLVE is safe, ensuring the execution conditions are always such that an applied update does not violate type-safety. Finally, JVOLVE is efficient, piggybacking its DSU services naturally on existing VM support for dynamic class loading, JIT compilation, and garbage collection, with the effect that all DSU-related overhead before or after an update is eliminated. By contrast, prior work (for C, C++, and Java) with comparable flexibility typically incurs larger time and space overheads, and these overheads are felt regardless of the number or timing of updates. Using JVOLVE, we successfully applied dynamic continuous updates corresponding to 17 of the 19 releases that occurred over more nearly two years' time, one update per release, for two open-source programs, the Jetty web server and the Java e-mail server. Our results indicate that the VM is well-suited to supporting DSU, and that DSU-enhanced VMs have the potential to significantly improve software availability.

1. Introduction

Software is imperfect. To fix bugs and adapt software to the changing needs of users, 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 [28, 25]; maintenance costs [32]; and at the least, inconvenience for users, which can translate into a serious security risk if patches are not applied promptly [3]. Dynamic software updating (DSU) addresses these problems by updating programs while they run. DSU is appealing because it is general-purpose: it requires no special software architecture and neither extra nor redundant hardware. The challenge is to make DSU *safe* enough that updating a program is as correct as deploying it from scratch, *flexible* enough that it can support software updates that are likely to occur in practice, and *efficient* enough to have little or no impact on application performance.

Researchers have made significant strides toward making DSU practical for systems written in C or C++, supporting server feature upgrades [23, 8], security patches [3], and operating systems upgrades [29, 5, 19, 7]. Because enterprise systems and embedded systems—including safety-critical applications—are increasingly written in languages such as Java and C#, these languages would benefit from DSU support. Unfortunately, work on DSU for these languages lags behind work for C and C++. For example, while the HotSpot JVM [17] and several .NET languages [12] support on-the-fly method body updates, this support is too inflexible for all but the simplest updates—less than half of the changes in two Java benchmark programs we examined could be supported. Other approaches proposed in the literature [27, 20, 26] are more flexible but impose substantial a priori space and time overheads and have not been proven on realistic applications.

This paper presents the design and implementation of a dynamic updating system called JVOLVE that we have built into Jikes RVM, a Java research virtual machine. The paper's key contribution is to show that modest extensions to existing VM services naturally support DSU that is flexible, safe, and imposes no a priori space or time overheads.

JVOLVE DSU support is quite flexible. Dynamic updates may add new classes or change existing ones. A change to a class may add new fields and methods, or replacing existing ones, and these replacements may have different 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 instance fields. A default transformer is automatically generated by the system at runtime—it initializes new and changed fields to a default value, and retains the values of unchanged fields—or the user may provide a custom one.

JVOLVE loads new and updated classes into a running program via the standard class loading facility. JVOLVE triggers compilation of the modified classes and invalidates existing compiled code for modified and *transitively-modified* methods, which are those methods that rely on the prior method's representation, such as methods in which the JIT previously inlined the modified methods. The JIT compiler then naturally recompiles each invalidated method when the program next tries to execute it, just as it would for a newly-loaded method. When a dynamic update to some class changes its instance fields, JVOLVE piggybacks on top of a whole-heap GC to find and transform existing instances of that class. When the collector encounters such an object, it creates a new object corresponding to the new class definition. At the conclusion of GC, JVOLVE applies the object and class transformers to initialize the new objects' fields and static class fields, respectively.

JVOLVE imposes no overhead during normal execution. The class loading, recompilation, and garbage collection modifications of JVOLVE's VM-based DSU support are modest and their overheads 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 typical of C and C++ that, for example, use a compiler or dynamic rewriter to insert levels of indirection [23, 26] or trampolines [7, 8, 3], respectively, which affect performance during normal execution. VMs also have the advantage of better memory management support. Rather than requiring each allocation to pad objects in case they become larger due to an update [23], managed languages have the flexibility to copy objects, and thus grow them only when necessary.

JVOLVE piggybacks on normal bytecode verification, part of classloading, to ensure that updated classes are type-safe. To avoid type errors that could result due to the timing of an update [23, 5], JVOLVE only permits updates to take place at a *safe point*. Such a point occurs when no running thread's activation stack refers to (1) an updated class, or (2) any class that either inlines an updated class or calls a method of an updated class whose signature has changed. Both parts of condition (2) are to handle changes in representation due to recompilation. This safety condition is similar to conditions proposed in prior work [30, 6], but is comparably much simpler, and accepts the large majority of updates we considered.

