The Dissertation Committee for Suriya Subramanian
certifies that this is the approved version of the following dissertation:

# Dynamic Software Updates: A VM-Centric Approach

Committee:

Kathryn S. McKinley, Supervisor

Steve Blackburn

Michael Hicks

Calvin Lin

Keshav Pingali

# Dynamic Software Updates: A VM-Centric Approach

**by**

**Suriya Subramanian, B.E., M.S.C.S**

**DISSERTATION**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**DOCTOR OF PHILOSOPHY**

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2010

To Amma, Appa, and Shriram.

# Acknowledgments

I am deeply grateful to my advisor Kathryn McKinley for guiding me through grad school with utmost interest and dedication. She was flexible enough to let me do what I wanted, even if it was outside her area. In the days before my first paper submission, she gave me undivided attention and worked through the night until we submitted the paper. At times it seemed like she cared more about my success and well-being than I did myself. I wouldn't have gotten this far without her unflinching positive attitude and encouragement. As I leave grad school, inspired by Kathryn as a researcher and a human being, I will strive to always give my best to those around me.

Over the past three years, Michael Hicks' meticulous attitude towards research has been a positive influence. Always prompt to reply to e-mails, he invariably shot down most of my arguments and imparted some of his vast knowledge in the field of dynamic software updating. I thank him for all his advice and for introducing us to scrum for research [41].

Steve Blackburn has always been eager to offer technical advice and guidance. I thank Steve, Keshav Pingali, and Calvin Lin for their insightful questions and helpful feedback.

My research has benefitted enormously from my interactions with Mike Bond. I thank him for brainstorming engaging research problems as well as for patiently answering technical questions. For their personal and technical support, I thank members of the Speedway research group: Jenn Sartor, Bert Maher, Byeongcheol Lee, Katie Coons, Dimitrios Prountzos, Ivan Jibaja, Alex

Loh, and Na Meng.

This research would not have been possible without JikesRVM. I thank JikesRVM's developers for maintaining an excellent research infrastructure, and the community, especially Dave Grove and Filip Pizlo, for helpful advice.

I am thankful to Gem Naivar, Tom Horn, Lydia Griffith, and Gloria Ramirez for all their guidance and help. My thanks are also due to UT and NSF for directly and indirectly funding my research.

Among my friends, I am indebted to Smriti for her patient listening in times good and bad, and my roommates, Sibi and Ramtilak, for all their support. I thank Smita, Urmila, and other volunteers of Vibha for all the good times and Vishaal, Harish, and Rohan for numerous music sessions that helped keep me sane. My time in Austin was made memorable by several nice folks. I thank them all.

I wouldn't have reached here without my family. My parents Swarnam and Subramanian showered me with love and affection, gave me a great education, and instilled in me the importance of sincerity and hard work. My brother Shriram has been my best cheerleader. To my family, I dedicate this dissertation.

# Dynamic Software Updates: A VM-Centric Approach

Publication No. _____

Suriya Subramanian, Ph.D.
The University of Texas at Austin, 2010

Supervisor: Kathryn S. McKinley

Because software systems are imperfect, developers are forced to fix bugs and add new features. The common way of applying changes to a running system is to stop the application or machine and restart with the new version. Stopping and restarting causes a disruption in service that is at best inconvenient and at worst causes revenue loss and compromises safety. Dynamic software updating (DSU) addresses these problems by updating programs while they execute. Prior DSU systems for managed languages like Java and C# lack necessary functionality: they are inefficient and do not support updates that occur commonly in practice.

This dissertation presents the design and implementation of JVOLVE, a DSU system for Java. JVOLVE's combination of *flexibility*, *safety*, and *efficiency* is a significant advance over prior approaches. Our key contribution is the extension and integration of existing Virtual Machine services with safe, flexible, and efficient dynamic updating functionality. Our approach is flexible enough to support a large class of updates, guarantees type-safety, and imposes no space or time overheads on steady-state execution.

JVOLVE supports many common updates. Users can add, delete, and change existing classes. Changes may add or remove fields and methods, replace existing ones, and change type signatures. Changes may occur at any level of the class hierarchy. To initialize new fields and update existing ones, JVOLVE applies *class* and *object transformer* functions, the former for static fields and the latter for object instance fields. These features cover many updates seen in practice. JVOLVE supports 20 of 22 updates to three open-source programs—Jetty web server, JavaEmailServer, and CrossFTP server—based on actual releases occurring over a one to two year period. This support is substantially more flexible than prior systems.

JVOLVE is safe. It relies on bytecode verification to statically type-check updated classes. To avoid dynamic type errors due to the timing of an update, JVOLVE stops the executing threads at a *DSU safe point* and then applies the update. DSU safe points are a subset of VM *safe points*, where it is safe to perform garbage collection and thread scheduling. DSU safe points further restrict the methods that may be on each thread's stack, depending on the update. Restricted methods include updated methods for code consistency and safety, and user-specified methods for semantic safety. JVOLVE installs return barriers and uses on-stack replacement to speed up reaching a safe point when necessary. While JVOLVE does not guarantee that it will reach a DSU safe point, in our multithreaded benchmarks it almost always does.

JVOLVE includes a tool that automatically generates default object transformers which initialize new and changed fields to default values and retain values of unchanged fields in heap objects. If needed, programmers may customize the default transformers. JVOLVE is the first dynamic updating system to extend the garbage collector to identify and transform all object

instances of updated types. This dissertation introduces the concept of object-specific state transformers to repair application heap state for certain classes of bugs that corrupt part of the heap, and a novel methodology that employes dynamic analysis to automatically generate these transformers. JVOLVE's eager object transformation design and implementation supports the widest class of updates to date.

Finally, JVOLVE is efficient. It imposes no overhead during steady-state execution. During an update, it imposes overheads to classloading and garbage collection. After an update, the adaptive compilation system will incrementally optimize the updated code in its usual fashion. JVOLVE is the first full-featured dynamic updating system that imposes no steady-state overhead.

In summary, JVOLVE is the most-featured, most flexible, safest, and best-performing dynamic updating system for Java and marks a significant step towards practical support for dynamic updates in managed language virtual machines.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Evolving software is a fact of life. Developers constantly fix bugs and add new features to software. These software changes make their way to deployed systems. However, many systems ranging from medical equipment to communication and transportation systems to financial systems, must be always available. For online stores, brokerages, and exchanges an hour of downtime can mean significant revenue loss running into thousands or even millions of dollars [73, 66, 30, 69]. Mission critical applications such as satellites cannot tolerate downtime [18]. When it comes to end-user applications and operating systems, downtime is both expensive and inconvenient [92]. In spite of these concerns, the most common update method used today is to stop a running system, install an update, and restart with the new version. In one study of nearly 6,000 outages of high-availability applications, 75% were for planned hardware and software maintenance [49]. Another study of the IT industry finds 80% of all downtime to have been planned, and 15% of downtime is attributed to software updates [85]. Moreover, security issues force system administrators to patch their systems ever more frequently [11]. With Internet access being ubiquitous, and application and OS vulnerabilities in unpatched systems being exploited more and more, delaying updates poses real security risks [3, 5, 44].

As a solution to these problems, researchers have proposed Dynamic

Software Updating (DSU). DSU is a general-purpose approach to updating running programs. DSU dynamically patches a program running an old version of an application, updating running code and data to a state consistent with a newer version. DSU's appeal is that it can be applied generally without the need for redundant hardware or explicit special-purpose system designs [74]. An ideal general-purpose DSU system should be *flexible*, *safe*, and *efficient* [40, 60].

**Flexible.** A DSU system should be *flexible* enough to handle the type of changes developers typically make between versions. While it is not possible to handle every possible software update, DSU desires to support most software version changes encountered in practice.

**Safe.** Dynamically updating to the new version should be as *safe* as starting the new version from scratch. The system should guarantee to developers and users that the update process does not introduce type or process state errors, and that the updated version executes as intended.

**Efficient.** DSU should provide an *efficient* upgrade process and efficient normal execution. DSU support should add no performance overhead over normal application execution, and the update process itself should be quick. Preparing the patch and testing the update itself requires additional development effort, but this effort should be minimal.

Researchers have made significant strides toward making DSU practical for systems written in C or C++, supporting server feature upgrades [64, 21, 51], security patches [3], and operating systems upgrades [75, 8, 9, 10, 52, 20, 5]. Unfortunately, work on DSU for managed languages such as Java, C#, Python

and Ruby lags behind work for C and C++. However, the use of Java and other dynamic languages is increasingly common, powering systems both small and large [34, 55, 54, 70, 24]. Websites and web-services are increasingly written in dynamic languages such as Python and Ruby. Java powers both large complex systems running stock exchanges and millions of small cell phones and other handheld devices. Even safety-critical applications such as those that control aircraft originally written in Ada, are migrating to Java. Dynamic updating support for managed languages has lagged behind pervasive use of these languages. For example, while the HotSpot JVM [81] and some .NET languages [56] support on-the-fly method body updates, this support is too inflexible for all but the simplest updates [78]. Academic approaches [72, 53, 67, 13] offer more flexibility, but have never been demonstrated on realistic applications, and furthermore, these prior designs impose substantial space and time overheads on steady-state execution.

In this dissertation, we present JVOLVE, a Java Virtual Machine (JVM) that provides general-purpose dynamic updating of Java applications. JVOLVE's combination of flexibility, safety, and efficiency is a major advance over prior approaches. Our key contribution is to show how to extend and integrate existing Virtual Machine (VM) services to support dynamic updating that is flexible enough to support the largest class of updates to date, guarantees type-safety, and imposes no space or time overheads on steady-state execution.

**Flexibility.** A practical DSU system must support changes to software that developers typically make between versions. Prior studies on C applications [61], the Linux kernel [68] and Java applications [25] show that changes can usually be categorized into addition of new methods and types, modification of

method definitions, and changes to the type signatures of methods and data fields. JVOLVE supports these and other common updates. Users can add, delete, and change existing classes. Changes may add or remove fields and methods, replace existing ones, and change type signatures. Changes may occur at any level of the class hierarchy. Changes can alter data structures, e.g., replace a list with a hash map. As a testament to its flexibility, our experience shows that JVOLVE supports 20 of 22 updates to three open-source programs—Jetty web server, JavaEmailServer, and CrossFTP server—based on actual releases occurring over 1 to 2 years.

**Overview of JVOLVE's approach.** JVOLVE consists of 1) an offline source analysis tool called the Update Preparation Tool (UPT) that uses jClasslib Java Bytecode Viewer library, and 2) a JVM with DSU support built on top of JikesRVM, a research VM. UPT identifies differences between source versions and generates an update specification that includes *class* and *object transformer* functions that specify logic to update application data from the old to the new version. The JVOLVE VM takes in this specification and performs the update. Upon being notified that an update is available, JVOLVE pauses the application, performs the update atomically in a single step, and resumes application execution of the updated version. Only old code runs before the update and only new code runs after the update. Application data stored in globals, locals, and the heap always corresponds to the newest version of the application, a property called *representation consistency*. To maintain representation consistency JVOLVE suspends the running program at a *DSU safe point* and converts all object instances to conform to the newest version before it restarts the application.

**Update Safety.** JVOLVE relies on Java's semantics to ensure that the old and new application versions are independently consistent and *type-safe*. JVOLVE ensures a type-safe update process by allowing only unchanged methods to be active on stack when performing the update [77, 64, 8]. JVOLVE relies on the developer and testing process to specify other methods that should not be executing during an update in order to maintain correct application semantics with the new version [37, 63]. For example, consider a method `foo` that makes two consecutive methods invocations to `bar` and `baz`, and requires for correctness that both calls are of the same version. If both methods are updated, the developer should not allow an update when `foo` is active, to prevent an update in between calls to `bar` and `baz`.

To process an update, JVOLVE pauses execution at a *DSU safe point*. DSU safe points are a subset of VM *safe points* where it is safe to switch application threads and perform Garbage Collection (GC). DSU safe points have the aforementioned restrictions on what methods can be active on stack to guarantee a safe update. In situations where restricted methods are active on stack, JVOLVE installs return barriers [90] on these methods that inform the VM that an unsafe method has returned and that it may now be safe to perform an update. JVOLVE's use of return barriers increases the likelihood of an application reaching a *DSU safe point*, but does not guarantee reaching one in multithreaded programs [62]. JVOLVE utilizes OSR to recompile restricted methods with unchanged method bodies that are active on stack. In this case, JVOLVE promptly performs the update. In our experience, if a safe point can ever be reached, JVOLVE's support is sufficient to reach it.

**Updating code.** JVOLVE makes use of Just-in-time (JIT) compilation to

efficiently update the code. JVOLVE invalidates existing compiled code and installs new bytecode for all changed method implementations. When the application resumes execution these methods are JIT-compiled when they are next invoked. The adaptive compilation system naturally optimizes updated methods further if they execute frequently.

**Updating data.** To update live object instances of changed classes, JVOLVE makes use of Garbage Collection (GC), and is the first dynamic updating system to do so. JVOLVE initiates a whole-heap GC, which identifies existing object instances of updated classes. As the last phase of GC, JVOLVE initializes and transforms all updated objects to their new versions using default or user provided object transformers for each updated type. Because JVOLVE identifies all objects it will update before actually invoking object transformers, the system can control the order in which objects are transformed. To our knowledge, JVOLVE is the only system with this functionality. In the current implementation, JVOLVE relies on the developer to specify the required order.

In JVOLVE's natural and intuitive object transformation model, the transformer function receives two objects: an object of the old type corresponding to the state before the update, and an object of the new type corresponding to the state after the update. For global variables, JVOLVE invokes class transformers where the programmer has access to any object reachable from global variables, i.e., static fields of classes. The UPT generates default object and class transformers that are simple, and the developer can write a more sophisticated function if required.

**Specializing and automating transformers.** In this dissertation, we also explore state transformers with logic that depends on the state of the object being transformed, rather than only performing a uniform action for all objects. We use such transformers to repair erroneous application state that results from executing a buggy program version. As part of the update, the dynamic updating system runs state transformers on a subset of objects with corrupted state. We use these transformers to fix memory leaks in real applications. In this dissertation we also present a methodology that starts with a bugfix and instruments the application to mark objects that would be corrupted in a run of the old version of the program. We then perform a novel dynamic analysis that infers a predicate that distinguishes between marked and unmarked objects in the application and automatically generates object transformers that repair the state of objects identified by the predicate. We are the first to consider systematic, accurate repair of buggy application state during dynamic updating.

**Efficiency.** JVOLVE imposes no overhead on steady-state execution. During an update, JVOLVE employs classloading and garbage collection. After an update, the adaptive compilation system will incrementally optimize the updated code in its usual fashion. Eventually, the code is fully optimized and running with no additional overhead. The zero overhead in steady-state execution for a VM-based approach is in contrast to DSU techniques for C, C++, and previous proposals for managed languages. These prior approaches use a compiler or dynamic rewriter to insert levels of indirection [64, 67] or trampolines [20, 21, 3, 5], which add overhead during normal execution.

We assessed JVOLVE by applying it to updates corresponding to one

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

In summary, the main contributions of this dissertation are:

1. New techniques that extend and integrate standard virtual machine services for managed languages to support a flexible, safe, and efficient dynamic software updating service.

2. The first implementation to employ garbage collection to update objects in the heap and a flexible model that allows object transformers to enforce execution order.

3. The first systematic methodology for developing state transformers to repair application state, employing a novel dynamic analysis to automatically generate the transformers.

4. The design, implementation, and evaluation, using real-world applications, of JVOLVE, a Java VM with fully-featured support for dynamic software updating, that is distinguished from prior work in its realism, flexibility, technical novelty, and high performance.

This work demonstrates a significant step towards supporting flexible, efficient, and safe updates in managed code virtual machines. We believe that our design, implementation, and results show that this technology, together with a rigorous testing regime, is ready to be adopted and be a part of future virtual machines.

# Chapter 2

# Background

This chapter presents an overview of the dynamic updating problem. It discusses the semantics of updates, explains safety guarantees that DSU systems provide, and presents mechanisms that systems employ to support DSU. This chapter focuses on DSU system requirements and mechanisms, whereas Chapter 6 covers how prior systems chose among them.

The goal of DSU is to avoid application downtime in the face of software updates. Researchers have addressed the problem of dynamic updating in various contexts such as standalone and server applications, distributed computing, distributed object stores, and databases for various types of code updates, and types of code and data updates. This dissertation considers updating a single application process, changing code and data to be as expected by the new version.

## 2.1 Updating Code

The most primitive functionality any dynamic updating system must support is the ability to call new versions of updated methods. Dynamic updating systems in all types of contexts, be it in a compiled language like C, or in a managed language runtime, or in a distributed computation framework, resort to some form of indirection to call the new version of a function.

Systems for C/C++ such as Ginseng [64] and K42 [75] use indirection for each function call. Each function call goes through a table that points to the latest version of the function. At update time, the DSU system updates table entries to point to new method versions. As a result, all future calls to a method invoke its latest version. All systems using this approach pay an additional overhead for all method calls during normal execution.

KSplice [5], a dynamic updating system for the Linux kernel uses trampolines to achieve indirection. In the absence of an update, kernel execution happens normally. At update time, Ksplice overwrites the first few instructions of an updated method with a call instruction to the new version of a method. Future function calls to an updated method call the old body, which transfers execution to the newest version. With this approach, Ksplice pays almost zero execution time overhead.

Function indirection in managed languages usually comes for free. To make a function call in an interpreted language, the interpreter gets the method body by looking it up by name in a dictionary. Dynamic updating systems that work in the context of an interpreter only have to update this dictionary to point to the new method versions.

In a managed language Virtual Machine that compiles code down to machine code, all non-inlined calls typically go through either a Virtual Method Table (VMT) for virtual calls, or a global table for calls to static methods. These tables point to the latest compiled version of each method. When the VM compiles a method at a higher level of optimization because the method is executed frequently, it updates the table to point to the new version. JVOLVE extends this functionality for dynamic updating, by rewriting table entries to point to the new method version. If the compiler has previously inlined a

changed method into an unchanged calling method, JVOLVE also rewrites the table entries of these calling methods containing an inlined changed method. We are aware of no other system that handles inlining.

## 2.2 Updating data

The most essential and challenging feature of any dynamic updating system is to change application state — stored in local and global variables, and in heap allocated objects — to conform to the semantics, type specification, and concrete representation of the new version. Dynamic updating systems by Hjálmtýsson and Gray [42] and Duggan [28] allow multiple versions of a type to coexist, where code and data objects from the old and new program versions interact freely with each other. In this section, we restrict our discussion to a model where all data values in the application are logically of the latest version of their corresponding type, a property called *representation consistency* [77]. A system that maintains representation consistency transforms all objects to correspond to their new type at update time, or transforms each object when the application next accesses it.

In order to satisfy the semantics of the update, updating systems use automatically-generated or programmer-written *state transformers* that return new program state from old state. Depending on the dynamic updating context, state transformers operate on stack state, global variables, heap objects, or database tables. This section discusses mechanisms that systems employ to support updates to application data and the semantics of updating data using state transformers.

### 2.2.1 Implementation mechanisms

To update data, a system must address two questions. First, how does the concrete representation of objects facilitate updating? Second, when are object transformers invoked?

**Concrete representation** Systems such as Ginseng create a wrapper type for each updateable type in the application. The system instruments the program so that all object accesses go through the wrapper types. The wrapper type uses padding to allow a new version to add fields to a type. The advantage of padding is that it is straightforward to implement and integrates seamlessly with the rest of the application. Objects declared as local variables and those allocated dynamically on the heap are all update ready and are treated the same. The disadvantages are that it wastes space and that a type cannot grow larger than the initially allocated space.

An alternative approach is to use indirection where a field of the object points to the additional fields in the updated types. Such an approach is employed in the K42 operating system [75]. Indirection allows types to grow arbitrarily large in size, but adds a memory access to dereference the indirection pointer for each access to fields of the new version.

Another approach is to retain the same representation of objects as in a system without dynamic updating. With this approach, the system allocates new objects during update time and appropriately copies over contents from old objects. However, such a system has to ensure that pointers to objects are changed during the update to point to the newly allocated objects. JVOLVE is the first to implement this functionality. JVOLVE does so by extending the Virtual Machine's garbage collector, as explained in detail in Section 3.5.5.

**Transforming objects** A design decision that dynamic updating systems make is when and how to invoke state transformers on objects in the application. One approach is to lazily transform objects after the update [64, 17, 72]. The system instruments every data access to check whether the concerned object is of the latest type and invoke its state transformer if the object is not up to date. The disadvantage of this approach is that the system must always incur the overhead of instrumentation and the addition of a field in every object to keep track of the version number. The advantage is that lazily transforming objects amortizes the cost of invoking state transformers by spreading it across application execution.

The other alternative is to eagerly transform all objects during update time which requires a way to access all objects in the application. With this model, the programmer must specify how to explicitly trace and transform objects starting from global variables [40], or the system must maintain a registry of all live objects in the application [75]. JVOLVE which implements eager transformation piggybacks on the garbage collector to trace and identify live objects that need updating, and updates each such object based on programmer specification.

### 2.2.2 Semantics of state transformers

The state of a running process consists of values of local and global variables, heap data, and one or more Program Counters (PCs) indicating the current execution point of all running application threads. A state transformer maps state from the old version of a program to state as expected by the new version. The semantics of the update is as intimately tied to the definition of the state transformer function, as it is to the definition of the old and new

program versions themselves.

For instance, a state transformer that initializes all variables to the value "unknown" and the PC to the start address of the new version, is equivalent to stopping the old version of the program and restarting the new one. Such a transformer would not be very useful and would defeat the purpose of DSU. A useful and meaningful transformer has to come with an understanding of semantics, both of the application and the update. There might be updates where it is impossible to have a meaningful state transformer. Currently, DSU systems rely on the programmer to provide such a state transformer function.

As an example, consider a bugfix where a programmer used a 32-bit counter in an older version and changed it to a 64-bit counter in a newer version, presumably because the 32-bit counter was insufficient to represent real-world values of the counter. The best any state transformer can hope to do in this scenario is to copy the old counter value into the new version and pad the higher order bits with zeros. If the counter had indeed wrapped around in the old version, there would be no way for the transformer to be aware of this fact and know what the higher order bits should be. Leaving the higher order bits as zeros might or might not affect useful execution of the updated version. What is acceptable totally depends on the application and update semantics and the expectations of the developers and users.

