Dynamic Software Updates: A VM-Centric Approach

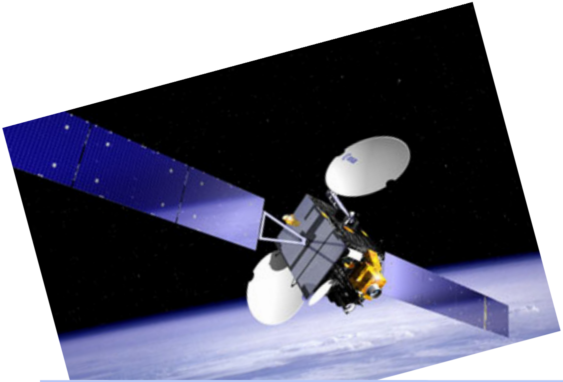
Suriya Subramanian

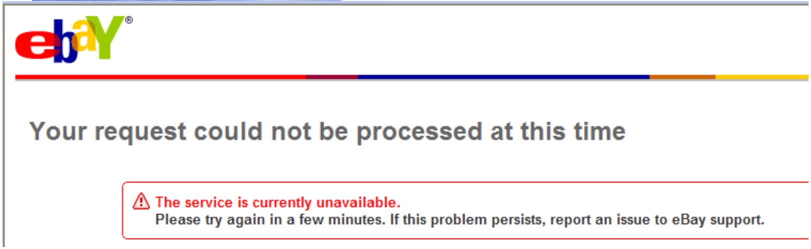
Department of Computer Science
The University of Texas at Austin

April 15, 2010

Motivation

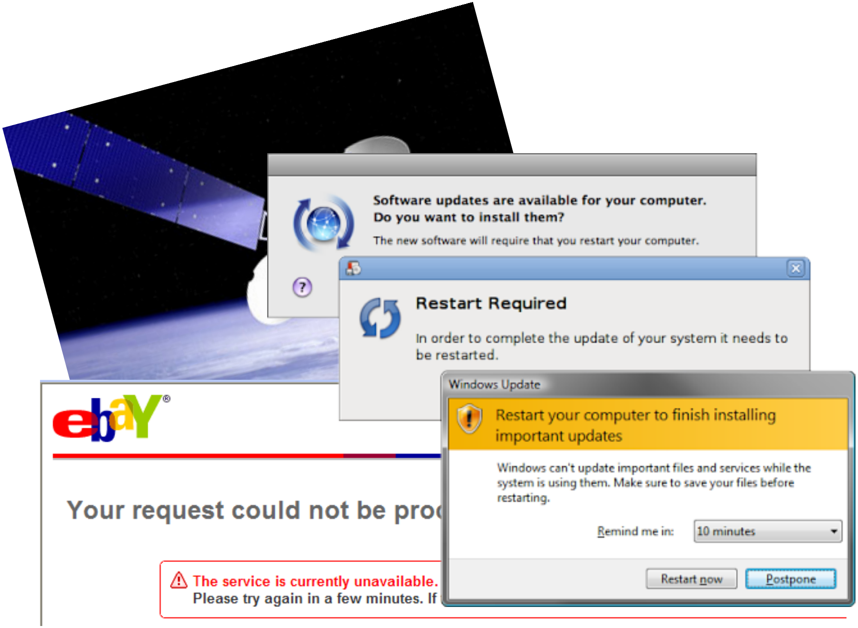
- Software applications change all the time
- Deployed systems must be updated with bug fixes, new features
- Updating typically involves: stop, apply patch, restart





Your request could not be processed at this time

The service is currently unavailable.
Please try again in a few minutes. If this problem persists, report an issue to eBay support.



Software updates are available for your computer.
Do you want to install them?

The new software will require that you restart your computer.

Restart Required

In order to complete the update of your system it needs to be restarted.

