Rubah: DSU for Java on a stock JVM

Luís Pina Luís Veiga
Technical University of Lisbon / INESC-ID
[luis.pina/luis.veiga]@ist.utl.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 only 5–9% 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 three substantial, long-running applications: the H2 database, the Voldemort key-value store, and the Jake2 implementation of the Quake 2 shooter game.

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.

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 can reconnect on restart after an update [17].

General-purpose systems developed for C and C++ have been applied to dozens of realistic applications, tracking changes according to those applications release histories [8, 10, 15, 16]. Increasingly, important on-line services are written in managed languages like Java. For example, Twitter has moved most of its major infrastructure to Java [23], and the Java-based Voldemort noSQL database is used by companies like Linkedin. While several DSU systems for Java have been developed [21, 22, 24] 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 [8], 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 [22], the DVM [24], and JDrums [21].
- 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.
- Rubah enjoys good steady-state performance: supporting updating imposes 5–9% 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 [20] 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 precipitates the target

[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.

object's transformation and then removes itself, to avoid adding further 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 three substantial applications: the H2 SQL relational database; the Voldemort key-value store, used in practice by LinkedIn; and Jake2, a Quake2 port translated to 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 three versions of H2 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 77 lines of code for all the updates we tested.

Performance experiments using benchmarks for Voldemort and H2 found that Rubah imposes 5% and 9% overhead, respectively. We also found that our state transformation algorithms reduce the pause at update-time. The parallel algorithm performed nearly 5 times faster than the single-threaded version for larger heaps. However, for larger heaps, the total pause time can still be quite high; e.g., 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.

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 [8], 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

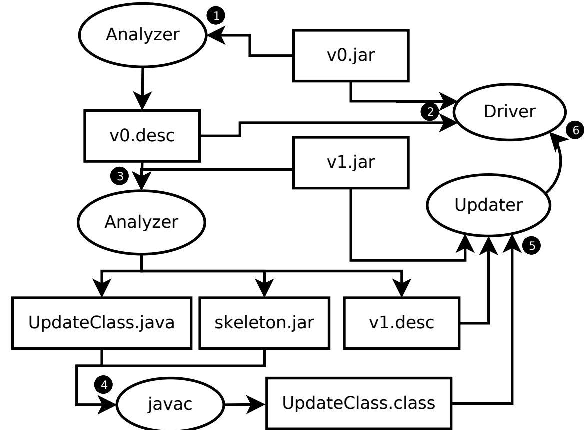


Figure 1. Deploying a program, and preparing and installing an update for it, using Rubah. Square boxes represent artifacts: Compiled code (jar/class), source code (java), or update descriptors (desc). Round boxes represent tools: Rubah's driver, analyzer, and updater, and the unmodified Java compiler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

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 he update. In the second stage, *state transformation*, the driver initiates (and may complete) the modification of object instances whose class has 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 [12].

```
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).

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 will be in methods like `process`, which will 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 [10]. 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. The state relevant to the update is not in the middle of being modified. In general, update points are best placed at the head of long-running loops to ensure both safety and availability [9].

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 [7]; or (2) have each thread voluntarily wake-up from I/O calls frequently and reach an update point before 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 (see Section 5.1). In the example, `process` will throw 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.

Calling method **`Rubah`**`.update` when an update is available results in it throwing an `UpdatePointException`; 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

[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.

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.

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. Besides application threads and the main thread, Rubah also supports control-flow migration of thread pools.

Note that some systems, like UpStare [13], 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.

```
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.

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 keeps the total number of bytes ever written to a particular store. v_1 renames field `writeCount` to `writeCountBase` and introduces two new fields, `readCount` and `writeCount`, to 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 in the expected way, e.g., by running tests and ensuring that an update mid-test does not violate expected behavior [6, 9].

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. Rubah is the first DSU system to develop a parallel eager transformation algorithm. 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 (the differences from Java are made for brevity and 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. All other objects map to themselves. We write `visited[key] = val` to associate `key` with `val`, and retrieve the current mapping by writing `visited[key]`; if no mapping for `key` exists, this expression yields $\perp$.
- Visiting each field in an object, used to compute the transitive closure of the object graph, is written using a for-like 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 primitives 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 [11], and is used by many DSU systems [8, 10, 22, 24]. We improve on this basic idea by performing tracing in parallel, using multiple threads. Here we present single- and multi-threaded variants of our algorithm, for presentation purposes.