To assess JVOLVE, we used it to apply nearly two years' worth of the changes corresponding to releases of two open-source applications, JavaEmailServer (an SMTP and POP server) and the Jetty web server. JVOLVE successfully applies 17 of the 19 updates—the two updates we could not apply changed classes with infinitely-running methods, and thus no safe point could be reached. This restriction is not limited to JVOLVE, however; of all of the general-purpose DSU systems of which we are aware, only Ginseng [23] (which works for C programs) would have some hope of supporting these changes. Performance experiments with Jetty confirm updated applications enjoy the same performance as those started from scratch, except during the update itself. Microbenchmark results show that the pause time due to an update depends on the size of the heap and fraction of objects that must be transformed. We find there is a high per-object cost to using a transformer as compared to simply copying the bytes. However, most updates to Jetty and JavaEmailServer transformed only a small fraction of the heap objects.

In summary, the main contribution of this paper is JVOLVE, a VM-based approach for supporting dynamic software updating that is distinguished from prior work in its realism, technical novelty, and high performance. We believe this approach is a promising step toward supporting highly flexible, efficient, and safe updates in managed code virtual machines.

2. Dynamic Updates in JVOLVE

This section overviews how JVOLVE performs dynamic updating, including the sorts of updates that JVOLVE supports, and how the developer participates in the process.

2.1 System Overview

Figure 1 illustrates the dynamic update process. It shows the VM running the current version of the program and at top left, the source files. Meanwhile, developers are working on a new version, whose source files are shown at the bottom left. When the new version is ready–it compiles correctly and the developer has fully tested it using standard procedures—the developer passes the old and new versions of the source tree to JVOLVE's *change detector*. This tool identifies those classes that have been added or changed and copies them into a separate directory.

At this point, the developer 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

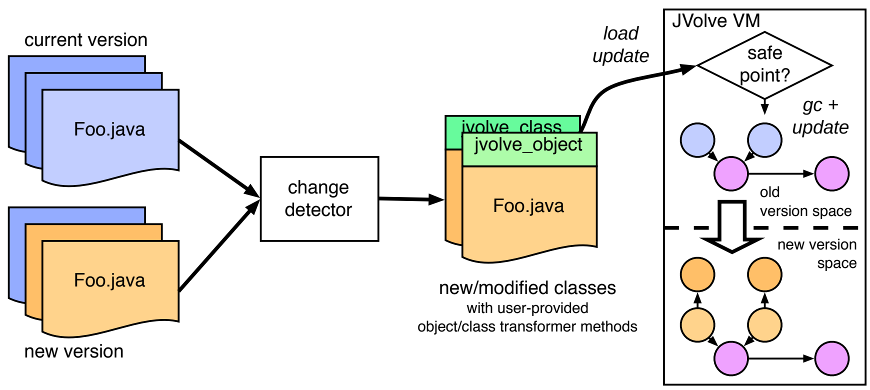

Figure 1. Dynamic Software Updating with JVOLVE

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 copies over 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 then 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. Next, the VM waits for all the threads to stop at a DSU *safe point* [15, 23] at which all threads are stopped and none of their activation stack's contain *updated methods*. Updated methods include all the directly updated methods and any methods that depend on them or updated classes, e.g., due to inlining. Section 3.2 describes some additional restrictions on safe points.

The VM then adds any new entries to the method dispatch table, and invalidates any updated method implementations. Changed methods are then compiled as a matter of course the next time the program invokes them (and the old implementations can never be accessed). 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 at the last invokes the class transformers. At this point, the update is complete and the threads resume execution.

2.2 Update Model

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

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

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 are the simplest and most commonly supported updates [17, 12, 11, 14, 26, 29, 16], because these changes do not require DSU safe points; they can be applied at any time and preserve type safety. Permitting only method implementation updates however prevents many common changes [22]. For example, Section 4 shows that over half of the updates to Jetty and JavaEmailServer add fields and/or change method signatures.

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 can enforce update ordering constraints. None of the program versions we observed made this type of update.

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, as desired). 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 [13].