Windows Update

Restart your computer to finish installing important updates

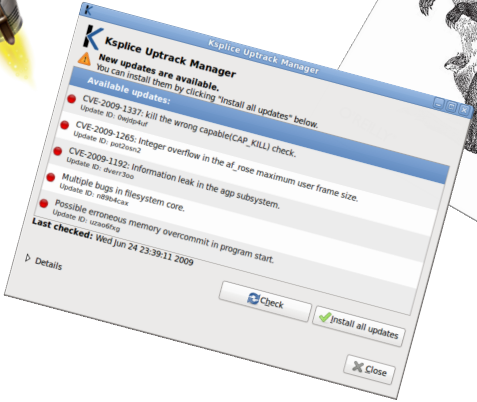
Windows can't update important files and services while the system is using them. Make sure to save your files before restarting.

Remind me in: 10 minutes

Your request could not be proc

The service is currently unavailable.
Please try again in a few minutes. If

Dynamic Software Updating in the real world



Ksplice Uptrack Manager

New updates are available.
You can install them by clicking "Install all updates" below.

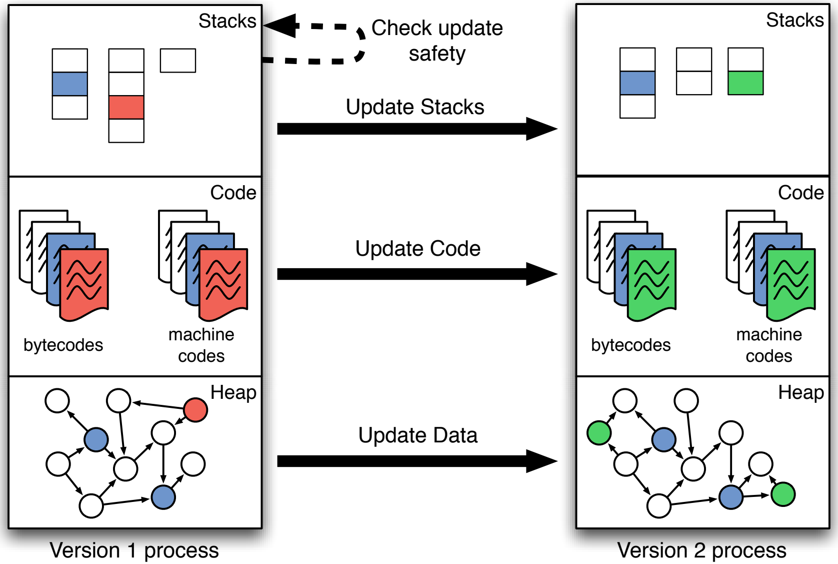
Available updates:

- CVE-2009-1337: kill the wrong capable(CAP_KILL) check.
  Update ID: 0wjdp4uf
- CVE-2009-1265: Integer overflow in the af_rose maximum user frame size.
  Update ID: pot2osn2
- CVE-2009-1192: Information leak in the agp subsystem.
  Update ID: dverr3oo
- Multiple bugs in filesystem core.
  Update ID: n89b4cax
- Possible erroneous memory overcommit in program start.
  Update ID: uzao6fxg

Last checked: Wed Jun 24 23:39:11 2009

The fundamental problem is losing state because of downtime.