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 [8]) 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.

Figure 4 shows the parallel state transformation algorithm. 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, create an instance of the new class, and transforms the object (lines 7 to 14). `Rubah.convert` calls instance conversion methods in a hierarchical way similar to how Java calls constructors [5]. 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 looks up an object in the map, so in the single-threaded variant, 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 fin-

```
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

ishes when the task threads complete with an empty task pool.[3]

Multiple threads will be reading from and writing to the `visited` map concurrently and there is a danger of data 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** goes to write the new object but finds the map does not contain $\perp$, and thus some other thread beat it to writing an object there, it will simply return the existing object. Then, on line 16 of `migrate`, the object `mapped` will be different than `newObj` and so `migrate` will simply complete (since the existing object has already been set for traversal by whatever thread put it there).

[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).

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 o to the last possible moment, which is just before the program uses o for the first time after an update takes place; at this point we say the object is *visited*. Laziness avoids the significant pause that would occur for large heaps.

To implement lazy transformation, Rubah uses proxies to intercept when the program manipulates unvisited objects. When the program reaches a proxy, we say that such proxy is *dereferenced*. At that moment, Rubah transforms the proxied object, creates proxies for the objects that the transformed object refers to, and transfers the control flow to the transformed object.[4]

Rubah modifies the original program so that every object can behave like a proxy to itself by setting a flag. When the proxy flag is set, methods start by redirecting the control flow to Rubah's API before executing the original method body. Lines 28 to 31 show a possible implementation that we shall use throughout this section. We discuss the actual implementation in Section 4.3.

3.3.1 Correctness conditions

The lazy algorithm is correct if the new program code can find only transformed program state. This intuition is captured by the following invariants:

Invariant 1: Visited objects only refer to other visited objects or to proxies

Invariant 2: Objects are not proxied after being visited

Invariant 1 is a safety property that ensures the object graph has a clear *frontier* between the visited and unvisited program state that is composed of proxies. This frontier starts by being the root references and expands outward as more proxies get dereferenced.

Invariant 2 is a liveness property that ensures that the state transformation makes progress because it does not allow for an object to be constantly proxied and dereferenced in sequence. This situation might lead to stack overflow, as the call stack grows at each proxy-dereference, or to live-lock if two program threads end up installing proxies to an object and dereferencing another constantly and mutually.

3.3.2 Algorithm

Figure 5 shows the lazy state transformation algorithm. We assume that the algorithm starts by running the loop that starts in line 9 for every root reference. 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 run-

[4] During this section we assume that proxies can only be dereferenced through method invocations for the sake of simplicity. We discuss how Rubah handles all other ways in which proxies may be dereferenced, such as field access, in Section 4.3.

ning any constructors (line 20), runs the conversion code to transfer the state from the outdated object (line 21), proxies the new object (line 22), marks the 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.) Objects referred to by the root references are transformed only once: aliased objects are detected by line 13 and each aliasing reference is set to the correct object in line 14. For now, assume that the **CAS** operations on lines 14, 23, and 24 always succeed; their role will 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 the each paused thread's `run/main` method at the new version, beginning the process of control-flow migration. Assuming for now 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), installing proxies for all the unvisited objects that the proxy refers to. It then dereferences the proxy (line 5), and marks the object as visited (line 6). Invariant 1 remains true when the program flow reaches line 31 because the new object only refers to proxies and other transformed objects.

If an object is aliased, several threads might find it concurrently and try to install a proxy for it. All these threads will race to mark the outdated object as visited. 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 object. Therefore, all the conversion methods must be idempotent.

Let us assume that two threads T_1 and T_2 race to map the same object o_1 which is referred to by o_2`.f`. Furthermore, consider that 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 will (correctly) fail for T_2. This race is also the reason for the **CAS** operation in line 14.

3.3.3 Ensuring Progress

The only two lines that install proxies are line 22 and line 26, for objects that need to be transformed and for objects that do not, respectively. It is trivial to ensure invariant 2 for transformed objects: They are proxied before being globally visible, at line 23. Once they are dereferenced (line 5) they cannot be proxied again because they are marked as visited.