Running Example. Consider the following update from JavaEmailServer, a simple SMTP and POP e-mail server written in Java that we use as a running example throughout the paper. 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 ac-

```java
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

```java
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

counts in the server. Moving from version 1.3.1 to 1.3.2, there are two basic 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, forwarded addresses are represented as an array of instances of a new class, `EmailAddress`, rather than `String`, which 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 `EmailAddresses` instead of an array of `Strings`.

2.3 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. For our example, 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 `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 user initializes the new field by referring to the old field. The function allocates a new array of `EmailAddresses` initialized using the `Strings` 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 that of past work [27, 20]. In particular, transformers may only use old objects to initialize new objects; the only safe access to a new object is through the `to` argument. Transformers may safely copy the contents of `from` fields. These fields may also be dereferenced if the update has not changed their class, or if it has, once the referenced objects are transformed to conform to the new class definition. At the moment this is achieved by invoking a VM function, but ultimately we plan to provide more automatic support. Finally, object transformers may not call methods on the old object. For example in Figure 3, class `v_1_3_1_User` is defined in terms of the fields it contains, while the methods have been removed. As explained in Section 3.4, these limitations stem from a goal to keep 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 merely 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

JVOLVE's Update Preparation Tool (UPT) (described in Section 3.1) generates templates for the `jvolve_object` and `jvolve_class` methods that contain the default transformers and that the user may then modify [15, 23].

3. VM Support for DSU

This section describes how we implement DSU by extending common virtual machine services. We discuss our particular implementation choices in the context of Jikes RVM [2, 1], a high-performance [31] Java-in-Java Research VM. Our current JVOLVE implementation uses Jikes RVM's dynamic classloader, JIT compiler, thread scheduler, and copying garbage collector. Specifically, both *class updates* and *method body updates* (see Section 2.2), require VM classloading, JIT compilation, and thread scheduling support. *Class updates* additionally require garbage collection support. Jikes RVM also provides on-stack-replacement support which we hope to leverage in future work.

The update process in JVOLVE proceeds in four steps. First, a standalone tool prepares the update. When the update is ready, the user signals JVOLVE, which waits until it is safe to apply the update. At this point JVOLVE stops running threads, loads and JIT-compiles 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.

3.1 Update Preparation

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 on top of jclasslib,[1] a bytecode viewer and library for examining (bytecode) class files. UPT first finds *class updates* — classes whose signature has changed due to field or method additions or deletions, or due to method signature changes. UPT finds methods whose bodies have changed and classifies them as *method body updates*. It simply compares the bytecodes to make these classifications. Finally, UPT determines *indirect method updates*, which are methods whose source code is unchanged but refer to a field or method in an updated class. Such methods must be recompiled to correct field or methods offsets that changed because of the update. (Once an update is loaded, the VM also adds methods to this list similarly affected by 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.

As mentioned previously, JVOLVE generates the default transformer internally. In addition it provides templates for object and class transformer functions, that the user may modify if necessary. The user presents the list of updated classes and user transformers to JVOLVE.

3.2 Stopping and Resuming Program Execution

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 piggyback on *VM safe points*, but also restrict the methods that may be on the stack. To safely perform VM services such as thread scheduling, garbage collection, and JIT compilation, Jikes RVM (and most production VMs) insert 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 the DSU event handler detects an update, it sets the yield flag, then checks the stack. If the stack contains a restricted method, it gives up, and the user can try again later.

Similar to most other DSU systems [27, 20, 3, 11, 17, 12, 8, 29], 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

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

`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 the new version of the program is itself internally type correct. If a programmer changes the type of a method `m`, for the program to have compiled properly, any methods that call `m` must do so at the right 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 JVOLVE could change the signature of method `m` and some old caller could call it—the update must also include all callers of `m`.

However, restricting methods belonging to updated classes is not enough—we must also restrict those methods updated indirectly, i.e., all methods identified by UPT. This is because these methods depend on the particular field and method offsets of the classes to which they refer, so if a class changes its signature then its callers must be recompiled to reflect that change. Finally, we must also 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.

Discussion. While simple and largely efficacious, there are several possible enhancements to our basic approach for establishing safe points.