Dynamic software updating



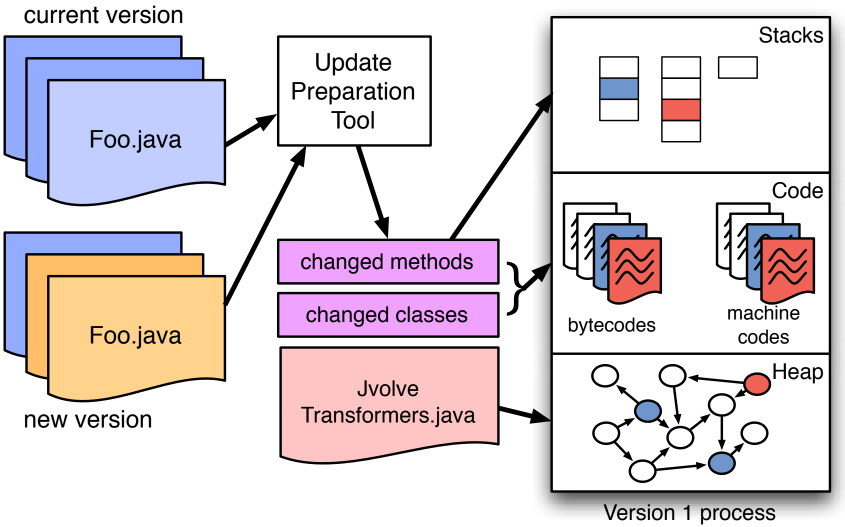
Thesis contributions

JVOLVE - a DSU-enabled Java Virtual Machine

Safe	Guarantees type-safe updates Relies on programmer for semantic-correctness
Flexible	Supports method body and signature changes across class hierarchies
Efficient	No overhead during normal execution
Easy to use	The stock application is DSU-ready No rewriting/recompilation required

Outline

JVOLVE - System overview



Supported updates

- Changes within the body of a method

```
  public static void main(String args[]) {
    System.out.println("Hello, World.");
+   System.out.println("Hello again, World.");
  }
```

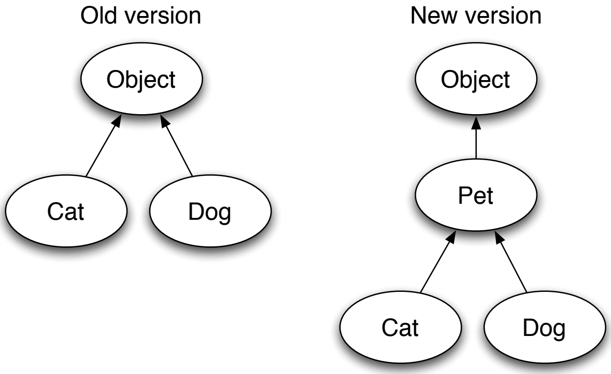
- Class signature updates
  - Add, remove, change the type signature of fields and methods

```
  public class Line {
-   private final Point2D p1, p2;
+   private final Point3D p1, p2;
    ...
  }
```

- Signature updates require an object transformer function
- Changes can occur at any level of the class hierarchy

Unsupported changes

- Renaming classes
- Changes to class hierarchy



Example of an update (JavaEmailServer)

```
  public class User {
    private final String username, domain, password;
-   private String[] forwardAddresses;
+   private EmailAddress[] forwardAddresses;
    public User(...) {...}
    public String[] getForwardedAddresses() {...}

    public void setForwardedAddresses(String[] f) {...}

  }

  public class ConfigurationManager {
    private User loadUser(...) {
       ...
       User user = new User(...);
       String[] f = ...;

       user.setForwardedAddresses(f);
       return user;
    }
  }
```

Example of an update (JavaEmailServer)

```
  public class User {
    private final String username, domain, password;
-   private String[] forwardAddresses;
+   private EmailAddress[] forwardAddresses;
    public User(...) {...}
-   public String[] getForwardedAddresses() {...}
+   public EmailAddress[] getForwardedAddresses() {...}
-   public void setForwardedAddresses(String[] f) {...}
+   public void setForwardedAddresses(EmailAddress[] f) {...}
  }

  public class ConfigurationManager {
    private User loadUser(...) {
       ...
       User user = new User(...);
-      String[] f = ...;
+      EmailAddress[] f = ...;
       user.setForwardedAddresses(f);
       return user;
    }
  }
```

Example of an update (JavaEmailServer)

```
public class v131_User {
  private final String username, domain, password;
  private String[] forwardAddresses;
}
public class JvolveTransformers {
 ...
 public static void jvolveClass(User unused) {}
 public static void jvolveObject(User to, v131_User from) {
    to.username = from.username;
    to.domain = from.domain;
    to.password = from.password;
    // to.forwardAddresses = null;
    int len = from.forwardAddresses.length;
    to.forwardAddresses = new EmailAddress[len];
    for (int i = 0; i < len; i++) {
      to.forwardAddresses[i] =
        new EmailAddress(from.forwardAddresses[i]);
}}}
```