As another simple example, consider an application that stores points with x and y co-ordinates in a 2-dimensional space. A feature of a newer version might be that the application now supports a 3-dimensional space with points having a z co-ordinate as well. In this case as well, a state transformer function cannot hope to obtain an accurate representation of the new version's state. Setting the z co-ordinates to zero of existing points in the application

might in fact work. It also seems intuitive that setting z co-ordinates to values other than zero might cause the application to work improperly. Such an inference has to come from the developer with an understanding of the real-world semantics of the application.

In this work, we assume that a correct and safe state transformer does indeed exist, and ask what safety guarantees such as update correctness, type safety, transaction safety, and representation consistency that a DSU system can provide.

## 2.3 Safety of updates

Supporting updates to code and data in a system should not compromise its safety. By safety, we mean that we want to make guarantees that a DSU system and the update are *valid* and that the update and the application would not perform *illegal* operations that are usually disallowed by normal execution semantics. For example, the update should not lead the new version to crash because it accessed an invalid memory location, or dereferenced a null pointer, or accessed an object of a different type than it expected.

**Update validity** One guarantee a dynamic updating system may want to make is that the update is *valid*. Gupta et al. offer the following definition of update validity [38, 37]. A process or a running program $P$ is a pair $(\Pi, s)$, where $\Pi$ refers to the program's code and $s$ to its state. The state as mentioned above comprises locals, globals, heap data, and the current PC. An update to $P$ is a pair $(\Pi', S)$ where $\Pi'$ is the new version's code and $S$ is the state transformer function. Applying the update involves applying the state transformer function on the old state. The PC value of the old state is called

the *update point* and the resulting new state's PC specifies the instruction at which to resume execution. The updated process is $(\Pi', s')$ where $s' = S(s)$. An update is valid if and only if the new program resuming execution at state $s'$ eventually goes to a *reachable* state $s''$. We call $s''$ to be reachable if the new program starting from its initial state on the same set of inputs at some point reaches state $s''$. Gupta et al. showed that the problem of deciding whether or not an update is valid for a state transformer in the general case is undecidable.

Undecidability means that we cannot come up with a general purpose algorithm that, given $\Pi$, $\Pi'$, $S$ and $s$, can say whether or not an update is valid. However, they show that update validity can be verified formally by restricting at which points an update takes place and what code the state transformer can contain. These restrictions are too conservative. They only admit simple changes to applications. While it might be impossible to guarantee update validity, in general, we consider the following safety properties.

**Type safety** Type safety is a well-understood and highly desired property of real programming languages. A type-safe system guarantees that any data element accessed by code is of the right type expected by the code. A type-safe DSU system guarantees type-safety of transformer code and new program code that runs after the update.

DSU systems that support realistic changes provide a way to update user defined types that change from the old to the new version. Each type $t$ that has changed representation from type $\tau$ in the old to type $\tau'$ in the new version requires a *type transformer* function of type $\tau \rightarrow \tau'$. To keep update semantics intuitive, DSU systems enforce that at any given point of time, there is exactly one representation of a type $t$ and that is the newest representation,

a property called *representation consistency*. To provide type safety, a DSU system guarantees that no code is run, during or after the update, that expects a representation of an earlier version.

**Activeness safety** A simple way to support representation consistency and type safety is by not allowing any changed or deleted methods to be active on stack at an update point. This restriction is called *activeness safety* [86, 5, 3, 9]. Activeness can be checked with a simple and accurate dynamic test that walks all application stacks and looks for changed or deleted methods that are active at a potential update point. It can also be enforced with a conservative static analysis that examines the call graph of the old version. With either approach, activeness safety guarantees type safety as follows. Consider the set of all methods in the old and new version of the application. Some methods exist in the old version but not in the new, either because these methods are changed, or removed in the new version. Conversely, some methods exist in the new but not in the old version. Presumably, there are unchanged method bodies that are common to both the old and the new version. In a type-safe language, the old and the new program versions are independently type-safe. Activeness safety restricts active methods at an update point to the intersection between the old and new versions. At update time, type transformer functions convert all object representations to conform to the new version. After the update, the application can only execute the new version methods, which are type-safe by definition. JVOLVE uses activeness safety because it is simple, guarantees type-safety, requires only a list of changed methods, and is very efficient to check at update time.

Restricting modified methods to be not active at update time can be too

```
1 function foo() {
2   ...
3   access type t1;
4   ...
5   access type t2;
6   ...
7 }
```

t1 is changed in the new version, t2 is not

Figure 2.1: Simple function illustrating con-freeness safety

constraining for multithreaded programs and for changes that affect methods high in the call chain. These limitations stand in the way of correctly dynamically updating more programs [40, 4]. A system with activeness safety would never be able to update, for instance, an application that prints its version number at the start of its main method, because the main method would always be active. An alternative is to allow old methods to run to completion after the update, but invoke new version bodies for future method calls.

**Con-freeness safety** Stoyle et al. have defined a property called con-freeness of an update that ensures type safety and have developed a static updatability analysis that answers whether or not an update point in the program would violate con-freeness [77]. An update point $p$ is said to be con-free if code that comes after $p$ (which would run after the update) does not concretely access any updated type. Consider the simple function `foo` shown in Figure 2.1. `foo` concretely accesses objects of two different types `t1` and `t2` respectively. `t1` is a type whose representation is changed in the new version, while `t2`'s representation remains the same. The update process runs type transformers for all objects of type `t1`. Let us look at con-freeness at update points corresponding to line numbers 2, 4, and 6. Line 2 is *not* con-free for the update

```
1 transaction {
2   ...
3   foo();
4   ...
5   bar();
6   ...
7   baz();
8   ...
9 }
```

Figure 2.2: Simple transaction region marked by the programmer

because the function will expect a type `t1` object of the old representation, but encounter a new version one. Line 4 is con-free for the update because `t2`'s representation is unchanged between the old and new versions. Line 6, of course, is con-free as it is the end of the method with no unsafe access possible. Stoyle et al. test for con-freeness with a flow-sensitive backwards dataflow analysis. Con-freeness safety is less restrictive than activeness, but recent work shows that exploiting some of these additional update points can lead to incorrect updates in real applications [39].

**Transaction safety** Transaction safety is a guarantee that a marked transaction fully obeys either the semantics of the old version or that of the new version. Consider the simple example in Figure 2.2, where the programmer has marked a region of code as a transaction. Let as assume that code in the transaction is itself unchanged but methods `foo`, `bar` and `baz` might have changed. Line 2 is always a safe update point since the entire transaction will run the new code. Line 8 is also a safe update point since the entire transaction would have run the old code. If only one of the three called methods is updated, all program points in the transaction are update safe, i.e., if the program point occurs before the call to the changed method, the transaction

will have the semantics of the new version, whereas if the program point occurs after the call to the changed method, the transaction will have the semantics of the old version. Now, consider that both `foo` and `baz` are changed in the new version. Line 4 and 6 are unsafe points because the transaction would run the old version of `foo`, but run the new version of `bar`, violating transaction safety.

### 2.3.1 Assuring safety by testing

When program analyses fail to provide formal guarantees of correctness and safety, software developers use testing to develop confidence that their programs execute correctly. The safety of Dynamic Software Updating should be informally assured the same way. Hayden et al. [39] have devised a framework that exhaustively tests updating an application. Their starting point is a set of regression tests that is already used on a daily basis to assure developers that the application runs correctly. Testing that the program can be safely updated at all possible program points for each regression test is prohibitively expensive. However, they employ a novel dynamic analysis that minimizes the space of update points by grouping them into equivalence classes. Update points in an equivalence class all produce the same execution behavior for a particular program trace. Update points that do not pass all regression tests should be marked as unsafe when updating a production system. Results from their work show that activeness safety and con-freeness safety very closely approximate update correctness, but are not sufficient to guarantee correct program execution.

## 2.4 Update Timing

The previous section dealt with safety properties at each program point for a specific update. In this section, we discuss update timing, or how a dynamic updating system ensures that an application reaches a program point that is safe for the update.

**Update points in a program** DSU systems provide API calls that check for and perform an update, and typically allow either the programmer or the compiler to instrument program update points. Note that these points are potential update points, and it is not necessarily safe to perform an update at these points. JVOLVE uses method entry points, method exit points, and loop backedges as potential update points. These points are the same as *safe points* in a Virtual Machine. At these safe points, the VM can safely switch threads and perform garbage collection. Upstare, a DSU system for C, has the same update points as JVOLVE. In JVOLVE, the compiler already instruments these safe points for normal program execution, whereas in Upstare, the DSU system has to instrument the program to make it updateable.

The developers of Ginseng observed that programs that benefit most from dynamic updating are typically structured as long-running event processing loops. Each loop iteration is usually independent of each other and processes a particular transaction. The start of each loop iteration serves as a *quiescent point* where there are no partially-completed transactions, and all global state is consistent. The common use case scenario in Ginseng is for the developer to mark update points at the start of outer loop iterations. Ginseng performs a static analysis to ensure that it is safe to update at a marked program point, and updates the application during runtime.

**Return barriers** In JVOLVE, a user can trigger an update at any time during program execution. While JVOLVE will suspend the application at the earliest VM safe point it encounters, this point is not necessarily safe for the update. For instance, a modified method might be active on stack, violating activeness safety discussed in Section 2.3. In such situations, JVOLVE installs return barriers that trigger an update after all unsafe methods have returned. Return barriers are most useful for long-running or frequently-invoked unsafe methods, which are likely to be on stack almost all the time. Return barriers are not sufficient if an unsafe method contains an infinite loop, and would never return. There are at least two good solutions which deal with infinite loops — stack reconstruction, used first in Upstare, discussed in Section 2.4.1, and loop extraction, used first in Ginseng.

**Loop extraction** In loop extraction, the programmer can mark potentially unsafe long running loops, and the compiler will extract the loop out into its own function, that is called on each loop iteration. If an update changes the loop body, the extracted function will be unsafe for the update, but the update can happen after it returns and later loop iterations call the new version's function. Because of the code change, state used by the loop across iterations might have to be updated as well. Ginseng automatically generates state transformers for this loop state.

### 2.4.1 Updates to active methods

A dynamic updating system can improve flexibility by performing updates to methods that are active on stack. Supporting updates to active methods in their full generality makes it impossible to guarantee even non-semantic

safety properties such as type-safety or correct execution that respects the language specification. However, actual changes to real applications drive DSU systems to support some changes to active methods. Systems lie on various points in the spectrum from no support at all to more general support for arbitrary code changes.

Systems that enforce activeness safety, do not support updates to active methods. As mentioned in Section 2.3, these systems are type-safe but very restrictive. They do support a simple update that changes a version number string printed by the main method of the application, since the main method is always active. Systems that enforce con-freeness safety, for instance Ginseng [64], support such updates by allowing existing methods to run the old version, while executing newer versions for future invocations.

When considering changes to methods, it is important to mention the level of abstraction at which we compare method versions. In systems that compile source code to machine code, either statically in the case of C or dynamically in the case of Java, the source code may be the same across versions, while the compiled machine codes is different, as explained in more detail in Section 3.3. When DSU systems refer to changes, they usually do so at the compiled code level. Jvolve enforces activeness safety at the bytecode level, i.e., only methods with unchanged bytecodes can be active at update time. At the machine code level, Jvolve performs On-Stack Replacement. Jvolve recompiles the method to generate machine code that conforms to the new version of the application and switches execution of an active method to this new code.

**Stack extraction** In order to support arbitrary changes to active methods, a DSU system must extract an active method's stack state, transform it to satisfy the new version of the method, and transfer execution to the right instruction in the new method's body. Upstare is the only dynamic updating system we are aware of, that supports updates to active methods [51]. Programmers write a stack transformer function that takes takes the old version's stack and returns a new one. Upstare's *stack extraction* support takes the old version's stack, applies the programmer specified transformer and resumes execution after the update. This model relies heavily on developer expertise and testing.

### 2.4.2 Multithreaded applications

The above discussion on update points focussed mainly on single-threaded applications. Other than stack reconstruction, which can update unsafe methods in more than one thread, all other mechanisms fail for multithreaded programs. Unsurprisingly, the challenge of synchronizing multiple threads to all simultaneously be at a safe update point is hard. A DSU system has to suspend a thread at a safe point, while waiting for other threads to also reach respective safe points. This waiting can adversely affects application throughput, and in the worst case may deadlock the application. Neamtiu et al. address this problem in their work on Stump [62]. Stump allows the developer to specify a few program points (in each thread) to be update safe. The system then uses static analysis and runtime support to expand this list to other program points with safe behavior. The runtime system synchronizes across threads, resuming them if they have waited for too long.

## 2.5 Conclusion

In this chapter, we presented an overview of the dynamic software updating problem. We discussed mechanisms for updating code and data that meet desirable safety guarantees while processing the update in a timely fashion.

# Chapter 3

# JVOLVE System

This chapter discusses the JVOLVE system. We present a high level view of the system, supported changes, and the JVOLVE Virtual Machine implementation.

## 3.1 Introduction

Figure 3.1 illustrates the dynamic updating process. The left portion depicts the work done offline. The developer writes the old and new versions of the application, and tests them as part of the software development process. The Update Preparation Tool (UPT) examines source code of the old and new versions of the application and prepares a dynamic patch. The user feeds the patch to a process with dynamic-updating support, depicted on the right. The figure shows two processes, one running version one of an application, showing stacks, code and heap; and another running version two of the same application. The goal is to transition from a process running an old version of the application, to one running the new version, on the fly, without stopping and restarting the application.

We assume that developers write and fully test both the old and new versions using standard development practices without anticipating that the application is going to be updated dynamically. Testing is already a well es-

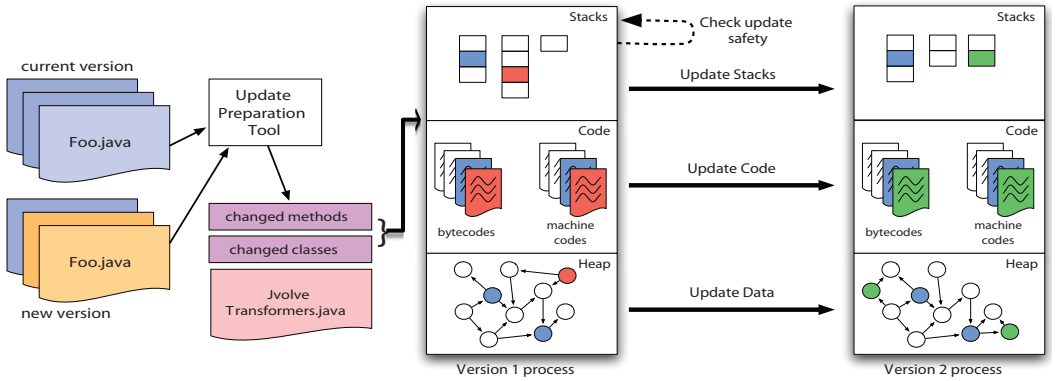



Figure 3.1: Overview of the Dynamic Updating process.

tablished part of software development. With dynamic updating, developers should test the update process, in addition to testing their applications. When it comes time to perform the update, the developer provides the source code for the old and new versions to JVOLVE's Update Preparation Tool (UPT). The UPT compares these two versions and provides a specification for the update. The specification consists of two major parts. First, it contains information about code changes that inform the VM when it is safe to perform the update, what old methods to invalidate, and what new method bodies to load. Second, it informs the VM how to deal with changes to data. The VM uses this specification to transform classes and objects in the heap to conform to their new version. The programmer has the option to modify the update specification. Specifically, the UPT does not reason deeply about the semantics of data structure changes. The programmer may need to modify the UPTs output to obtain a correct program.

Given an update specification, the user signals the running VM to apply the update. The VM loads the new class files and schedules the update. The VM scheduler signals an interrupt, which stops all threads at VM safe points,

where it is safe to perform thread scheduling and garbage collection. JVOLVE then checks if the VM is at a *DSU safe point*. DSU safe points require that no thread's activation stack contains a *restricted* method, which is part of the specification.

Restricted methods are of three categories: (1) methods changed by the update, (2) methods whose bytecode is unchanged but whose compiled representation may change, and (3) methods specified by the user or testing process for semantic reasons. If restricted methods are on stack, the VM installs return-barriers for (1) and (3), and performs on-stack-replacement for (2) to reach a DSU safe point. Section 3.5.2 describes how JVOLVE reaches a safe point in detail.

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

## 3.2 Supported Changes

We have designed a simple, yet flexible update model that supports updates that we have seen to be common in practice.

```
1 private int x;
2 private static int y;
3 public int getX();
4 public static int getY();
```

(a) Old version

```
1 private double x;
2 private static double y;
3 public double getX();
4 public static double getY();
```

(b) New version

Figure 3.2: Examples of an update that changes class signature

**Method body changes** Programmers may change method bodies. Method body updates are the simplest and most commonly supported change [81, 56, 29, 36, 67, 75, 42]. DSU systems can preserve type safety by simply invoking the new method the next time the program executes the method. However, restricting updates to only method bodies prevents many common changes [61]. Chapter 5 shows that over half the releases of Jetty, JavaEmailServer, and CrossFTP, the programs we studied, change more than method bodies.

**Class signature changes** Programmers may also change class signatures in various ways. The class signature includes all fields and methods defined by the class and those inherited from super classes. A programmer may change method signatures by changing the type or number of method arguments. They may add or delete virtual and static field members, and change the types or access modifiers of existing members. These changes may occur at any level of the class hierarchy. For example, programmers may delete a field from a parent class and this change will propagate correctly to the class's descendants.

Figure 3.2 provides examples of class signature changes. Each line in the figure defines either a field or a method and represents a class signature change

```
1  public class User {
2    private final String username, domain, password;
3    private String[] forwardAddresses;
4    public User(...) {...}
5    public String[] getForwardedAddresses() {...}
6    public void setForwardedAddresses(String[] f) {...}
7  }
8  public class ConfigurationManager {
9    private User loadUser(...) {
10      ...
11      User user = new User(...);
12      String[] f = ...;
13      user.setForwardedAddresses(f);
14      return user;
15   }
16 }
```

(a) Version 1.3.1

```
1  public class User {
2    private final String username, domain, password;
3    private EmailAddress[] forwardAddresses;
4    public User(...) {...}
5    public EmailAddress[] getForwardedAddresses() {...}
6    public void setForwardedAddresses(EmailAddress[] f) {...}
7  }
8  public class ConfigurationManager {
9    private User loadUser(...) {
10      ...
11      User user = new User(...);
12      EmailAddress[] f = ...;
13      user.setForwardedAddresses(f);
14      return user;
15   }
16 }
```

(b) Version 1.3.2

Figure 3.3: Changes to JavaEmailServer `User` and `ConfigurationManager` classes from version 1.3.1 to version 1.3.2

in the new version. Internally, UPT does not view this update as changing the signature of individual fields and methods. Instead, it views the update as removing the field `int x` and adding a new field `double x`, and similarly for the methods. Irrespective of how UPT views this change, it is pertinent to note that any reference to these fields and methods in the new version must conform to the new type signature. The Java to bytecode compiler will ensure that the standalone new version of the program is well-formed.

JVOLVE does not support permutations of the class hierarchy, e.g., reversing a super-class relationship. While this change may be desirable in principle, in practice, it requires sophisticated transformers that enforce update ordering constraints. None of the program versions we examined make this type of change. JVOLVE also does not support renaming a class, though this functionality should be easy to add.

**Example.** Consider the following update from JavaEmailServer, a simple SMTP and POP e-mail server. Figure 3.3 illustrates a pair of classes that change between versions 1.3.1 and 1.3.2. JVOLVE fully supports these changes. JavaEmailServer uses the class `User` to maintain information about e-mail user accounts in the server. Moving from version 1.3.1 to 1.3.2, there are three differences. First, the method `loadUser` fixes some problems with the loading of forwarded addresses from a configuration file (details not shown). This change is a simple method update. Second, the array of forwarded addresses in the new version contains instances of a new class, `EmailAddress`, rather than `String`. This change modifies the class signature of `User` since it modifies the type of `forwardedAddresses`. Finally, the class's `setForwardedAddresses` method is also altered to take an array of `EmailAddress`es instead of an array

of `Strings`, and code from `loadUser` accommodates this change as well.

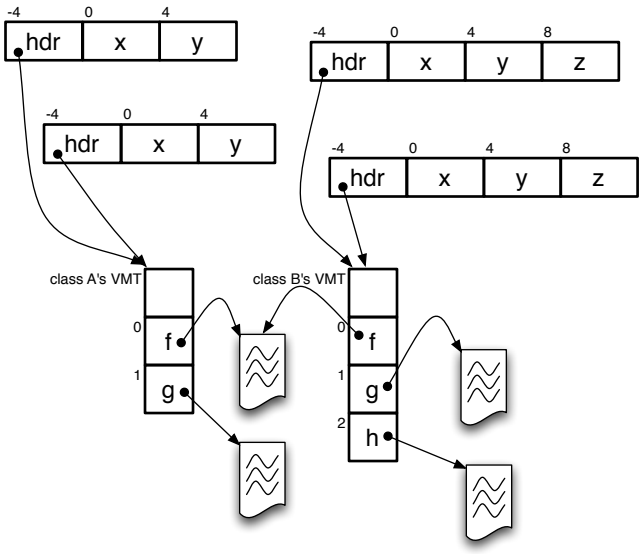
## 3.3 VM object model and method dispatch

JVOLVE's dynamic application of an update closely follows UPT's static specification, and is related to how a JVM would support field accesses and method calls. In languages such as C and Java accessing a field involves reading to or writing from an offset from a structure or object's address. This offset is determined at compile-time. In a language like Java, by compile-time we mean the time when bytecode is translated to machine code. Method calls in C, involve jumping to a memory location that contains the machine code for the called method. Method calls in Java are similar, except that a JVM needs to support *virtual method dispatch*. There can be different definitions of the same method and which one is invoked depends on the object in hand. JVMs use Virtual Method Tables which contain pointers to compiled machine codes of methods. A method call looks up the contents of a particular slot in the VMT and jumps to that address.

