# Rubah: DSU for Java on a stock JVM

Luís Pina Luís Veiga
Instituto Superior Técnico, Universidade de Lisboa
INESC-ID Lisboa
luis.pina@tecnico.ulisboa.pt luis.veiga@inesc-id.pt

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

## Abstract

This paper presents Rubah, the first *dynamic software updating* system for Java that: is *portable*, implemented via libraries and bytecode rewriting on top of a standard JVM; is *efficient*, imposing essentially no overhead on normal, steady-state execution; is *flexible*, allowing nearly arbitrary changes to classes between updates; and is *non-disruptive*, employing either a novel eager algorithm that transforms the program state with multiple threads, or a novel lazy algorithm that transforms objects as they are demanded, post-update. Requiring little programmer effort, Rubah has been used to dynamically update five long-running applications: the H2 database, the Voldemort key-value store, the Jake2 implementation of the Quake 2 shooter game, the CrossFTP server, and the JavaEmailServer.

***Categories and Subject Descriptors*** C.4 [*Performance of Systems*]: Reliability, availability, and serviceability

***Keywords*** Dynamic Software Updating; Java; JVM

## 1. Introduction

As on-line services go global, an increasing number of systems require constant availability, and as a matter of convenience many other systems would prefer it. A common technique for ensuring high availability is *rolling upgrades*, enabled by a load balancer that distributes requests among many back-end servers. These servers can be taken off-line on a rolling basis when they become idle, and then upgraded and re-entered into service. For this approach to work, interesting state must be kept external to the server (e.g., in a DBMS) and connections must be fairly short lived (so that servers quickly become idle). These requirements are sometimes infeasible or too inefficient.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*OOPSLA '14*, October 20–24, 2014, Portland, OR, USA.
Copyright © 2014 ACM 978-1-4503-2585-1/14/10. . . $15.00.
http://dx.doi.org/10.1145/2660193.2660220

An alternative approach to rolling upgrades is *dynamic software updating* (DSU). This technique works by updating a process *in place*, patching the existing code and transforming the existing in-memory state. By not shutting down the updated program, DSU addresses the shortcomings of rolling upgrades. First, it preserves active, long-running connections (e.g., to databases, media streaming, FTP and SSH servers), which can immediately benefit from important program updates (e.g., security fixes). Second, it preserves in-memory server state. Doing so is extremely valuable for in-memory databases, gaming servers and many other systems, that rely on the relatively low expense and high performance of commodity RAM, to maintain large data sets in the heap. This problem is acute enough that Facebook uses a custom version of memcached that keeps in-memory state in a ramdisk to which it reconnects on a post-update restart [2].

General-purpose systems developed for C and C++ have been applied to dozens of realistic applications, tracking changes according to those applications release histories [3–6]. Increasingly, important on-line services are written in managed languages like Java. For example, Twitter has moved most of its major infrastructure to Java [7], and the Java-based Voldemort noSQL database is used by companies like Linkedin. While several DSU systems for Java have been developed [8–10] they all have shortcomings that inhibit practical usage.

This paper presents Rubah, the first full-featured, portable DSU system for Java with good performance. Rubah implements DSU as *whole program updates*, in the style of Kitsune [6], a DSU system for C we developed previously.[1] Compared to prior DSU systems for Java, Rubah has several advantages (further comparisons are in Section 6):

- Rubah works by bytecode rewriting, enhancing its portability; no changes to the underlying JVM are required, unlike past systems such as Jvolve [10], the DVM [9], and JDrums [8].
- Rubah is extremely flexible, handling release-level updates. As far as we are aware, no prior system can handle the same range of updates Rubah can.

[1] Kitsune is the Japanese word for *fox*, a shape shifter. Rubah is the Indonesian word for fox; natives of the island of Java speak Indonesian.

- Rubah enjoys good steady-state performance: supporting updating imposes negligible (-1.0–2.5%) overhead on normal execution for our benchmarks when using a production-quality VM, whereas prior systems either did not work with production VMs (Jvolve used Jikes) or imposed high overheads (e.g., DuSTM [11] imposed overheads of more than 50% on similar benchmarks).

In addition, Rubah uses two novel algorithms to reduce the pause in application execution while the application's state is being transformed. Rubah's *parallel* algorithm speeds up the standard algorithm by parallelizing it. Rubah's *lazy* algorithm injects *proxy objects* that mediate access to outdated instances; when accessed, the proxy triggers the target object's transformation and then removes itself, to avoid adding further indirection overhead. The proxy implementation and data structures are wait-free, which means that the original program cannot deadlock/livelock due to an update.

We have used Rubah with Oracle's production HotSpot VM to dynamically update five long-running applications: the H2 SQL relational database; the Voldemort key-value store, used in practice by LinkedIn; Jake2, a Quake2 port translated to Java; CrossFTP, an FTP server written in Java; and JavaEmailServer, an POP3/SMTP server written in Java. We modified these applications by hand to support updates in Rubah and found that the amount of effort required correlates with the application's control-flow structure rather than with the application's size. This effort added from 29 to 267 lines of code and is a one-time effort: Once the first version supports Rubah, subsequent versions require little, if any, modification. We also wrote code to transform the state between four versions of CrossFTP, three versions of H2 and JavaEmailServer, and two versions of Voldemort. This effort must be done for each supported version and is a function of the number of classes with a different representation between versions. Rubah automates the majority of this process and so we only had to write a total of 114 lines of code for all the updates we tested.

Performance experiments using benchmarks for Voldemort, H2, and CrossFTP found that the overhead Rubah imposes (-1.0%–2.5%) is well within the noise on modern systems [12]. We also found that our state transformation algorithms reduce the update-time pause. The parallel algorithm performed nearly 4 times faster than the single-threaded version for larger heaps. However, for larger heaps, the total pause time can still be high (tens of seconds to minutes). By contrast, when using the lazy algorithm on real updates to H2 and Voldemort, the pause time was typically 2–3 seconds, regardless of the heap size, and the application recovered 90% of the steady-state performance in 30 seconds or less.

In summary, Rubah represents the first portable, performant, full-featured DSU system developed for Java, and represents an important step toward practical use.

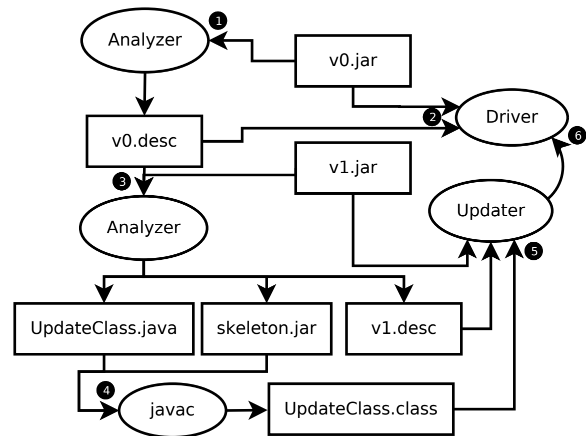



**Figure 1.** Deploying a program, and preparing and installing an update for it, using Rubah. Square boxes represent artifacts: Compiled code (jar/class), source code (java), or update descriptors (desc). Round boxes represent tools: Rubah's driver, analyzer, and updater, and the unmodified Java compiler (javac).

## 2. Dynamic Software Updating with Rubah

This section describes how Rubah supports dynamic updating for Java programs, with its design inspired by Kitsune's approach for C [6], but employing new algorithms (Section 3) and a novel implementation strategy (Section 4).

### 2.1 Workflow

The workflow for using Rubah is given in Figure 1. Prior to deploying the initial version of a program (which we call "version 0" or $v_0$), that version's code (`v0.jar`) is given to the Rubah *analyzer* tool, which produces a *version descriptor* (`v0.desc`) that contains meta-data, such as the list of all updatable classes, for that version. The program is executed by Rubah's *driver*, which takes the application's classes and the descriptor. The driver uses a custom classloader that intercepts each class that the application loads and performs a semantics-preserving bytecode transformation that adds support for future updates to the loaded class, most notably in support of state transformation, discussed below.

Once a new version of the program is available (which we call "version 1", or $v_1$), the developer prepares a dynamic update by passing the new code (`v1.jar`) and the $v_0$ descriptor to the analyzer, which produces, along with the $v_1$ descriptor, an *update class* (`UpdateClass.java`) that describes how existing objects should be changed to work with the new code. The programmer can customize this class as needed, and then compile it using the analyzer-produced `skeleton.jar` as a placeholder for the old-version classes.

The dynamic update is deployed by the *updater*, which signals the running driver, providing the new code and the update class. The driver then deploys the update in three stages. In the first stage, *quiescence*, the driver gets each thread to a point at which it is safe to perform the update.

In the second stage, *state transformation*, the driver initiates (and may complete) the modification of object instances whose class changed (according to the update class). In the final stage, *control flow migration*, each thread is restarted and shepherded to a point equivalent to the one at which the update took place. At this point, the update is logically complete. Future versions repeat steps 3–6 in the figure.

This approach is extremely flexible. Rubah permits changing any class in an arbitrary manner, with few exceptions, whereas past approaches often limit which classes can be changed, and in what ways. For Rubah, the only classes that cannot be updated are the Java runtime classes and libraries (e.g., Java collections). *Updatable classes* can directly reference non-updatable classes but not the reverse, due to issues involving the bootstrap class path of a Java application [13]. Of course, library classes do not directly reference application classes, so this restriction poses no practical difficulty.

Rubah requires the programmer to write (or retrofit) the program so that the update process works properly. In particular, to help with quiescence, the programmer must insert *update points* that identify safe moments to perform updates. The programmer must also add code to perform control flow migration. Finally, for each new version that comes out, the programmer may also need to customize the default update class. In the remainder of this section we describe what must be done, using an example.

