Dynamic Software Updates for Java: A VM-Centric Approach

Anonymous Submission to PLDI 2008, Please Do Not Distribute

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) instead updates 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 explores the advantages and challenges of DSU for managed languages like C# and Java. The paper's key contribution is the design and implementation of an enhanced JVM, which we call JVOLVE, that provides safe, flexible, and efficient DSU. JVOLVE's DSU support piggybacks naturally on existing VM services, including dynamic class loading, JIT compilation, and garbage collection to efficiently implement method and class changes. These changes include additions, modifications, and replacements of fields and methods, including changes that alter class signatures. Most prior work on DSU has focused on C or C++ and incurs time and space overheads regardless of the number or timing of updates. In contrast, by piggybacking on existing VM services, JVOLVE has no a priori overheads before or after an update. Using JVOLVE, we successfully applied dynamic updates corresponding to 16 of the 19 releases that occurred over more than a year's 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 ideally 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 [27, 24]; maintenance costs [31]; 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 compared to other approaches for on-line updates, such as those based on load balancing [32], because it is general-purpose and requires no redundant hardware. The challenge is to make DSU *safe* enough that updating a program is no more dangerous than 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 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 [28, 5, 19, 7]. Because enterprise systems and embedded systems—including safety-critical applications—are increasingly written in managed languages such as Java and C#, these languages would benefit from DSU support. Unfortunately, work on DSU for managed languages lags behind work for C and C++. For example, while the HotSpot JVM [17] and several .NET languages [11] support on-the-fly method body updates, this support is too inflexible for all but the simplest changes. Comparable systems in the research literature [26, 20, 25] are more flexible but impose substantial overheads and, to our knowledge, fail to assess their flexibility by attempting to update 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 key contribution is to show that modest extensions to existing VM services naturally support DSU in a flexible, efficient manner.

In JVOLVE, dynamic updates may add new classes or change existing classes in a running program. Supported changes include additions, modifications, and replacements of fields and methods, and method and field replacements may have different type signatures. JVOLVE makes use of *class* and *object transformers* to initialize new or modified static and instance fields, respectively. These transformers, which resemble Neamtiu et al.'s *type transformers* [23], can be provided by the programmer or a default transformer can be used. The type correctness of all updates is checked by the normal compilation and loading process.

JVOLVE loads new and updated classes into a running program via the standard class loading facility. JVOLVE compiles the classes with its JIT compiler and adds or updates pointers to the compiled methods in the virtual dispatch table. While our current implementation does not support inlining, the design is straightforward and only requires the JIT to provide an inlining log to JVOLVE. JVOLVE piggybacks on top of a whole heap garbage collection (GC) to find existing objects whose class has changed. When the collector encounters such an object, it creates a new object corresponding to the new class definition and the object transformer initializes it using the old instance data as needed. At the conclusion of GC, JVOLVE applies the class transformers to update static class fields.

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

approaches that, for example, use a compiler or dynamic rewriter to insert levels of indirection [23, 25] or trampolines [7, 8, 3], respectively, which affect performance during normal execution. VMs also have the advantage of better memory management support. Rather 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 on demand.

As recognized in previous work [23, 5], updates that change the types of objects are challenging because a poorly-timed update may cause a type error. JVOLVE permits type-altering updates by controlling the times at which updates are applied, in two ways. First, JVOLVE delays an update until no thread refers to an updated class on its activation stack. This is sufficient to avoid type errors. Second, when traversing the object heap to initialize new objects from old ones, we ensure that the transformer function only accesses objects of the *old* version and/or creates new objects. We ensure that this condition is met by a simple analysis of the transformation functions. While others have proposed more flexible and sophisticated techniques for type-altering updates [29, 6], our approach is equally safe. Since it is also simple, it requires only a straight forward analysis of a proposed update and needs no programmer annotations. We have found that our approach is sufficient to support the updates we have experimented with so far.