We illustrate how a typical VM supports field accesses and method calls with a simple example shown in Figure 3.4. The class `A` defines two integer fields `x` and `y`. Objects of type `A` have these fields laid out contiguously. Objects of `class B` have an additional integer field `z`. In managed languages, the runtime system maintains a header field for all objects, which it uses to look up at runtime the type of a particular object instance. The runtime also uses the object header to look up the type's VMT to invoke methods. `class A`'s VMT has method `f()` at slot 0 and method `g()` at slot 1. The class `B` overrides method `g()`. In addition, `class B` defines a new method `h()` that takes slot 2. Figure 3.4 (b) shows VMTs of `A` and `B`. VMT slots point to

```
1  public class A {
2    private int x, y;
3    public void f() {
4      g();
5    };
6    public void g() {
7      this.y = 0;
8    }
9  }
10
11 public class B
12           extends A {
13   private int z;
14   public void g() {
15     this.z = 0;
16   }
17   public void h() {}
18 }
```

(a) Java source code




(b) Objects and VMTs in the heap

```
1  aload_0
2  invokevirtual <A.g>
3  return
```

(a) Bytecode for A.f()

```
1  The stack pointer points to "this"
2  EDX := this
3          MOV     EDX       [ESP]
4  The VMT is at offset -4
5  ECS := VMT
6          MOV     ECX    -4[EDX]
7  Send this parameter in EAX
8  EAX := this
9          MOV     EAX       [ESP]
10 Call function g()
11 g() is at offset 8 within VMT
12         CALL 8[ECX]
```

(d) Machine code for the `invokevirtual` instruction

Figure 3.4: Simple example of method and field accesses to illustrate how inheritance is implemented in Java

compiled machine code of respective methods. `A.f()` and `B.f()` both point to the same code, while `A.g()` and `B.g()` point to different methods since `B` overrides `g()`. The figure also shows objects of type `A` and `B`. The header field of these objects (chosen arbitrarily to be at offset -4) allow access to their respective VMTs.

Figure 3.4 (d) shows function `f()`'s generated machine code. The call to `g()` refers to VMT slot 1 (at offset 8). This call invokes either `A.g()` or `B.g()` based on the type of the object making the call. Whenever the VMT slot of a method is known at compile time, callers of that method use the offset in their machine code. Similarly, the machine code usually contains offsets of fields as well — for instance, functions `A.g()` and `B.g()` referring to offsets for `y` and `z` respectively. The interface a VM presents to the compiler consists of the following — Virtual Method Table with methods at various slots, and objects with fields laid out at various offsets.

## 3.4 JVOLVE's view of updates

JVOLVE groups updates presented by the UPT into two categories — those that change the exposed representation of a class, and those that don't. JVOLVE handles updates as follows.

- Method body changes: JVOLVE invalidates any old machine code, loads the new bytecode of the method, and lazily generates new machine code.
- Class signature changes:
    1. The update changes the number and offsets of fields within an object. JVOLVE creates new object instances that conform to the

layout of the new version and copies fields appropriately. JVOLVE also needs to initialize fields that do not exist in the old version. Section 3.4.1 talks about such changes and how JVOLVE transforms objects to conform to the semantics of the application.

2. The update changes the number and offsets of method slots in a class' Virtual Method Table. JVOLVE creates new VMTs and points all object instances to this new VMT.

- Methods that are not changed by the update but that refer to classes with signature changes. The machine code of such methods will have field and method offsets that are invalid in the new version. JVOLVE invalidates old codes, and lazily recompiles such methods to generate machine code afresh. We refer to such methods as *indirect updates*.

JVOLVE, with respect to the implementation of dynamic updating, does not view classes as monolithic collections of fields and methods. Instead it views them as method bodies that need to be updated, and structure instances whose fields need to be aligned and have proper values to conform to the semantics of the new version.

### 3.4.1 Class and Object Transformers

For classes whose signatures have changed, an object transformer method initializes a new version of the object based on the old version. For example, consider the class `Point` in Figure 3.5. The `Point` class in the old version represents points in a 2-dimensional space with fields `x` and `y`. After the update, the new version represents points in a 3-dimensional space with the additional field `z`. The object transformer's job is to modify each ob-

```
1 class Point {
2   double x, y;
3 }
```

(a) Old version

```
1 class Point {
2   double x, y, z;
3 }
```

(b) New version

```
1 class JvolveTransformers {
2   public static void jvolveObject(
3     Point to, old_Point from) {
4     to.x = from.x;
5     to.y = from.y;
6     to.z = 0.0;
7   }
8 }
```

(c) Default UPT-generated object transformer

Figure 3.5: Example of a simple class and the default object transformer

ject instance of type `Point` in the heap to conform to its new class definition. Class transformers serve a similar purpose and update static fields, rather than instance fields. The UPT generates default class and object transformers automatically, retaining unchanged fields and initializing new or changed ones. The default object transformer, show in Figure 3.5 (c) for our changed `Point` class copies fields `x` and `y` from an old object to a transformed object and initializes `z` to zero.

For our example from JavaEmailServer in Figure 3.3, the UPT identifies that the `User` and `ConfigurationManager` classes have changed, and produces default object transformers. The programmer elects to modify the object transformer for the class `User`, as illustrated in Figure 3.6.

Object and class transformer methods are simply `static` methods in the class `JvolveTransformers`, which is created by UPT and loaded by Jvo-

```
1  public class v131_User {
2    private final String username, domain, password;
3    private String[] forwardAddresses;
4  }
5  public class JvolveTransformers {
6   ...
7   public static void jvolveClass(User unused) {}
8   public static void jvolveObject(User to, v131_User from) {
9      to.username = from.username;
10     to.domain = from.domain;
11     to.password = from.password;
12     // default transformer would have:
13     //   to.forwardAddresses = null
14     int len = from.forwardAddresses.length;
15     to.forwardAddresses = new EmailAddress[len];
16     for (int i = 0; i < len; i++) {
17       String[] parts =
18         from.forwardAddresses[i].split("@", 2);
19       to.forwardAddresses[i] =
20         new EmailAddress(parts[0], parts[1]);
21     }
22   }
23 }
```

Figure 3.6: `User` object transformer for update from JavaEmailServer version 1.3.1 to version 1.3.2

LVE at update time. The class transformer method `jvolveClass` takes an instance of the new class as a dummy argument. Standard overloading in Java distinguishes the `jvolveClass` methods for different classes. (In our example, `jvolveClass` does nothing.) The object transformer method `jvolveObject` takes two reference arguments: `to`, the uninitialized new version of the object, and `from`, the old version of the object. We prepend a version number to the names of old classes to distinguish them from the new versions. Based on the UPT specification, but before the VM loads the `JvolveTransformers` class, the VM renames the old class in all its internal data structures. This renaming makes the class name space and the `JvolveTransformers` class type-correct.

In our example, the VM renames the old version of `User` to class `v131_User`, which is the type of the `from` argument to the `jvolveObject` method in the new `User` class. The `v131_User` class contains only field definitions from the original class; all methods have been removed since the updated program may not call them, as discussed below.

A typical transformer initializes a new field to its default value (e.g., `0` for integers or `null` for references) and copies references to the old values. In the example, the first three lines simply copy the previous values of `username`, `domain`, and `password`. A more interesting case is the field type change to `forwardedAddresses`. The default transformer function would initialize the `forwardedAddresses` field to `null` because of the type change. The customized update function in Figure 3.6 instead allocates a new array of `EmailAddress`es and initializes them to substrings of the `String` objects from the old array.

Because the transformer class is separate from the old and new object classes, the Java type system would normally forbid the transformer, access to their `private` fields. There is no obvious solution to this problem that conforms to the Java type system during an update. We could define object transformers as methods of the old changed classes, which would grant access to the old fields, but not the new ones. Defining transformers as methods of the new changed class has the reverse problem. Also, the Java type system would disallow writes to `final` fields from within the transformer functions. A `final` field is "write once" fields and can be written to only in constructor methods. To avoid these problems, we compile our transformation class separately. We extend the JastAdd Java-to-bytecode extensible compiler [32] to ignore access modifiers (e.g., `private` and `protected`) and allow methods

to assign to `final` fields only during an update. Bytecode that ignores these modifiers would not normally verify. JikesRVM, on which JVOLVE is built, does not implement a bytecode verifier. Aside from this exceptional case, JVOLVE classes are compiled normally and would pass verification. The VM executes these Java functions normally, because they are otherwise standard Java. Since the transformation class is only active and available during the update, after the update the system no longer accesses the transformer functions. Separating transformers from updated classes avoids cluttered class files at run-time, and makes dynamic updating more transparent to developers.

Supported in its full generality, a transformer method may reference any object reachable from the global (`static`) namespace of both the old and new classes, and read or write fields or call methods on the old version of an updated object and/or any objects reachable from it. JVOLVE presents a more limited interface (similar to past work [72, 53]). In particular, the only access to the new class namespace is via the `to` pointer, whose fields are uninitialized. The old class namespace is accessible, with two caveats. First, fields of old objects may be dereferenced, but only if the update has not changed the object's class, or if it has, after the referenced objects are transformed to conform to the new class definition. Second, no methods may be called on any object whose class was updated. In Figure 3.6 class `v131_User` is defined in terms of the fields it contains; no methods are shown. As explained in Section 3.5.5, these limitations stem from the goal of keeping our garbage collector-based traversal safe and relatively simple. This interface is sufficient to handle all of the updates we tested. Section 4.1 goes into detail on how our implementation influences our model for object transformers, its limitations and alternative approaches.

An alternative programming model would be that transformers could dereference `from` object fields and see the *old* objects, rather than the transformed ones. Boyapati et al. [17] implement this model, as described in Section 6.3. Our experience and that of others [8, 64, 62, 51] indicate that our model expresses many updates well. We leave to future work a detailed investigation of the semantics and expressiveness of both models.

## 3.5 Implementation

This section describes how JVOLVE supports dynamic updating by extending common virtual machine services. JVOLVE is built on JikesRVM, a high-performance Java-in-Java Research VM [2, 83]. JVOLVE integrates and extends JikesRVM's dynamic classloader, JIT compiler, thread scheduler, copying garbage collector (GC), and support for return barriers and on-stack replacement to implement dynamic updating.

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

### 3.5.1 Preparing the update

To determine the changed classes and methods for a given release, we wrote Update Preparation Tool (UPT). UPT is built using jclasslib, a Java bytecode library [31]. UPT examines differences between the old and new classes provided by the user, and groups them into the following categories

described in Section 3.2.

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

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

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

UPT generates default object and class transformer functions for all class updates, which the programmer may optionally modify. After compiling the transformers with our custom JastAdd compiler (described in Section 3.4.1), the user initiates the update by signaling the JVOLVE VM and providing the new version of the application, the update specification file, and the transformers class file.

### 3.5.2 DSU safe points

JVOLVE enforces various update safety properties by restricting at what points in the program an update can happen, called *DSU safe points*. DSU safe points occur at *VM safe points* but further restrict the methods on the threads' stacks. These restrictions provide sensible update semantics: no code from the new version executes before the update completes, and no code from the old version executes afterward. These restrictions also ensure that the update is type-safe, that the compiled version of methods are consistent with

the update, and that the update respects program semantics. As mentioned in Section 3.1 and above, we divide restricted methods into three categories: (1) methods whose bytecode has changed; (2) methods whose bytecode has not changed but that access an updated class; and (3) methods the user blacklists.

We next discuss why these restrictions improve the safety and semantics of updates, and then describe the actions JVOLVE takes to reach a DSU safe point.

**Semantics of DSU safe points.** To understand why category (1) methods are restricted, consider the update from Figure 3.3. Assume the thread is stopped at the beginning of the `ConfigurationManager.loadUser` method. If the update takes effect at this point, the new implementation of `User.setForwardedAddresses` will take an object of type `EmailAddress[]` as its argument. However, if the old version of `loadUser` were to resume, it would still call `setForwardedAddresses` with an array of `String`s, resulting in a type error.

Preventing an update until changed methods are no longer on the stack ensures type safety because when the new version of the program resumes, it will be self consistent. If a programmer changes the type signature of a method `m`, for the program to compile properly, the programmer must also change any methods that call `m`. In our example, the fact that `setForwardedAddresses` changed type necessitated changing the function `loadUser` to call it with the new type. With this safety condition, there is no possibility that the signature of method `m` could change and some old caller could call it—the update must also include all updated callers of `m`. These cases must be considered by dynamic updating systems. Our choice of restricted methods is similar to other DSU systems [72, 53, 3, 29, 81, 56, 21, 75].

**Ensuring type-safety** JVOLVE does not allow methods with changed bytecodes to be active on stack when performing the update. This restriction by itself ensures type-safety of JVOLVE's update process. Let us consider the set of all method bodies in the old and new versions of the application. Some methods exist in the old version and not the new, either because these these methods are modified, or are completely removed in the new version. Similarly, some methods exist in the new version but not in the old one. Several method bodies (bytecodes) are common to both the old and new versions. From standard Java semantics, the old and the new versions are individually type-safe. JVOLVE requires that only methods common to both versions be active on the stack at the time of the update. This restriction together with the fact that we transform all heap objects to conform to their new type signature, ensures that the application is type-safe after the update.

**Ensuring correct compiled code** Category (2) methods stem from indirect updates as mentioned in Section 3.3. Suppose some method `getStatus` calls method `getForwardedAddresses` from our example, but `getStatus` source code and bytecode has not changed from versions 1.3.1 to 1.3.2. Nevertheless, `getStatus`'s *machine code*, produced by the JIT compiler, may need to be recompiled. For example, if the new compiled version of `getForwardedAddresses` is at a different offset than before, then the VM must recompile `getStatus` to correctly refer to the new offset. An update may also change field offsets in modified classes, which requires recompiling any class that accesses them as well. Ginseng [64] and POLUS [21], two DSU systems for C, likewise consider functions as changed if their source code is the same but they access data types whose (compiled) representation is different. Note that a VM would not need

to restrict category (2) methods if it used an interpreter that looked up offsets at each access.

Note that when the VM JIT compiler uses inlining, we may need to increase the number of restricted methods to include those into which the compiler inlined restricted methods. In particular, if a category (1), (2), or (3) method `m` is inlined into method `n`, we also restrict `n` (and recompile it lazily after the update) to prevent the old `m` from running after the update. JikesRVM initially compiles a method with its *base*-compiler, which generates machine code but does not apply sophisticated optimizations. Based on run-time profiling information, the VM may recompile the same method later using its optimizing compiler, which performs standard optimizations, including inlining. It performs inlining of small, frequently used methods; cost-based inlining for larger methods; and may inline multiple levels down a hot call chain [6]. As a consequence, JVOLVE restricts inlined callers of restricted methods.

**Ensuring program specific semantics** Even if a method has not changed, a user may need to manually blacklist it. For example, suppose a method `handle` calls methods `process` and `cleanup`, and the method `cleanup` initializes a field that it uses. Now suppose we update this program to move the initialization statement into `process`, because `process` needs to use the field as well. In both versions, the field is properly initialized when the program runs from scratch. However, suppose that JVOLVE applies the update and the thread running `handle` yields in between the calls to `process` and `cleanup`. In this case, `handle`'s bytecode has not been changed, so we could go ahead with the update. But if we did, then the program would have called the old `process` method, which did not perform any initialization, and then would

call the new `cleanup` method, which performs no initialization either, since the new version `process` does it, leading to incorrect semantics. To avoid such *version consistency* problems [63] the programmer should include `handle` in the restricted set. DSU testing can also help produce such a list [39]. Our benchmarks discussed in Chapter 5 did not require manual restrictions, but a DSU system must support it to provide correctness in many cases.

While these restrictions informally assure the safety of updates, more work is needed to formally define update semantics and guarantee safety, as mentioned in Section 2.3.

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

If any thread is running a restricted method, JVOLVE defers the update and installs a *return barrier* [90] on the topmost restricted method of each thread. A generic return barrier replaces the specified method return branch back to the next instruction in the calling method with a branch instead to *bridge code*, which performs some special action and then executes the return branch. We added this generic return barrier functionality to JikesRVM. This

technology is standard in other VMs. Our bridge code restarts the update process. When a restricted method returns, the thread will block and JVOLVE will restart the update process, which will either reach a DSU safe point, or the VM will insert more return barriers. If JVOLVE does not reach a safe point within 15 seconds, it aborts the update (the length of the timeout is arbitrary, and can be configured by the user). However, with some additional care and stack state transformation, we proceed with some updates despite category (2) methods active on stack, as described next.

### 3.5.3 On-Stack Replacement to lift category (2) restrictions

JVOLVE reduces the number of restricted methods in category (2) by leveraging VM support for On-Stack Replacement (OSR) [19, 43]. State-of-the-art VMs use *adaptive* strategies to selectively compile and recompile methods at increasing levels of optimization as they get invoked more number of times, i.e. become *hot*. Usually, after recompilation, the next method invocation runs the optimized version. However, some methods are long-running and VMs need a mechanism to transition from an actively running compiled version of a method to a more optimized version. JikesRVM normally uses OSR to replace a *base*-compiled version of an active method with an optimized version. We observe that for category (2) restricted methods, the situation is much the same. An unchanged, on-stack method requires recompilation, in our case to fix any changed offsets. If the stack only contains unchanged and category (2) methods, JVOLVE first performs OSR on the category (2) methods, and then starts the update. JVOLVE currently supports OSR only for *base*-compiled category (2) methods. We leave engineering JVOLVE to support OSR for optimized methods to future work.

```
1 class C {
2   static int sum(int c) {
3     int y = 0;
4     for (int i = 0; i < c; i++) {
5       y += i;
6     }
7     return y;
8   }
9 }
```

(a) A simple example showing method `sum`

```
1  0 iconst_0
2  1 istore_1
3  2 iconst_0
4  3 istore_2
5  4 goto 14
6  7 iload_1
7  8 iload_2
8  9 iadd
9  10 istore_1
10 11 iinc 2 1
11 14 iload_2
12 15 iload_0
13 16 if_icmplt 7
14 19 iload_1
15 20 ireturn
```

(b) bytecode for `sum`

Running thread: MainThread
Frame Pointer: 0xSomeAddress
Program Counter: 16
Local variables:
    L0 (c) = 100;
    L1 (y) = 1225;
    L2 (i) = 50;
Stack expressions:
    S0 = 50;
    S1 = 100;

(c) State extracted from the stack frame

```
1  ldc 100
2  istore_0
3  ldc 1225
4  istore_1
5  ldc 50
6  istore_2
7  ldc 50
8  ldc 100
9  goto 16
10 0 iconst_0
11 ...
12 16 if_icmplt 7
13 ...
14 20 ireturn
```

(d) Version of `sum` with specialized prologue

Figure 3.7: Example of JikesRVM's OSR mechanism [35]

JikesRVM has an excellent OSR functionality [35] that is simple and mostly compiler independent, expect for extracting stack state. The OSR mechanism does not depend on the compilers used to compile the two different versions of methods. OSR in JikesRVM takes effect when the thread running the to-be-recompiled method reaches a yield point. After reaching the yieldpoint, JikesRVM recompiles the topmost method on the thread's stack and modifies the thread's PC to switch to the newly recompiled method body. The key challenge in making OSR work is to correctly transition the PC to the new compiled code. The VM must construct a new stack frame as expected by the new code and find the PC value in the new code that corresponds to the old one. JikesRVM first examines the currently active stack frame and extracts the values of local variable. It then generates a special prologue, in bytecode, that will set local variables to their correct values. The last bytecode instruction of this special prologue jumps to the bytecode at which to resume execution. JikesRVM makes this transition compiler-independent by expressing it in bytecode. The optimizing compiler can compile the method with this special prologue. Jumping to the start of the method will allow the VM to set up the stack frame as expected by the new version and resume execution at the right location. Figure 3.7 shows an example of JikesRVM's OSR transition mechanism. This example is taken from Fink et al. [35]. Figure 3.7 shows the specialized prologue that sets up the stack frame.

We extend JikesRVM's OSR facilities to support multiple stack activation records, and multiple stack frames on the same stack. This later addition makes it more likely to reach a DSU safe point when more than one category (2) method precedes a changed method on the stack. Given this support, Jvolve ignores *base*-compiled category (2) methods when testing for a safe

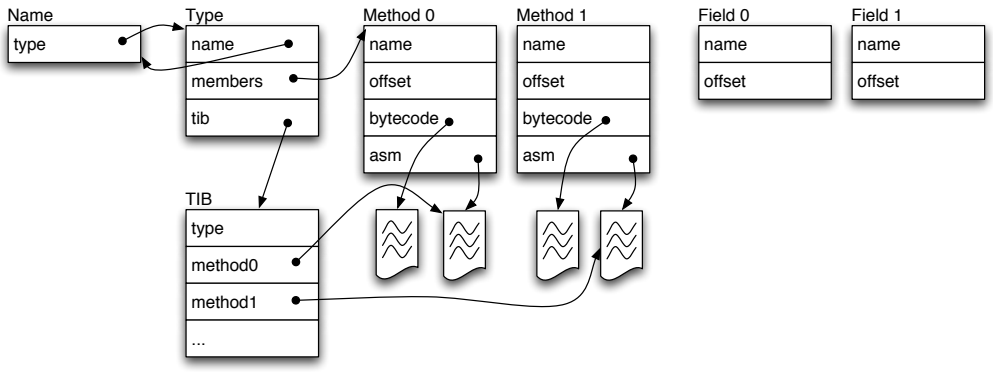



Figure 3.8: JikesRVM meta-data schema for each class

point. If any *base*-compiled category (2) methods are on stack at an otherwise DSU safe point, JVOLVE uses OSR to replace them. Once JVOLVE reaches a DSU safe point, it next installs the modified classes.