### 2.2 Example

Figure 2 shows a simplified version of a method from the H2 database that we modified to support updating. The changes we made are highlighted. While much of figure constitutes modifications, bear in mind that most of the application logic is in methods like `process`, which require no changes. In our experience, code changes to support updating are small, requiring on the order of 100 lines of code, and stable, typically requiring no changes between versions.

Ignoring the highlighted code for now, we can see that this method handles client connections. The method starts by parsing the client data and negotiating the protocol parameters (lines 3 to 8). Then, it executes every client command by calling method `process` (line 14) inside an infinite loop (lines 11 to 22). Method `process` blocks until the client issues the next command, executes that command, and returns.

Note the complex handling of exceptions, typical in server methods. All recoverable exceptions thrown inside the `process` method are sent back to the client (line 20), and non-recoverable exceptions are logged (line 27, which catches exceptions re-thrown by the `sendError` method). A **`finally`** block ensures that the connection is closed when the server method exits (line 30).

### 2.3 Update Points and Quiescence

The light gray highlighted code in the figure is related with update points [3]. In Rubah, update points are simply calls to method **`Rubah`**`.update`. This method takes a string as its sole argument, which is a label intended to identify logically distinct program points. The example in Figure 2 shows an update point placed on line 13. This is a good place to put an update point because the program is *quiescent*:[2] At this point it has finished processing the last client command and has not started to process the next one. State relevant to an update is not in the middle of being modified.

```
1  public void run() {
2    if (!Rubah.isUpdating()) {
3      transfer.init();
4      trace("Connect");
5      // Parse client version
6      // Negotiate protocol params
7      transfer.flush();
8      trace("Connected");
9    }
10   try {
11     while (!stop) {
12       try {
13         Rubah.update("process");
14         process();
15       } catch (UpdateRequestedException e) {
16         continue;
17       } catch (UpdatePointException e) {
18         throw e;
19       } catch (Throwable e) {
20         sendError(e);
21       }
22     }
23     trace("Disconnect");
24   } catch (UpdatePointException e) {
25     throw e;
26   } catch (Throwable e) {
27     server.traceError(e);
28   } finally {
29     if (!Rubah.isUpdateRequested())
30       close();
31   }
32 }
```

**Figure 2.** Example adapted from H2 TcpServerThread featuring logic related with update points (gray highlight) and control-flow migration (black highlight).

We designed Rubah to be directly applicable to programs with a control-flow structure similar to this example, which in our experience is typical of high-availability programs [15], i.e., each thread has a long-running loop to process internal or external requests, and the head of this loop is a point where the program is naturally quiescent.

When an update becomes available, the program may be blocked waiting for some I/O operation. To avoid an undue delay to the update, Rubah requires the program to either: (1) Use non-blocking sockets and `select` operations, which are blocking but can be interrupted without closing the socket [16]; or (2) have each thread voluntarily wake-up from I/O calls frequently and reach an update point be-

[2] Note that our definition of quiescence differs from (and is not comparable to) that of some prior work [14], which defines it to mean that all updated functions are inactive, i.e., not running.

fore blocking again. Rubah provides an API that simplifies retrofitting a program to use non-blocking I/O, if needed; we used this API for the H2 database, CrossFTP server, and JavaEmailServer (see Section 5.1). In the example, `process` throws an `UpdateRequestedException` if interrupted by an update. This exception is caught on line 16 and the loop soon reaches the update point on line 13.

Method **`Rubah`**`.update` throws an `UpdatePointException` when an update is available; unhindered, this exception will ultimately reach a Rubah-provided wrapper for a thread's `run` (or `main`) method, where it is caught and dealt with. The thread wrapper is implemented in the class `RubahThread`, which is a drop-in replacement for class `java.lang.Thread` that applications must use. Of course, the exception may be caught by intervening **`catch`** blocks in the application, so the developer may need to make changes to avoid this (lines 18 and 25). The developer also needs to ensure that the exception does not change any state by being propagated, therefore actions within finally blocks must be guarded to account for possible updates (line 29).

When all threads have been stopped at update points, the program is quiescent, and the update may take place. This happens in two steps: *state transformation*, which loads in new and updated classes and transforms existing objects to use those new classes, and *control-flow migration*, which returns the threads to their logically correct positions in the (new) application code. We defer discussion of state transformation to Section 2.5 and discuss control-flow migration next, completing the explanation of the code in our example.

### 2.4 Control-flow Migration

The goal of the control-flow migration is to return each program thread to an update point in the new version that is equivalent to the point at which it stopped in the previous version. Rubah begins control-flow migration by re-starting each thread's (possibly updated) `run` method (or the `main` method for the main thread, if it is still alive). Each thread eventually reaches, and blocks at, an update point with same label as the update point at which the thread quiesced originally. Once all threads have so blocked, control-flow migration is complete, and all threads may continue. Besides application threads and the main thread, Rubah also supports control-flow migration of thread pools.

When a thread starts for the first time, it typically performs initialization actions that should not be re-performed during control-flow migration. In our example, lines 3 to 8 negotiate protocol parameters with the client, and this negotiation should not be repeated, post update. To avoid initialization code, Rubah provides API calls that the developer can use to determine whether a thread is running for the first time or as a result of an update. In our example, line 2 guards the initialization code with a call to **`Rubah`**`.isUpdating` which returns **`true`** if called while performing the control-flow migration and **`false`** otherwise.

```
1 class UpdateClass {
2   void convert(
3       v0.org.h2.store.PageStore o0,
4       v1.org.h2.store.PageStore o1) {
5     o1.readCount = 0L;
6     o1.writeCount = 0L;
7     o1.writeCountBase = o0.writeCount;
8   }
9 }
```

**Figure 3.** Example adapted from H2 of an update class with a single instance conversion method.

Note that some systems, like UpStare [17], attempt to perform control-flow migration automatically. Following Kitsune, Rubah prefers the manual approach because (a) it makes the updating process manifest in the program code and thus easier for the programmer to reason about, and (b) it imposes less overhead than would full support for program-wide stack unwinding and rewinding (as in UpStare).

### 2.5 State transformation

Prior to restarting each thread, Rubah performs state transformation to convert the existing program's objects to use the updated classes. Conceptually, this happens by visiting each object in the heap that might have been affected by an update and *transforming* it to work with the new version's code. In most cases this transformation is simple; e.g., version $v_0$ of a class has two fields while version $v_1$ has three, and the newly added field is initialized with its default value. In rare cases the transformation is more involved, and so the programmer can specify what to do in the update class.

Figure 3 shows an example of an update class, which specifies the transformation. This example has a single *instance conversion method* that transforms instances of class `org.h2.store.PageStore` by taking an existing instance `o0` that belongs to version $v_0$ and using it to initialize the equivalent new instance `o1` that shall take `o0`'s place in $v_1$.

Update classes have one instance conversion method for each class that has a different set of fields from version $v_0$ to version $v_1$. Even if the set of fields is the same, with regards to name and type, the developer can define instance conversion methods to account for fields whose semantics changes. If a field has changed neither name nor type, then Rubah copies its value from the old to new version by default; the developer can override this behavior by assigning to the field in the conversion method. Update classes may also define *static conversion methods* to transform static fields.

In Figure 3, field `writeCount` in $v_0$ tracks the total number of bytes ever written to a particular store. Version $v_1$ renames field `writeCount` to `writeCountBase` and introduces two new fields, `readCount` and `writeCount`, that track how many bytes were read/written since the store was opened. The transformation code copies the value from

the renamed field in line 7 and sets to zero the two new fields in lines 5 and 6. This transformation code makes the store act as if it was opened when the update took place.

The arguments of the conversion method in Figure 3 are *skeleton classes*, which as the name implies, have been stripped of a lot of the original's contents: all methods are removed, and all fields are made public (so as to be accessible to the update class's code). Each class is placed in a distinct namespace, depending on its version, allowing the developer to refer to version $v_0$ or $v_1$ unambiguously and still use the regular Java compiler to compile the update class.

Rubah's analyzer generates a default update class that the programmer may customize. The analyzer compares $v_0$ and $v_1$ and matches fields by owner class name, field name, and field type. It generates a conversion method for each class with unmatched/changed fields that initializes those fields to a default value (`0`, **`false`**, or **`null`**). The developer then "fills in the blanks."

Rubah's state transformation algorithms are responsible for finding outdated instances and updating them via the update class. We have developed two algorithms, a *parallel* one and a *lazy* one, which have different tradeoffs. These algorithms constitute one of the main contributions of this work, and so are discussed in detail in the next section.

Once the programmer has suitably modified the update class, the update can be tested for correctness, e.g., using the strategies proposed in our prior work [15, 18]. In brief: For tests whose outcome should be the same in both the old and new version (e.g., because the tested features are unaffected by the update), we can run the test and update the program at various moments during the test, making sure the test still passes. For tests relevant only to the new version, e.g., because they test a new feature, we can establish the initial state using old-version commands, updating at any time while doing so, and then complete the test at the new version, ensuring it still passes. We may need to construct new tests when the new-version behavior is incompatible with old-version behavior but the update supports a kind of intermediate view. This could happen, say, when the new version imposes a connection limit, but that limit is exceeded when the update takes place; we might nevertheless like to preserve existing connections until going below the limit. A new test would have to be written to establish that the update does indeed ensure this behavior. More details can be found in our prior papers.

## 3. State Transformation Algorithms

Once the current version of the program becomes quiescent, Rubah initiates the process of transforming the program's *outdated* objects, that is, those whose (own or ancestor) class has been changed by the update.

Rubah supports two novel state transformation algorithms. The first, *parallel* algorithm transforms all outdated objects eagerly, using multiple threads, while the program is stopped. The second, *lazy* algorithm transforms each outdated object as late as possible, just before the program attempts to use the object after the update takes place. This section describes each algorithm in detail.