```
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         p.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).

For all the other objects, reasoning about progress is more subtle. When they are proxied (line 26), they already are globally visible but not marked as visited. To understand why this is different than transformed objects, consider the following scenario: An object o is aliased by two fields o_1`.f` and o_2`.f`. Thread T_1 traverses o_1`.f`, proxies o into o_P, dereferences o_P, and stops executing after line 5. Then thread T_2 traverses o_2`.f` and proxies o again because o is not marked as visited yet, so the test on line 13 fails.

In this case, object o goes back and forth from proxied and dereferenced. This does not violate invariant 2, however, because object o only becomes visited at line 6. If this line executes while o is proxied as o_P, the next time o_P gets dereferenced it goes back to o in line 5. From this point on, o cannot become proxied again because it is marked as visited and thus passes the test in line 13.

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 scenario that we are following. However, there is a limited number of threads. So, there is a bounded number of times which an object can be proxied and dereferenced in sequence. Assuming that the map `visited` is wait-free, it follows that there is a bound on the number of steps required

to mark each object as visited. Therefore, we can state that the lazy state transformation algorithm is *wait-free*.

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 Class replacement

After the updater signals that an update is ready (step 6 in Figure 1), the driver will load the new classes. For those classes that have changed their method bodies but not their class signature (i.e., the method and field types are unchanged), Rubah uses HotSwap support [18] (a standard JVM feature) to replace the method bodies with their new versions. To update classes whose signatures have changed, Rubah uses bytecode rewriting, described next, to give each updated class a distinct name. During state transformation, references to objects of the old class will be redirected to (transformed) instances of the new one.

4.2 Bytecode transformation

The Rubah driver uses a custom classloader to rewrite all classes, updatable or not, as they are loaded. Bytecode rewriting consists of three parts:[5]

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. Changing the name of all classes might break some reflection calls, such as `Class.forName`. Rubah rewrites all invocations of these methods to call Rubah's API instead, which provides the same semantics and accounts for name mangling.

2. Type Erasure. Rubah replaces occurrences of updatable types in all fields and method arguments with `java.lang.Object`, and adds casts to the code that uses them. It does this for the following reason.

Suppose an updatable program has three classes, A, B, and C, such that A has a field that refers to B and B has one that refers to C. Consider also that a particular update changes class C's signature and how class B interacts with class C. In this case, Rubah needs to generate an updated class C_1 for C_0. It also needs to generate an updated class B_1 to update all references (in fields and methods) from C_0 to C_1. But now class B has also changed and, following the same rationale, Rubah ends up generating class A_1 even though class A did not change and does not use directly any updated class.

[5] Bootstrap class packages, such as the `java.lang`, are re-written before execution and placed in a bootstrap jar that the JVM must load.

Replacing updatable types with `java.lang.Object` eliminates this problem, since the types of declarations will never change. In our example, Rubah generates C_1 and rewrites the methods in B_0 so that they refer to C_1 instead of C_0. But class A_0 is left completely unchanged.

3. Hash Code. When we update an object o_{old} with a replacement o_{new}, we must take care that, post-update, for all objects p_{old} updated to p_{new}:

– if o_{old}`.equals(`p_{old}`)` then o_{new}`.equals(`p_{new}`)`
– if o_{old}`.hashCode() ==` p_{old}`.hashCode()` then o_{new}`.hashCode() ==` p_{new}`.hashCode()`.

These properties are particularly important for collections.

If `equals` is overridden by the program, it is up to the programmer to ensure that the update class preserves equality behavior. Likewise, overridden `hashCode` methods must be ensured equal semantics by the programmer.

Non-overridden `equals` methods just implement == (i.e., pointer) equality, which Rubah ensures by construction. Non-overridden `hashCode` must also be ensured as having the required semantics by Rubah. It does this by injecting an integer field to every updatable class to hold the hash-code, and overrides the `hashCode` method to return the contents of that field. Rubah also injects code to initialize the field on every constructor's exit. Thus the hashcode field can simply be copied when an object is transformed.