First, we could use a static analysis to determine that some methods may be removed from the restricted method set. For example, Stoyle et al.'s static analysis [30] would determine the update to `User` could be permitted even while `loadUser` is running, but only after the call to `setForwardedAddresses`. Static analysis however cannot enable updates to methods that are essentially *always* on the stack, such as infinite loops that perform event processing. Updates to methods containing such loops will be indefinitely delayed. To update in this case, past work has proposed extracting out the contents of an infinite loop into separate methods as a part of compiling updatable software, i.e., before the first execution [23]. Therefore, if the contents of the loop change, the change is limited to the extracted method body, but the loop itself and the type signature of the extracted method will remain the same. This source modification enables updates to the loop body because there are windows in which it is not active, i.e., just before or just after a call to the extracted method. A similar technique could be implemented at update-time using *on-stack replacement*, which is provided in many VMs. The user would indicate the correspondence between an infinite loop in the old version of the method and the loop in the new version, and indicate how to map between the local variables used in the two cases.

Our current retry policy is quite simplistic. If there is but a small window in which the all forbidden methods are inactive, only a very lucky user will find that window. Alternatively, we could delay an update, and insert stack barriers [9] to try again when conditions are ripe.

A final possible enhancement is that the compiler could take care to keep the field layout the same whenever possible, reducing the number updated method and thus the number of methods that cannot be on the stack at update time. We leave exploration of these ideas to future work.

3.3 Loading and Compiling Modified Classes

Once the program reaches a safe point, the JVOLVE initiates the update. There are two main steps in this process. First, JVOLVE loads the changed classes and then arranges for them to be compiled properly by updating the metadata for the existing classes and adding metadata for new classes. 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. The remainder of this section covers the first step, and the next subsection covers the second.

In Jikes, a class has several data structures. Each class has a corresponding `VM_Class` 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 perform collection. In addition to this metadata, `VM_Class` points to the *type information block* (TIB) for the class, which maps a method's offset to its actual implementation. Jikes RVM chooses to always compile a method directly to machine code, where the compilation takes place 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 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, JVOLVE renames the old metadata for the class (to use it during object updates), and installs the new `VM_Class` and corresponding metadata for the newly-compiled class. Installing this information takes two steps. First, the VM updates several Jikes data structures to indicate that the newly-loaded class is now the up-to-date version. These include the JTOC (Java Table of Contents) for `static` methods and fields. In addition, the VM invalidates the TIB for all updated

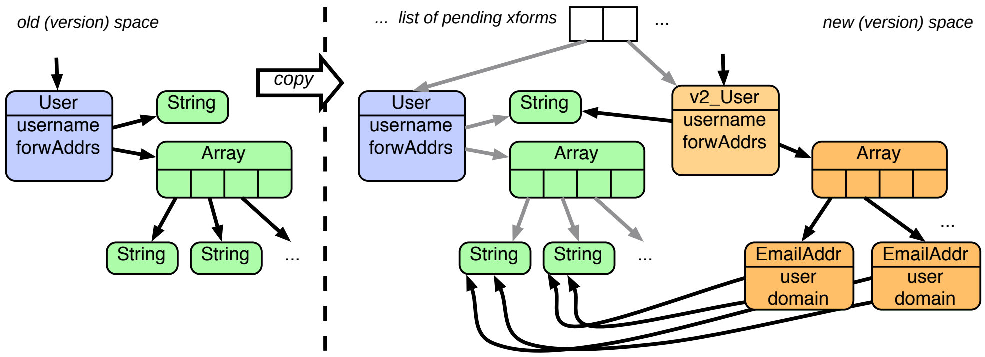

Figure 4. Running object transformers following GC

methods. Just as with a newly-loaded method, the JIT will compile the updated method when it is first invoked following the update. The second step happens when the garbage collector traces objects affected by the change: it updates them to point to their new TIB (as well as applying their object transformer, described next).

3.4 Applying Class and Object Transformers

Existing objects whose class signatures have changed must be transformed via their object transformer methods. We modify the Jikes semi-space copying collector 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 perform a collection.

A semi-space copying collector normally works by traversing the pointer graph in the heap (called *from-space*) starting at the roots and performs 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 be 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 pointers to the copy of the old object and the new object in an update log. 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, or `jvolve_object`, if present), 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. They will be reclaimed at the next collection. 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 by 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, the fields of old objects can be dereferenced safely so long as all objects they point to are up-to-date. If one of the fields points to an object whose class has been updated, it is possible that the pointed-to object has not yet been transformed (i.e., since it appears later in the update log). To transform it, we could find the old object by scanning the remainder of the update log and then pass both old and new objects to the `jvolve_object` method (or the default transformer). 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 (this pointer is not shown in the figure).