### 3.1 Notation

The algorithms are presented in Java-like pseudocode, with the differences from Java made for readability:

- Brackets are omitted, and indentation determines scope.
- We use a map `visited` to keep track of visited objects. The map associates outdated objects with their transformed versions (or with themselves, if they have not changed). We write `visited[key] = val` to associate `key` with `val`, and retrieve the current mapping by writing `visited[key]`; if no mapping for `key` (yet) exists, this expression yields $\perp$.
- Visiting each field in an object, used to compute the transitive closure of the object graph, is written using notation: `for (Field f : obj)` ... `obj.f` ....
- We use atomic *compare and swap* (CAS) to ensure safe concurrency. The expression `CAS(lval, expectVal, setVal)` atomically sets the *l-value* `lval` to `setVal` assuming that `lval`'s contents are currently `expectVal`, in which case `setVal` is returned, otherwise the current contents are. Thus, if `obj.f=0`, then the expression `CAS(obj.f,0,1)` sets `obj.f` to be `1`, and returns `1`, at which point the expression `CAS(obj.f,0,2)` would make no change to `obj.f` and return `1`. We assume the map supports atomic semantics so that `map[key]` can be used as an l-value, i.e., `CAS(map[key],expect, newKey)` denotes an atomic map insertion.

We explain how we actually implement some of these notational conveniences in our prototype in the next section.

### 3.2 Parallel state transformation

The simplest way to transform the program state is to do so *eagerly*, while the program is stopped. A single thread can, starting from the root references, follow each object reference transitively until all the program state is visited and transformed. This is very similar to a *stop-the-world* tracing garbage collection algorithm [19], and is used by many DSU systems [3, 6, 9, 10]. We improve on this basic idea by performing tracing in parallel, using multiple threads.

For the purposes of state transformation, we consider the root references to be the static fields in all loaded classes and the fields in all stopped `java.lang.Thread` objects. We do not consider local variables to be roots, as their stacks are unwound during quiescence; our experience (with Rubah and Kitsune [6]) is that values in locals at update-time are rarely needed, but if they are the programmer can store them away (e.g., in a hashtable) temporarily.

```
 1  Map visited;
 2  TaskQueue queue;
 3
 4  migrate (Object obj) =
 5    if (visited[obj])
 6      return visited[obj];
 7    Class c = obj.getClass();
 8    Class newC = Rubah.mapClass(c);
 9    Object newObj;
10    if (newC != c)
11      newObj = Rubah.new(newC);
12      Rubah.convert(obj, newObj);
13    else
14      newObj = obj;
15    Object mapped = map(obj, newObj);
16    if (mapped != newObj)
17      return mapped;
18    traverse(newObj);
19    return newObj;
20
21  ST:traverse (Object obj) =
22    for (Field f : obj)
23      obj.f = migrate(obj.f);
24
25  ST:map(Object pre, Object post) =
26    visited[pre] = post;
27    return post;
28
29  MT:traverse (Object obj) =
30    for (Field f : obj)
31      Task t = new Task()
32        { obj.f = migrate(obj.f); }
33      queue.add(t);
34
35  MT:map(Object pre, Object post) =
36    return CAS(visited[pre], ⊥, post);
```

**Figure 4.** Parallel state migration algorithm

Figure 4 shows the parallel state transformation algorithm as well as a single-threaded variant, for comparison. The main code is in the `migrate` method. The `traverse` and `map` methods differ for the single- and multi-threaded variants, and their code is prefixed with labels `ST` and `MT`, respectively, in the figure.

The algorithm calls `migrate(o)` for each root object `o`. This method starts by looking up the object in the map (line 5). If not present, it proceeds to map the old class to the new one by calling method **`Rubah`**`.mapClass` (line 8) which, for argument class `c`, returns either class `c` if the update does not modify `c`; or the updated version of outdated class `c`. For outdated objects, the algorithm creates an instance of the new class, and transforms the object (lines 11 and 12).

**`Rubah`**`.convert` calls instance conversion methods in a hierarchical way similar to how Java calls constructors [20]. Let us consider the case in which classes `A` and `B` are updatable, class `B` extends `A`, class `N` is non-updatable, and class `A` extends `N`. In this case, to transform instances of class `B`, Rubah: (1) copies all fields inherited from class `N`, (2) copies all unchanged fields from class `A`, (3) calls `A`'s conversion method to transform `A`'s updated fields, (4) copies all unchanged fields from class `B`, and (5) calls `B`'s conversion method to transform `B`'s updated fields.

After transforming the object, the algorithm marks the object as visited (lines 15 to 17) and traverses the transformed object (line 18). In the single-threaded variant of the algorithm, traversal is done by the method `ST:traverse`, which simply calls `migrate` for every field that the object has. In this variant, `ST:map` simply updates an object in the map, so the condition on line 16 is always false.

The multi-threaded algorithm uses a **`TaskQueue`** to coordinate state transformation among multiple threads. The multi-threaded object traversal (method `MT:traverse`) creates tasks to do object transformation for each field (line 33). Each task, itself, creates further tasks and the algorithm finishes when the task threads complete with an empty queue.[3]

Multiple threads read from and write to the `visited` map concurrently and there is a possibility of races. In particular, it is possible for one thread to read the map (line 5), find it empty, and then create a new object to store in the map (line 15). Before this new object is stored in the map, however, another thread could follow the same path, and ultimately overwrite the object stored by the first thread, leading to an inconsistency in the transformed heap.

The multi-threaded algorithm solves this problem by using **`CAS`** in its `MT:map` implementation. If the **`CAS`** attempts to write the new object but finds the map does not contain $\perp$, and thus another thread won the race to write an object there, it simply returns the existing object. Then, on line 16 of `migrate`, the object `mapped` is different from `newObj` and so `migrate` simply completes since the existing object has already been set.

### 3.3 Lazy state transformation

Lazy state transformation takes place while the program is running. The goal is to postpone the transformation of each object to the last possible moment. Laziness avoids the significant pause that would otherwise occur for large heaps.

To implement lazy transformation, Rubah uses proxies to intercept control when the program is about to dereference an object *o* (i.e., read/write one its fields or call a method) that is not completely up to date. The proxy does the necessary work to bring the object up to date before allowing the program to continue. To simplify the presentation, we present the algorithm as if every object can behave like a proxy to itself by setting a flag, rather than using a separate object. The start of each method is modified to check the proxy flag, and perform the necessary work if the flag is set, before executing the original method body.[4] We discuss the actual implementation in Section 4.3.

[3] We notate tasks (line 31 to line 32) using braces, which form the boundary of a closure: `obj` and `f` are free variables inside task `t` resolved to those in the lexical scope (i.e., the variables defined in lines 29 and 30, respectively).

[4] We discuss how Rubah handles all other ways in which proxies may be dereferenced, such as field access, in Section 4.3.

```
1  Map visited;
2
3  LAZYmigrate (Object obj)
4    LAZYtraverse(obj);
5    obj.isProxy = false;
6    visited[obj] = obj;
7
8  LAZYtraverse (Object obj)
9    for (Field f : obj)
10     Object ref = obj.f;
11     if (ref.isProxy)
12       continue;
13     else if (visited[ref])
14       CAS(obj.f, ref, visited[ref]);
15       continue;
16     else
17       Class c = ref.getClass();
18       Class newC = Rubah.mapClass(c);
19       if (c != newC)
20         Object p = Rubah.new(newC);
21         Rubah.convert(ref, p);
22         p.isProxy = true;
23         p = CAS(visited[ref], ⊥, p);
24         CAS(obj.f, ref, p);
25       else
26         ref.isProxy = true;
27
28 Object method(Object ... args)
29   if (this.isProxy)
30     LAZYmigrate(this);
31   // Rest of original method
```

**Figure 5.** Lazy conversion algorithm. Note that this algorithm assumes that all field accesses from outside a class are via methods (we validate this assumption in our implementation).

#### 3.3.1 Correctness conditions

Any state transformation algorithm is correct if, once the program threads are restarted after reaching quiescence, they only run up-to-date code and access up-to-date state. This is trivially true for the parallel algorithm.

For the lazy algorithm to guarantee correctness, it ensures that the restarted threads will only ever use objects that are *safe to access*.

**Invariant 1:** After the update, the program only uses objects that are safe to access

An object `o` is safe to access if and only if (1) `o`'s class is not outdated, and (2) either `o` is a proxy (i.e., its proxy flag is set) or all of its fields are safe to access. By ensuring this invariant, we ensure that whenever the program uses an object, the object is either up to date or a proxy. In the latter case, the algorithm updates the proxied object's fields so that they are safe to access and uninstalls the proxy before letting the program use the (now up-to-date) object. This approach causes the object graph to have a clear *frontier* between the up-to-date and partially updated program state that is composed of proxies. This frontier starts at the root references and expands outward as more proxies get dereferenced.

In addition to this core invariant, the lazy algorithm maintains another important invariant, which is that all objects mapped to by `visited` are safe to access.

**Invariant 2:** If `visited[o] = p` then `p` is safe to access

As we shall see shortly, this invariant helps ensure that after converting a proxy object to one that is up to date, the latter is safe to access.

#### 3.3.2 Algorithm

Figure 5 shows the lazy state transformation algorithm. The algorithm first handles the roots by running the loop on line 9, where each field `obj.f` considered is a root reference (e.g., a **`static`** field). Lines 17 to 19 test if each referred object needs to be transformed. If not, the algorithm simply proxies the object (line 26). Otherwise, the algorithm creates an object of the new class without running any constructors (line 20), runs the conversion code to initialize the new object using the state of the outdated object (line 21), proxies the new object (line 22), marks the old object as visited (line 23), and sets the original reference to point to the new object (lines 24). (Recall from Section 3.1 that the first argument of **`CAS`** is treated as an l-value, not an r-value.) Note that assigning `visited[ref]` to `p` on line 23 satisfies invariant 2 because `p` is not outdated, and is a proxy. Objects are transformed only once: aliased proxies are skipped (lines 11–12) and aliased objects are set to the correct, safe-to-access object (lines 13–15). For now, assume that the **`CAS`** operations on lines 14, 23, and 24 always succeed; their role shall become evident later on this section.