To assess JVOLVE, we used it to apply over a year's worth of the changes corresponding to releases of two open-source applications, JavaEmailServer (an SMTP and POP server) and the Jetty web server. JVOLVE could successfully apply 16 of the 19 updates we tried, and we believe we could support the remaining 3 with some small changes to some of our tools, and a bit more work. Performance experiments with Jetty confirmed 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 on the 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, few of the updates to Jetty and JavaEmailServer transformed more than a very small fraction of the total heap's 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 first step toward supporting highly flexible, efficient, and safe updates in managed code virtual machines.

2. Related Work

We discuss of related work on supporting DSU in Java and other managed languages, and on supporting DSU in C and C++. DSU for managed languages falls into two categories: special-purpose VMs and libraries and/or classloader-inserted or compiler-inserted code modifications. Support for DSU in C and C++ often combines compiler and runtime system support. Existing approaches widely vary in their update flexibility, safety guarantees, and run-time overhead. Broadly speaking, JVOLVE provides flexibility and safety guarantees comparable to the most flexible approaches proposed to date, but has superior performance and a more realistic and thorough evaluation.

2.1 Edit and Continue Development

Debuggers have long supported *edit and continue* (EnC) to make some implementation modifications visible to active debugging sessions. This avoids stopping and restarting during debugging. For example, Sun's HotSwap VM [17, 9], .NET Visual Studio for C# and C++ [11], and library-based support [10] for .NET applications all provide EnC. However, these systems are more inflexible than JVOLVE, typically supporting only code changes within method bodies. This limitation however reduces safety concerns, and programmers need not write class or object transformers.

2.2 Dynamic Software Updating in Managed Languages

Approaches with special VM support. JDrums [26] and the Dynamic Virtual Machine (DVM) [20] both provide VM support for DSU with a programming interface similar to JVOLVE, but their implementation imposes overheads during normal execution, whereas JVOLVE has zero overhead and a richer evaluation.

Both VMs update lazily, whereas JVOLVE performs updates eagerly, as part of a full GC. Lazy updating has the advantage that the pause due to an update can be amortized over subsequent execution. JDrums works by trapping 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. The DVM is similar, but does some eager conversion incrementally. 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 upgrading too, 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 VM has been evaluated on updates derived from realistic applications. The DVM paper reports no experience at all and papers on JDrums present only a toy phone book example.

Gilmore et al. [13] 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 ultimately 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 strictly more powerful 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 prohibit transformers from dereferencing fields whose class has been modified. They can only copy references from the old to the new version, which is sufficient for the updates we have seen. We are investigating similar support in JVOLVE.

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. [25] supports DSU via source-to-source translation in which they conceptually introduce 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

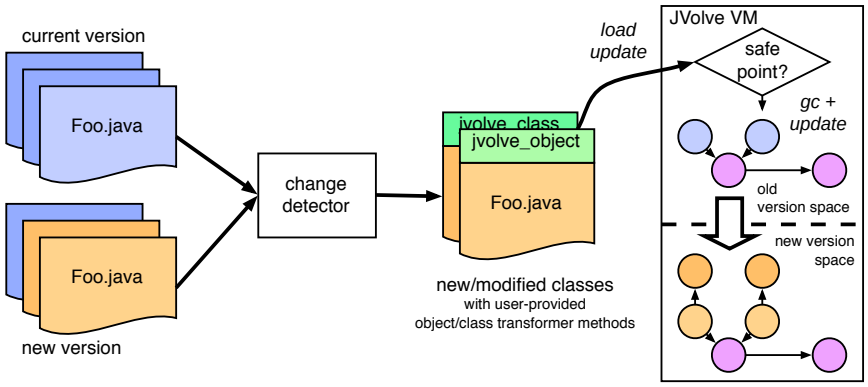


Figure 1. Dynamic Software Updating with JVOLVE

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.

2.3 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 [28, 5, 7, 18, 19]. These systems are far more mature than the systems described above, in many cases with substantial updating experience. One goal of our research is to address the challenges of managed languages, to reach the same level of maturity.

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, 28, 5] or dynamically-inserted trampolines to redirect function calls [3, 7, 8]. Both cases impose persistent overhead on normal execution. 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 overheads on normal execution, whereas a VM-based approach has no a priori overheads.