Performance. The type erasure and the hash-code transformations are the main sources for the steady-state overhead that Rubah imposes. However, the JIT compiler masks most of the overhead. For type erasure, most of the type erased fields and methods always hold instances of the same type. The JIT compiler is able to realize this and generate optimized code for the common case. As for the hash-code, the JIT inlines calls to the `hashCode` method, thus reducing them to a simple field read operation in most cases.

4.3 State transformation

Our implementation largely follows the state transformation algorithms given in Section 3, with two exceptions: the `visited` map is implemented as an added field, and the `isProxy` field is actually implemented as a proxy object.

Visited map. The `visited` map from Section 3 conceptually 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 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 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 proxies are just a flag that changes how methods work. A direct implementation would require adding an extra field to every class, and adding a dynamic test to the start of each method. Instead, Rubah generates a proxy class for each class that it loads, extending it and overriding all of the original class'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.

In HotSpot, the vtable is placed inside a `_klass` structure and every object points to its `_klass` at a fixed offset. (Jikes has a similar object layout.) The JIT engine uses the `_klass` reference to resolve virtual method invocation, the GC uses the `_klass` reference to find relevant information about the object, e.g., its size and layout, while traversing the heap. Turning an object into a proxy and back is just a matter of installing a different `_klass`. Rubah does exactly that using unsafe operations present in class `sun.misc.Unsafe`.

Changing the `_klass` of an object is potentially unsafe because the code that the JIT emits assumes that the `_klass` does not change. Violating this invariant makes the JVM crash when invoking methods over objects with an unexpected `_klass`. 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 proxy and the proxied `_klass` agree on this metadata, this optimization does not cause the GC to crash the JVM.

Changing the vtable 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.

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.

[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`**.

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 that each object's vtable is in a fixed placed accessible to unsafe operations. As far as we know, IBM's Jikes and OpenJDK both satisfy these assumptions.

5. Evaluation

This section presents the details and results of our experimental evaluation of Rubah. We used Rubah to dynamically update three applications: **H2**, an SQL DBMS written in Java; **Voldemort**, a key-value store used by LinkedIn; and **Jake2**, a Java port of the shooter game Quake 2. All three applications are long-running, and maintain important in-memory state (the database contents for the first two, and the game state in the last) that would be lost on restart. All of our code is available at `http://web.ist.utl.pt/~luis.pina/oopsla14`.

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 5.2.1) How much slower is the normal operation of the Rubah-retrofitted version of an application than its unmodified version?

Per update overhead (Section 5.2.2) 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. The first two columns list the application versions and their size. We retrofitted three versions of H2,

Version	Original Code	Modifications	Update Class
H2			
1.2.121	36882 / 428	267 / 9	-
1.2.122	37182 / 428	Same	106 / 45
1.2.123	37084 / 428	Same	40 / 30
Voldemort			
1.5.3	51679 / 518	175 / 7	-
1.5.4	51667 / 518	Same	12 / 2
Jake2			
0.9.5	40077 / 257	29 / 2	-

Table 1. Programmer effort to support Rubah. Column *original code* shows the total lines of code, excluding comments and blank lines, and number of files on the original application. Column *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.

two of Voldemort, and one of Jake2 (as previous versions lack sufficient differing functionality to be of interest).

The third column counts the number of files and lines affected by our retrofit of the application to use Rubah. For all three, we added update points to long-running loops and added control-flow migration as described in Sections 2.3 and 2.4, respectively. For H2, we also changed blocking I/O calls to use Rubah's equivalent calls that use non-blocking I/O and can be interrupted (accounting for 134 LOC); Voldemort already uses non-blocking I/O and Jake2 polls I/O frequently rather than blocking. Consistent with our experience with dozens of updates to six C applications using Kitsune [8], 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 and Voldemort. Retrofitting an application to support Rubah is, therefore, a modest one-time cost.

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

5.2 Performance

We conducted an experimental evaluation of Rubah's performance using our three applications. 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 (benchmark).

Experimental setup. 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 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 [3] 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* that TPC-C uses to multiply 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 [**?**].

For Jake2 we do not have an automated performance benchmark. Since we have no client process, therefore, we do not measure steady-state overhead but rather only the pause time resulting from applying a v0v0 update to the (idle) process loaded with game state.