At this point all root references refer to proxies. Invariant 1 is therefore true and Rubah can safely start running each paused thread's `run/main` method at the new version, beginning the process of control-flow migration. Assuming that all accesses to objects are via method calls, then the next method call on an object will be to a proxy. We assume all methods have been modified according to the bottom of the figure: the program calls method `LAZYmigrate` (line 30), which traverses the proxy (line 4) using `LAZYtraverse`. As explained above, this method ensures all of the proxy's fields are safe to access. Each field is already a proxy (line 11), is made into a proxy (lines 22 and 26), or was previously visited (line 13), in which case we update it with the new version from the map. Invariant 2 ensures that this new version is safe to access. Once a proxied object is traversed, `LAZYmigrate` uninstalls the proxy (line 5), and marks the object as visited (line 6) by mapping the object to itself. Doing so satisfies invariant 2 since the object's fields are all safe to access. This fact also ensures invariant 1 when, at line 31, we start running the object's code.

Now let us revisit the uses of **`CAS`** in the algorithm. If an object is aliased, several threads might find it concurrently and try to transform it. All these threads race to mark the outdated object as visited. We consider that objects become visited only when they are registered in the `visited` map

such that, for object o, visited[o] $\neq \perp$. Line 23 ensures that only one thread wins the race and all the threads use the same transformed object. As a consequence, the conversion methods that the developer writes may be called more than once for the same outdated object. Therefore, all the conversion methods must be idempotent.

There are two more uses of **CAS** we can justify with an example. Suppose two threads $T_1$ and $T_2$ race to mark the same object $o_1$ as registered, which is referred to by $o_2$.f. Furthermore, suppose $T_1$ wins and $T_2$ is not scheduled to run for a long while after executing line 23. $T_1$ finishes running method LAZYtraverse, then finishes running method LAZYmigrate, and then starts executing the new program version's code. Suppose that now $T_1$ performs $o_2$.f = $o_3$ while executing the program code. At this point $T_2$ runs again and executes line 24. $T_2$ cannot be allowed to perform $o_2$.f = p because that would overwrite $o_3$, thus changing the program's semantics and introducing an error. That is why line 24 has a **CAS** operation, **CAS**($o_2$.f, $o_1$, p), which in this case (correctly) fails for $T_2$. This race is also the reason for the **CAS** operation in line 14.

Assuming that the visited map is wait-free, this algorithm is trivially *wait-free*: all operations are guaranteed to finish in a bounded number of steps because there are no loops.

# 4. Implementation

Rubah is the first DSU system for Java that is both full-featured (flexibly handling release-level updates) and VM-independent. This section details how Rubah's driver actually performs a dynamic update once one becomes available. Our implementation is written in roughly 9KLOC of Java, and makes use of the ASM bytecode rewriting tool [1].

## 4.1 Name Mangling

Rubah renames updatable classes to distinguish those of different (past and future) versions. A class named AppClass gets renamed to AppClass__0 in version $v_0$ and AppClass__1 in version $v_1$. For brevity, in the following text we write $C_0$ for C__0 and $C_1$ for C__1. Changing the name of a class might break some reflection calls, such as **Class**.forName. Rubah rewrites all invocations of these methods to call Rubah's API instead (e.g. **Rubah**.classForName), which provides the same semantics and accounts for name mangling.[5]

## 4.2 Class replacement

After the updater signals that an update is ready (step 6 in Figure 1), the driver will load the new classes. Rubah generates a new class $C_1$ for each class $C$ in the new version of the program, even if class $C$ is unchanged from the previous version. As a consequence, Rubah must transform all instances of $C_0$ to instances of class $C_1$. But executing the state transformation algorithms for objects of classes that did not change would be inefficient. Rubah takes advantage of how the JVM lays out objects in memory to avoid such transformations.

HotSpot finds the class to which any object belongs by looking for that object's _klass reference at a fixed offset within the object (Jikes and OpenJDK have a similar object layout). If two classes $A$ and $B$ define the same fields in the same order, we can turn an instance of $A$ into an instance of $B$ by simply modifying the _klass reference.

Rubah uses the unsafe operations available in class sun.misc.Unsafe to manipulate _klass references. If the structure of a class $C$ does not change between versions, Rubah turns all instances of class $C_0$ it finds into instances of class $C_1$ by setting the _klass reference. Note that the bodies of the methods might change between versions. In this case, installing a new _klass also changes the vtable of the object, effectively installing the new methods. This technique is analogous to using HotSwap [21] to install new code for loaded classes. However, this technique does not require the JVM to run in debug mode, which we have found adversely affects JIT performance.

Changing the _klass reference of an object is potentially unsafe because the code that the JIT emits, e.g. when inlining, assumes that the _klass does not change. As such, changing the _klass could crash the JVM. We developed Rubah carefully to ensure that this violation only happens in methods inside of Rubah that never get inlined in the program's code and that such methods never perform any virtual method invocation that might reach the vtable. The GC also uses the _klass. However, it assumes far less than the JIT engine about the structure of the _klass and just looks for the relevant metadata at a fixed offset for all objects. Given that both the old and the new _klass agree on this metadata, this optimization does not cause the GC to crash the JVM.

## 4.3 State transformation

Our implementation of state transformation largely follows the algorithms given in Section 3, with two exceptions: (1) the visited map is often implemented as an added field rather than entirely as a separate data structure, and (2) the isProxy field is actually implemented by manipulating the _klass pointer to refer to a proxy class.

***Visited map.*** The visited map from Section 3 marks objects as visited and maps outdated $v_0$ object instances to their $v_1$ equivalents as they are transformed. Rather than implement the map entirely as a separate data structure, Rubah adds an extra instance field to updatable and non-updatable classes called $forward that points to an object's updated version. This approach adds a small per-object overhead, but avoids adding the extra memory pressure at update-time that

[5] The added version suffix should not be too confusing for developers to see during debugging, nor should replacing field accesses with accessor methods, as described in Section 4.3. The standard compiler already inserts accessor methods for use by inner classes to access containing-class private fields.

a separate data structure would impose. It also permits more fine-grained concurrency control: reading or writing the forwarding pointer can be done with a regular compare-and-swap operation.

Unfortunately, not all classes can be changed to add this new field. For instance, the JVM directly accesses the fields in all `java.lang.Reference` subclasses by their index. Adding a field changes the index and makes the JVM crash. Also, arrays cannot have extra fields. In these cases, Rubah uses an adaptation of `java.util.concurrent.ConcurrentHashMap` that provides the same semantics as `java.util.IdentityMap`. This map supports an atomic operation that checks if a key is present, otherwise inserting a mapping in a single step.

***Lazy Proxies.*** Section 3.3 suggests that a proxy is just an object whose added `isProxy` flag is set, where the flag changes how methods work. Checking this flag would degrade performance at the entrance of every method. Furthermore, given that the JVM JIT optimizer aggressively inlines small methods, the flag check would increase the code size of all methods, making the JIT compiler miss inline opportunities for small methods and thus generate slower code. It would also require an extra field in every class.

Instead, Rubah generates a proxy class to hold the proxy code and turns regular objects into proxies, and proxies back into regular objects, by manipulating the `_klass` reference with unsafe operations in the same way we describe in Section 4.2.

Rubah generates a proxy class $C_P$ for each class $C$ that it loads. $C_P$ extends $C$ and overrides all of $C$'s methods, redirecting the control flow to Rubah's API.[6] Thus, proxies inherit the fields of the classes they extend, having the same layout as the object they proxy, with the only difference between an object and a proxy is the vtable it keeps.

Changing the vtable through the `_klass` pointer makes proxies intercept virtual method invocations. However, besides those, proxy classes must also intercept other ways that the proxied object might be manipulated, which are field accesses and non-virtual method calls.[7]

For field accesses, Rubah rewrites all field accesses so that they are made through accessor methods, which can be overriden and intercepted by proxy objects. When such accessors are called from within the class's own methods, the JIT safely optimizes the call away by inlining; the only overhead will be due to accesses from outside the class.

[6] Rubah removes all **`final`** modifiers from classes and methods (but not fields) it loads to ensure that every class and method can be proxied. There are some classes in the `java.lang` package that do not support this, such as `java.lang.String`, but these classes are never proxied.

[7] In Java, calls to private methods are non-virtual, as are calls to methods via **`super`**.

For non-virtual calls, there is no issue if the call is made via **`this`** or **`super`**, since the current object cannot be a proxy. The only time the receiver of a non-virtual call can be a proxy is when invoking a private method of a different object (having the same class). Rubah places a check before the invocation to ensure that the other object is not a proxy; if it is, it must be transformed. This case is very rare and the extra call does not add any measurable overhead in practice.

***Wait freedom.*** The pseudocode of the algorithm given in Figure 5 checks the `isProxy` flag on line 29 to determine if an object is a proxy, and calls `LAZYmigrate` if so, before continuing with the body of the original method. In our actual implementation, the `_klass` pointer has been modified to point to a proxy class whose methods consist simply of a call to `LAZYmigrate`, followed by a call to **`this`**`.method(args)`. Because `LAZYmigrate` resets the `_klass` pointer to that of the original object (equivalent to the resetting of the `isProxy` flag on line 5 in the pseudocode), this call executes the correct method. However, there is a possibility that another thread could re-proxy the object after line 5 executes. As such, we can view the conditional on line 29 as being a **`while`** loop, instead of an **`if`**. Now we must be concerned: Is it possible that a thread will be stuck in the while loop forever, thus violating wait freedom? Fortunately, the answer is 'no'.