### 3.5.4 Installing modified classes

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

JikesRVM represents classes with several internal data structures. Figure 3.8 shows the information JikesRVM associates with a class. Each class has an `RVMClass` meta-object that describes the class. It points to other meta-objects that describe the class's method and field types and offsets in an object instance. The compiler and garbage collector query this metadata. The compiler hard codes these offsets in generated machine code when accessing fields and when calling methods. These offsets are statically known when a class is loaded. JikesRVM and other VMs lay out fields and methods in Virtual Method Tables (VMTs) in such a way that a given field or method has the

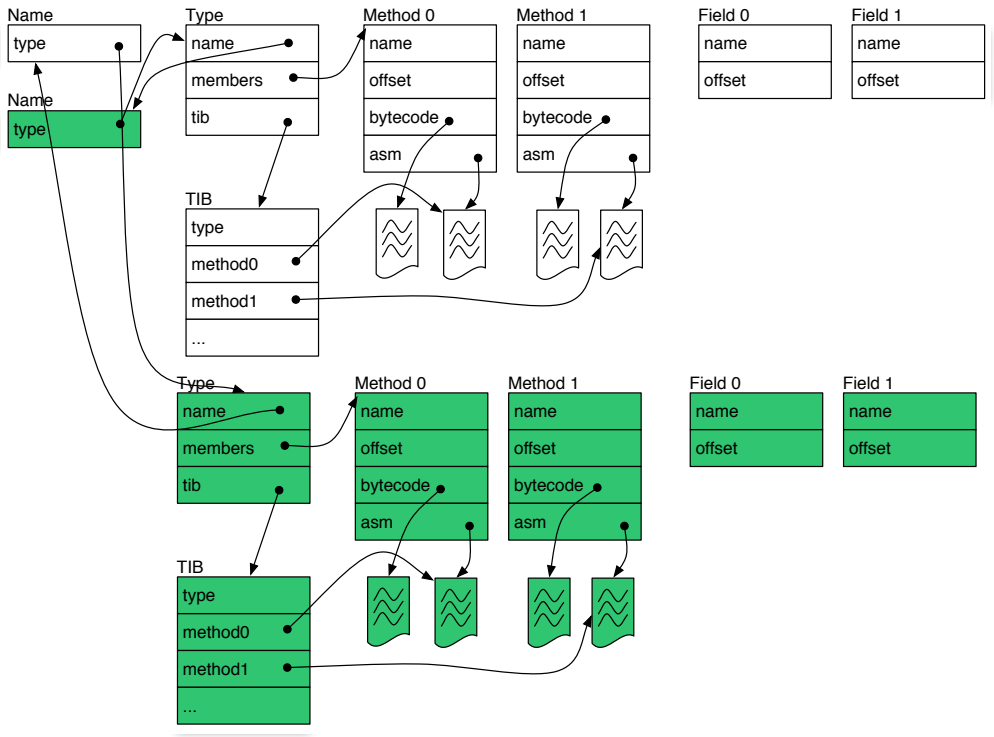



Figure 3.9: JikesRVM meta-data showing the old class name pointing to a newer class after an update

same offset or slot in all subclasses. As explained in Section 3.3, in VMTs, subclass methods share the same slot as superclass methods they override. This sharing eases *dynamic dispatch* — which method is called is decided at runtime based on the object in hand. When a program invokes a method on an object, the generated code indexes the object's VMT at the correct offset and jumps to the machine code. JikesRVM uses an array called the Type Information Block (TIB) for its VMT. The first entry of the array points to the `RVMClass` meta-object for that class. The garbage collector uses this entry to look up the type of a particular object. The rest of the entries in the array serve as VMT slots.

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

For a class with changed signature, the class's number, type, and order of fields or methods may have changed, which in turn impacts the class's metadata, including its TIB. JVOLVE modifies existing class metadata as follows. First, it changes the old class's metadata to use a modified class name, e.g., metadata for class `User` is renamed to `v131_User` in our example update from Figure 3.3 and 3.6. At this point, it is as if `class User` never existed. JVOLVE reads in `User`'s class file as it normally does and sets up metadata for the new version and updates its data structures to indicate that the newly-loaded class is now the up-to-date version. Note that all TIB entries for the newly-installed class are invalid, so all methods in the class will be compiled on demand. JVOLVE invalidates the TIB entries and other data structures for the old class so that they can be garbage-collected.

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

are relatively rare events, any added overhead due to recompilation will be short-lived.

As mentioned in Section 3.5.3, JVOLVE uses on-stack replacement to update active category (2) methods. The VM triggers on-stack replacement after the application resumes execution when callees of category (2) methods return. In order to resume execution, JVOLVE must update data in the heap. JVOLVE initiates a full-heap garbage collection and transforms objects of updated classes, which we get to next.

### 3.5.5 Applying Transformers

We modify the JikesRVM semi-space copying collector [14, 22] to update changed objects as part of a collection. The collector traverses the heap, copies reachable objects, and creates additional empty copies for all updated live objects. After collection, JVOLVE walks through these updated objects, runs their object transformers, and sets their fields appropriately. We first examine JikesRVM's semi-space copying collector and the tricolor abstraction [26, 46] it maintains, and then present JVOLVE's modifications.

In the tricolor abstraction, the GC maintains colors for each object as it performs a full-heap traversal to compute the transitive closure of reachable objects. Objects have one of three colors: white objects are yet to be visited; black objects have been visited and their immediate children have been visited has well; grey objects have been visited but their children might not have been visited. In this abstraction, collectors maintain the invariant that no black object can point to a white object. At the end of the traversal, there are no grey objects. Objects that are unreachable are white and are freed by the collector. Reachable objects are black and survive the collection.

```
1  function GC(roots):
2     Q := empty
3     for r in roots:
4         r.color = GREY
5         Q.add(r)
6     while Q not empty:
7         object = Q.pop()
8         scanObject(object)
9
10 // precondition: object is GREY
11 function scanObject(object):
12     for field in object:
13         object.field = moveObject(object.field)
14     object.color = BLACK
15
16 // precondition: object is either a forwarding
17 // pointer or a WHITE object. Either way,
18 // object resides in from space.
19 function moveObject(object):
20     if object is real:
21         copy = copyObject(object)
22         copy.color = GREY
23         Q.add(copy)
24         object.FP = copy
25         return copy
26     if object is a forwarding pointer:
27         return object.FP
28
29 // precondition: object is WHITE
30 function copyObject(object):
31     copy = malloc() # in to space
32     memcopy(copy, object)
33     return copy
```

Figure 3.10: Semi-space copying collector pseudo-code

The semi-space collector maintains the tricolor abstraction as follows. The heap is divided into two spaces — the currently active heap with all objects, called the *from-space*, and an empty heap called *to-space*. During the traversal, the collector copies reachable objects from from-space to to-space and leaves unreachable ones in from-space. At the end of collection, the entire from-space is reclaimed. Initially, all objects in from-space are colored white. The traversal begins at *roots*. The roots include statics, stack-allocated local variables, and references in registers. The compiler generates a *stack map* at every VM safe point (a superset of DSU safe points). The stack map enumerates all register and local variables on the stack that reference heap objects. The roots are not objects themselves, but are references to objects. The roots reside outside the heap and are not copied. For the purposes of the tricolor abstraction, the roots are initially colored gray.

The copying collector walks through the list of grey objects and *scans* them (see line 11 in Figure 3.10). Scanning an object involves going through all reference fields of that object, moving the referred objects over to to-space, and setting the field to the address newly moved to. When moving an object, the collector leaves behind a *forwarding pointer* in its place that points to the new location. This pointer ensures that an object is moved only once and the forwarding pointer gives the to-space address of the object. Later, if the collector encounters a forwarding pointer when processing a reference, it sets the reference to the value of the forwarding pointer. Moving an object to to-space turns it grey, while scanning an object makes it black. The fields of black objects point to other objects in to-space, which are either grey or black. The fields of grey objects point to white objects (or forwarding pointers), which are in from-space. The traversal ends when there are no more grey objects.

```
1  function moveObject(object):
2      if object is real:
3          copy = copyObject(object)
4          copy.color = GREY
5          Q.add(copy)
6          if object.type.newVersion != null:
7             newCopy = malloc()
8             add newCopy to update log
9             newCopy.type = object.type.newVersion
10            // newCopy is currently empty
11            object.FP = newCopy
12            return newCopy
13         else:
14            object.FP = copy
15            return copy
16     if object is a forwarding pointer:
17         return object.FP
```

Figure 3.11: JVOLVE's modification to JikesRVM's semi-space copying collector

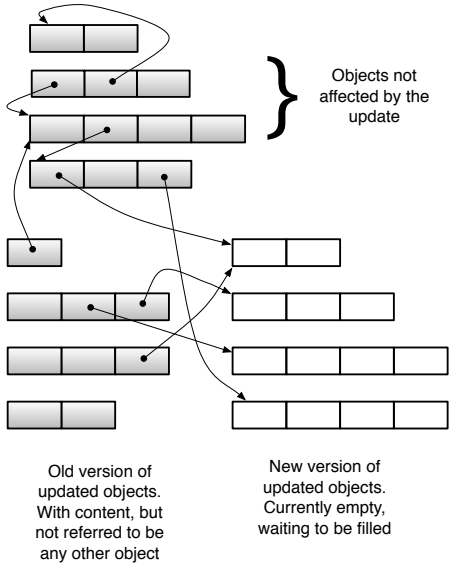



Figure 3.12: A view of the *to* semi-space immediately after garbage collection

JVOLVE's modified collector works in much the same way as the regular semi-space collector just described. As shown in Figure 3.11, it differs in how it handles objects whose class signature has changed. In this case, it allocates a copy of the old object *and* an *empty* new object according to the new class definition, which may have a different size compared to the old one. The collector initializes the new object to point to the Type Information Block (TIB) of the new type, and installs the forwarding pointer in from-space to this new version. Next, the collector stores a pair of pointers in its *update log*, one to the copy of the old object and one to the new object. The collector continues scanning the old copy. The key fact to note is that because forwarding pointers point to the empty new copy of the object, references to updated types all point to these empty objects. Figure 3.12 illustrates this situation. The old version of updated objects are filled with content while references point to empty new versions of these objects.

After the collection completes, JVOLVE adds another phase. It first executes transformers for all classes and then for all objects. JVOLVE goes through the update log and invokes the object transformer, passing the old and new object pair as arguments. Recall from Section 3.4.1 that the `jvolveObject` functions receive an object of the old version and an object of the new version as parameters. These parameters come from the update log, with the old object filled with content and the new object empty waiting to be filled by the transformer function. Once JVOLVE processes all object pairs, the log is deleted, making the duplicate old versions unreachable. As Figure 3.12 shows, old version objects are unreachable as no references point to them, and the next garbage collection cycle will free them. While we did not do so in our implementation, we could put these objects in a special space and reclaim the

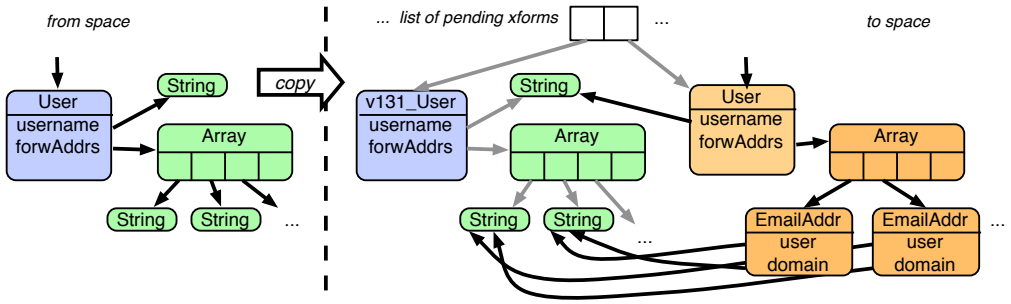



Figure 3.13: Running object transformers following garbage collection

space immediately after the update. We now look at two simple updates to illustrate how JVOLVE applies object transformers.

**Example 1.** Figure 3.13 illustrates a part of the heap at the end of the GC phase while applying the update from Figure 3.6 (forwarding pointers are not shown). On the left is a depiction of part of the heap prior to the update. It shows a `User` object whose fields point to various other elided objects. After the copying phase, all of the old reachable objects are duplicated in to-space. The transformation log points to the new version of `User` (which is initially empty) and the duplicate of the old version, both of which are in to-space. The transformer function can safely copy fields of the `from` object. The figure shows that after running the transformer function, the new version's `username` field points to the same object as before, and the new version's `forwardAddresses` field points to a new array of new `EmailAddress` objects. The `EmailAddress` constructor called from within the transformer function initialized these objects by referring to the old e-mail `String` values and assigning fields to point to substrings of the given `String`.

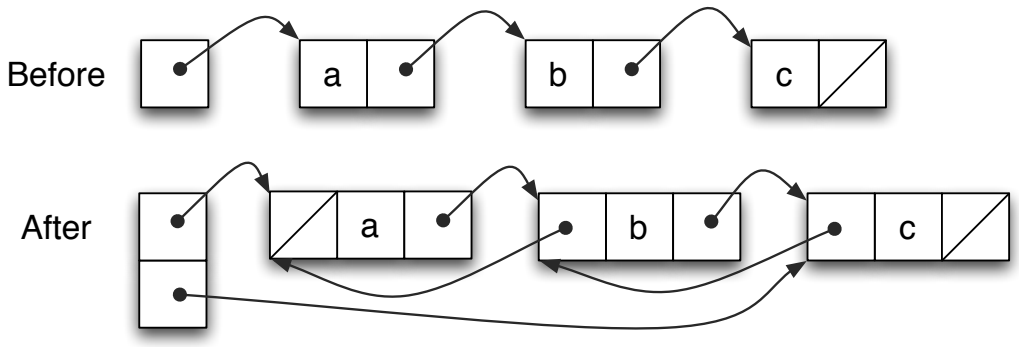



Figure 3.14: A look at the structure of an example linked list before and after the update

**Example 2.** We explore our object transformation model by looking at another example. Figure 3.14 shows an example linked list before and after the update. In this example, a singly-linked list in the old version becomes a doubly-linked list in the new. Figure 3.15 shows the code for the old and new versions, and stub classes and default transformers generated by the Update Preparation Tool (UPT). The update involves two classes with signature changes: `LinkedList` which adds a `tail` field, and `Node` which adds a `prev` field. The object transformers must set these additional fields appropriately to create a doubly-linked list out of a singly-linked one. The `jvolveObject` functions generated by default will correctly copy the `data` and `next` fields for each node in the linked-list and correctly set `head` to point to the start of the list. We modify `Node`'s transformer to set each node's `next` node's `prev` field to point back to itself, i.e., `from.next.prev = from`. Setting the `tail` of the list to point to the last node is trickier. We need some way to traverse the list to get to the end. At the time we traverse the list, not all nodes might have been transformed. In order to support this update, JVOLVE provides a way

```
1 public class LinkedList {
2   class Node {
3     Node next;
4     int data;
5   }
6   private Node head;
7 }
```

Old version code

```
1 public class LinkedList {
2   class Node {
3     Node prev;
4     Node next;
5     int data;
6   }
7   private Node head;
8   private Node tail;
9 }
```

New version code

```
1 public class r0_LinkedList {
2   public LinkedList.Node head;
3   public class Node {
4     public LinkedList.Node next;
5     public int data;
6   }
7 }
```

Stub classes for the old version

```
1  public class JvolveTransformers {
2    public static void jvolveObject(
3        LinkedList.Node to, r0_LinkedList.Node from) {
4      to.prev = null; // no such field in from
5      to.next = from.next;
6      to.data = from.data;
7    }
8    public static void jvolveClass(LinkedList.Node unused) {}
9    public static void jvolveObject(
10       LinkedList to, r0_LinkedList from) {
11     to.head = from.head;
12     to.tail = null; // no such field in from
13   }
14   public static void jvolveClass(LinkedList unused) {}
15 }
```

Default UPT-generated transformer

Figure 3.15: An update that goes from a singly-linked to a doubly-linked list

for the programmer to request that an object be transformed on-demand as the programmer traverses the list to retrieve the `tail` element of the list. Section 4.1 explores this example in more detail and presents alternative object transformation models.

## 3.6 Conclusion

We presented a detailed view of the JVOLVE Virtual Machine. In the next chapter we look at JVOLVE's state transformer model and a novel way to automatically generate state transformers.

# Chapter 4

# State Transformers: Models and Automation

This chapter discusses JVOLVE's state transformation model in more detail. We present JVOLVE's per-type object transformation model; our use of state transformers to repair application state for specific bugfixes; and a methodology for automating state transformer generation to ease programmer burden.

## 4.1 Object Transformation Model

This section explains JVOLVE's object transformation model and compares it to other alternatives by revisiting the singly-linked to doubly-linked example introduced in Section 3.5.5.

Any dynamic updating system must convert old process state at the time of the update to the one expected by the new version at the point it resumes execution. The system must convert state maintained in global static data, method stack variables, and heap data as required for the new version. Even ignoring semantics of the application, at the very least, the system needs to represent data correctly in the form expected by the new version.

In an object-oriented programming language like Java, data is primarily stored as objects in the heap. An object is statically-typed[1], and is an instance

[1]In contrast, dynamically-typed languages such as Python may change fields and methods

of a particular `class`. A dynamic updating system for Java should convert heap objects to the new version in a class-specific manner, leading to class or type-specific object transformation functions [64, 75]. Such a type-specific transformation function specifies how to obtain an object of the new version's type given an object of the old one. As explained in Section 2.2, there exist two main design choices on when to execute these transformation functions. One design is to lazily transform objects. In a lazy approach, objects are transformed when they are first accessed by the application after the update. A lazy approach requires that application code be instrumented to check the state of an object and transform it if necessary. The other design alternative is to eagerly identify and transform all objects at once. For JVOLVE, we followed the eager approach because we did not want to incur the steady-state performance overhead that comes with instrumenting code and because we could efficiently transform the entire heap by piggybacking on garbage collection. We first discuss design choices with the eager transformation model and then present the lazy transformation model and how one might implement it in JVOLVE.

### 4.1.1 Eager transformation models

For the rest of this discussion, we assume that the interface to specify object transformers is the `jvolveObject` function. This function accepts two parameters: an object corresponding to the old version called `from` and an object for the new version called `to`. The programmer specifies how to initialize the `to` object with data from the `from` object. The interesting question in this and other models is, what the types of the `from` and `to` parameters should be.

---

of an object over time, and objects of the same class need not be consistent.

We require access to the both the old and new version's definition, yet we must not break Java's type system in order to utilize the strong type-safety guarantees provided by the language. The definitions of the `from` and `to` objects depend on what view of the heap the developer uses during transformation time and vice versa. In one, the programmer has access to the old version's code and any pointer dereference will return an object that is consistent with the old version. We call this model the "old world" model. In the "new world" model, the programmer has access to the new version's code and any pointer dereference will return an object of the new version. The old world model has some natural advantages over the new world model. Since the old world model presents the well-formed old heap from before the update, any code or data access on this heap will be safe. The new world model exposes the transformation function to a heap that is yet to be populated. Dereferencing pointers to objects whose contents are yet filled can result in incorrect data values or null pointer exceptions. Before transformer code can access an object, the programmer or the compiler has to ensure that its contents are correctly filled in. In JVOLVE we implemented the new world model because it integrated naturally with our VM implementation.

Figure 4.1 shows an example of an update with two classes `X` and `Y`. Class `X` has a field `y` pointing to an object of type `Y`. The update adds an additional field each to classes `X` and `Y`. Figure 4.2 shows the object transformers for classes `X` and `Y` in the old and new world models. In the old world model, the `from` parameters are of the old types `X` and `Y`. The types of the `to` parameters are artificially constructed stub classes that correspond to the definition of the new versions of `X` and `Y`. Note that fields of the `new_X` and `new_Y` classes, specifically, `new_X.y` refers to an old version object, thus maintaining the old

```
1 public class X {
2   private Y y;
3 }
4 public class Y {
5   private int i;
6 }
```

(a) Old version

```
1 public class X {
2   private String s;
3   private Y y;
4 }
5 public class Y {
6   private int i, j;
7 }
```

(b) New version

Figure 4.1: Example of a simple update where the field of an updated class refers to another updated class

world invariant. Conversely, in the new world model the `to` parameters are of the new version's types while the `from` parameters are stub classes that correspond to the definition of the old version. The field `old_X.y` refers to a new version object.

In the new world model, the stub classes `old_X` and `old_Y` have the same fields (both types and names) as defined in the old version and provide access to the fields of old-version objects in a type-safe manner. These stub classes are used by the Java-to-bytecode compiler to compile object transformer functions. However, as explained in Section 3.5.4, JVOLVE does not load these classes. It merely renames the old version of the class that is already loaded by the application to the names of these stubs. This renaming eliminates the naming conflict between old and new class names. In the old world model, the runtime replaces stub classes `new_X` and `new_Y` used while transforming objects with the new versions of `X` and `Y` before the application resumes execution.

The simplest meaningful transformer functions possible are those that copy fields from the old-version object to the new one. In both models, the

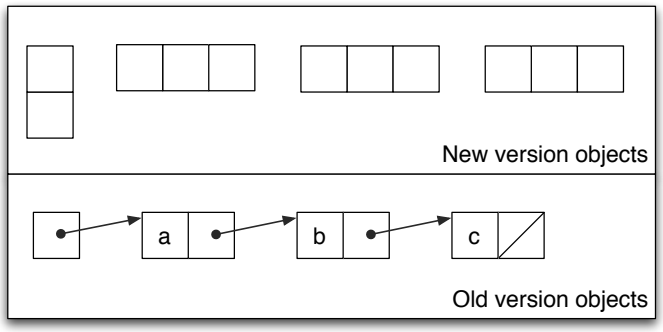
```
1  // new_X.y refers to the old version
2  // The new X and Y do not exist, yet
3  class new_X { String s; Y y; }
4  class new_Y { int i, j; }
5
6  class JvolveTransformers {
7    public static void
8        jvolveObject(new_X to, X from) {
9      to.s = null; to.y = from.y;
10   }
11   public static void
12       jvolveObject(new_Y to, Y from) {
13     to.i = from.i; to.j = 0;
14   }
15 }
```

(a) Old World Model: Stub classes and transformers

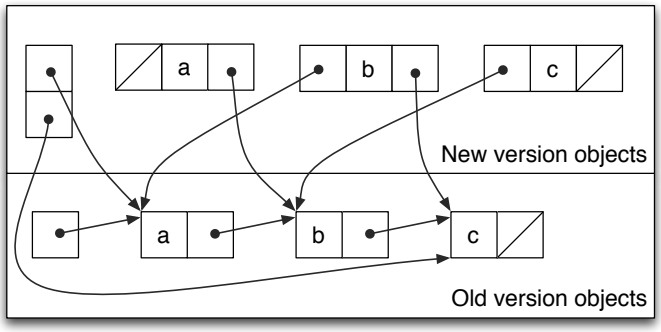
```
1  // old_X.Y refers to the new version
2  // The old X and Y do not exist now
3  class old_X { public Y y; }
4  class old_Y { public int i; }
5
6  class JvolveTransformers {
7    public static void
8        jvolveObject(X to, old_X from) {
9      to.s = null; to.y = from.y;
10   }
11   public static void
12       jvolveObject(Y to, old_Y from) {
13     to.i = from.i; to.j = 0;
14   }
15 }
```

(b) New World Model: Stub classes and transformers

Figure 4.2: Stub classes and transformers for the update in Figure 4.1




(a) View of the heap before running object transformers




(b) View of the heap after running object transformers

Figure 4.3: Old World Model: A view of the linked lists, before and after running transformation functions

programmer can copy both primitive and reference fields. In the old world view, reference fields point to old version objects, while in the new world view they point to new version objects. In the old world view, the dynamic updating system changes every reference to an old version object to its corresponding new one, before the application resumes execution.