5.2.1 Steady state overhead

Rubah-enabled applications are going to be somewhat slower than the original ones due to the changes made to the application during bytecode transformation (which, in part, involve calls to the Rubah run-time library). Table 2 shows that Rubah imposes little overhead during steady-state exe-

Version	Vanilla	Rubah	Overhead
H2			
1.2.121	372.7 ± 6.9	412.0 ± 10.0	10.5%
1.2.122	375.7 ± 4.4	407.1 ± 3.2	8.4%
1.2.123	373.4 ± 2.8	405.8 ± 7.2	8.7%
Voldemort			
1.5.3	478.0 ± 3.8	499.0 ± 5.1	4.4%
1.5.4	475.4 ± 1.7	500.5 ± 2.4	5.2%

Table 2. Elapsed time (in seconds) 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 $(Rubah/Vanilla) - 1$.

cution: About 5% for Voldemort and 9% for H2. For these measurements, we configured H2's benchmark to run 256K transactions on a database with a scale factor of 32; for Voldemort we ran 25M operations over a key-value store populated with 5M entries of size 128 bits.

5.2.2 Per-update overhead

Installing an update will temporarily slow down the application. Under all circumstances, we must wait for the threads to quiesce and we must wait for Rubah to HotSwap outdated method code. When transforming the program state eagerly, the application will remain paused while Rubah threads traverse and transform the heap; when transforming state lazily, the application begins running quickly, but will briefly pause each time it must transform an object.

Parallel transformation. We configured each benchmark to install an update 10 seconds after it finishes populating the server with test data. We repeat 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 how long Rubah takes to transform the program state in each case. Rubah achieves speedups using up to 12 threads on H2 and 16 on 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.

We expect the pause time to increase as heap size increases. Table 4 shows the length of the pause, from the client process's point of view, which grows as the heap grows. For larger heaps, the pause is quite pronounced. Figure 6 visualizes this effect. The figure presents plots where the x-axis is elapsed time, and the y-axis is transactions/-operations per second. The left column on Table 4 shows the results when using parallel state transformation. We configured H2's benchmark to run 256K transactions over a

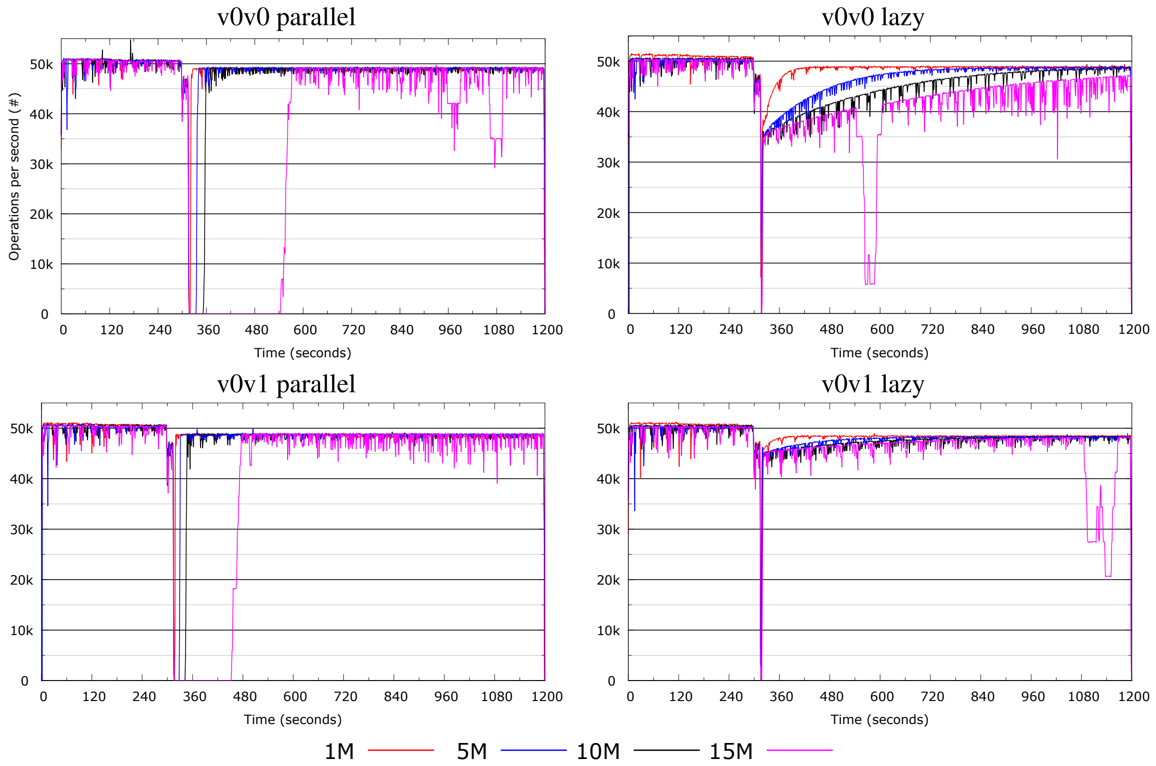
Num. threads	v0v0		v0v1	
	H2			
1	60.8 ± 1.5	1	15.7 ± 0.2	1
2	34.4 ± 0.8	1.8	9.2 ± 0.1	1.7
4	20.3 ± 0.3	3.0	7.0 ± 0.2	2.2
8	13.7 ± 0.5	4.4	6.0 ± 0.2	2.6
12	13.3 ± 0.3	4.6	5.9 ± 0.2	2.6
16	13.0 ± 0.1	4.7	6.3 ± 0.3	2.5
	Voldemort			
1	80.3 ± 1.0	1	37.1 ± 1.2	1
2	68.3 ± 1.2	1.2	43.5 ± 2.7	0.8
4	37.3 ± 1.2	2.2	26.1 ± 3.3	1.4
8	22.4 ± 0.8	3.6	16.2 ± 0.8	2.3
12	20.3 ± 1.1	3.9	14.4 ± 0.5	2.6
16	18.9 ± 1.4	4.2	13.2 ± 0.7	2.8