Consider the following scenario: An object $o$ is aliased by two objects such that $o_1$`.f` $=$ $o_2$`.f` $=$ `o`. Thread $T_1$ traverses $o_1$`.f`, proxies $o$ (e.g., on line 26), and then calls a method on $o$. Because $o$ is a proxy, this prompts Rubah to traverse it, eventually executing line 5. But before $T_1$ can execute line 6, suppose thread $T_2$ traverses $o_2$`.f` and, because $o$ is not marked as visited yet, the test on line 13 fails and $T_2$ re-proxies $o$. At this point, thread $T_1$ executes line 6 and returns from method `LAZYmigrate`. The guard in the notional while loop, replacing the **`if`** on line 29, is true and `LAZYmigrate` is called again. However, notice that thread $T_1$ marked $o$ as visited in line 6. The next time a thread finds $o$ while traversing an object, the conditional on line 13 is true, so the object is not be specifically re-proxied again. Therefore, once $T_1$ executes line 5 the second time, the object will be de-proxied permanently.

The worst case scenario is if half of the threads in the application behave as $T_1$ and the other half as $T_2$, alternately, as in the sketched scenario. However, because there is a bounded number of threads, there is a bounded number of times that a proxy can be installed and uninstalled in sequence for the same object. Assuming that the map `visited` is wait-free, it follows that there is a bound on the number of steps required for each proxy to break out of its while loop. Therefore, we can state that the implementation of the lazy state transformation algorithm remains *wait-free*.

### 4.4 Portability assumptions

Rubah was tested on Oracle's HotSpot JVM. It does not modify any part of it, but it relies on a number of assumptions about it. In particular, Rubah (1) uses "unsafe operations" to read fields directly, circumventing access checks and bounds checks, and to compare-and-swap on arbitrary memory locations; and (2) assumes the JVM lays out fields in the same order along the class hierarchy, and places each object's vtable in a fixed location accessible to unsafe operations. Besides Oracle's HotSpot, IBM's Jikes and OpenJDK also satisfy these assumptions.

## 5. Evaluation

This section presents our experimental evaluation of Rubah. We used Rubah to dynamically update five applications: **H2**, an SQL DBMS written in Java; **Voldemort**, a key-value store used by LinkedIn; **Jake2**, a Java port of the shooter game Quake 2; **CrossFTP**, an FTP server; and **JavaEmailServer**, a POP3/SMTP mail server. All five applications are long-running, and maintain important in-memory state (the database/store contents, the game state, and/or the protocol's state for each client) that would be lost on restart. All of our code is available at `http://web.ist.utl.pt/~luis.pina/oopsla14`.

We used Rubah to install application releases as dynamic updates. The first three columns of table 1 list the application versions, their size, and how they changed between releases. H2 changed considerably in the releases we considered: Among other changes, developers implemented support for new SQL commands/idioms and full-text search, and improved the performance of H2's page store. Voldemort did not change as much: the new release fixes a race and improves throttling when cleaning up data after rebalancing a server cluster. CrossFTP added support for new configuration options for the PASV command and the international character encoding for directory lists. JavaEmailServer added support for limiting the maximum size for incoming messages, maximum delivery attempts before dropping a message, and relaying messages based on the recepient's address. We evaluate Rubah along three axes:

**Programmer effort** (Section 5.1) How difficult is it to retrofit an application to use Rubah? How difficult is it to write an update class (which describes how to transform the application's state)?

**Steady-state overhead** (Section 5.3) How much slower is the normal operation of the Rubah-retrofitted version of an application than its unmodified version?

**Per update overhead** (Section 5.4) How is the performance of an application negatively affected while the update is being installed? That is, how long is the application paused and/or its performance degraded?

### 5.1 Programmer Effort

Table 1 assesses the programming effort to use Rubah on our applications. We retrofitted four versions of CrossFTP, three versions of H2 and JavaEmailServer, two of Voldemort, and one of Jake2 (as other versions lack sufficiently different functionality). The fourth column counts the number of files and lines affected by our retrofit of the application to use Rubah. For all five, we added update points to long-running loops and added control-flow migration as described in Sections 2.3 and 2.4, respectively.

We placed update points so that each applications reaches them shortly after an update is available. The following paragraphs briefly describe how each application reaches an update point, considering idle and active scenarios.

When idle, all applications wait either for new clients to connect, or for new requests from connected clients. We retrofitted each application so that it can be interrupted while waiting. For H2, CrossFTP, and JavaEmailServer, we changed I/O calls to use Rubah's equivalent interruptible calls that use non-blocking I/O[8] (accounting for 134 and 49 LOC, respectively); Voldemort already uses non-blocking I/O; and Jake2 polls I/O frequently rather than blocking.

When active, each application processes requests from clients: H2 processes SQL commands, Voldemort processes read/store operations, Jake2 processes network frames, CrossFTP processes FTP commands, and JavaEmailServer processes POP3/SMTP commands. We retrofitted each application so that it finishes processing the current requests it already started processing at the time an update became available, and reaches an update point before starting to process any new requests.

Some applications might take a long time processing requests. For instance, CrossFTP RETR/STOR commands involve sending/receiving an arbitrarily large file over the network. To avoid large periods of quiescence, while the server is not accepting new clients/requests because an update is available but it cannot start the update process because it is executing such a command, we took advantage of the presence of a transfer buffer that CrossFTP fills before sending/receiving data and added an update point reached when the buffer gets filled.

Consistent with our experience with dozens of updates to six C applications using Kitsune [6], the number of changes required is relatively small and not strongly correlated with program size, but rather with its control structure—notice that Jake2 required only 29 lines changed compared to 267 for H2, but is actually larger. Moreover, as indicated by the table, no new changes were required for subsequent versions of H2, Voldemort, and JavaEmailServer. We expect that retrofitting an application to support Rubah is, like Kitsune, a modest, largely one-time cost.

[8] Rubah's I/O library does not support SSL at this point, so we had to comment out CrossFTP's code that uses SSL. Supporting SSL is just a matter of engineering effort.

| Version | Release Code (#lines / #files) | Release Changes (#classes / #methods / #fields) | Retrofit Modifications (#lines / #files) | Update Class (#stub / #mod) LOC |
|---|---|---|---|---|
| **H2** | | | | |
| *1.2.121* | 40119 / 98 | - | 267 / 9 | - |
| *1.2.122* | 40566 / 98 | 63 / 149 / 12 | Same | 106 / 45 |
| *1.2.123* | 40655 / 99 | 44 / 86 / 3 | Same | 40 / 30 |
| **Voldemort** | | | | |
| *1.5.3* | 87516 / 517 | - | 175 / 7 | - |
| *1.5.4* | 87539 / 517 | 8 / 12 / 2 | Same | 12 / 2 |
| **Jake2** | | | | |
| *0.9.5* | 85408 / 256 | - | 29 / 2 | - |
| **CrossFTP** | | | | |
| *1.07* | 18221 / 161 | - | 224 / 8 | - |
| *1.08* | 18108 / 161 | 9 / 20 / 1 | Same | 16 / 1 |
| *1.09* | 18173 / 160 | 30 / 58 / 4 | +4 / Same | 47 / 2 |
| *1.11* | 18435 / 161 | 10 / 34 / 11 | Same | 51 / 23 |
| **JavaEmailServer** | | | | |
| *1.3.3* | 2368 / 20 | - | 183 / 6 | - |
| *1.3.4* | 2447 / 20 | 5 / 11 / 1 | Same | 26 / 2 |
| *1.4* | 2529 / 20 | 7 / 17 / 3 | Same | 55 / 9 |

**Table 1.** Changes between releases and programmer effort to support Rubah. Column *release code* shows the total lines of code, excluding comments and blank lines, and number of files on the original application. Column *release changes* shows the code changes between the previous release, in terms of modified classes, methods, and fields. Column *retrofit modifications* shows how many lines of code we added/modified to support Rubah and how many files were changed. Column *update class* shows the LOC of the automatically generated update class file and the number of its lines we added/modified.

For H2, Voldemort, CrossFTP, and JavaEmailServer, we developed update classes to implement state transformation between the supported versions; the fifth column provides some data about these classes. We can see that stub update classes eased the burden placed on the developer: The maximum number of lines that we had to modify was 45. We tested our updates to H2, Voldemort, and CrossFTP by running standard benchmarks (described shortly), updating while they were underway, and confirming the integrity of the final results.

### 5.2 Performance Experiments: Setup

We conducted an experimental evaluation to measure Rubah's influence on an application's steady state performance, and its performance at update time. Measurements were carried out on a machine equipped with two Intel Xeon E5520 processors (8 physical cores, 16 logical) and 24GB of RAM running Ubuntu 10.04 (Linux kernel 2.6.32). We used the Oracle JVM version 1.7.0_25 with HotSpot 64-Bit Server VM (build 23.25-b01) configured to use a maximum heap size of 16GB for the server and 2GB for the client.

In all of our experiments, we start the application server process and then launch a separate client process that executes a performance benchmark that interacts with the server and measures its performance. To measure *steady-state overhead* we compare the performance of the unmodified server with that of the Rubah-enabled one—no updates are performed. To assess *per-update overhead* we update the Rubah-enabled server in the middle of the benchmark run and measure the performance impact of doing so. In addition to performing a real update from one version to the next (which we call a *v0v1* update), we also consider a *v0v0* update, which installs the same version that the program is running, but considers all classes incompatible and transforms all the updatable program state, copying (and not just adjusting the `_klass` pointer) all instances while the program state traversal takes place. This is a good approximation of a worst case scenario.

To measure H2's performance, we used the TPC-C benchmark available in the DaCapo benchmark suite [22] as the client process. We can configure the TPC-C benchmark with the number of transactions to run and the size of the database to create before running the workload. The database size is expressed in terms of a *scale factor* with which TPC-C multiplies the number of rows in several tables it creates. The H2 server keeps all data in memory.

Voldemort ships with a performance benchmark that we used as the client process. The benchmark has several configurable parameters. The most interesting are: The number of operations to perform, number of key-value pairs created before running the workload, the size (in bytes) of each stored key, and the ratio of read and write operations performed by the workload. Besides these parameters, we extended the benchmark with support to run the workload for

a fixed period of time (as opposed to a fixed number of operations). We configured Voldemort's server in a single node setting, with all the data in memory. The benchmark executes a realistic mix of 95% read and 5% write operations [23].