3. Dynamic Updates in JVOLVE

This section overviews the process of performing dynamic updates in JVOLVE, including the sorts of changes JVOLVE can support, and the way in which the developer participates in the process.

3.1 System Overview

Figure 1 illustrates the process of performing dynamic updates with JVOLVE. We begin with the VM running the current version of the program, whose source files are shown at the top left of the figure. At the same time, developers are working on the new version[1], whose source files are shown at the bottom left. When the new version is ready–it compiles correctly and has been fully tested using standard procedures—we pass 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 programmer may need to write some number of *object* and/or *class transformers*, although JVOLVE provides defaults. The 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 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 software 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 loads the transformers and the new user class files and starts the updating process. The first step is for the VM's JIT compiler to compile the loaded classes. Next, the VM waits for a *safe point* [15, 23] at which to apply the update. Among other criteria, JVOLVE requires that no updated method currently be on the activation stack of any running thread (see Section 4.2). Once a safe point is reached, the VM stops all threads and begins transforming the program. The VM replaces overwritten method implementations in modified classes by overwriting their entries in the virtual dispatch table. It also adds any new methods to the table. The VM thus ensures that all future calls will be to the new versions of methods. Next, the VM initiates a full copying garbage collection to update the state of existing objects whose classes have been changed. When the collector encounters an object in the old space, it allocates sufficient space for the new object and then runs the object transformer to initialize that object, passing the old copy of the object as an argument. Class transformers are essentially a special case of object transformers and are applied to class objects once the collection is finished. At this point, the update is complete and the threads resume execution.

3.2 Simple Updates

To ensure that we arrive at a practical solution, we have designed JVOLVE to be as simple as possible while supporting updates that we believe are important in practice. JVOLVE supports the following kinds of dynamic updates:

[1] Recent work in refactoring can help users prepare their updates [14]

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

Method updates: These updates add or delete methods to classes or change the internal implementation of a method.

Method signature updates: These updates change a method signature, e.g., adding, modifying, or deleting parameters and return values.

Class signature updates: These updates add, modify, or remove fields from a class.

These changes may occur at any level of the class hierarchy. For example, an update that deletes a field from a parent class will affect the class's descendants.

This 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. These updates support common refactorings, such as dividing a method into multiple methods, renaming a class or interface, changing types, renaming fields [12].

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 accounts 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`. This change modifies the class signature of `User`—the type of the `forwardedAddresses` field is different. There is a corresponding method signature update as well; `setForwardedAddresses` now takes an array of `EmailAddresses` as an argument instead of an array of `Strings`.

Update Support and Limitations. JVOLVE supports this interesting JavaEmailServer update, but it does not support arbitrary changes. In particular, JVOLVE does not allow alterations of the class hierarchy, e.g., reversing a super-class relationship. While this update may be desirable in principle, in practice there are two problems. First, it may be quite challenging to actually implement this change—the process of transforming old objects from the old to the new class hierarchy is complex. Second, complex changes make it harder to reason that an update's semantics will be correct. To see why, consider that a common limitation of existing code updating systems is that they support only method body updates, not method signature updates [17, 11, 10, 13, 25, 28, 16]. The reason is that such changes can take effect at any time and preserve type safety, whereas changes to method signatures must take timing into account. However, permitting only method implementation updates prevents many common changes [22] including the example in Figure 2. Therefore, any practical system must trade off flexibility with availability—the more flexible the update, the greater burden on the system to ensure that updates are well-timed.