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.

database with a varying scale factor and Voldemort's benchmark to run for 20 minutes over a server with a varying number of key-value pairs; there is a different line for each variation, listed for each application. We run the benchmark for *v0v0* and *v0v1* (1.2.121 to 1.2.122 for H2's case).

We install an update at time T=60 seconds for H2 and T=300 seconds for Voldemort. Thus at these 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 believe this happens due to JIT confusion: After the update, there are more types for the JIT compiler to reason about, i.e., more concrete implementations of virtual methods inherited by updatable types (Rubah never unloads the outdated types). We are investigating how to improve the situation.

Lazy transformation. The lazy algorithm performs state transformation as needed, on a per-object basis, thus amortizing the performance cost over time. The right column of Figure 6 shows the performance for lazy state transformation. The key feature is the far smaller drop in performance at update-time, after which performance slowly rises, depending on the heap size. Returning to Table 4, we can see that pauses for lazy updates on H2 *v0v0* increase linearly; this is because much of the heap is reached (in an array) prior to completing a single transaction. For all other lazy updates, the update pause is constant regardless of the total heap size, and is quite small compared to the parallel algorithm.



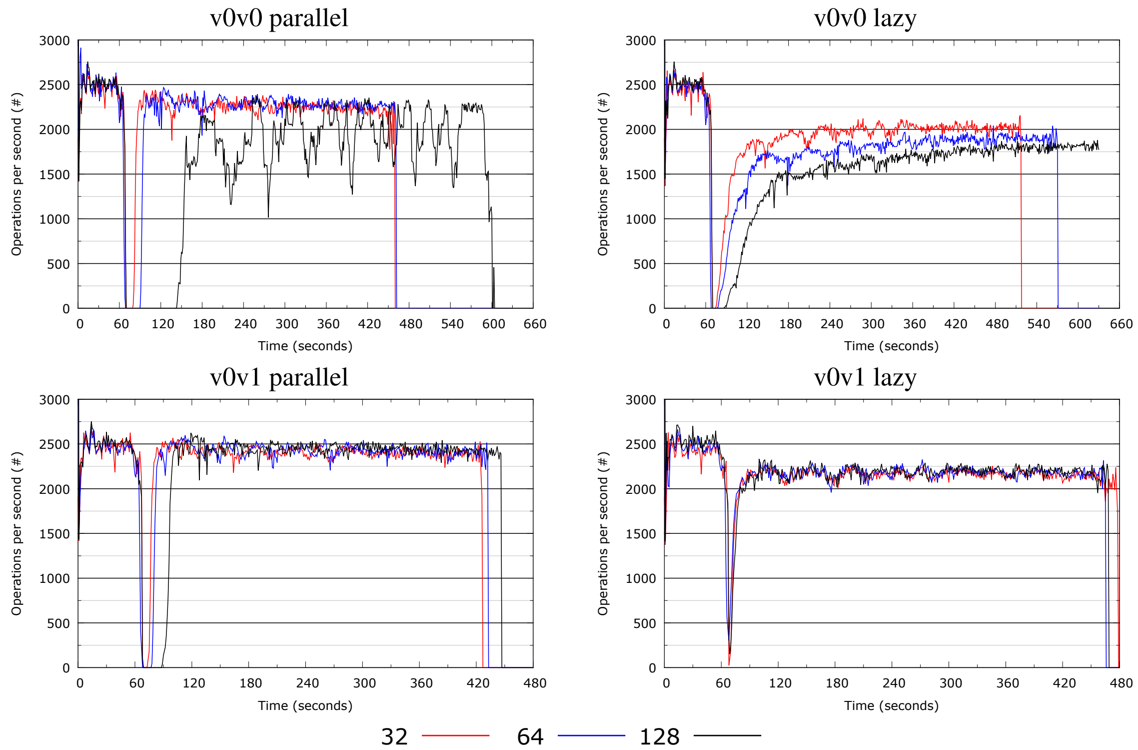


Figure 6. Plotting of Rubah's performance while installing an update under varying heap sizes. Each line shows the performance for a different-sized database (the line label is the scale factor for H2 and the number of key-value pairs for Voldemort). We report the average of 10 benchmark runs. For parallel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

Size	v0v0		v0v1	
	Parallel	Lazy	Parallel	Lazy
H2				
32	13.0 ± 0.5	7.0 ± 0.5	7.5 ± 0.5	3.0 ± 0.1
64	24.5 ± 0.5	13.0 ± 0.7	13.0 ± 0.5	3.0 ± 0.5
128	83.0 ± 3.6	23.0 ± 0.6	27.0 ± 0.8	2.5 ± 0.5
Voldemort				
1M	5.1 ± 0.1	1.3 ± 0.1	4.2 ± 0.1	1.9 ± 0.2
5M	20.5 ± 0.5	1.3 ± 0.4	15.1 ± 0.3	1.5 ± 0.3
10M	40.1 ± 1.4	1.8 ± 0.3	29.8 ± 0.7	1.8 ± 0.5
15M	242.3 ± 3.8	1.7 ± 0.3	151.4 ± 7.8	2.8 ± 0.4
Jake2				
	1.5 ± 0.1	1.2 ± 0.1	-	-

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.

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 runs close to the total heap size. 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.