We must also 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 a cycle is found. In our current implementation, the programmer indicates whether to transform a child object before or after the parent at the start of the `jvolve_object` function. It should be straightforward to determine when this case automatically, through a read barrier during collection in the `jvolve_object` code, or even simple analysis of the `jvolve_object` bytecode.

Discussion. Our approach of requiring an extra copy of all updated objects adds temporary memory pressure, since copies will persist until the next GC. We could instead copy the old versions to the end of from-space, rather than to to-space. Assuming they all fit, they will be immediately reclaimed when the GC is complete. We could attempt to avoid extra copying altogether by invoking object transformer functions during collection. This approach is more complicated because it may require 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 that follows forwarding pointers before dereferencing an object, and that determines whether an object has been transformed yet. We leave exploration of these ideas to future work.

We use a stop-the-world garbage collection-based approach that requires the application to pause for the duration of a full 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 [27, 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], in which inserted checking code is removed once all objects have been updated.

4. Experience

To evaluate JVOLVE, we used it to update two open-source servers written in Java: the Jetty webserver[2] and JavaEmailServer,[3] an SMTP and POP e-mail server. 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 a year and a half of each program's lifetime. Of 19 updates we considered, JVOLVE could support 17 of them—the two updates we could not apply changed classes with infinitely-running methods, and thus no safe point could be reached.

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

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

Ver.	# classes added	# changed classes	methods add	methods del	methods chg	fields add	fields del
5.1.1	0	26	4	1	38/0	0	0
5.1.2	1	17	0	0	12/1	0	0
5.1.3	3	22	19	2	59/0	10	1
5.1.4	0	14	0	4	9/6	0	2
5.1.5	0	61	21	4	112/8	5	0
5.1.6	0	7	0	0	20/0	5	6
5.1.7	0	10	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 1. Summary of Release Updates to Jetty

To our knowledge, no existing general-purpose DSU system for Java could support these updates either, and indeed typical support for dynamic updating (changing method bodies only) would fail to support 10 of 19 of them. JVOLVE 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. The remainder of this section describes this experience.

4.1 Jetty webserver

Jetty is a widely-used webserver written in Java, in development since 1995. It supports static and dynamic content and can be embedded within other Java applications. The Jikes RVM 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 1 shows a summary of the changes in each update. Each row in the column tabulates the changes relative to the prior version.

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. Table 2 breaks down the restricted method set for each update into its constituent parts. Column 3 contains the total number of methods in the program at runtime, where the number in parentheses is the number these the compiler inlined when using aggressive optimization (which 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 during runtime 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 a class update, the second is the number of methods whose bodies were updated, the third is the number of methods indirectly updated, and the fourth sums these, with the number of the total that were inlined written in parentheses. The final

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 3. Summary of Release Updates to JavaEmailServer

column is the total number of methods in the restricted set; it differs from the fourth column by the number of (transitively) inlined callers of the restricted methods.

The table shows that both indirect method calls and inlining have a non-negligible impact on increasing the size of the restricted set, though inlining is small by comparison because all callers of an updated class's methods are *already* included in the indirect set, so inlining these methods adds no further restriction. In the case of Jetty, none of this had any impact on the update failing: version 5.1.3 changed the bodies of the infinite loops of active threads' `run` methods, and thus reaching a safe point was not possible.

4.2 JavaEmailServer

For JavaEmailServer we looked at 10 versions—1.2.1 through 1.4—spanning a duration of about two years. The final version of the code consists of 20 classes and about 5000 lines of code. 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. 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 threads with infinite processing loops (e.g., to accept POP and SMTP requests). Because these classes 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]. We could avoid this transformation by using a limited form of on-stack replacement (OSR), as mentioned earlier. Note that a similar transformation would not work for our other problematic updates because it was the `run` methods themselves whose code was changed, and not some method that it calls.

Config.	Req. rate (/s)	Resp. time (ms)
5.1.5 (JVOLVE)	361.3 +/- 33.2	19.2
5.1.6 (Jikes)	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 4. Throughput measurements for Jetty webserver

5. 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. We report on this experiment in Section 5.1.