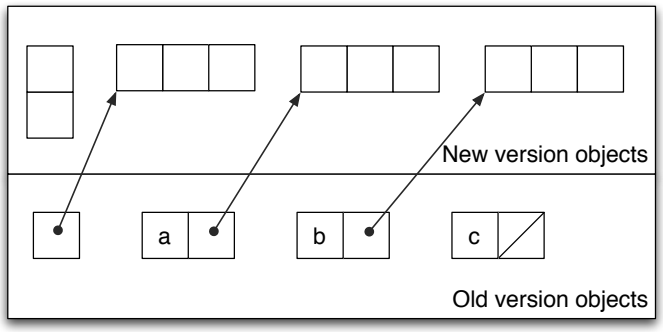
**Updating from a singly-linked list to a doubly-linked list** We revisit the singly-linked to doubly-linked example from Figure 3.15 to explain trans-

```
1  public static void jvolveObject(
2              r1_LinkedList to, LinkedList from) {
3    to.head = from.head;
4    Node prev = null;
5    Node current = from.head;
6    while (current != null) {
7      prev = current;
8      current = current.next;
9    }
10   to.tail = prev;
11 }
```

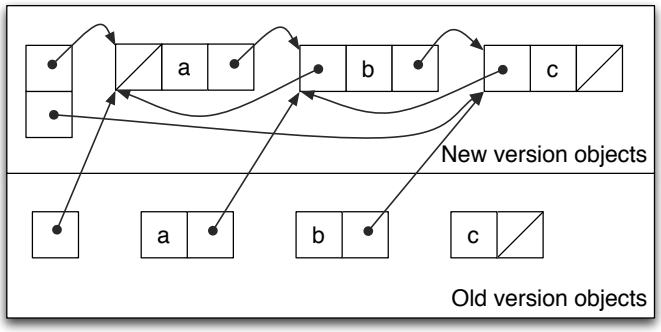
Figure 4.4: Old World Model: Object Transformers to convert a singly-linked list into a doubly-linked list

formers in the old world and new world models. As mentioned in Section 3.5.5, it is straightforward to set the `head` pointer of the doubly-linked list. Setting the `tail` pointer to the last node involves traversing the linked list. In the old world model, the list is well-formed and the transformer code obtains the last node with a simple traversal of the list. The new world model requires special handling since it involves traversing a list whose nodes are yet to be populated.

We first present the old world model. Figure 4.3 shows the heap before and after running object transformers. The heap is logically divided into objects of the old version and those of the new version. All pointers refer to objects of the old version. The object transformer functions given in Figure 4.4 populate the empty new version objects. The code to set the `tail` field of the list involves a simple list traversal. After transforming all objects the dynamic updating system scans the heap and converts all pointers to refer to the new version objects and the application can resume execution. After this phase, the heap is identical to the new world heap after transformation shown in




(a) View of the heap before running object transformers




(b) View of the heap after running object transformers

Figure 4.5: New World Model: A view of the linked lists, before and after running transformation functions

Figure 4.5 (b).

Converting the singly-linked list to a doubly-linked list in the new world model is a bit more involved. Figure 4.5 shows a view of the heap before and after running object transformers in the new world model. All pointers refer to objects of the new version. The heap is ready to be used by the application immediately after running the transformers.

In order to set the `tail` of the list to point to the last node, the trans-

```
1  public static void jvolveObject(
2              LinkedList to, r0_LinkedList from) {
3    to.head = from.head;
4    Node prev = null;
5    Node current = from.head;
6    while (current != null) {
7      prev = current;
8      if (! VM.is_transformed(current)) {
9        r0_Node current_old = VM.old_version_object(current);
10       current = current_old.next;
11     } else {
12       current = current.next;
13     }
14   }
15   to.tail = prev;
16 }
```

(a) Explicitly traversing old and new version objects

```
1  public static void ensure_transformed(Object o) {
2    if (! is_transformed(o)) {
3      // The jvolveObject method corresponding to the
4      // object's type is invoked using reflection.
5      jvolveObject(o, old_version_object(o));
6    }
7  }
8  public static void jvolveObject(
9              LinkedList to, r0_LinkedList from) {
10   to.head = from.head;
11   Node prev = null;
12   Node current = from.head;
13   while (current != null) {
14     prev = current;
15     VM.ensure_transformed(current);
16     current = current.next;
17   }
18   to.tail = prev;
19 }
```

(b) Explicitly transforming new version objects

Figure 4.6: New World Model: Object Transformers to convert a singly-linked list into a doubly-linked list

former code needs to traverse the list to get to the end. At the time we traverse the list, not all nodes might have been transformed, because we cannot always control the order in which objects are transformed. There are two ways to support this update. One is to allow the programmer to access old version objects from the new world view by making a request to the runtime. The other is to provide a way for the programmer to request that an object be transformed on-demand as the programmer traverses the list to retrieve the `tail` element of the list.

**1. Explicitly traversing old and new-version objects.** We could traverse the list using both objects of the old version and objects of the new version. It is important to note that, the system we currently have ensures type-safety. All reference fields, in the old and new objects, point to objects of the right type. The issue though is that fields in new objects might be `null` because these objects have not been transformed yet. The VM could use metadata in object headers to provide an Application Programming Interface (API) that allows a programmer to get the old-version object corresponding to a new-version object and vice versa. The programmer should carefully traverse the list, getting the old-version object of a new-version object, and following the `next` field of the old-version object. Figure 4.6 (a) shows the object transformer in this model.

**2. Explicitly transforming new-version objects.** Another approach would be to traverse the linked list, while ensuring that the nodes are transformed before we dereference them. Figure 4.6 (b) shows the object transformer in this model. The inherent issue we face is that we want to enforce an

order in which objects are transformed. JVOLVE provides an API that allows the programmer to transform a particular object on demand. The programmer could invoke this function `ensure_transformed` for each node as they traverse the list. Note that, this VM function calls `jvolveObject`, the object transformer if an object is not yet transformed. This mechanism could potentially lead to a cyclic dependency, i.e., transforming an object might require that the same object already be transformed. Currently, we do not address this problem and leave it to the programmer to guarantee acyclicity. We could imagine a more sophisticated approach that analyzes transformation functions and automatically chooses the order in which they are invoked. Alternatively, we could automatically instrument the transformer function to make the required API calls.

### 4.1.2 Discussion

While the object transformer functions written for the old world and new world models indicate that the old world model provides a simpler logical model for object transformation, we implemented the new world model for simplicity of implementation and efficiency. The old world model requires two traversals of the heap — one pass to identify objects that need to be transformed, and another pass to scan and convert all pointers that refer to old version objects to point to new version ones. The old version model also requires two steps to load classes of the new version. A first step to load stub classes to run transformers and another step to load the real classes of the new version after the update. The new world model on the other head integrates cleanly with our VM implementation. JVOLVE renames old classes to stub classes and loads new version classes in a single step. JVOLVE's modified

garbage collector walks through the heap, identifies objects that need to be updated, creates additional copies for such objects while ensuring that all references point to these new copies. After invoking object transformers on all updated objects, the application can resume execution in the new version.

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

### 4.1.3 Lazy transformation model

We use a stop-the-world garbage collection-based approach that requires the application to pause for the duration of a full heap GC in order to perform the update. An alternative to this model is applying object transformers lazily [72, 53, 17, 64, 21]. In a lazy model, the system instruments the application to check, at each dereference, whether an accessed object is up-to-date and invoke its object transformer if necessary. The main drawback of this approach is that it adds overhead during steady state execution. However, the lazy approach is useful in application contexts that cannot tolerate the pause caused by a full heap GC.

The lazy approach can be implemented in JVOLVE by performing the

```
1  function getfield(object, field):
2      value = object.field
3      new_value = ensure_transformed(value)
4      if (value != new_value):
5          setfield(object, field, new_value)
6      return new_value
7
8  function getstatic(class, field):
9      // similar to getfield
10     // uses setstatic
11
12 function aaload(array, index):
13     // similar to getfield
14     // uses aastore
15
16 function ensure_transformed(o):
17     if o.isForwardingPointer():
18         return o.ForwardedValue
19     else if o.isUpToDate():
20         return o
21     else:
22         new_o = new Object()
23         jvolveObject(new_o, o)
24         o = (forwarding pointer to new_o)
25         return new_o
```

Figure 4.7: Lazy object transformation model implementation

following steps. 1) At update time, invoke object transformers on all stack local variables. 2) Maintain the invariant that any object accessed by the application is up-to-date. We can maintain this invariant by adding read-barrier code to all object dereferences. An object can be dereferenced in Java only using one of the following three opcodes: `getfield` that returns a field from an object; `getstatic` that returns the value of a global variable; and `aaload` that return an array element. The read-barrier code to check whether or not an object is transformed can use the same forwarding pointer mechanism

used by the garbage collector while scanning the heap. Figure 4.7 shows the code that implements this functionality. The first time an object is accessed after the update, it will be transformed leaving a forwarding pointer in its place. Future accesses to the object that go through the forwarding pointer return the transformed object and update the corresponding reference. The forwarding pointer will be garbage collected away when no field refers to it.

While the lazy transformation model reduces pause time, it is not as flexible as the stop-the-world eager transformation model. When using the lazy transformation model, the programmer must take special care to ensure that application correctness is not affected by when objects are transformed. For example, in the update that goes from a singly to doubly-linked list, an object's `prev` field is set by its previous object's transformer and not by itself. No general mechanism for obtaining the previous object exists without eagerly transforming the list. When the application accesses an object's `prev` field and the field is `null`, there is no way to know if it is actually `null` or whether it is yet to have its value set. Thus, lazy transformation limits the types of updates a dynamic updating system can support.

## 4.2 Repairing Application State

The state transformer model presented above makes the implicit assumption that a to-be-updated application's execution state is correct. In particular, dynamic patches that are used to initialize the new version's state by examining the current state assume that this state is well-formed. Most times, this assumption is correct. Perhaps the bug does not corrupt the heap, or has not yet been exercised, or only causes incorrect input/output processing. However, in some cases, this assumption can be faulty.

```
1  public void foo() {
2    List<String> ll = new LinkedList<String>();
3    // populate the list
4    while (...) {
5      ll.add(...);
6    }
7    if (...) {
8      // some long running loop
9      while (...) {
10       // read elements from ll
11     }
12   } else {
13     // another long running loop
14     while (...) {
15       // never use ll
16     }
17   }
18 }
```

(a) Object is reachable, but never accessed by the application

```
1  public void process(HashSet hs, Order order) {
2    hs.add(order);
3    if (...) {
4      if (order.processed) {
5        ...
6        // forgets to remove order from hs
7      }
8    } else {
9      hs.remove(order);
10   }
11 }
```

(b) Failing to remove objects from a collection

Figure 4.8: Examples of "memory leaks" in Java

Consider a memory leak. After a while, there are many reachable, but dead objects in memory that need to be freed. JVOLVE and other existing dynamic updating systems can apply a provided fix to the code to prevent further leaks, but DSU researchers have paid little attention to the problem of finding and freeing existing objects once the code is fixed. In this section, we explore source code updates that fix memory leaks in real programs, and our extending state transformers to repair application state at update time by cleaning up existing leaked objects. While this discussion is specific to one bug type, we use it to gain insight to more general mechanisms for fixing heap corruption.

### 4.2.1 Memory leaks in Java

In the context of a garbage collected language like Java, where unreachable objects are automatically reclaimed, researchers define a memory leak as follows. Any object that is still reachable from the roots of the heap (globals and locals) but will not be accessed by the application in the future is considered a memory leak. Consider the simple program in Figure 4.8 (a). The function defines a local variable that points to a linked list. After populating the list, execution reaches an if-then-else conditional. The linked list is accessed only in the true branch of the if code block. If the conditional is false, the linked list will not be accessed again in the future and can be freed. This example leak is not too problematic because it does not grow over time. More problematic are leaks that continue to grow and eventually exhaust memory.

Another commonly used definition of a memory leak is semantic and typically involves collections of objects such as an array, linked list, or hash-map. Leaky objects are those that are reachable, but is never used by the

| Application | Patch | Description of leak |
|---|---|---|
| jEdit | SVN r8329 | Application stores unnecessary information about closed buffers that do not correspond to any file in the file system. |
| | SVN r5178 | Code that splits a line into tokens continues to maintain references to that line. |
| | SVN r14027 | Similar to r5178, application but maintains a reference to an object of class TokenHandler. A TokenHandler knows how to split a string of a particular syntax (C, Java, Verilog, etc.) into tokens. |
| Eclipse IDE | Bug #115789 | Application maintains reference counts to objects in a collection. In some cases, it neglects to correctly decrement this reference count such that application logic never reclaims the object. |

Table 4.1: Memory leak fixes to real applications

application for any real purpose. Consider Figure 4.8 (b). The programmer fails to remove from a hashset an object that is not required by application in the future. Note that if the set is reachable, the object is not only reachable, but also likely to be read by the program, when the hashset library re-hashes buckets in the set. We define objects that will not used by application logic in the future as leaks. Developers do inadvertently introduce such leaks in applications and fix them in future versions [16, 47]. In this work, when we dynamically apply the source patch that fixes the leak, we also remove leaky objects as part of state transformation.

### 4.2.2 Fixing corrupt heap state for leaks

We considered a total of four leaks in two Java applications jEdit and Eclipse IDE. jEdit is a popular text editor used by programmers. jEdit provides common features such as syntax highlighting, folding, and automatic indentation, and advanced features such as plugin support, macro recording, and a built-in macro language. Eclipse IDE is a widely-used multi-language Integrated Development Environment (IDE). Eclipse is well known for its modularity and plugin architecture. While these applications are not prime candidates for dynamic updating, they are complex, easily available, and can run for a long time. They exhibit the same long-running loop structure of mission-critical always-on programs and are good candidates to demonstrate our work.

Table 4.1 shows descriptions of four leaks, three in jEdit and one in Eclipse IDE. For all leaks, we wrote object transformers that fixed the existing leak in the application during update time. JVOLVE is not yet robust enough to update the patch for Eclipse IDE. Therefore we simulated the update and the object transformer from within the application, by doing the following. We let the application initially run as normal. After some number of iterations of the outer loop, we invoked the object transformer code that would fix the leak, and in future iterations we called the updated code without the leak. JVOLVE correctly updates all three jEdit patches and runs object transformers that fix the leaks at update time. We now discuss the individual leaks and their object transformers.

**jEdit SVN r8329** Figure 4.9 shows source code patch for revision 8329 that fixes the leak and the object transformer that repairs the leak. jEdit

```
1   public class EditPane {
2     private void handleBufferUpdate(BufferUpdate msg) {
3       if(msg.getWhat() == BufferUpdate.CREATED) { ...
4       } else if(msg.getWhat() == BufferUpdate.CLOSED) { ...
5 +       Buffer b = msg.getBuffer();
6 +       if (b.isUntitled()) {
7 +         // the buffer was a new file so I do
8 +         // not need to keep its info
9 +         Map carets = (Map) getClientProperty(CARETS);
10 +        if (carets != null)
11 +          carets.remove(b.getPath());
12        }
13      } else if(msg.getWhat() == BufferUpdate.SAVED) { ...
14      } else ...
15      }
16    }
17  }
```

(a) Patch applied by SVN revision 8329.
Lines 5–11 are added in the new version

```
1  public static void jvolveObject(EditPane ep) {
2    Map<String, CaretInfo> carets =
3                     ep.getClientProperty("Buffer->Caret");
4    if (carets != null) {
5      for (String path : carets.getKeys()) {
6        Buffer b = jEdit.getBuffer(path);
7        if (b.isClosed() && b.isUnitled()) {
8          carets.remove(path);
9        }
10     }
11   }
12 }
```

(b) State transformer

Figure 4.9: jEdit leak and fix: SVN revision 8329

```
1    public class TokenMarker {
2      private TokenHandler tokenHandler;
3      public void markTokens(TokenHandler th)  {
4        this.tokenHandler = th;
5        // lots of processing
6        ...
7        ...
8  +     this.tokenHandler = null;
9      }
10   }
```

(a) Patch applied by SVN revision 14027.

```
1  public static void jvolveObject(TokenMarker tm) {
2    tm.tokenMarker = null;
3  }
```

(b) State transformer

Figure 4.10: jEdit update: SVN revision 14027

maintains the last cursor position of each file in a hashmap. This information is meaningless for an "untitled buffer," i.e., an editor buffer that does not correspond to any file in the file system. jEdit creates an untitled buffer when a user opens a new empty buffer. After such a buffer is closed, jEdit has no use for its cursor position, and this information should be removed from the hashtable. Versions prior to r8329 failed to do so, and as a result leaked memory. Figure 4.9 (b) shows the object transformer function that repairs the leak. The function walks though all keys in the `CARETS` hashtable, and removes those entries that correspond to untitled buffers.

**jEdit SVN r5178 and r14027** The leaks fixed by versions r5178 and r14027 and similar and occur in the same function, but to two different fields. We only show the code for version r14027 here. Figure 4.10 shows the source patch

```
1    class Editor { int refCount; }
2
3    class History {
4      Editor editor;
5      void dispose() {
6        this.editor = null;
7      }
8    }
9
10   public class NavigationHistory {
11     ArrayList<History> history;
12     ArrayList<Editor> editors;
13
14     void disposeEntry(History h) {
15       if (h.editor == null) return;
16       h.editor.refCount--;
17       if (h.editor.refCount == 0)
18         editors.remove(h.editor);
19       h.dispose();
20     }
21
22     void updateNavigationHistory() {
23       for (History h : history)
24         if (...) {
25           history.remove(h);
26 -         h.dispose();
27 +         disposeEntry(h);
28         }
29     }
30   }
```

Figure 4.11: Eclipse IDE memory leak patch: Bug #115789

```
1  public static void jvolveObject(NavigationHistory nh) {
2    for (Editor e : nh.editors)
3      e.refcount = 0;
4    for (History h : nh.history)
5      if (h.editor != null)
6        h.editor.refcount++;
7    for (Editor e : nh.editors)
8      if (e.refcount == 0)
9        nh.editors.remove(e);
10 }
```

Figure 4.12: State transformer that fixes Eclipse IDE memory leak bug #11-5789

and object transformer for this leak. jEdit calls function `markTokens` when it needs to split a string into tokens based on the type of the file being edited. Each file type (C, Java, Verilog, etc.) has special logic to split text in a line, embedded in an object of type `TokenHandler`. The leaky jEdit version fails to set the field `TokenMarker.tokenHandler` to `null`. The object transformer is simple, it sets the leaky field to `null`. It is safe to execute the transformer only when the `markTokens` function is not active on stack. Since the update changes `markTokens`, it will not be active at a DSU safe point.

**Eclipse IDE #115789** Figure 4.11 shows the source for a memory leak in Eclipse IDE. In the class `NavigationHistory`, the IDE maintains two lists: one of `History` objects called history, and another of `Editor` objects called editors. Each `History` object has an `editor` field. Thereby, each object in the history list point to an object in editors list. An editor object may remain in the editors list only as long as it is pointed to by some history object. To enforce this property, the program maintains a reference count field with each editor object. This field represents the number of history objects pointing

to that particular editor object. Whenever the application removes a history object, it should decrement the reference count of the editor object that the history object points to, and remove the editor object from the editors list if the count is zero. Function `disposeEntry` implements this functionality. In one instance, the programmer failed to follow this procedure, causing a memory leak. Figure 4.12 shows the object transformer that frees the leak at update time. The transformer walks though all history objects, recomputes reference counts for editor objects, and frees editor objects with a reference count of zero.

## 4.3 Automating State Transformer Generation

Thus far, we have relied on the programmer to provide heap transformer functions. In this section, we explore techniques to automate generation of the transformer based on the patch to ease programmer burden.

This section presents a technique that starts from a patch and automatically produces state transformers that repair application state. Our methodology takes into account the code fix for a particular heap-corrupting bug, and uses a combination of dynamic and static analysis to discover predicates on heap objects. We employ the garbage collector to invoke object transformers on relevant heap objects satisfying the discovered heap predicates. While our focus is on memory leaks, our technique is general and we have applied it to other sorts of conditional heap updates as well.

Our technique works as follows. Suppose we have a Java program in which field `f` of class `C` is the source of a memory leak, and the fix is to insert a line `x.f = null;` in some method `foo`. Figure 4.13 shows the patch for this method. The new version sets field `f` to `null` causing the object pointed to by `f`

```
1    public void foo() {
2      C x;
3      ...
4      if (...) {
5        ...
6      } else {
7        ...
8  +     x.f = null;
9      }
10   }
```

Figure 4.13: Example of a simple patch that fixes a memory leak

to be garbage collected and freed. Given this fix, we would like to dynamically update the program to correct the code of `foo`. The patch prevents further leaks, and our object transformer needs to modify the existing leak to free already-leaked objects. We use a combination of dynamic and static analysis to automatically generate the object transformer function.

The object transformer function must find existing objects `x` of class `C` (or a subclass of `C`) and assign `null` to `x.f`. But we must be careful to only null out `x.f` if that object is actually leaked; if we null out `x.f` while the program is still using it, then the update will introduce incorrect behavior or a crash. We thus need to distinguish leaked objects from in-use ones. In Figure 4.13, the fixed version of `foo` introduces the statement `x.f = null;` on line 8. The *leaky objects* consist of all objects $o$ that have been bound to `x` during execution (of the old program) that have reached this line. If the fix had been in place, `x.f = null` would have been executed, and thus the `f` field of these objects would have been set to null. All other objects of class `C` (or subtypes of `C`) that have *not* been bound to `x` at line 8 are not leaking, and so we should leave their `f` field alone. We call these the *non-leaky objects*.