Finally, note that the post-update drop of peak performance is larger for the lazy case, compared to the parallel one. We speculate this is due to the many proxies that are in the object graph. The JIT ends up optimizing for transforming proxies, which is a frequent operation immediately after the update. This optimization is, obviously, less performant when the program reaches steady-state after the update and most of the proxies are already transformed.

Jake2. Table 4 shows that the update time for Jake2, both lazy and parallel versions, is quite small. We confirmed that the update was non-disruptive by playing several matches of Quake2 and updating in the middle; the Quake2 client already tolerates network latency and the Jake2 server keeps a very small program state.

6. Related Work

As mentioned in Section 2, Rubah employs the same approach to whole program updates as Kitsune [8], 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, Kitsune'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* [4] 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 [19] and DUSTM [20] 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 [18], for the purposes of enabling "stop-edit-continue" development (JRebel also targets this domain). Full-featured DSU is supported by the JVolve [22] and DCE VMs [24], though even these do not support some changes to the class hierarchy, which Rubah does. Since the DSU services are located inside the JVM itself, these systems can 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. And ultimately, that similar performance could be obtained while imposing only a fraction of the intrusion to VM code as previous works.

The JDrums [21] 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 [16] and multi-threaded [15] 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 [2] 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 [11]: 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 modest 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[17] 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.

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

[19] A. Orso, A. Rao, and M.J. Harrold. A technique for dynamic updating of java software. In *ICSM*, 2002.

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

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

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

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

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

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