Sections 4.2 and 4.4 provide additional details on the safety, flexibility, and implementation concerns of this simple update model and possible enhancements.

```
public class v_1_3_1_User {
  private String username, domain, password;
  private 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

3.3 Class and Object Transformers

For our example, the JVOLVE change detector will identify that both the `User` and `ConfigurationManager` classes have changed. At this point, the programmer may elect to write object and/or class transformers for the changed classes or use the defaults. For our example, the user elects to write both a class and object transformer for the class `User`. The code is 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 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, we can see that in the first few lines the username, domain, and other fields are simply copied from their previous values. More interesting is the case when a field changes its type, as is the case for the forwardedAddresses field. In this case, the user must initialize the new field by referring to the old field. The transformer function allocates a new array of EmailAddresses initialized using the Strings from the old array. Note that if the default transformer function was used instead, the first three fields would be copied as shown, and the forwardedAddresses field would be initialized to *null* because it has changed type.

Supported in its full generality, a class transformer method may reference any object reachable from the global (static) namespace of both the old and new classes, and the object transformer code may additionally 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 [26, 20]. In particular, we do not allow transformers to refer to new objects via class variables; the only new object the transformer can access safely is the one referenced by its to argument. We also do not permit transformers to dereference pointers to an (old) object whose signature has changed. Transformers can copy the pointers themselves, but may not dereference them. Finally, object transformers may not call methods on the object being changed. This is why, 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 4.4, these limitations stem from a goal to keep our garbage collector-based traversal safe and simple. While limited, this interface is sufficient to handle all of the updates we have tested that were not otherwise supported.

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. It would be straightforward to generate templates for the jvolve_object and jvolve_class methods that contain the default transformers and that the user may then modify [15, 23], but we have not yet done so.

4. VM-Supported Dynamic Software Updates

This section describes how we implement DSU by extending existing virtual machine (VM) services that are common in most Java VMs. In particular, our approach can take advantage of the following common VM services: dynamic classloading, interpretation, just-in-time (JIT) compilation, thread scheduling, on-stack-replacement, and garbage collection. We discuss our particular implementation choices in the context of Jikes RVM [2, 1], a high-performance [30] Java-in-Java Research VM. Our current JVOLVE implementation uses Jikes RVM's dynamic classloader, JIT compiler, thread scheduler, and copying garbage collector. Specifically, all three update types, *method, method signature,* and *class signature* (see Section 3.2), require VM classloading, JIT compilation, and thread scheduling support, while class signature updates also require garbage collection support. Jikes RVM does not have an interpreter and although it provides on-stack-replacement, JVOLVE does not currently use it.

The update process in JVOLVE proceeds in four steps, described in detail next. 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.

4.1 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,[2] a bytecode viewer and library for examining (bytecode) class files. UPT first finds classes whose signature has changed, such as by adding and removing fields or methods, or by changing the signature of fields and methods. All these classes require class and method signature updates. UPT finds methods whose bodies have changed and classifies them as method updates. It simply compares the bytecodes to make these classifications.

Finally, UPT determines the transitive relationships between these changes and other methods. Even though the implementation of a particular Java method is unchanged, it may refer to changed classes. Therefore, we need to recompile them because they may contain code that uses field offsets or has other dependencies. For example, if a field offset changes, the code must be recompiled to reflect the new offset. Therefore, if a method refers to a changed class UPT includes it in the method updates. UPT also includes method updates for methods that refer to constant pool entries of changed classes. As mentioned previously, users may write object transformer functions and these are included in the update as well. In the future, we will augment UPT to produce a default transformer that the user can modify. Right now, JVOLVE generates the default transformer internally. The user presents the list of updated classes and user transformers to JVOLVE.