The problem with literally implementing this idea is that the bug is discovered, and the fix discerned, only after the program is deployed. At this point, the patch has no information about which objects reached line 8 and which did not. To remedy this problem, as part of update preparation, we attempt to generate a *heap predicate* offline, that unambiguously distinguishes the leaky and non-leaky objects. Then we generate a dynamic patch to correct the heap:

```
1 for all objects x of class D < C:
2    if heap_pred(x):
3        x.f = null
```

The code to fix a particular object's state (line 3) comes from the code patch and is applied to all objects of type D satisfying the heap predicate. We execute this code at update time, together with other object and class transformers. We use a modified version of the JVOLVE garbage collector to traverse the heap to find the matching objects and null their fields.

We generate the heap predicate by performing a dynamic analysis. We run the old program and *mark* all objects of class C that reach the said line. In our current implementation, we modify the concerned class and explicitly add a boolean field named `__marked`. To mark an object, we set this field to `true`. In addition, each time program execution reaches a legal update point, we dump the heap. We use the Sun JVM's heap dumping support. Our goal is to now discover a heap predicate that distinguishes marked objects from the unmarked ones, i.e., `(heap-predicate(x) == true)` if and only if `(marked(x) == true)`. We do this by feeding information about marked objects to an invariant detection tool. For our work, we use Daikon, a tool that discovers dynamic invariants in a running program and across multiple runs

of the program [33]. We construct a special Java program and run it through Daikon. The program contains a function whose parameters match the signature of the concerned class. The main method reads each marked object from the snapshot and calls this function with the fields of the object as parameters. We control how deep to look for invariants by passing the contents of referred objects, objects pointed by them, and so on. We also pass the type and count of references pointing to marked objects. We run this Java program though Daikon which infers invariants among the parameters of our specially constructed function. The invariants that hold across all runs can be used as predicates to identify marked objects. However, we can use only those predicates that definitely *do not* hold for any unmarked object. To do so, we negate each predicate identified by Daikon and check whether the negated predicate holds for every unmarked object in all heap snapshots. A predicate that is satisfied by every marked object and is not satisfied by even a single unmarked object can be used at runtime to distinguish marked and unmarked objects. We use the conjunction of all predicates identified by this process as our heap predicate to identify leaky objects.

There are two caveats that we need to keep in mind. First, this is a dynamic analysis, so we need adequate test coverage to be convinced that the predicates we have inferred are general. Second, we need to consider that some objects should be unmarked if the program subsequently writes to the field that the patch modifies. In the example in Figure 4.13, the absence of line 8 in the old version is a leak. However, it might be the case that there is a future legitimate write to field `f` that makes the object non-leaky. Therefore, whenever there is a write to field `f`, we unmark the concerned object, i.e., change its identification to non-leaky. In general, whenever a leak occurs in

the old version we mark the concerned object as leaky, and after any access to that object that invalidate its leaky state we mark it as non-leaky.

This approach can be generalized for other simple patches. For example, a patch that either adds a boolean field in the new version, or fixes incorrect setting of the field in the old version. In both cases, we run the new version and discover dynamic invariant relationships between the value of the concerned boolean field and other objects in the heap. We use this discovered predicate to set the new field's value with the following state transformer:

```
1 for all objects x of class D < C:
2     if heap-pred(x):
3         x.boolean_field = true
```

### 4.3.1 Invariants discovered from real fixes

We applied this methodology to three real fixes and present the automatic state transformers we obtained. Two of them are similar, and are for the memory leaks fixed in jEdit by SVN revisions 5178 and 14027, mentioned in Table 4.1. The other patch is from the Bittorent client Azureus. The new version adds a boolean field `isExpanded` for a GUI element.

**jEdit fixes in SVN revision 5178 and 14027** Figure 4.10 shows the patch and transformer for the leaky field `tokenHandler`. For this version, we discover that the following heap predicate distinguishes leaky objects from the non-leaky ones: `(o.tokenHandler != null)`. This predicate validates our transformer in Figure 4.10 (b), which sets all leaky fields to `null`.

**Adding a new boolean field in Azureus** The version number of the fix is `a04dc20b6b` in our Git repository of Azureus' CVS repository. This bugfix corresponds to commits made around 2010/02/14. The new version adds a field `isExpanded` to class `BaseMdiEntry`. The GUI element corresponds to a tree that displays data, and the `isExpanded` field specifies the state of a particular node in the tree. For this update, the default transformer generated by our Update Preparation Tool (UPT) would set the boolean field to `false`. However, a dynamic analysis on the new version of the application produces the following invariants among objects of type `BaseMdiEntry`:

```
1 this.iconBarEnabler.visible == this.sidebar.visible
2 this.isExpanded == this.soParent.paintListenerHooked
```

The second invariant set the value of the `isExpanded` field in the object transformer. Stephen Magill collaborated on studying this fix, ran it through Daikon, and discovered the above heap predicate.

## 4.4 Conclusion

In this chapter, we presented JVOLVE's new world state transformation model in comparison to an alternative old world model. We show that the models are similar and more flexible than lazy models. While the new world model makes implementation simpler and efficient, the old world model leads to more elegant transformation functions.

We extend our model to more sophisticated transformers with logic dependent on the state of each object. We use such transformers to repair application state that the old program incorrectly created. To simplify creation of such transformers, we introduce a dynamic analysis technique to discover

predicates that describe and distinguish heap objects that need repair. We use the discovered invariants to automatically generate state transformers, thereby reducing some of the effort required of the programmer. We also use these discovered invariants to help the programmer gain confidence in the correctness of their own transformer functions.

# Chapter 5

# Evaluation

To evaluate JVOLVE, we used it to update three open-source servers written in Java: Jetty webserver [88], JavaEmailServer [23], and CrossFTP server [76]. 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 instances or create duplicate instances with built-in special purpose redirection functionality.

We explored updates corresponding to releases made over roughly one to two years of each program's lifetime. None of these programs were maintained with DSU support in mind. Of the 22 updates we considered, JVOLVE could support 20 of them—the two updates we could not apply changed classes with infinitely-running methods, and thus no safe point could be reached. To our knowledge, no existing DSU system for Java could support all these updates; indeed, previous systems with simple support for updating method bodies would be able to handle only 9 of the 22 updates. Although JVOLVE cannot support every update, it would substantially reduce required downtime. It is the first DSU system for Java that has been shown to support changes to realistic programs as they occur in practice over a long period of time.

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

## 5.1 Performance

JVOLVE's impact on performance can be divided into the following. 1) the steady state impact of implementing dynamic updating support, 2) the pause in application execution to perform the update, and 3) the cost of recompiling updated and invalidated methods upon resuming execution. The latter is very hard to measure and can be grouped with the steady state performance impact. Our experiments show that the main performance impact of JVOLVE is the pause time while applying an update. Once updated, the application eventually runs without further overhead. To confirm this claim, we measured the throughput and latency of two Jetty versions while running on stock JikesRVM and on JVOLVE after dynamically updating to the next version. Section 5.1.1 shows that the performance of these configurations is essentially identical.

The cost of applying an update is the time to load any new classes, invoke a full heap garbage collection, and apply the transformation methods on objects belonging to updated classes. Roughly, the time to suspend threads and check that the application is at a safe-point is less than a millisecond, and classloading time is usually less than 20ms. The update disruption time is primarily due to the GC and object transformers, and is proportional to the size of the heap and the fraction of objects being transformed. We wrote a simple microbenchmark to measure these overheads. The results reported in Section 5.1.2 show that object transformation is the dominant cost.

We conducted all our experiments on an Intel Core 2 Quad machine running at 2.4 GHz machine with 2 GB of RAM. The machine ran Ubuntu 7.10 on Linux kernel version 2.6.22. We implemented JVOLVE on top of JikesRVM (SVN r15532) and built a FullAdaptiveSemiSpace configuration of the VM.

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

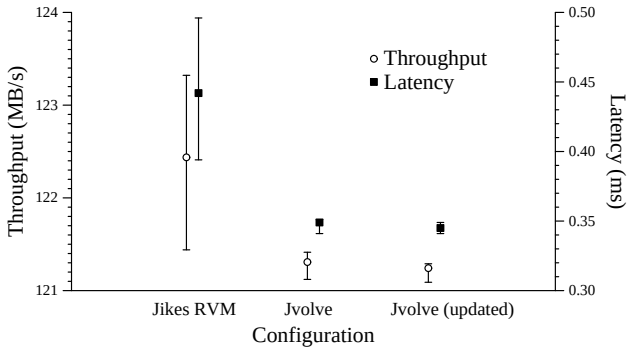



Figure 5.1: Throughput and latency measurements for Jetty webserver version 5.1.6 showing median and semi-interquartile range

The FullAdaptive configuration means that the VM's code is compiled at the highest level of optimization while creating the VM's boot image (Full) and the application code is compiled adaptively adaptively (Adaptive) [1]. In adaptive compilation, the VM baseline compiles an application method the first time it is executed, and recompiles the method at increasing levels of optimization as it gets invoked more often. The SemiSpace configuration refers to JikesRVM's semi-space garbage collector.

### 5.1.1 Jetty Webserver performance

To see the effect of updating on application performance, we measured Jetty under various request rates using httperf, a webserver benchmarking tool [58], and determined Jetty's saturation rate to be roughly 800 new con-

nection requests/second. We then measured Jetty's performance by issuing connections at this rate. Each connection makes 5 serial requests for a 40 Kbyte file. Httperf reports average throughput and average per-request latency over a 60 second period. We ran this experiment 21 times and report the median and quartiles of the throughput and latency reports. With 21 runs, the range between the quartiles serves as a 98% confidence interval [71]. In order to eliminate network traffic effects, we ran the server on two cores of a quad-core machine and the client on another core.

Figure 5.1 shows our results in tabular form and plotted graphically. The second and third columns of the table report the median throughput and the range between the two quartiles. The third column and fourth column report the median latency and the inter-quartile range. The first and second rows illustrate the performance of Jetty version 5.1.6 running on stock JikesRVM and JVOLVE, respectively. The third row shows the performance on JVOLVE of Jetty 5.1.6 dynamically updated from version 5.1.5 prior to the start of the experiment. The performance of the two JVOLVE configurations are statistically indistinguishable. The two configurations' corresponding inter-quartile ranges largely overlap. The performance of JVOLVE is also quite similar to the performance of stock JikesRVM. There are many small differences between JVOLVE and the stock implementation that change VM code size, code layout, and garbage collection behavior. These differences may impact performance directly and they may indirectly affect other elements of the VM, such as the timing of garbage collections and JIT optimizations (such indirect effects make VMs notoriously difficult to benchmark [15, 59]).

### 5.1.2 Microbenchmark performance

The two dominant factors that determine JVOLVE update time are the time to perform a GC, determined by the number of objects, and the time to run object transformers, determined by the fraction of objects being updated. To measure each cost, we devised a simple microbenchmark that creates an array of objects and transforms a specified fraction of these objects when a dynamic software update is triggered. The microbenchmark has two simple classes, `Change` and `NoChange`. Both contain three integer fields, and three reference fields that are always `null`. The update adds an integer field to `Change`. The user-provided object transformation function copies the existing fields and initializes the new field to zero. We measure the cost of performing an update while varying the total number of objects and the fraction of objects of each type. The number of objects is the maximum that can fit in heap sizes 32, 64, 128 and 256 MB. Note that JikesRVM's heap includes VM data structures as well. We measure the running time in a generous heap, five times the minimum required size, such that the only collections are those DSU triggers. We report the median of 21 runs.

Table 5.1 shows the elapsed time while varying the number of total objects and the fraction of the objects that are updated. The variance was insignificant, so we do not report it. The first group of rows reports garbage collection time, the second group reports the time to transform all updated objects, and the final group reports the total update time. While the total time includes the time to load and install the updated classes, synchronize running threads, and find a DSU safe point, it is roughly equal to the sum of GC time and transformation time. The first column of the table shows the number of objects in the test, and the second column the heap size, which is five times

| # objects in '000s | Heap (MB) | | Fraction of updated objects 0% | 10% | 20% | 30% | 40% | 50% | 60% | 70% | 80% | 90% | 100% |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 280 | 160 | | 78 | 81 | 83 | 89 | 99 | 103 | 108 | 113 | 113 | 120 | 120 |
| 770 | 320 | Garbage collection | 148 | 165 | 181 | 195 | 213 | 223 | 237 | 249 | 262 | 269 | 278 |
| 1,760 | 640 | time (ms) | 313 | 347 | 382 | 416 | 449 | 478 | 506 | 534 | 558 | 583 | 601 |
| 3,670 | 1,280 | | 615 | 694 | 763 | 833 | 900 | 965 | 1019 | 1076 | 1129 | 1181 | 1217 |
| 280 | 160 | Running transformation functions (ms) | 0 | 13 | 23 | 34 | 43 | 54 | 62 | 74 | 84 | 93 | 104 |
| 770 | 320 | | 0 | 33 | 63 | 91 | 116 | 145 | 173 | 201 | 231 | 262 | 292 |
| 1,760 | 640 | | 0 | 77 | 143 | 207 | 269 | 333 | 397 | 464 | 534 | 604 | 674 |
| 3,670 | 1,280 | | 0 | 160 | 299 | 429 | 560 | 693 | 827 | 975 | 1119 | 1263 | 1405 |
| 280 | 160 | | 82 | 99 | 109 | 128 | 147 | 161 | 174 | 192 | 202 | 218 | 228 |
| 770 | 320 | Total DSU pause | 153 | 202 | 249 | 291 | 334 | 372 | 414 | 455 | 498 | 535 | 576 |
| 1,760 | 640 | time (ms) | 316 | 429 | 530 | 627 | 723 | 816 | 908 | 1002 | 1097 | 1191 | 1281 |
| 3,670 | 1,280 | | 618 | 859 | 1065 | 1269 | 1466 | 1663 | 1850 | 2054 | 2253 | 2448 | 2627 |

Table 5.1: Microbenchmark results: JVOLVE update pause time (in ms) for various heap sizes

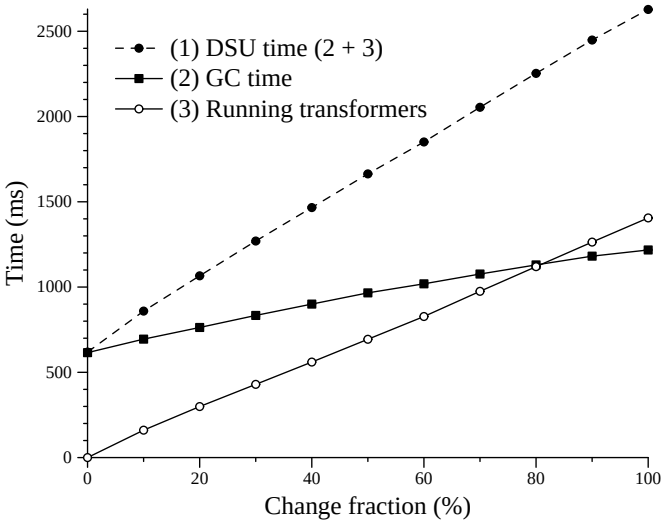



Figure 5.2: Microbenchmark pause times with a heap size of 1280 MB containing 3.67 million objects

the minimum heap size to run the microbenchmark. Columns 4 though 14 show pause times for varying fractions (from 0% to 100%) of updated objects.

To shed light on the results in the table, Figure 5.2 plots collection time, transformer time, and total update time for the microbenchmark with 3.67 million objects in a 1280 MB heap. The figure shows that the costs of garbage collection and transformation increase as a function of the number of changed objects. The slope of the "GC time" line illustrates the cost to deal with one object of the update type. This cost includes creating an additional copy of the transformed object; creating an update log entry with a pointer to the old and new copy; and caching a pointer to the old copy from the new copy. The slope of the "Running transformers" line illustrates the cost of accessing the update log and actually running the transformer for one object. This extra processing to handle transforming objects increases the total pause time with

all objects updated by roughly four times compared to the pause time with no object updated. The "Running Transformers" line is steeper than the "GC time" line, revealing that the cost of running transformers is higher than the extra copying cost incurred during GC.

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

## 5.2 Applications

We now take a look at our three applications in detail. We present their application structure, changes between versions, our success at updating them, and some non-trivial transformer functions.

### 5.2.1 Jetty webserver

Jetty is a popular webserver written in Java. It supports static and dynamic content and can be embedded within other Java applications. It is used by various popular projects such as Eclipse, Google App Engine, and the Hadoop map-reduce framework [45]. JikesRVM, and thus JVOLVE, is not able to run the most recent versions of Jetty (6.x). Therefore we considered 11 versions, starting at 5.1.0, released in November, 2004 through 5.1.10, released in January, 2006. Version 5.1.10 contains 317 classes and about 45,000 source

| Ver. | Date | SLOC | # classes added | # changed | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | classes | methods | | | fields | |
| | | | | | add | del | chg | add | del |
| 5.1.0 | Nov 2004 | 42,981 | | | | | | | |
| 5.1.1 | Dec 2004 | 43,073 | 0 | 14 | 4 | 1 | 38/0 | 0 | 0 |
| 5.1.2 | Jan 2005 | 43,277 | 1 | 5 | 0 | 0 | 12/1 | 0 | 0 |
| 5.1.3* | Apr 2005 | 43,612 | 3 | 15 | 19 | 2 | 59/0 | 10 | 1 |
| 5.1.4 | Jun 2005 | 43,578 | 0 | 6 | 0 | 4 | 9/6 | 0 | 2 |
| 5.1.5 | Nov 2005 | 44,027 | 0 | 54 | 21 | 4 | 112/8 | 5 | 0 |
| 5.1.6 | Nov 2005 | 43,948 | 0 | 4 | 0 | 0 | 20/0 | 5 | 6 |
| 5.1.7 | Dec 2005 | 44,081 | 0 | 7 | 8 | 0 | 11/2 | 9 | 3 |
| 5.1.8 | Dec 2005 | 44,082 | 0 | 1 | 0 | 0 | 1/0 | 0 | 0 |
| 5.1.9 | Dec 2005 | 44,084 | 0 | 1 | 0 | 0 | 1/0 | 0 | 0 |
| 5.1.10 | Jan 2006 | 44,102 | 0 | 4 | 0 | 0 | 4/0 | 0 | 0 |

Table 5.2: Summary of updates to Jetty

lines of code (SLOC) (lines of code excluding comments and blank lines, as reported by the sloccount tool [87, 89]).

Table 5.2 shows a summary of the changes in each update. For each version, the first three columns list the version number, release date and SLOC. For versions 5.1.1 through 5.1.10, each row tabulates the changes relative to the prior version. The fourth column shows the number of classes that the new version added. The fifth column shows the number of classes that contained changes from the previous version. Columns 6 though 10 enumerate changes such as the addition, deletion and modification of methods; and the addition and deletion of fields. For the eighth column listing changed methods, the notation $x/y$ indicates that $x + y$ methods were changed, where $x$ changed in body only, and $y$ changed their type signature as well. For dynamic updating systems that only support changes to method bodies, only the first and last three of the ten updates could be supported, since the rest either change method signatures and/or add or delete fields.

```
1  // There are different listeners for various
2  // virtual hosts Each listener has multiple
3  // threads
4  public class HttpListener {
5    public void run() {
6      while (true) {
7        if (job == null)
8          wait();
9        if (job != null)
10         handle(job);
11     }
12   }
13 }
14
15 // Maps a collection of HttpListener objects
16 // which generates requests and HttpContext
17 // objects which maintain collection of
18 // HttpHandlers
19 public class HttpServer {
20   public void start() {
21     for (HttpContext c : contexts) {
22       c.start();
23     }
24     for (HttpListener l : listeners) {
25       l.start();
26     }
27   }
28
29   public static void main(String args[]) {
30     HttpServer server = new HttpServer();
31     HttpContext context = server.getContext();
32     context.addHandler(...);
33     server.addListener(...);
34     server.start();
35   }
36 }
```

Figure 5.3: Jetty webserver code: High level organization

Figure 5.3 shows the high-level organization of the application. The main class of the application `HttpServer` services Hypertext Transfer Protocol (HTTP) requests by maintaining a map between a collection of `HttpListener` objects and `HttpContext` objects. The `HttpListener` objects listen for client requests. The application supports virtual hosts (multiple websites on the same server listening possibly at different addresses) by having a `HttpListener` object for each virtual host. The listener also maintains a fixed pool of threads to service client requests, thereby avoiding thread creation overhead for each request. The application uses complex producer-consumer synchronization between the client sockets and thread pools. `HttpContext` objects provide context such as filesystem path prefix, classpath, and resource location for `HttpHandler` objects. Each handler supports a different type of request—regular file request, running servlets and web applications, returning error messages, and so on. When the server is idle, threads in the thread pools wait to be woken up by listeners which wait on client sockets.

#### 5.2.1.1 State transformer functions in Jetty

Between the default class and object transformers the Update Preparation Tool (UPT) generated and those we wrote by hand, we successfully wrote dynamic updates to all versions of Jetty that we examined. The update to version 5.1.2 demonstrates the usefulness of automatically generated default transformers in the common case. Version 5.1.2 changes the access protection of method `setHttpContext` in class `HttpResponse` to `public`. This method is now called by methods outside class `HttpResponse` and JVOLVE correctly loads these method bodies to update the application. Since the update changes the class signature of `HttpResponse`, the entire class has to be reloaded again,

```
1  import org.mortbay.http.HttpResponse;
2  import org.mortbay.http.r_5_1_1_HttpResponse;
3
4  public final class JvolveTransformers {
5    public static void jvolveObject(HttpResponse to,
6                            r_5_1_1_HttpResponse from) {
7      to._status = from._status;
8      to._reason = from._reason;
9      to._httpContext = from._httpContext;
10   }
11   public static void jvolveClass(HttpResponse unused) {
12     HttpResponse.log = r_5_1_1_HttpResponse.log;
13     HttpResponse.__100_Continue =
14           r_5_1_1_HttpResponse.__100_Continue;
15     HttpResponse.__101_Switching_Protocols =
16           r_5_1_1_HttpResponse.__101_Switching_Protocols;
17     ...
18     HttpResponse.__505_HTTP_Version_Not_Supported =
19           r_5_1_1_HttpResponse.__505_HTTP_Version_Not_Supported;
20     HttpResponse.__507_Insufficient_Storage =
21           r_5_1_1_HttpResponse.__507_Insufficient_Storage;
22     HttpResponse.__statusMsg =
23           r_5_1_1_HttpResponse.__statusMsg;
24     HttpResponse.__Continue =
25           r_5_1_1_HttpResponse.__Continue;
26   }
27 }
```

Figure 5.4: UPT-generated transformers for the update to Jetty v5.1.2

and JVOLVE must run class and object transformers for this class. Figure 5.4 shows the default transformer that UPT generated. In this situation, where there is no semantic change, the default transformers correctly set the fields in the new version. Moreover, the fact that the `HttpResponse` class has over fifty fields make the default transformers extremely useful.