To evaluate CrossFTP, we implemented an FTP benchmark. Existing FTP benchmarks focus on measuring the bandwidth of file transfer, typically downloading/uploading the same file as many times as possible over the duration of the benchmark. This workload does not exercise the parts of a server that deal with other FTP commands, e.g. browsing the file structure. Our benchmark connects to a remote FTP server, randomly browses the directory structure in a depth-first manner, and downloads the first file it finds. The benchmark spawns multiple threads, each one representing one client. We have CrossFTP server a directory tree that is $D = 2$ levels deep, with each non-leaf folder containing $W = 10$ sub-folders, and each leaf folder containing a file. Files have random contents with sizes in the range 2MB-300MB following an exponential distribution with a mean size of 50MB. We chose these parameters so that the workload resembles a repository of binary software packages used by current GNU/Linux distributions, which are typically made available through an FTP server.

Jake2 and JavaEmailServer lack an automated performance benchmark. As such we only measure the pause time resulting from applying an update to an idle process; for Jake2 this is a v0v0 update after loading the game state, and for JavaEmailServer it is v0v1 update after startup.

### 5.3 Steady-state overhead

Table 2 reports the time H2's benchmark took to run 256K transactions on a database with a scale factor of 32; the time Voldemort's benchmark took to run 25M operations over a key-value store populated with 5M entries of size 128 bits, and the bandwidth CrossFTP's benchmark used after a 5 minute run with 8 client threads. From this data, we can claim that Rubah essentially imposes no overhead in normal execution: The performance difference ranges from $-1.0\%$ to $2.5\%$, which is well within the noise on modern systems [12].

### 5.4 Per-update overhead

Installing an update temporarily slows down the application. Rubah pauses the application while waiting for the threads to quiesce, and then while loading the new classes. On top of that, when transforming the program state eagerly, the application remains paused while Rubah threads traverse and transform the heap; when transforming state lazily, the application resumes execution, but it briefly pauses each time it must transform an object. We measure these two effects with two experiments. The first experiment measures the benefits of parallelization to eager state transformation, and the second measures the pause for both the eager and lazy case as well as the impact on post-update performance.

| **Version** | **Vanilla** | **Rubah** | **Overhead** |
|---|---|---|---|
| **H2** Elapsed time (seconds) | | | |
| *1.2.121* | $350.5 \pm 6.4$ | $351.4 \pm 3.9$ | $0.3\%$ |
| *1.2.122* | $348.8 \pm 7.0$ | $350.0 \pm 3.5$ | $0.8\%$ |
| *1.2.123* | $347.1 \pm 7.0$ | $350.0 \pm 3.5$ | $0.8\%$ |
| **Voldemort** Elapsed time (seconds) | | | |
| *1.5.3* | $469.1 \pm 3.1$ | $471.6 \pm 1.3$ | $0.5\%$ |
| *1.5.4* | $469.1 \pm 2.5$ | $473.7 \pm 3.5$ | $1.0\%$ |
| **CrossFTP** Bandwidth (Mbps) | | | |
| *1.07* | $829.9 \pm 5.5$ | $811.1 \pm 15.6$ | $2.3\%$ |
| *1.08* | $827.8 \pm 3.7$ | $813.7 \pm 6.0$ | $2.5\%$ |
| *1.09* | $801.8 \pm 4.9$ | $809.7 \pm 5.5$ | $-1.0\%$ |
| *1.11* | $803.5 \pm 14.1$ | $809.1 \pm 3.4$ | $-0.7\%$ |

**Table 2.** Results of benchmark runs, with and without Rubah, thus reporting steady-state performance. Reported values are the median and semi-interquartile range of 10 benchmark runs. Overhead is computed by $(Rubah/Vanilla) - 1$.

| Num. Threads | **v0v0** | | **v0v1** | |
|---|---|---|---|---|
| | Time (sec) | Speedup | Time (sec) | Speedup |
| | **H2** | | | |
| *1* | $31.2 \pm 0.7$ | 1 | $18.8 \pm 0.7$ | 1 |
| *2* | $19.0 \pm 0.5$ | 1.7 | $12.3 \pm 0.5$ | 1.5 |
| *4* | $12.6 \pm 0.3$ | 2.5 | $9.2 \pm 0.2$ | 2.0 |
| *8* | $10.0 \pm 0.2$ | 3.1 | $8.2 \pm 0.4$ | 2.3 |
| *12* | $9.3 \pm 0.3$ | 3.3 | $8.1 \pm 0.3$ | 2.3 |
| *16* | $9.2 \pm 0.2$ | 3.4 | $7.8 \pm 0.2$ | 2.4 |
| | **Voldemort** | | | |
| *1* | $42.5 \pm 1.0$ | 1 | $29.2 \pm 0.9$ | 1 |
| *2* | $39.5 \pm 1.0$ | 1.1 | $30.4 \pm 1.1$ | 0.9 |
| *4* | $22.0 \pm 0.8$ | 1.9 | $18.0 \pm 0.9$ | 1.6 |
| *8* | $13.7 \pm 0.7$ | 3.0 | $12.1 \pm 1.0$ | 2.5 |
| *12* | $12.6 \pm 0.7$ | 3.3 | $10.6 \pm 0.5$ | 2.8 |
| *16* | $12.0 \pm 0.4$ | 3.5 | $10.7 \pm 0.4$ | 2.7 |

**Table 3.** Elapsed time (in seconds) of parallel state transformation. The first column under each benchmark is the median time and semi-interquartile range, in seconds, required to transform the program state. The second column is the speedup relative to one thread. Reported values are the average and standard-deviation of 10 benchmark runs. The H2 benchmark used a database with a scale factor of 32 and the Voldemort benchmark used a key-value store with 5M entries.

#### 5.4.1 Parallelizing state transformation

We configured each benchmark to install an update 10 seconds after populating the server with test data, and measured the time Rubah took to perform parallel transformation. We ran the experiment for both *v0v0* and *v0v1* updates (1.2.121 to 1.2.122 in H2's case) and with a varying number of transformation threads.

Table 3 reports the results for H2 and Voldemort. CrossFTP, Jake2, and JavaEmailServer keep such a modest amount of program state that increasing the number of threads does not influence the state transformation time. We thus exclude them from this experiment. Comparing to single-threaded transformation, Rubah achieves speedups using up to 16 threads on H2 and Voldemort, despite the fact that the test machine has only 8 physical CPUs. The *v0v0* case has more work to do per object, therefore sees a higher speedup than the *v0v1* case. In Voldemort's case, changing from 1 to 2 threads yields little or no speedup, and sometimes slows down because 2 threads create a much larger number of in-flight conversions, thus creating a larger task queue and triggering more garbage collections. Adding more threads amortizes this added memory pressure.

#### 5.4.2 Update-time performance

Our second experiment considers how the length of the pause caused by the state transformation varies with algorithm/heap size, and how an update affects the program's subsequent performance.

We approximate the pause due to an update by measuring the maximum server latency that the client ever experiences during the benchmark run; we expect that the pause induced by an update will dwarf the normal latency a client would experience. For H2, we keep track of the time each successful[9] SQL command takes to execute; for Voldemort, we keep track of each store read/write; and, for CrossFTP, we keep track of the time the server takes to reply to each CWD/LIST command and the time it takes to fill a 4MB transfer buffer when downloading a file.

For H2, the benchmark executes 256K transactions, while for Voldemort we run its standard benchmark for 20 minutes. We install an update at time T=60 seconds for H2 and T=300 seconds for Voldemort, to give each program time to reach peak performance. For CrossFTP, we ran our benchmark for five minutes and installed the update at T=10 seconds. For Jake2 and JavaEmailServer we perform the update after the servers are loaded and initialized, while idle (since we had no good automated benchmark). We run the benchmarks for *v0v0* and *v0v1* (1.2.121 to 1.2.122 in H2's case, 1.08 to 1.09 in CrossFTP's case, and 1.3.4 to 1.4 in JavaEmailServer's case).

***Jake2, CrossFTP, and JavaEmailServer.*** Table 4 presents the measured pause times. For Jake2, both the parallel and lazy algorithms induce short pauses. We confirmed that the update was non-disruptive by playing several matches of Quake2 while performing the update; the Quake2 client already tolerates network latency and the Jake2 server keeps a very small program state.