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. The cost of loading new classes is negligible (and hard to measure). Therefore the update disruption time is due to the GC and object transformers, and the cost of these is proportional to the size of the heap and the fraction of objects being transformed. We wrote a simple microbenchmark to measure this. Section 5.2 reports 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 2.9.1. The VM was configured with one virtual processor and utilized only one of the machine's CPUs.

5.1 Jetty performance

To see the effect of updating on application performance, we measured Jetty under various configurations using httperf,[4] a webserver benchmarking tool. We used httperf to issue roughly 100 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.

Table 4 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

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

Jetty	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
5.1.1	Yes	1374 (375)	25/25	3/5	35/43	63/73 (35)	67
5.1.2	No	1374 (375)	326/382	4/6	42/45	370/433 (97)	373
5.1.3	Yes	1384 (374)	82/82	5/6	15/16	101/104 (24)	101
5.1.4	Yes	1380 (372)	14/80	39/60	13/15	62/155 (17)	62
5.1.5	Yes	1394 (378)	203/219	3/3	16/19	222/241 (40)	223
5.1.6	Yes	1394 (380)	186/187	1/2	53/69	239/258 (74)	243
5.1.7	Yes	1402 (379)	0/0	1/1	0/0	1/1 (1)	1
5.1.8	Yes	1402 (379)	0/0	0/1	0/0	0/1 (0)	0
5.1.9	Yes	1402 (379)	0/0	4/5	0/0	4/5 (2)	6

Table 2. Impact of safe point restrictions on updates to Jetty

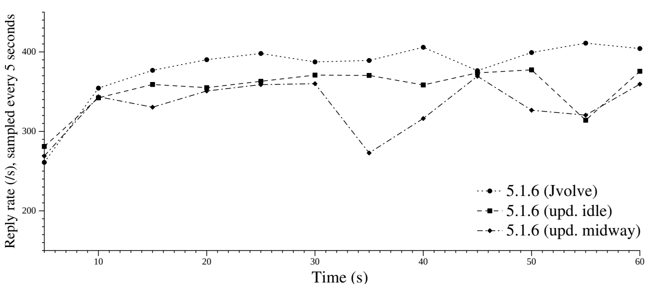

Figure 5. Webserver request rate over time

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, thus causing the system to pause during the measurement, which correspondingly affects the processing rate. We can see this pictorially in Figure 5, which plots throughput (the Y-axis) over time for the last three configurations in Table 4. 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, with 99.6% of the time due to GC and 0.01% due to transformer execution.

5.2 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, 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 5 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 setup. 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 original 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	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 5. JVOLVE pause time (in ms) for various heap sizes

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.

6. Related Work

We discuss of related work on supporting DSU in Java and other managed languages, and on supporting DSU in C and C++. Broadly speaking, JVOLVE is one of the most flexible systems proposed to date, and when compared to systems with similar flexibility, demonstrates superior performance and a more realistic and thorough evaluation.

6.1 Edit and Continue Development

Debuggers have long provided *edit and continue* (EnC) functionality that permits limited modifications to program state to avoid stopping and restarting during debugging. For example, Sun's HotSwap VM [17, 10], .NET Visual Studio for C# and C++ [12], and library-based support [11] for .NET applications all provide EnC. These systems are all less flexible than JVOLVE, typically supporting only code changes within method bodies. This limitation reduces safety concerns, and programmers need not write class or object transformers, but as discussed in Section 4, more than half of the updates we saw in practice would be disallowed.

6.2 DSU in Managed Languages

Approaches with special VM support. JDrums [27] and the Dynamic Virtual Machine (DVM) [20] both implement DSU support for Java within the VM, providing a programming interface similar to JVOLVE. However, their implementations impose overheads during normal execution, whereas JVOLVE has zero overhead and a richer evaluation.

Both VMs update *lazily*. For example, JDrums traps object pointer dereferences to check whether a new version of the object's class is available. If so, the VM runs the object transformer function(s) to upgrade the object. By contrast JVOLVE performs updates eagerly, as part of a full GC. (The DVM performs updates lazily as JDrums, but does some eager conversion incrementally.) 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.