We now describe updates where default transformers were not sufficient. In Version 5.1.3 class `NCSARequestLog` added two boolean fields `_logLatency` and `_logCookies`. `_logLatency` specifies whether the application should in-

```
1 public class HttpContext {
2   private List
3     _systemClasses;
4 }
```

(a) Version 5.1.5

```
1 public class HttpContext {
2   private String[]
3     _systemClasses =
4       new String [] {...};
5 }
```

(b) Version 5.1.6

```
1 public static void jvolveObject(
2   HttpContext to, r_5_1_5_HttpContext from) {
3     to._systemClasses = null;
4 }
```

(c) Default transformer

```
1  public static void jvolveObject(
2    HttpContext to, r_5_1_5_HttpContext from) {
3    if (from._systemClasses == null) {
4      to._systemClasses = null;
5    } else {
6      to._systemClasses =
7        new String[from._systemClasses.size()];
8      int i = 0;
9      for (Object o : from._systemClasses) {
10       to._systemClasses[i] = (String) o;
11       i++;
12     }
13   }
14 }
```

(d) User generated transformer

Figure 5.5: Object transformer from Jetty version 5.1.5 to 5.1.6

clude the latency to process HTTP requests when it logs information about a request, and `_logCookies` controls whether or not that application logs cookies information. The default transformer sets these boolean fields to `false`. However, the actual values to set these fields to depend on the configuration parameters when running the new version. In the same update, Class `Pool` which manages a pool of threads to handle HTTP requests adjusts the number of threads based on system load, added a `private` field `_lastShrink` to store the time the number of threads were last shrunk. Jetty uses this field to prevent shrinking available threads frequently. Setting this field to the default value of zero should not pose any major problems because this field will have the correct time the next time the pool is shrunk.

The update from version 5.1.5 to 5.1.6 changed the type of two fields `_systemClasses` and `_serverClasses` from `List` in the old version to `String[]` in the new, in class `HttpContext`. Figure 5.5 shows the changes to one of these fields. In Figure 5.5 (c), UPT's default transformer sets the field `_systemClasses` to `null` (similar to the example with JavaEmailServer, Figure 3.6). Since the new version's field declaration (Figure 5.5 (b)) already declares the value of these fields, the developer writing to object transformer has to decide whether to use this value or convert the value stored in the old version's list into a String array. We choose the latter option, shown in Figure 5.5 (d).

#### 5.2.1.2 Reaching a safe point in Jetty

For the updates we considered, we tried to study how our safe point restrictions (Section 3.5.2) inhibited a dynamic update. Other than the update to 5.1.3, all versions immediately reached a safe point every time, with no need

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

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

Table 5.3: Impact of safe point restrictions on updates to Jetty

of return barriers.

For each version, starting at 5.1.0, we ran Jetty under 20% load (160 connections per second from our httperf experiments). After 30 seconds we tried to apply the update to the next version; if a safe point could not be immediately reached, we deemed the attempt as failed. We tried five such attempts for each version, starting up from the server from scratch for each attempt. The results are presented in Table 5.3. Column 2 shows the number of times out of five such runs where the application reached a safe point. The methods whose presence on a thread stack precluded the application from reaching a safe point are mentioned below the table. For the update to 5.1.3, the offending method was always active because it contained an infinite loop. The other updates either always succeeded, or did after a small number of retries.

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

Column 3 contains the total number of methods in the program at runtime, where the number in parentheses is the number of those which the compiler inlined when using aggressive optimization. This provides an upper bound on the effect of inlining in reaching a safe point. The next group

of columns contains the restricted method set. Each column in the group specifies the number of methods loaded at run time by the VM, followed by the total number of methods in that category in the program. The first column in this group is the number of methods in classes involved in a class update. Recall that when a class is updated, say by adding a field, all its methods are considered restricted (see section 3.5.2). The second column in this group is the number of methods whose bodies are updated, the third is the number of category (2) or indirectly updated methods, and the fourth sums these, with the number of methods that were inlined written in parentheses. The final two columns list the total number of methods in the restricted set; they differ from the first number in the fourth column by the number of inlined callers of the restricted methods that were not already restricted. The final column lists the actual number of restricted methods when JVOLVE's OSR capability is enabled.

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

| Ver. | Date | SLOC | # classes | | # changed | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | add | del | classes | methods | | | fields | |
| | | | | | | add | del | chg | add | del |
| 1.2.1 | Dec 2002 | 2,841 | | | | | | | | |
| 1.2.2 | Feb 2003 | 2,841 | 0 | 0 | 3 | 0 | 0 | 3/0 | 0 | 0 |
| 1.2.3 | Jul 2003 | 2,924 | 0 | 0 | 7 | 0 | 0 | 14/2 | 12 | 0 |
| 1.2.4 | Sep 2003 | 2,961 | 0 | 0 | 2 | 0 | 0 | 4/0 | 0 | 0 |
| 1.3* | Oct 2003 | 2,305 | 4 | 9 | 2 | 11 | 3 | 6/9 | 12 | 5 |
| 1.3.1 | Oct 2003 | 2,307 | 0 | 0 | 2 | 0 | 0 | 4/0 | 0 | 0 |
| 1.3.2 | Nov 2003 | 2,359 | 0 | 0 | 8 | 4 | 2 | 4/2 | 3 | 1 |
| 1.3.3 | Nov 2003 | 2,368 | 0 | 0 | 4 | 0 | 0 | 3/0 | 0 | 0 |
| 1.3.4 | Feb 2004 | 2,447 | 0 | 0 | 6 | 2 | 0 | 6/0 | 2 | 0 |
| 1.4 | Jul 2004 | 2,529 | 0 | 0 | 7 | 6 | 1 | 4/1 | 6 | 0 |

Table 5.4: Summary of updates to JavaEmailServer

### 5.2.2 JavaEmailServer

For JavaEmailServer, we considered 10 versions—1.2.1 through 1.4—spanning a duration of about two years. Version 1.4 consists of 20 classes and about 2500 SLOC. Table 5.4 shows the release date, source lines of code and summary of updates to each new version.

Figure 5.6 shows the high-level structure of JavaEmailServer's code. The application creates multiple Post Office Protocol (POP) and Simple Mail Transfer Protocol (SMTP) protocol handler threads. The number of threads can be configured by the user. Each thread listens on its respective port (110 for POP and 25 for SMTP), waiting for new client connections. Upon receiving a client request, the server thread communicates with the client, services the client's command, and goes back to listening again. The main loop of POP and SMTP servers are in `Pop3Processor.run()` and `SMTPProcessor.run()` respectively. The POP server authenticates users, reads their e-mail messages from the filesystem and sends them to the client through the network. The

SMTP server reads in e-mail messages from the network and writes them to the filesystem. These messages may be meant to be delivered to local users or relayed to other servers on the internet. Message delivery is handled by a single `SMTPSender` thread, which polls the filesystem for new messages and writes them into user mailboxes, or relays them to other servers. The number of threads and therefore the maximum number of clients that can simultaneously be serviced is fixed and known when the application starts. The server will ignore clients that try to connect when all server threads are responding to requests. At any moment, the `run()` methods of all threads and based on server load, additional methods to handle the POP and SMTP protocols and methods to deliver messages are active on stack.

#### 5.2.2.1 Updates to JavaEmailServer

Approaches that only support updates to method bodies will be able to handle only four of the updates shown in Table 5.4. We could successfully construct updates for all versions we examined and we could successfully apply all of them except the update to version 1.3. For all updates but the one from 1.3.1 to 1.3.2 (shown in Figure 3.3), default transformers were sufficient.

Version 1.3 has 20% fewer SLOC than its prior version because the update reworked the entire configuration framework of the server. Among other things, this version removes a GUI tool for user administration and adds several new classes that implement a file-based configuration system. As a result, many classes are modified to point to a new configuration object. Among these classes are threads with infinite processing loops that accept POP and SMTP requests. Because these threads are always active, the safety condition can never be met and thus the update cannot be applied.

```
1  // Listen on socket for POP client connections
2  // Handle a client
3  public class Pop3Processor implements Runnable {
4    public void run() {
5      while (true) {
6        Socket client = serverSocket.accept();
7        handleCommands();
8      }
9    }
10 }
11
12 // SMTPProcessor is similar to Pop3Processor
13
14 // Poll the filesystem and deliver messages
15 public class SMTPSender implements Runnable {
16   public void run() {
17     while (true) {
18       if (messageToBeDelivered)
19         deliverMessages();
20       sleep();
21     }
22   }
23 }
24
25 public class Mail {
26   public static void main(String args[]) {
27     // Create 5 POP threads
28     for (int i = 0; i < 5; i++)
29       new Thread(new Pop3Processor()).start();
30
31     // Create 5 SMTP threads
32     for (int i = 0; i < 5; i++)
33       new Thread(new SMTPProcessor()).start();
34
35     // Create one thread that delivers mail
36     new Thread(new SMTPSender()).start();
37   }
38 }
```

Figure 5.6: JavaEmailServer code: High level organization

| Ver. | SLOC | | # classes | | # changed | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Total | Without JmDNS | add | del | classes | methods | | | fields | |
| | | | | | | add | del | chg | add | del |
| 1.05 | 13,852 | 13,852 | | | | | | | | |
| 1.06 | 13,926 | 13,926 | 4 | 1 | 1 | 0 | 0 | 3/0 | 1 | 0 |
| 1.07 | 18,081 | 14,066 | 0 | 0 | 3 | 4 | 0 | 14/0 | 5 | 0 |
| 1.08 | 18,108 | 14,093 | 0 | 1 | 3 | 2 | 0 | 10/0 | 0 | 2 |

Table 5.5: Summary of updates to CrossFTP server

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

### 5.2.3 CrossFTP server

CrossFTP server is an easily configurable, security-enabled File Transfer Protocol (FTP) server. CrossFTP allows simple configuration through a Graphical User Interface (GUI) and more advanced customization using configuration files. The GUI interface also allows detailed monitoring of server operation and a real-time look at active clients and the commands they issue. We did not use the GUI interface and therefore do not consider changes to that part of the program.

We looked at 4 versions of CrossFTP—1.05 through 1.08, details shown in Table 5.5—spanning a duration of more than a year. Version 1.08 contains about 18,000 SLOC across 161 classes. Version 1.07 added over four thousand SLOC by including code from JmDNS, a Java implementation of multi-cast DNS [84]. We did not exercise this functionality in our runs.

```
1  public class FtpServer implements Runnable {
2    // main loop of server
3    public void run() {
4      while (true) {
5        Socket cl = serverSocket.accept();
6        RequestHandler rh = new RequestHandler(cl);
7        (new Thread(rh)).start();
8      }
9    }
10 }
11
12 // Handle a single client
13 public class RequestHandler implements Runnable {
14   private BufferedReader reader;
15   private FtpWriter writer;
16   private boolean isClosed;
17
18   // main loop of server-client conversation
19   public void run() {
20     while (!isClosed) {
21       String command = reader.readLine();
22       service(command, writer);
23     }
24   }
25 }
26
27 // Creates a single server thread
28 public class CommandLine {
29   public static void main(String args[]) {
30     FtpServer server = new FtpServer();
31     (new Thread(server)).start();
32   }
33 }
```

Figure 5.7: CrossFTP code: High level organization

Figure 5.7 shows the high level organization of the server's code. The server has one active thread, an instance of `FtpServer` which listens on a socket for client connections. The main loop of the server is shown in function `FtpServer.run()`. For each client, the server spawns a new thread, an instance of the class `RequestHandler`. At any point of time, the application is running one server thread and multiple connection threads, one for each active client connection. The main loop to handle a client is given in `RequestHandler.run()`. The server reads a command from the client, services the command and waits for the next one. In steady state, `FtpServer`'s `run` method, `RequestHandler`'s `run` method and methods involved in processing particular FTP commands from the client are active on stack.

#### 5.2.3.1 Updates to CrossFTP

JVOLVE successfully applies all three updates to this application. For all three updates, the default transformers we sufficient. Note that since all updates either add or delete fields, simple method body updating support on its own would be insufficient.

JVOLVE could only apply the update from version 1.07 to 1.08 when the server was relatively idle. The server runs a new `RequestHandler` thread to process each FTP session, and the `RequestHandler.run()` method was changed by the update. JVOLVE installs a return barrier on this method, but with many active sessions, this method is essentially always on stack, preventing the update.

## 5.3 Conclusion

Our evaluation shows that JVOLVE is the first full-featured dynamic updating system that imposes no steady-state overhead. To perform an update, JVOLVE pauses the application for a modest time period, roughly equal to a full heap garbage collection pause. JVOLVE is the most versatile dynamic updating system for Java and supports 20 of 22 real-world updates written over a one to two year period to three open-source server programs.

# Chapter 6

# Related Work

Researchers have designed and implemented dynamic updating solutions in a variety of contexts, ranging from theoretical models of updates and safety, to language design for dynamic updating, to practical DSU systems for C and Java, and to operating systems with DSU support. In this chapter, we compare our VM-centric approach to DSU with related work on implementing DSU for managed languages, C, and C++.

## 6.1 Dynamic Software Updating for C/C++

There are several substantial systems for dynamically updating C and C++ programs that target server applications [42, 3, 64, 21, 51, 62] and operating systems components [75, 8, 20, 48, 52]. While these systems are mature and offer substantial updating experience, the flexibility afforded by Jvolve is comparable or superior.

Jvolve's timing restrictions and Java's type safety also provide comparable or superior safety. Because C is a type-unsafe language, DSU systems for C have to restrict certain unsafe C idioms and perform conservative analysis to enforce type-safety of updates. The lack of a VM is a disadvantage for C and C++ DSU. For example, because a VM-based JIT can compile and recompile replacement classes, it imposes no steady-state execution overhead.

By contrast, C and C++ implementations must use either statically-inserted indirections [42, 64, 75, 8, 51] or dynamically-inserted trampolines to redirect function calls [3, 20, 21, 5]. Both cases impose persistent space and time overhead on normal execution and inhibit optimization. Likewise, because these systems lack a garbage collector, they either do not update object instances at all [5], update them lazily [64, 21] or perform extra allocation and bookkeeping to locate the objects at update-time [8].

Because these systems lack support for on-stack replacement, they must pre-compile potentially long-running methods specially, so that they can be updated while they run. These techniques impose time and space overheads on steady-state execution, and in some cases limit update flexibility. Some prior systems [62, 51, 21] have focused on means to reach DSU safe points quickly, and JVOLVE is comparable in support when it comes to single-threaded applications. For multithreaded applications, STUMP's synchronization of multiple threads to reach a safe point [62] and Upstare's support to update active methods on stack [51] are superior to what JVOLVE provides. While we did not implement such features in JVOLVE, its model does not preclude such support.

### 6.1.1 K42 Operating System

K42 is an object-oriented research operating system with dynamic updating support out of IBM Research. Updates are performed at the granularity of an *object instance*. K42 is structured as a set of objects, with objects exporting an interface by declaring a virtual base class. All objects that are updated must provide state transfer functions to export the old version to a common format and import from the common format to the new version. K42 reaches a safe point by reaching a quiescent state. The OS services requests

by creating a new kernel thread for each request. To perform an update, K42 restricts new accesses to updated objects, waits until all prior requests/threads have completed, and then performs the update. Finally, to support access to the new version of an object, K42 uses an *object translation table*. Each object in the system has an entry in the table, and all object invocations go through the table. At update time, K42 modifies the table to invoke the new versions of an object.

### 6.1.2 Ksplice

Ksplice [5] is a dynamic updating system for the Linux kernel. Ksplice is very easy to use and takes as input a source patch to the currently running kernel, and generates a binary patch that can be applied to the running kernel. Ksplice only supports changes to method bodies, and does not support function or type signature changes. However, this flexibility has been sufficient to support most patches that fix security vulnerabilities. Like all C systems, Ksplice needs some form of function indirection to jump to the new version of a method. Ksplice does so by installing *trampolines* at update time. Ksplice overwrites the first few instructions of the old version's method body with a jump instruction to the new method. Future calls to an updated method jump through this trampoline to the latest version's body. Ksplice uses activeness check and prevent updates when changed methods are active on some thread's stack.

### 6.1.3 Ginseng

Ginseng [64, 60] is a DSU system for C server applications, and is very flexible in supporting changes to method bodies, method signatures, structure

definitions, and global data. Ginseng uses standard techniques needed of DSU systems without a managed runtime to support updates to code and data. Ginseng uses function indirection to make calls to the right version of a method and "type wrapping" to check that accessed data has the right version of a type. Ginseng's offline patch generator generates state transformers for updated types, and the runtime uses padding to accommodate additional space required by new versions of objects. The application invokes state transformers lazily when it first accesses an object after the update.

While Ginseng does not support changes to active methods on stack, programmers may annotate long running loops, extract loop bodies into their own methods, and then update between loop iterations. To guarantee type-safety, Ginseng must first deal with the type-unsafeness of C. Ginseng prohibits the use of certain C idioms that are *unsafe*. Ginseng also introduced a notion of safety called *transaction safety* in which programmers mark transaction regions. The system's analysis restricts certain update points within a region to ensure that a transaction runs entirely as the old version or as the new version.

### 6.1.4 Upstare

Upstare [51, 50] supports dynamic updates to multi-threaded C programs. Upstare performs a whole-program compilation and extensively instruments an application to make it update-compatible. Upstare uses function indirection for function calls, instruments function entry and exit points, and loop back edges to guarantee "immediate updates" when the user signals the application to update. These instrumentation points are precisely the same VM safe points that JVOLVE uses. Like JVOLVE, Upstare allows the

programmer to specify update points by making API calls from within the application, or initiate an immediate update upon receiving a signal from a user. Upstare suspends all application threads and performs atomic updates, just like JVOLVE does. It relies on complex synchronization to suspend all but one application thread, the *update co-ordinator* thread which performs the update. Upstare cannot perform an update when threads are waiting on blocking system calls. This restriction is especially problematic for multithreaded server application threads which are often blocked waiting for requests. JVOLVE's thread synchronization mechanism is much simpler and supports blocking system calls in a straightforward manner.

Upstare's support for *stack reconstruction* is unique among dynamic updating systems. Upstare extracts the state of a function's stack, and reconstructs it as expected by the new version of a function. This reconstruction support enables even modified functions to be active on stack. Stack reconstruction requires the programmer to specify state transformers for the local variables of functions, and correspondence between execution points in the old and new function bodies. JVOLVE restricts its On-Stack Replacement (OSR) support to methods with identical bytecodes in the old and new versions. As a result, JVOLVE currently does not require a stack state transformer or a mapping or execution. There is no conceptual reason why JVOLVE wouldn't be able to support an extended OSR, but for the additional burden on the programmer.

## 6.2 Dynamic Software Updating for managed languages

Researchers have adopted several approaches to bring DSU support to managed languages. These include special-purpose compilation, class loaders,

or VM-support. The main drawback of approaches that do not change the VM are inflexibility and high overhead.

### 6.2.1 Edit and continue development

Debuggers and IDEs have long provided *edit and continue* (E&C) functionality that permits limited modifications to program state to avoid stopping and restarting during debugging. For example, Sun's HotSwap VM [81, 27], .NET Visual Studio for C# and C++ [56], and library-based support [29] for .NET applications all provide E&C. These systems restrict updates to code changes within method bodies. While this restriction reduces safety concerns and obviates the need for class and object transformers, the resulting systems are inflexible. A request for enhanced dynamic updating has the fourth highest number number of votes among enhancement requests recorded in the Hotspot VM bug database [79, 80]. These systems cannot perform more than half of the updates discussed in Chapter 5.

### 6.2.2 Solutions without VM-support

JRebel [91] is a productivity tool targeting Java EE (Enterprise Edition) developers. JRebel watches changes to the source tree of a web application under development and applies these changes to a running VM. Developers do not have to restart their application after every change, thereby speeding up development. JRebel is implemented on top of the Sun Hotspot VM's instrumentation API. When loading a class, JRebel rewrites the bytecodes of all methods, intercepting all method calls and field accesses. JRebel implements method and field accesses by performing a dictionary lookup, and incurring a high performance overhead, like in an interpreted language such as

Python or Ruby. When a new version of a class is available, JRebel updates the dictionary to point to the new method versions. While JRebel supports addition and deletion of fields, it does not update the state of existing objects, rendering updates type-unsafe. It also has no notion of update timing for safety. Because of the performance overhead and the lack of type safety, JRebel is suited mainly to ease debugging during development.

Barr and Eisenbach [7] support updates to Java libraries in a client server model. The system supports a subset of updates to Java programs that do not cause linker errors. Milazzo et al. [57] support dynamic updating in a distributed computing environment. Their work proposes a specialized software architecture to monitor updates, and distribute them across the application. Both systems use custom classloaders for binary-compatible and component-level changes respectively, but cannot support signature changes such as class field additions.

Orso et al. [67] use source-to-source translation for DSU by introducing a proxy object that indirectly accesses an object that may change. For each class C that might change in the future they produce a proxy for that class. All calls from clients of C are redirected to call the wrapper instead. When C is updated by some new class C', a new C' object is created and initialized using the old state of C and the wrapper is redirected to point to C'. This approach requires updated classes to export the same public interface, forbidding new public methods and fields.