[9] The TPC-C benchmark issues commands that timeout due to table locking made by other commands. We discard such unsuccessful commands.

| Size | v0v0 | | v0v1 | |
|---|---|---|---|---|
| | **Parallel** | **Lazy** | **Parallel** | **Lazy** |
| **H2** | | | | |
| *32* | $11.0 \pm 0.3$ | $3.3 \pm 0.2$ | $9.0 \pm 0.1$ | $3.1 \pm 0.1$ |
| *64* | $20.9 \pm 0.8$ | $3.7 \pm 0.4$ | $15.3 \pm 0.6$ | $3.7 \pm 0.1$ |
| *128* | $71.0 \pm 1.2$ | $4.0 \pm 0.5$ | $30.9 \pm 0.9$ | $3.7 \pm 0.3$ |
| **Voldemort** | | | | |
| *1M* | $4.9 \pm 0.3$ | $1.5 \pm 0.3$ | $4.4 \pm 0.4$ | $1.9 \pm 0.4$ |
| *5M* | $13.5 \pm 1.0$ | $1.6 \pm 0.6$ | $10.7 \pm 0.8$ | $2.2 \pm 0.5$ |
| *10M* | $24.7 \pm 1.8$ | $1.6 \pm 0.5$ | $19.1 \pm 2.1$ | $2.2 \pm 0.5$ |
| *15M* | $158.2 \pm 7.1$ | $1.8 \pm 0.5$ | $107.4 \pm 0.8$ | $2.4 \pm 0.4$ |
| **Jake2** | | | | |
| | $1.5 \pm 0.1$ | $1.2 \pm 0.1$ | - | - |
| **CrossFTP** | | | | |
| | $0.33 \pm 0.04$ | $0.35 \pm 0.08$ | $0.35 \pm 0.07$ | $0.44 \pm 0.06$ |
| **JavaEmailServer** | | | | |
| | $0.11 \pm 0.01$ | $0.09 \pm 0.01$ | $0.10 \pm 0.01$ | $0.09 \pm 0.02$ |

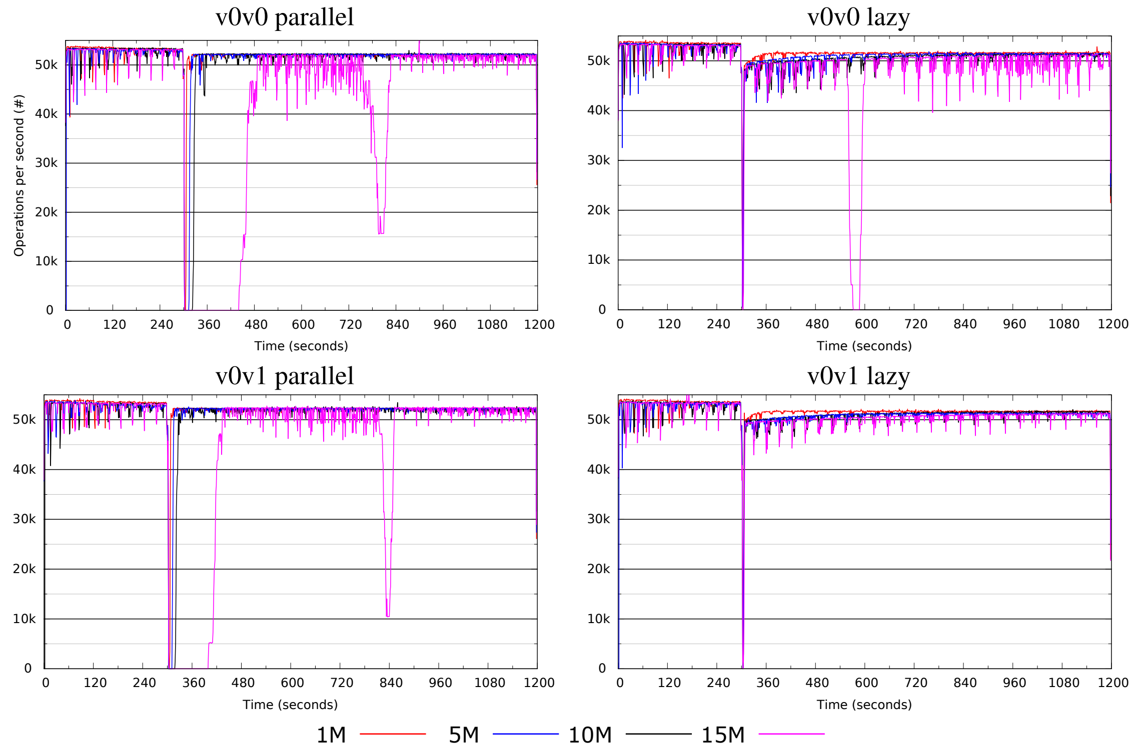
**Table 4.** Pause time (in seconds) required to install each update under various heap sizes. Reported values are the median and semi-interquartile range of 10 benchmark runs. The first column is the size that each benchmark used to populate the server with test data (scale factor for H2 and number of key-value pairs for Voldemort). The parallel transformation used 16 threads.

CrossFTP and JavaEmailServer also have small pause times. These programs keep a modest amount of program state—consisting of meta-data about each connected client (e.g. current working directory, transfer mode, permissions for CrossFTP; authentication state, list of messages, message being composed for JavaEmailServer)—so Rubah takes little time to traverse and transform their state. The parallel algorithm has slightly better results than the lazy algorithm. We report the result of using 16 threads. We interacted manually with JavaEmailServer through a telnet client connected to the POP3/SMTP port while an update was taking place to ensure that the server does not drop connections or session data due to the update process.

***H2 and Voldermort: Parallel transformation.*** Now we consider H2 and Voldemort's performance as impacted by Rubah's use of the parallel algorithm.

Table 4 shows the update-time pause for a variety of heap sizes. In particular, for H2 the **size** column reports the *scale factor* of the database over which the benchmark performs 256K transactions, while for Voldemort the **size** column reports the number of key-value pairs in the store. The pause times for the parallel algorithm, shown in the second and fourth columns, grow as the heap size grows, as expected. For larger heaps, the update pause causes a pronounced increase in the maximum latency.[10]

[10] Note that the results that Table 4 reports are not directly comparable to those of Table 3; e.g., the numbers in Table 3 for Voldemort are measured for an update taken at T=10 seconds, but for Table 4 the update is at T=300 seconds. For the latter, we measured a heap transformation time of 10.3 seconds for Voldemort v0v1 at 5M (out of the 10.7 second pause reported in Table 4), which is slightly less than the 10.7 seconds reported in Table 3, but within the reported error range.



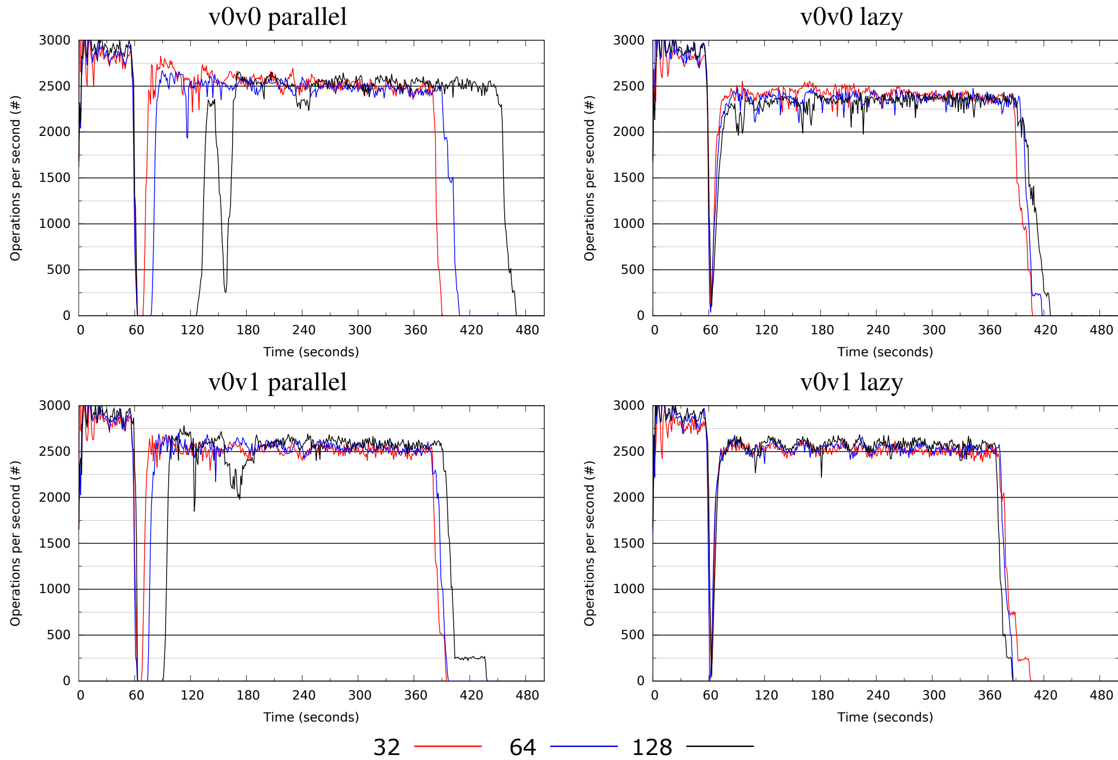




**Figure 6.** Plotting of Rubah's performance while installing an update under varying heap sizes. Each line shows the performance for a different-sized database (the line label is the scale factor for H2 and the number of key-value pairs for Voldemort). We report the average of 10 benchmark runs. For parallel transformation we used 16 threads. The occasional performance dips are due to garbage collections; their number and magnitude indicate the level of memory pressure.

Figure 6 visualizes the performance of H2 and Voldemort during the experiment. The figure presents plots where the x-axis is elapsed time, and the y-axis is transactions/operations per second. The left column of charts in the Figure show the results when using parallel state transformation. At the update times we see a performance drop, a pause, and then a rapid rise back toward the pre-update peak. There are two things to notice: First, the update pause increases with the heap size, particularly for *v0v0*, which must traverse all of the heap. Second, we see that performance does not completely return to its pre-update level. We observed a similar drop in steady-state performance on an experiment that traverses the whole heap starting from the root references without making any changes. We thus believe that the performance drop is due to a change in some internal JVM state triggered by the state traversal; we will investigate further to understand the reasons behind the performance drop. We installed a second update for the parallel v0v0 case and confirmed that the performance after the second update reached the same levels as the performance after the first update.

***Lazy transformation.*** Now we consider the use of the lazy algorithm with H2 and Voldemort. The right column of charts in Figure 6 plots performance when using lazy transformation. The key feature is the far smaller drop in performance at update-time, after which performance slowly rises, depending on the heap size. Table 4 shows that the update pause is constant regardless of the total heap size, and is quite small compared to the parallel algorithm.

Returning briefly to Figure 6, we note that the experiment also shows that lazy transformation does disproportionately better than the parallel transformation with larger heaps. For Voldemort, the 15M case consumes nearly the entire heap. The parallel transformation makes the GC thrash, triggering numerous full-GC cycles that are not able to free much memory. The lazy algorithm performs much better. It still triggers one full-GC cycle, but that one cycle actually frees enough memory to keep the GC from ever thrashing. We can see a similar thrashing pattern for H2's 128 *v0v0*, even though it happens after the update is installed.