4.2 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, and no code from the old version must run after the update. JVOLVE requires that the running system reaches a *DSU safe point* before applying updates. DSU safe points piggyback on *VM safe points*, but also restrict the methods on the stack. To safely perform VM services such as thread scheduling, garbage collection, and JIT compilation, the VM inserts yield points at every method entry, exit, and loop back edge. 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 uses this same functionality. When the DSU event handler detects an update, it sets the yield flag, then checks the stack. If the stack contains forbidden methods, it gives up, and the user can try again later. Alternatively, the system could set a timer or a stack barrier to try at a more opportune time.

To see why DSU safe points restrict the methods on the stack, consider what might happen if we were to apply the update from Figure 2 when a thread is running the method ConfigurationManager.loadUser. If the update is permitted to take effect immediately, the implementation of User.setForwardedAddresses will be modified so that it takes an object of type EmailAddress[]

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

as its argument. However, because the old version of `loadUser` is currently running, it will still call `setForwardedAddresses` with an array of `String`s, resulting in a type error.

To prevent these errors, prior work proposes using a static analysis to identify, in advance, the dependences a method has on its context at various points within its body [29, 23]. For example, such an analysis would determine that the prelude of `loadUser` depends on the type of `setForwardedAddresses`—it expects it to take an array of `String`s. After the call (even while `loadUser` is still running), the dependence goes away. When an update becomes available, it may take effect so long as it does not conflict with dependences imposed by the current state of the program. If it does, the update must be delayed.

Similar to most other DSU systems [26, 20, 3, 10, 17, 11, 8, 28], JVOLVE imposes a constraint that is equally safe but more coarse: it does not permit updates to classes whose methods are *active*: a method is active if it is on the activation stack of any running thread, i.e., either the method is currently running or it will be when it is returned to. This constraint ensures type safety because the new version of the program is itself type correct, as required before the update will commence. In particular, if a programmer changes the type of a method `m`, for the program to be type correct, any methods that call `m` must reflect the change. 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` and if any of these are active then the update will be delayed.

While simple, the drawback of this approach is that it fails to handle updates to methods that are on the stack. Static analysis could handle some of these cases. For example, the update to `User` would be permitted even while `loadUser` is running, but only after the call to `setForwardedAddresses`. That said, static analysis cannot easily 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 avoid this problem, past work has proposed extracting out the contents of an infinite loop into separate method 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. We leave such an implementation to future work.

4.3 Loading and Compiling Modified Classes

Once a safe point is reached, the VM then loads and compiles the classes. There are two main steps to updating the implementations of classes and objects once the JVM receives the updated classes. First, the changed classes and methods must be compiled and metadata for the existing classes in the VM must be replaced and/or updated to refer to the newly-compiled classes. Second, existing object instances must be changed 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.

Updated classes are first loaded by a dynamic classloader, a standard feature for all Java VMs, and then compiled. In Jikes, a compiled 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' 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, and the TIB method pointers are to this code. (To amortize the cost of classloading, each method's code is initialized to a stub that will compile the actual method body on the first call.) Each object instance contains a pointer to its TIB, to support dynamic dispatch. When an object's method is called, the generated code simply indexes the object's TIB at the correct offset and jumps to the machine code.