Both JDrums and the DVM are in the Sun JDK 1.2 VM, which uses an extra level of indirection (the *handle space*) to support heap compaction. Indirection conveniently supports object updates, but adds extra overhead. The DVM only works with the interpreter. Relative to the stock bytecode interpreter, which is already slow, the extra traps result in roughly 10% overhead. By contrast, JVOLVE imposes no overhead once an update is complete. Neither JDrums nor the DVM has been evaluated on updates derived from realistic applications—only a handful of toy updates have been considered.

More recently, Nicoara et al. developed PROSE, a system for 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 in support of run-time software evolution. As such, PROSE only supports updates to method bodies, with no support for signature or state changes. This flexibility is similar to the EnC implementations discussed above; indeed, PROSE builds on the HotSwap method replacement support in its Sun JDK implementation [17].

Gilmore et al. [14] propose DSU support for modules in ML programs. They use a programming interface that is similar to ours, but more restrictive. They also propose using copying GC to perform the update, as we do. They formalized an abstract machine for implementing upgrades using a copying garbage collector but did not implement it.

Boyapati et al. [6] support lazily upgrading objects in a *persistent object store* (POS). Though in a different domain, their programming interface is quite similar to JVOLVE and the other Java-based systems: programmers provide object transformer functions for each class whose signature has changed. Their object transformers are somewhat different than ours. Their system allows the object transformer of some class A to 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 may dereference fields via old objects, but if these fields point to objects whose classes have been updated, they will see the *new* versions (a semantics which is more typical). We plan to further explore the costs/benefits of transformer function semantics in future work.

Approaches using a standard VM. To avoid changing the VM, researchers have developed special-purpose classloaders, compiler support, or both for DSU. The main drawbacks of these approaches are typically less flexibility and greater overhead. Eisenbach and Barr [4] and Milazzo et al. [21] use custom classloaders to allow binary-compatible changes and component-level changes, respectively. The former targets libraries and the latter is part of the design of a special-purpose software architecture.

Orso et al. [26] support DSU via source-to-source translation by introducing what amounts to a proxy class that indirects accesses to objects that could change. This approach requires updated classes to export the same public interface—no new non-private methods or fields can be added to an updated class. 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 extra runtime support imposes both time and space overheads. By contrast, modifying the VM is much simpler, given its existing services. Our VM approach has no problems with native methods (since these are updated as well) or reflection, and it can be much more expressive, e.g., supporting signature changes.

6.3 Dynamic Software Updating for C and C++

Recently several substantial systems for dynamically updating C and C++ programs have emerged that target server applications [16, 3, 23, 8] and operating systems components [29, 5, 7, 18, 19]. These systems are more mature than most of the systems described above, in some cases with substantial updating experience. The flexibility afforded by JVOLVE is comparable or superior to most of these systems. Only Ginseng [23] imposes fewer restrictions on update timing than JVOLVE, but it (like most of the above systems) cannot update multi-threaded programs and need consider language features for object orientation.

The lack of a VM is a significant disadvantage in the implementation of DSU for C and C++. 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 [16, 23, 29, 5] or dynamically-inserted trampolines to redirect function calls [3, 7, 8]. Both cases impose persistent overhead on normal execution in and of themselves, and can inhibit optimization. Likewise, because these systems lack a garbage collector, they either update object instances lazily [23, 8] or perform extra allocation and allocator bookkeeping to be able to locate the objects at update-time [5]. Both approaches impose time and space overheads on normal execution, whereas JVOLVE's VM-based approach has no a priori overheads. Finally, the fact that C and C++ are not type-safe greatly complicates efforts to ensure that updates are safe.

7. Conclusions

This paper has presented 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. To show that it can support changes that occur in practice, we successfully applied updates that correspond to most of a year-and-a-half's worth of releases for two programs, the Jetty webserver and the JavaEmailServer mail 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, JIT compilation, thread synchronization, and garbage collection. 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.

References

[1] B. Alpern, D. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. Flynn Hummel, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM System Journal*, 39(1), February 2000.

[2] 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.

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

[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 ATC*, 2005.

[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] P. Cheng, R. Harper, and P. Lee. Generational stack collection and profile-driven pretenuring. In *Proc. PLDI*, pages 162–173, Montreal, Canada, June 1998.

[10] 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.

[11] 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.

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

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

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

[15] Michael Hicks and Scott M. Nettles. Dynamic software updating. *Trans. Prog. Lang. Syst.*, 27(6):1049–1096, November 2005.

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

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

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

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

[29] 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.

[30] 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.

[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.