Figure 6 also shows that the lazy algorithm converges very quickly to steady-state performance after the update. Several reasons contribute to this behavior: Converting an object lazily is fast, thus imposing a small performance penalty; the working set of each application is small and constant despite the heap size; and the rate of lazy object conversion drops quickly after the update.

***Post-update performance for lazy transformation.*** The post-update drop of peak performance is larger for the lazy case, compared to the parallel one. This happens due to decisions that the JIT compiler takes when optimizing the code immediately after an update, when proxies are present and used frequently. To diagnose this behavior, we ran Oracle's HotSpot JVM with debug flags that log the optimization decisions the JIT compiler makes.[11] The compilation logs show differences in how the JIT optimizes virtual method invocation after a lazy update.

The JVM supports method dispatch based on the runtime type of the object in which the method is executed — the *receiver*. The JVM uses the object's vtable to choose the most specific concrete method to execute when performing a virtual method invocation. However, looking up the vtable at each invocation is costly and the JVM optimizes for the common cases, as follows:

1. **Single method always invoked:** The JIT compiler inlines the method at the call site, protected with a trap that checks the receiver object type and ensures that the inlined code is correct for each call;
2. **Two methods always invoked:** Similar to 1. The JIT compiler inlines both methods after a conditional branch that jumps to the right method. A trap is also used to ensure correctness;
3. **More than two methods invoked** The JIT inlines the two most frequently invoked methods, as in 2. The JIT emits code to perform a *slow* virtual method call that consults the object's vtable for all the other concrete methods.

After a lazy update, some virtual method invocations are resolved to proxy methods. This affects the JIT compiler's optimization decisions, turning (a) case 1 into 2, (b) 2 into 3, or (c) changing the two methods inlined in case 3. For (a), the size of the optimized code increases due to the extra inlined method. This might prevent the optimized code from being itself inlined elsewhere, resulting in a performance penalty. For (b) and (c), the JIT might decide to inline proxy code. As the program executes after an update, the number of proxies found by the code drops because they get transformed. This optimization thus increases the number of slow virtual method invocations that use the vtable and bypass the inlined code, which in turn yields lower steady-state performance.

We are exploring solutions to this problem, such as implementing a mechanism to reset the JIT compiler on demand, dropping all the emitted code and collected performance metrics; or implementing a flag to keep the JIT compiler from inlining code belonging to particular Java classes. Both options require JVM changes. However, the changes required are minimal and completely backwards compatible.

## 6. Related Work

As mentioned in Section 2, Rubah employs the same approach to whole program updates as Kitsune [6], a DSU system for C; in particular, both systems employ the concepts of control-flow migration and update points. Kitsune's state transformation algorithm is like Rubah's parallel algorithm but uses a single thread. Due to C's weak type system, Kit-

[11] -XX:+UnlockDiagnosticVMOptions -XX:+LogCompilation

sune's compiler cannot always produce a traversal algorithm automatically, and so may require manual assistance. Kitsune uses a domain-specific language to specify state conversions; Rubah's update class is a more compact, and natural, representation for conversions in the Java context.

Rubah's update class bears some similarity to PJama's *bulk conversion* [24] routines. However, these routines work on offline updates of persistent object stores, rather than online updates to running Java programs. PJama also does not use skeleton classes to refer to old/new state unambiguously.

There have been several prior systems that support DSU for Java without requiring VM support. JRebel [25] allows unrestricted changes to the structure of a class (add/remove fields/methods) but not to the class position in the hierarchy, which Rubah supports. JRebel also does not support any state transformation besides the default Java initialization to added fields. DUSC [26] and DUSTM [11] work by inserting proxies as an indirection to every object, and paying the respective steady-state performance penalty, which can be as high as 50% for a similar H2 benchmark.

The JVM itself is a natural place to support DSU. The Oracle JVM supports dynamic updates to method bodies in existing classes [21], for the purposes of enabling "stop-edit-continue" development (JRebel also targets this domain). Full-featured DSU is supported by the JVolve [10] and DCE VMs [9], though even these do not support some changes to the class hierarchy, which Rubah does. Rubah also supports updates in a more timely fashion, as it does not require changed methods to be inactive (i.e., not on the call stack), while JVolve does. Instead, Rubah expects the programmer to use non-blocking I/O so that it can interrupt and unwind threads' call stacks prior to initiating an update, and use control migration code to restart those threads. As such, Rubah can install an update to CrossFTP that JVolve could not support (version 1.07 to 1.08). JVolve also could not install some JavaEmailServer updates for the same reason. Even though we did not test those updates, we believe Rubah would be able to install them.

Implementing DSU services inside the JVM itself makes JVolve and DCE VM able to take advantage of internal mechanisms, such as the garbage collection (GC) and JIT compiling, to implement efficient support for DSU. However, this approach is inherently non-portable. The goal of building Rubah was to show that similarly powerful mechanisms can be built outside the VM while imposing comparable performance and development costs.

The JDrums [8] JVM supports lazy updates using a technique analogous to Rubah's `$forward` field. JDrums also uses a conversion class to specify how to transform each class. Rubah is more flexible than JDrums and more performant: JDrums cannot transform data in each object's superclass, it does not support changing existing methods, and it executes only in interpreted mode.

Lazy updates have also been implemented for C. Ginseng implements lazy-updates for both single [4] and multi-threaded [5] programs. It uses a proxying approach similar to Rubah's. However, it ensures safety via a per-type mutex rather than via a wait-free algorithm and it does not remove proxies dynamically as objects are converted. POLUS [27] also allows for old and new data to co-exist while the update takes place, and converts old data when it is viewed by new code, on demand. POLUS tracks data changes at a coarser level than Rubah (using page protection).

Our state transformation algorithms have natural analogues in the GC literature [19]: Rubah's eager and lazy algorithms resemble *parallel* and *incremental, concurrent* GC, respectively. There is likely further gain in applying GC ideas to our state transformation algorithms, though DSU requirements are more stringent: GC may delay garbage identification and reclamation (e.g., floating garbage), while state transformation must always be applied prior to access, to bring objects up to date.

## 7. Conclusion

This paper has presented Rubah, the first full-featured, portable DSU for Java that enjoys good performance and is not difficult to use. Rubah's updating model is inspired by the that of the Kitsune updating system for C, inheriting its simplicity and flexibility. Rubah adds the novel notion of an *update class* for specifying how to update the program's state, two new algorithms for performing state transformation: one parallel algorithm that transforms all state at once, and one *lazy* algorithm that transforms state as demanded by post-update execution. Rubah imposes essentially no overhead on steady-state execution, and when using the lazy transformation algorithm imposes short pauses for real-world dynamic updates, recovering its steady-state performance fairly quickly. Rubah still has some performance shortcomings which we are currently addressing. In particular, parallel transformation is slow when using large heaps, and steady-state performance does not completely recover, post-update. We plan to continue to improve our approach, and expand it to new applications.

## Acknowledgments

We thank the anonymous reviewers for the helpful comments on early drafts of this paper. This work was supported by FCT — Fundação para a Ciência e a Tecnologia — under projects PTDC/EIA-EIA/113613/2009 and PEst-OE/EEI/LA0021/2014, by NSF grant CCF-0910530, and by the partnership between UMIACS and the Laboratory for Telecommunication Sciences.

## References

[1] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[2] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *USENIX NSDI*, 2013.

[3] Michael Hicks and Scott M. Nettles. Dynamic software updating. *TOPLAS*, 2005.

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

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

[6] Christopher Hayden, Edward Smith, Michail Denchev, Michael Hicks, and Jeffrey Foster. Kitsune: efficient, general-purpose dynamic software updating for c. In *OOPSLA*, 2012.

[7] Twitter moves from rails to java. `http://www.gmarwaha.com/blog/2011/04/11/twitter-moves-from-rails-to-java/`, 2011.

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

[9] Thomas Würthinger, Christian Wimmer, and Lukas Stadler. Dynamic code evolution for Java. In *PPPJ*, 2010.

[10] Suriya Subramanian, Michael Hicks, and Kathryn McKinley. Dynamic software updates: a VM-centric approach. In *PLDI*, 2009.

[11] Luís Pina and Joao Cachopo. DuSTM - Dynamic Software Upgrades using Software Transactional Memory. Technical Report 32/2011, INESC-ID Lisboa, 2011.

[12] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *ASPLOS XIV*, 2009.

[13] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java(TM) virtual machine. In *OOPSLA*, 1998.

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

[15] Christopher Hayden, Edward Smith, Eric Hardisty, Michael Hicks, and Jeffrey Foster. Evaluating dynamic software update safety using efficient systematic testing. *IEEE TSE*, 2012.

[16] Christopher Hayden, Karla Saur, Michael Hicks, and Jeffrey Foster. A study of dynamic software update quiescence for multithreaded programs. In *HotSWUp*, 2012.

[17] Kristis Makris and Rida A. Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. In *USENIX ATC*, 2009.

[18] Christopher Hayden, Stephen Magill, Michael Hicks, Nate Foster, and Jeffrey S. Foster. Specifying and verifying the correctness of dynamic software updates. In *VSTTE*, 2012.

[19] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall, 2012.

[20] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[21] Oracle(TM). Java SE 1.4 Enhancements. `http://download.java.net/jdk8/docs/technotes/guides/jpda/enhancements1.4.html`.

[22] Dacapo. `http://www.dacapobench.org/`.

[23] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. Tao: Facebook's distributed data store for the social graph. In *USENIX ATC*, 2013.

[24] Misha Dimitriev and Malcolm P. Atkinson. Evolutionary data conversion in the PJama persistent language. In *Proceedings of the Workshop on Object-Oriented Technology*, 1999.

[25] ZeroTurnAround. JavaRebel. `http://www.zeroturnaround.com/jrebel/`.

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

[27] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. Polus: A powerful live updating system. In *ICSE*, 2007.