For method updates, in which only the class' method bodies are changed, JVOLVE overwrites the entries in the TIB of the existing class to point to the new method bodies, and discards the old method bodies. All other meta-information about the class can remain unchanged. As a result, existing object instances can be left alone—their TIB pointers and object layout remains valid.

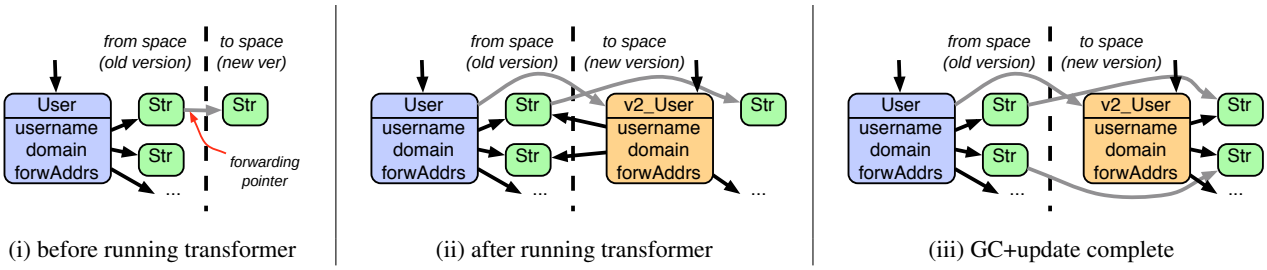
As we pointed out in Section 4.1, methods may have transitive dependences on others methods. Since all highly optimizing JIT compilers perform inlining, they also introduce transitive dependences. For example, if method `m` changes and it is inlined into another method, that method must also be recompiled. In this first prototype implementation, we prevent the compiler from inlining. To support updates with inlining, JVOLVE needs the compiler to keep track of all its inlining decisions. Then if a changed method has been inlined, JVOLVE will add that method to the list of methods to recompile and to the list of methods that must not be on the stack to safely perform the update. We expect to support inlining in the final version of the paper.

Method signature updates are treated the same as class signature updates. In these cases, the class' 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, several Jikes data structures are updated 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. Second, 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. We describe this process next.

4.4 Applying Class and Object Transformers

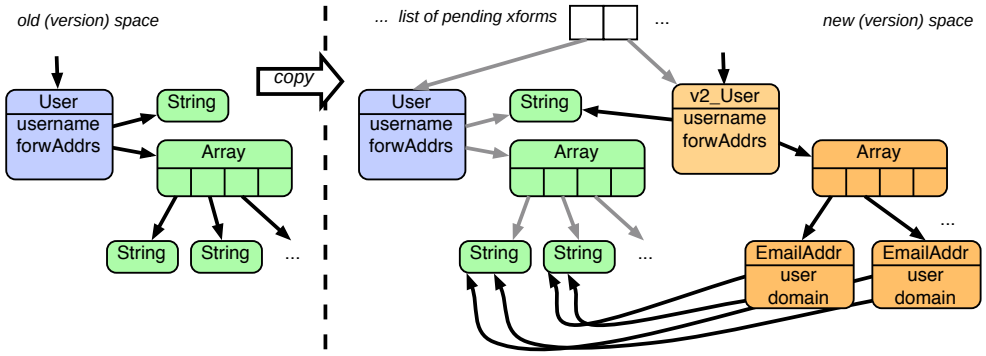
Existing objects whose class signatures have changed must be transformed via their object transformer methods. We have made simple modifications to the Jikes semi-space copying collector to allow JVOLVE to update changed objects as part of the collection. This process is safe because DSU safe points are a subset of the program points at which the VM ensures it is safe to perform GC. For example, the VM ensures that the stack maps are correct. Stack maps enumerate all registers and stack variables that contain references and are required to perform a collection.

A semi-space copying collector normally works by traversing the pointer graph in the existing heap (called *from-space*), and as it encounters objects it copies them to a new heap (called *to-space*). Any fields in the object that are also pointers are recursively scanned and copied in the same way. Once an object is copied, the



(i) before running transformer (ii) after running transformer (iii) GC+update complete

(a) Running the default transformer during the GC



(b) Running transformers following GC

Figure 4. Copying and transforming updated objects with the GC

collector overwrites its header with a forwarding pointer to the corresponding copy in to-space. If the old copy of the object is reached later during the traversal via another reference, the collector just redirects the reference to the new copy of the object. The 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 sufficient space for the new version of the object and initializes it to point to the TIB of the new type. Then it initializes the object's fields either according to the default object transformer, or via the user-provided one, if present.

The default transformer simply copies the pointer values from the corresponding fields in the two objects in the case that their types have not changed, and initializes any remaining fields to their default values. The top of Figure 4 illustrates the default transformation of an old `User` object from Figure 2. Part (i) illustrates the state of the system at the time the collector reaches the old `User` object. The collector has already copied the object to which the `userName` field points and has installed a forwarding pointer. It has yet to trace the second field or its target. The collector then allocates space for the new `User` object in to-space and runs the default transformer, which copies over the values in the fields `userName` and `domain`, which are pointers to from-space objects. At this point the collector continues tracing. Eventually it will scan the new `User` object in to-space and it will then update the `userName` to the forwarded pointer and will likewise copy the `domain` object. Part (iii) depicts the heap at the conclusion of the collection.

This approach works because only the values of fields are copied from old to new objects, and any object to which a field points is not dereferenced in the transformer. If the transformer dereferences an object, in this design, the object might already have been copied and thus be overwritten by a forwarding pointer, as with the `userName` object. To support user-defined `jvolve_object` functions that wish to dereference pointers, we amend our approach as follows. For each object that has a `jvolve_object` function, rather than allocate space only for the new version of the object in to-space, the collector also copies over the old version and scans it as usual. Then, rather than execute the transformer function immediately, the collector stores pointers to the duplicate of the old object and the newly-allocated object in an array. After the collection completes, JVOLVE goes through the array of old/new object pairs and invokes the relevant object transformer, passing the two pointers as arguments. Once all pairs have been considered, the array is deleted which will cause duplicate old versions to become unreachable (as long as no other object points to them) and thus they will be reclaimed at the next collection. At this point, JVOLVE also executes the class transformers.