Non VM-based approaches are in general limiting because they are not *transparent*—they make visible changes to the class hierarchy, and insert or rename classes. This approach makes it essentially impossible to be robust in the face of code using reflection or native methods. Moreover, the indirec-

tion imposes time and space overheads on steady-state execution. Our VM approach naturally supports reflection and native methods (these are updated as well), is more expressive, e.g., it supports signature changes, and imposes no overhead on steady-state execution.

### 6.2.3 VM support for DSU in managed languages

The PROSE system performs short-term, run-time patches to code for logging, introspection, and performance adaptation, rather performing general updates [65]. An Eclipse plug-in performs run-time bytecode instrumentation and a modified JIT performs method code replacement, using an API in the style of aspect-oriented programming. PROSE has the same update model as the E&C systems: it supports updates to method bodies but not class or method signature changes that require changes to object state. This flexibility is similar to the E&C implementations discussed above; indeed, PROSE builds on the HotSwap method replacement support in its Sun JDK implementation [81].

JDrums [72] and the Dynamic Virtual Machine (DVM) [53] both implement DSU for Java within the VM, providing a programming interface similar to Jvolve, but are lacking in two ways. First, neither JDrums nor DVM have ever been demonstrated to support updates from real-world applications. Second, their implementations impose overheads during steady-state execution. They both update *lazily* and use an extra level of indirection (the *handle space*). Indirection conveniently supports object updates, but adds extra overhead. For example, JDrums traps all object pointer dereferences to apply VM object transformer function(s) when the object's class changes. Lazy updating has the advantage that it amortizes pauses due to an update over subsequent

execution. The main drawback is that its overhead persists during normal execution, even though updates are relatively rare. DVM works only with the interpreter. Relative to this interpreter, which is already slow, the extra traps result in roughly 10% overhead.

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

### 6.2.4 Dynamic ML

Gilmore et al. [36] propose DSU support for modules in ML programs using a similar, but more restrictive programming interface compared with JVOLVE. They formalize an abstract machine for implementing updates using a copying garbage collector. Duggan [28] also proposes dynamic updates to ML programs, focusing on lazy updates to data type definitions. Neither approach was ever implemented.

### 6.2.5 Language support for DSU

UpgradeJ [13] is an extension to the Java language design supporting class upgrades, in two flavors: *revision upgrades*, which may modify method bodies, and *evolution upgrades*, which may add new methods and fields. Programmers control the effects of upgrades using *version annotations*, introduced by Bierman et al. [12]. For example, the programmer may write `o =`

`new Button[1=]()` to force `o` to always use version 1 methods, while writing `p = new Button[1+]()` or `p = new Button[1++]()` allows `p` to be revised or evolved, respectively. UpgradeJ's update model is easier to implement than JVOLVE's because it need not change existing object instances. Of course, the downside is a loss of flexibility. Many of the updates to our benchmark applications change field contents and layout. UpgradeJ does not support these updates. On the other hand, evolution upgrades add power over simple method body updates, and consequently enable more real-world updates to be supported [82]. There currently is no implementation of UpgradeJ.

## 6.3 Updates in a persistent object store

Boyapati et al. [17] support dynamic updates to classes kept in a *persistent object store* (POS). While the setting is different, their basic update model, and in particular their notion of object transformer function, is similar to ours. In their system, programmers manually write an object transformer that they view as a method on the old version of the updated class, i.e., the transformer method is type-safe with respect to the old class. In JVOLVE, object transformers may access the *new* versions of objects pointed to from the old class. Instead, Boyapati et al.'s transformers may access the *old* versions. To implement this model, they rely on *encapsulation* based on *ownership types*: if an object $a$ of class $A$ has an "owned" field pointing to an object $b$ of class $B$, then only $a$ can point to (and access) $b$. Encapsulation thus ensures the system will always transform $a$ before $b$, which makes the transformation algorithm more efficient. They rely on the programmer to enforce encapsulation, and describe how the compiler could automate language support for encapsulation in a non-standard type system. JVOLVE takes the opposite tack of

forcing old object fields to point to up-to-date objects, and thus requires no special language support. Moreover, JVOLVE's model follows that of earlier work [8, 64, 62, 51] which has proven its effectiveness on a half-dozen realistic applications across several years' worth of releases.

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

## 6.4 Summary

Table 6.1 summarises features, strengths, and limitations of four DSU systems for C/C++ — Ginseng, Upstare, Ksplice, and K42; updates in a persistent object store; and three DSU systems for Java — JDrums, DVM, and Jvolve. The table denotes advantages of a system with a '✓', disadvantages with '✗', and neutral features with a '•'. But for Upstare's generic stack reconstruction support, Jvolve is superior to every other DSU system in one or more ways.

This recent surge in the design and development of dynamic updating system shows a rising demand for highly available software and the limitations

| | C/C++ | | | | | Java | | |
|---|---|---|---|---|---|---|---|---|
| | Ginseng | Upstare | Ksplice | K42 | Boyapati et al. | JDrums | DVM | Jvolve |
| **Supported changes** | | | | | | | | |
| Method body | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Method types | ✓ | ✓ | × | ✓ | ✓ | ✓ | ✓ | ✓ |
| Data/type signatures | ✓ | ✓ | × | ✓ | ✓ | ✓ | ✓ | ✓ |
| Across class hierarchy | - | - | - | × | - | × | × | ✓ |
| Updates to active methods | × | ✓ | × | × | × | × | × | × |
| **Update timing** | | | | | | | | |
| Old changed code runs after update | • | | | | | | | |
| Atomic update implementation | | | • | | | | | • |
| Activeness safety | | | • | • | • | • | • | • |
| **Data changes** | | | | | | | | |
| No access indirection | × | × | - | × | × | - | - | ✓ |
| Does not use padding | × | × | - | ✓ | ✓ | - | - | ✓ |
| **State transformers** | | | | | | | | |
| Automatically generated | ✓ | ✓ | - | ✓ | ✓ | | | ✓ |
| Refer to old/new state (O/N) | O | O | - | O | O | - | - | N |
| Run Lazily/Eagerly (L/E) | L | L | - | L | L | - | - | E |
| **Flexibility** | | | | | | | | |
| Multi-threaded | ✓ | ✓ | ✓ | ✓ | ✓ | × | × | ✓ |
| **Performance** | | | | | | | | |
| No steady-state time overhead | × | × | ✓ | × | ✓ | × | × | ✓ |
| No steady-state space overhead | × | × | ✓ | ✓ | × | × | × | ✓ |

**Legend**

✓ Advantage × Disadvantage

• Neutral - Not applicable

Table 6.1: Comparison of DSU systems

in other systems reveal the difficulties inherent in combining performance, efficiency, and flexibility as we did in JVOLVE.

# Chapter 7

# Conclusion

This dissertation presents JVOLVE, a Java Virtual Machine with support for Dynamic Software Updating. JVOLVE is the most fully-featured, best-performing DSU implementation for Java to date. The key contribution of this dissertation is showing how to achieve safe, flexible, and efficient dynamic software updating by extending existing Virtual Machine services including dynamic class loading, thread synchronization, return barriers, on-stack replacement, just-in-time compilation, and garbage collection. We demonstrate JVOLVE's success by applying updates for one to two years worth of releases for three programs: Jetty webserver, JavaEmailServer, and CrossFTP server.

This dissertation explores state transformers with behavior that depends on each object's state, and uses such transformers to repair application state at update time for certain classes of bugs. This dissertation also introduces a novel dynamic analysis to automatically generate object-specific transformers.

Our work demonstrates how dynamic software updating support can be integrated into modern VMs, and that doing so has the potential to significantly improve software availability by reducing downtime.

# Bibliography

[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. F. 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), Feb. 2000.

[2] B. Alpern, D. Attanasio, J. J. Barton, A. Cocchi, S. F. Hummel, D. Lieber, M. Mergen, T. Ngo, J. Shepherd, and S. Smith. Implementing Jalapeño in Java. In *ACM SIGPLAN Conference on Object-oriented Programming Systems, Language, and Applications*, pages 314–324, Denver, Colorado, Nov. 1999.

[3] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz. OPUS: Online Patches and Updates for Security. In *USENIX Security Symposium*, pages 287–302, Baltimore, Maryland, Aug. 2005.

[4] J. L. Armstrong and R. Virding. Erlang—An Experimental Telephony Switching Language. In *13th International Switching Symposium*, Stockholm, Sweden, 1991.

[5] J. Arnold and F. Kaashoek. Ksplice: Automatic rebootless kernel updates. In *ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 187–198, Nuremberg, Germany, Apr. 2009.

[6] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *ACM SIGPLAN Conference on Object-oriented Programming Systems, Language, and Applications*, pages 47–65, Minneapolis, Minnesota, Oct. 2000.

[7] M. Barr and S. Eisenbach. Safe Upgrading without Restarting. In *IEEE International Conference on Software Maintenance*, pages 129–137, Amsterdam, The Netherlands, Sept. 2003.

[8] A. Baumann, J. Appavoo, D. D. Silva, J. Kerr, O. Krieger, and R. W. Wisniewski. Providing Dynamic Update in an Operating System. In *USENIX Annual Technical Conference*, pages 279–291, Anaheim, California, Apr. 2005.

[9] A. Baumann, J. Appavoo, R. W. Wisniewski, D. D. Silva, et al. Reboots are for Hardware: Challenges and Solutions to Updating an Operating System on the fly. In *USENIX Annual Technical Conference*, pages 337–350, Santa Clara, California, June 2007.

[10] A. Baumann, A. Baumann, D. D. Silva, O. Krieger, and R. W. Wisniewski. Improving Operating System Availability With Dynamic Update. In *Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, pages 21–27, Boston, Massachusetts, Oct. 2004.

[11] D. Berlind. Vista forces reboots. `http://news.zdnet.com/2422-13568_22-157931.html`.

[12] G. Bierman, M. Hicks, P. Sewell, and G. Stoyle. Formalizing Dynamic Software Updating. In *Workshop on Unanticipated Software Evolution*,

Warsaw, Poland, April 2003.

[13] G. Bierman, M. Parkinson, and J. Noble. UpgradeJ: Incremental typechecking for class upgrades. In *AITO European Conference on Object-Oriented Programming*, pages 235–259, Paphos, Cyprus, July 2008.

[14] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *International Conference on Software Engineering*, pages 137–146, Edinburgh, Scotland, May 2004.

[15] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffman, A. M. Khan, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. Wake Up and Smell the Coffee: Evaluation Methodology for the 21st Century. *Communications of the ACM*, 51(8):83–89, Aug. 2008.

[16] M. D. Bond and K. S. McKinley. Leak Pruning. In *ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 277–288, Washington, DC, Oct. 2009.

[17] C. Boyapati, B. Liskov, L. Shrira, C.-H. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. In *ACM SIGPLAN Conference on Object-oriented Programming Systems, Language, and Applications*, pages 403–417, Anaheim, California, Oct. 2003.

[18] J. Buisson, C. Carro, and F. Dagnat. Issues in applying a model driven approach to recongurations of satellite software. In *ACM Workshop on*

*Hot Topics in Software Upgrades*, pages 6:1–6:5, Nashville, Tennessee, Oct. 2008.

[19] C. Chambers and D. Ungar. Making pure object-oriented languages practical. In *ACM SIGPLAN Conference on Object-oriented Programming Systems, Language, and Applications*, pages 1–15, Phoenix, Arizona, Oct. 1991.

[20] H. Chen, R. Chen, F. Zhang, B. Zang, and P.-C. Yew. Live updating operating systems using virtualization. In *ACM SIGPLAN Conference on Virtual Execution Environments*, pages 35–44, Ottawa, Canada, June 2006.

[21] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew. POLUS: A POwerful Live Updating System. In *International Conference on Software Engineering*, pages 271–281, Minneapolis, Minnesota, May 2007.

[22] C. J. Cheney. A Nonrecursive List Compacting Algorithm. *Communications of the ACM*, 13(11):677–678, Nov. 1970.

[23] E. Daugherty. Java SMTP/POP EMail Server. `http://sourceforge.net/projects/javaemailserver`.

[24] J. deJong. Java Becoming Solution for Safety-Critical Applications. *Software Development Times*, Aug. 2007. `http://www.sdtimes.com/link/31052`.

[25] D. Dig and R. Johnson. How do APIs evolve?: A story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice*, 18:83–107, March 2006.

[26] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-Fly Garbage Collection: An Exercise in Cooperation. *Communications of the ACM*, 21(11):966–975, Nov. 1978.

[27] M. Dmitriev. Towards Flexible and Safe Technology for Runtime Evolution of Java Language Applications. In *Workshop on Engineering Complex Object-Oriented Systems for Evolution*, Tampa Bay, Florida, Oct. 2001.

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

[29] M. Eaddy and S. Feiner. Multi-Language Edit-and-Continue for the Masses. Technical Report CUCS-015-05, Columbia University Department of Computer Science, Apr. 2005.

[30] Eagle Rock Alliance Ltd. Cost of Downtime: Online Survey Results. `http://contingencyplanningresearch.com/2001%20Survey.pdf`, 2001.

[31] ej-technologies GmbH. jclasslib Java Bytecode Viewer. `http://jclasslib.sourceforge.net`.

[32] T. Ekman and G. Hedin. The Jastadd Extensible Java Compiler. In *ACM SIGPLAN Conference on Object-oriented Programming Systems, Language, and Applications*, pages 1–18, Montreal, Quebec, Canada, Oct. 2007.

[33] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of

likely invariants. *Science of Computer Programming*, 69(1–3):35–45, Dec. 2007.

[34] J. Fenn and A. Linden. *Hype Cycle Special Report for 2005*. Gartner Group, 2005.

[35] S. J. Fink and F. Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *International Symposium on Code Generation and Optimization*, pages 241–252, San Francisco, California, Mar. 2003.

[36] S. Gilmore, D. Kirli, and C. Walton. Dynamic ML without Dynamic Types. Technical Report ECS-LFCS-97-378, LFCS, University of Edinburgh, December 1997.

[37] D. Gupta. *On-line Software Version Change*. PhD thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur, Nov. 1994.

[38] D. Gupta, P. Jalote, and G. Barua. A Formal Framework for On-line Software Version Change. *IEEE Transactions on Software Engineering*, 22(2):120–131, 1996.

[39] C. M. Hayden, E. A. Hardisty, M. Hicks, and J. S. Foster. Efficient Systematic Testing for Dynamically Updatable Software. In *ACM Workshop on Hot Topics in Software Upgrades*, pages 9:1–9:5, Orlando, Florida, Oct. 2009.

[40] M. Hicks. *Dynamic Software Updating*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, August 2001.

[41] M. Hicks and J. S. Foster. Adapting Scrum to managing a research group. Technical report, University of Maryland, Department of Computer Science, June 2008.

[42] G. Hjálmtýsson and R. Gray. Dynamic C++ Classes, A lightweight mechanism to update code in a running program. In *USENIX Annual Technical Conference*, pages 65–76, New Orleans, Louisiana, June 1998.

[43] U. Hölzle and D. Ungar. A third-generation SELF implementation: reconciling responsiveness with performance. In *ACM SIGPLAN Conference on Object-oriented Programming Systems, Language, and Applications*, pages 229–243, Portland, Oregon, Oct. 1994.

[44] Internet Storm Center. Survival Time. `http://isc.sans.org/survivaltime.html`.

[45] Jetty webserver project. Powered by jetty. `http://docs.codehaus.org/display/JETTY/Jetty+Powered`, Dec. 2009.

[46] R. E. Jones and R. D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, July 1996.

[47] M. Jump and K. S. McKinley. Cork: Dynamic Memory Leak Detection for Garbage-Collected Languages. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 31–38, Nice, France, 2007.

[48] Y.-F. Lee and R.-C. Chang. Hotswapping Linux kernel modules. *Journal of Systems and Software*, 79(2):163–175, 2006.

[49] D. E. Lowell, Y. Saito, and E. J. Samberg. Devirtualizable Virtual Machines enabling general, single-node, online maintenance. In *ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–223, Boston, Massachusetts, Oct. 2004.

[50] K. Makris. *Whole-Program Dynamic Software Updating*. PhD thesis, Arizona State University, December 2009.

[51] K. Makris and R. Bazzi. Multi-Threaded Dynamic Software Updates Using Stack Reconstruction. In *USENIX Annual Technical Conference*, pages 397–410, San Diego, California, June 2009.

[52] K. Makris and K. D. Ryu. Dynamic and Adaptive Updates of Non-Quiescent Subsystems in Commodity Operating System Kernels. In *ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 327–340, Lisbon, Portugal, Mar. 2007.

[53] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime Support for Type-Safe Dynamic Java Classes. In *AITO European Conference on Object-Oriented Programming*, pages 337–361, Sophia Antipolis and Cannes, France, June 2000.

[54] R. Marejka. Java ME Technology: Everything a Developer Needs for the Mobile Market. *Sun Developer Network*, July 1998. `http://java.sun.com/developer/technicalArticles/javame/mobilemarket/`.

[55] L. Mearian. IBM Builds Java System for NYSE. *Computer World*, Dec. 2004. `http://www.computerworld.com/s/article/98376/IBM_Builds_Java_System_for_NYSE`.

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

[57] M. Milazzo, G. Pappalardo, E. Tramontana, and G. Ursino. Handling run-time updates in distributed applications. In *ACM Symposium on Applied Computing*, pages 1375–1380, Santa Fe, New Mexico, Mar. 2005.

[58] D. Mosberger and T. Jin. httperf: A Tool for Measuring Web Server Performance. *SIGMETRICS Performance Evalualtion Review*, 26:31–37, December 1998.

[59] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 265–276, Washington, DC, Oct. 2009.

[60] I. Neamtiu. *Practical Dynamic Software Updating*. PhD thesis, Department of Computer Science, University of Maryland, August 2008.

[61] I. Neamtiu, J. S. Foster, and M. Hicks. Understanding Source Code Evolution Using Abstract Syntax Tree Matching. In *International Workshop on Mining Software Repositories*, pages 1–5, St. Louis, Missouri, May 2005.

[62] I. Neamtiu and M. Hicks. Safe and Timely Dynamic Updates for Multi-threaded Programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–24, Dublin, Ireland, June 2009.

[63] I. Neamtiu, M. Hicks, J. S. Foster, and P. Pratikakis. Contextual Effects for Version-Consistent Dynamic Software Updating and Safe Concurrent

Programming. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 37–50, San Francisco, California, Jan. 2008.

[64] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical Dynamic Software Updating for C. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 72–83, Ottawa, Canada, June 2006.

[65] A. Nicoara, G. Alonso, and T. Roscoe. Controlled, Systematic, and Efficient Code Replacement for Running Java Programs. In *ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 233–246, Glasgow, Scotland, Apr. 2008.

[66] 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.

[67] A. Orso, A. Rao, and M. J. Harrold. A Technique for Dynamic Updating of Java Software. In *IEEE International Conference on Software Maintenance*, pages 649–658, Montreal, Canada, October 2002.

[68] Y. Padioleau, J. L. Lawall, and G. Muller. Understanding Collateral Evolution in Linux Device Drivers. In *ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 59–71, Leuven, Belgium, Apr. 2006.

[69] S. Parker. A simple equation: IT on = Business on. *The IT Journal*, 2001.

[70] F. Pizlo. Personal communication, based on experience at Fiji Systems LLC., Dec. 2009.

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

[72] T. Ritzau and J. Andersson. Dynamic Deployment of Java Applications. In *Java for Embedded Systems Workshop*, London, May 2000.

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

[74] Slashdot forum. Patch the Kernel Without Reboots. `http://tech.slashdot.org/article.pl?sid=08/04/24/1334234`, Apr. 2008. Consists of a lively technical debate about the benefits and drawbacks of in-place dynamic updates vs. using redundant hardware.

[75] C. Soules, J. Appavoo, K. Hui, D. D. Silva, G. Ganger, O. Krieger, M. Stumm, R. Wisniewski, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis. System Support for Online Reconfiguration. In *USENIX Annual Technical Conference*, pages 141–154, San Antonio, Texas, June 2003.

[76] Sourceforge.net. CrossFTP Server. `http://sourceforge.net/projects/crossftpserver`.

[77] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. *Mutatis Mutandis*: Safe and Flexible Dynamic Software Updating (full version). *TOPLAS*, 29(4):22, Aug. 2007.

[78] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic Software Updates: A VM-centric Approach. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, Dublin, Ireland, June 2009.

[79] Sun Developer Network Bug Database. Enhance Hot Code Replacement. `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4910812`, Aug. 2003.

[80] Sun Developer Network Bug Database. Top 25 RFE's (Request for Enhancements). `http://bugs.sun.com/bugdatabase/top25_rfes.do`, Apr. 2010.

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

[82] E. Tempero, G. Bierman, J. Noble, and M. Parkinson. From Java to UpgradeJ: an empirical study. In *ACM Workshop on Hot Topics in Software Upgrades*, pages 1–5, Nashville, Tennessee, Oct. 2008.

[83] The Jikes RVM Core Team. VM Performance Comparisons, 2007. `http://dacapo.anu.edu.au/regression/perf/head.html`.

[84] A. van Hoff. JmDNS: Java implementation of multi-cast DNS. `http://jmdns.sourceforge.net`.

[85] Vision Solutions. Understanding Downtime. May 2006.

[86] C. Walton. *Abstract Machines for Dynamic Computation*. PhD thesis, University of Edinburgh, 2001. ECS-LFCS-01-425.

[87] D. Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`.

[88] Wikipedia. Jetty Webserver. `http://en.wikipedia.org/wiki/Jetty_(web_server)`, 2010. [Online; accessed 22-January-2010].

[89] Wikipedia. Source lines of code. `http://en.wikipedia.org/wiki/Source_lines_of_code`, 2010. [Online; accessed 25-January-2010].

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

[91] ZeroTurnaround. JRebel. `http://www.zeroturnaround.com/jrebel`.

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

# Vita

Suriya Subramanian graduated from high school in 1999. He received a Bachelor of Engineering degree from College of Engineering, Guindy, Anna University in 2003. He received a Master of Science degree in Computer Sciences from the The University of Texas at Austin in 2006.

Permanent address: "Sri Ram Nilayam"
Plot No. 12, C R Ramakrishnan Puram (East)
Virugambakkam
Chennai - 600 092
Tamil Nadu
India

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.