The bottom of Figure 4 illustrates what the heap would look like at the end of the GC phase in order to support the `jvolve_object` function 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 objects; only the fields `userName` and `forwardedAddresses` are shown. 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` and the duplicate of the old version, both of which are allocated in to-space. JVOLVE then runs the object transformer function and accesses fields using the duplicate in to-space, and thus will never encounter any forwarding pointers. We can see that the new version of the object points to the same `username` field as before, but now points to a new array which points to new `EmailAddress` objects. These objects were initialized by passing in the old e-mail `String` values to the constructor, which assigns fields to point to substrings of the given `String`.

There are two disadvantages to this approach. First, we make an extra copy of an object that is to be upgraded by a user transformer function. While this copy will be dead by the next GC, it adds memory pressure. The second drawback is that an object transformer may only safely access the contents of, or invoke methods on, objects reachable via the `from` argument and only if those objects do not themselves have an object transformer function. This limitation occurs because we cannot be sure whether those objects have yet been initialized—it depends on their order in the array.

These disadvantages have not proved problematic in practice. Very often the default transformer is sufficient, in which case there is no extra copying. The custom object transformer functions we have written are similar to the one in Figure 3. By and large, they copy fields (either base types or pointers) from the old object to the new one. Transformers also tend to allocate new objects for changed or added fields, and these new objects may, once again, refer to objects directly pointed to by the old object. In the figure, the new `EmailAddress` objects point to the `String`s that used to serve as e-mail addresses in the old object.

Nevertheless, we are working to eliminate these limitations. To avoid extra copying, we can compile object transformer functions to be able to safely execute during collection. To do this, the compiler would insert a read barrier that follows forwarding pointers before dereferencing an object. In addition, to provide access to the fields of changed objects in transformer functions, we can customize the collector to completely process the children first using a postorder traversal. This traversal order would guarantee that any object reachable from the object transformer function is sure already initialized. Other programming interfaces are possible, too. For example, Boyapati et al. [6] take advantage of encapsulation information to allow transformer code to access the old versions of pointed-to objects. Our approach allows programs to only see the new versions of objects within transformers, similar to the *representation consistency* invariant proposed by Stoyle et al. [29]. We will let future experience guide our understanding of which programming interfaces are most useful.

Finally, one disadvantage of a garbage collection-based approach more generally is that it requires the application to pause for the duration of a full GC. The alternative would be to apply object and class transformers lazily, as they are needed [26, 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.

5. Experience

To evaluate JVOLVE, we use it to update two open-source servers written in Java: the Jetty webserver[3] and JavaEmailServer,[4] 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. Our experience was positive. Of 19 updates we considered, JVOLVE could support 16 of them. With a modest addition of functionality in our update preparation tool to handle anonymous classes, and a bit more work, we believe we could support the remaining 3 updates. To our knowledge, 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.

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

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

Ver.	# classes added	# changed classes	# changed methods			# changed fields	
			add	del	chg	add	del
5.1.1	0	26	4	1	38	0	0
5.1.2	1	17	0	0	13	0	0
5.1.3	3	22	19	2	59	10	1
5.1.4	0	14	0	4	15	0	2
5.1.5	0	61	21	4	120	5	0
5.1.6	0	7	0	0	20	5	6
5.1.7	0	10	8	0	13	9	3
5.1.8	0	1	0	0	1	0	0
5.1.9	0	1	0	0	1	0	0
5.1.A	0	4	0	0	4	0	0

Table 1. Summary of Release Updates to Jetty

5.1 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. JVOLVE was able to successfully update from each of version to the next for all versions except 5.1.3 and 5.1.5. Both of these updates were precluded by our update preparation tool because it fails to properly identify changes to anonymous inner classes. We believe this limitation is the only one needed to support these updates and we plan to have this functionality for the final paper.

5.2 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 support 8 of the 9 updates, all 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 adding several new classes to implement a file-based system. We believe we could support this change with a little more work by initializing instances of the new classes within a class transformer, referring to state reachable from static variables in the old classes. We aim to do so by the final paper.

As mentioned in Section 4.2, we have a simple safety condition that prevents updates to a method when it is on the stack. The release from version 1.3 to 1.3.1 updates the main loop, and this loop is always active. To implement this update, we manually extracted the body of the loop and made it a separate function. This change was sufficient to permit the update to take place. We could avoid this transformation by using a limited form of on-stack replacement (OSR), as mentioned earlier.

6. 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

Ver.	# classes		# changed					
	add	del	classes	methods			fields	
				add	del	chg	add	del
1.2.2	0	0	3	0	0	3	0	0
1.2.3	0	0	7	0	0	16	12	0
1.2.4	0	0	2	0	0	4	0	0
1.3	4	9	2	11	3	15	12	5
1.3.1	0	0	2	0	0	4	0	0
1.3.2	0	0	8	4	2	6	3	1
1.3.3	0	0	4	0	0	3	0	0
1.3.4	0	0	6	2	0	6	2	0
1.4	0	0	7	6	1	5	6	0

Table 2. Summary of Release Updates to JavaEmailServer

Config.	Req. rate (/s)	Resp. time (ms)
5.1.9 (JVOLVE)	42.9 +/- 6.6	129.5
5.1.10 (Jikes)	42.1 +/- 4.8	114.7
5.1.10 (JVOLVE)	41.2 +/- 7.6	123.8
5.1.10 (upd. idle)	41.0 +/- 7.4	114.6
5.1.10 (upd. midway)	38.6 +/- 7.2	132.1

Table 3. Jetty webserver numbers

started from scratch and following an update and found them to be essentially identical. We report on this experiment in Section 6.1.

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 6.2 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 Ubnutu 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.

6.1 Jetty performance

To see the effect of updating on application performance, we measured Jetty under various configurations using httperf, [5] 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 3 shows our results. The second column reports the 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 over the whole run. The first row illustrates the performance of Jetty version 5.1.9 using the JVOLVE VM, while the remaining measurements consider Jetty version 5.1.10 under various configurations. The second and third lines measure the performance of 5.1.10 started from scratch, under the Jikes and JVOLVE VMs, while the fourth line measures the performance of 5.1.10 updated from 5.1.9 before the benchmark starts. The performance of these three configurations is essentially

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

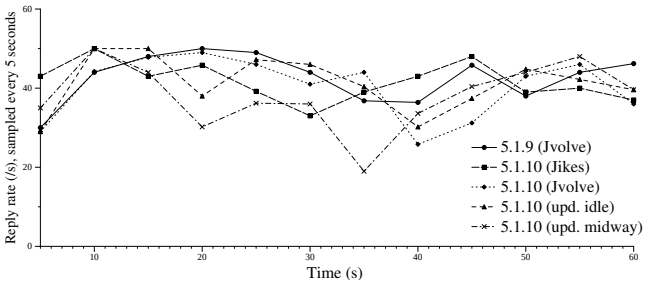


Figure 5. Webserver request rate over time

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 VM (with inlining disabled).

The last line of the table measures the performance of Jetty version 5.1.10 updated from version 5.1.9 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 all the configurations in Table 3. 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.

6.2 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 4 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.

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 reflec-

# 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 4. JVOLVE pause time (in ms) for various heap sizes

tion 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 the Jetty pause reported above.

7. Conclusions

This paper has presented JVOLVE, a Java virtual machine with support for dynamic software updating. JVOLVE is powerful enough to support many changes we observed 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] 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] J. Henkel and A. Diwan. CatchUp! Capturing and replaying refactorings to support api evolution. In *ICSE*, pages 274–283, St. Louis, MI, May 2005.

[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] 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.

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

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

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

[28] 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.

[29] 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.

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

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

[32] ZXTM: A high performance traffic manager. `http://www.zeus.com/products/zxtm/index.html`.