Stub generated by UPT for the old version

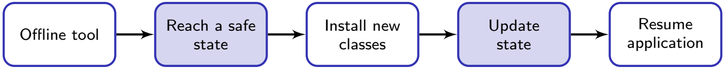
Default transformer copies old fields, initializes new ones to `null`

Example of an update (JavaEmailServer)

```
public class v131_User {
  private final String username, domain, password;
  private String[] forwardAddresses;
}
public class JvolveTransformers {
 ...
 public static void jvolveClass(User unused) {}
 public static void jvolveObject(User to, v131_User from) {
    to.username = from.username;
    to.domain = from.domain;
    to.password = from.password;
    // to.forwardAddresses = null;
    int len = from.forwardAddresses.length;
    to.forwardAddresses = new EmailAddress[len];
    for (int i = 0; i < len; i++) {
      to.forwardAddresses[i] =
        new EmailAddress(from.forwardAddresses[i]);
}}}
```

Stub generated by UPT for the old version

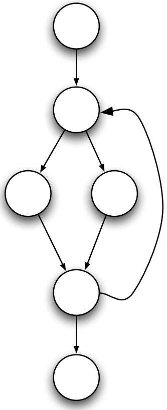
Update process



- Offline Update Preparation Tool
- JVOLVE VM
  - Reach a DSU safe point
  - Install new classes
  - Update state
  - Resume execution

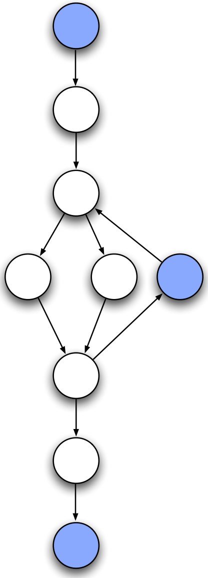
Safe point for the update

- Updates happen at “safe points”
- Safe points are VM yield points, and restrict what methods can be on stack
- Extend the thread scheduler to suspend all application threads
- If any stack has a restricted method, delay the update



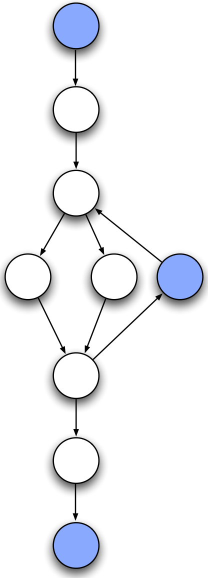
Safe point for the update

- Updates happen at "safe points"
- Safe points are VM yield points, and restrict what methods can be on stack
- Extend the thread scheduler to suspend all application threads
- If any stack has a restricted method, delay the update



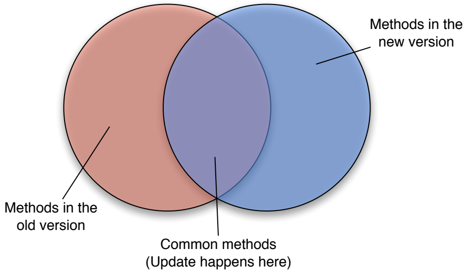
Safe point for the update

- Updates happen at “safe points”
- Safe points are VM yield points, and restrict what methods can be on stack
- Extend the thread scheduler to suspend all application threads
- If any stack has a restricted method, delay the update



Restrict changed methods (activeness safety)

- Update happens atomically
- Only old code runs before update, only new after update
- Do not allow changed methods to be active on stack
- Guarantees type safety
- Old and new version methods are independently type-safe


Methods in the new version
Methods in the old version
Common methods (Update happens here)

Restricted methods

(1) Methods changed by the update

(2) Methods identified by the user as unsafe based on semantic information about the application

Install return barriers that trigger DSU upon unsafe method's return

(3) Methods whose bytecode is unchanged, but compiled representation is changed by the update

- Offsets of fields and methods hard-coded in machine code
- Inlined callees may have changed

Utilize on-stack replacement to recompile base-compiled methods

Restricted methods

(1) Methods changed by the update

(2) Methods identified by the user as unsafe based on semantic information about the application

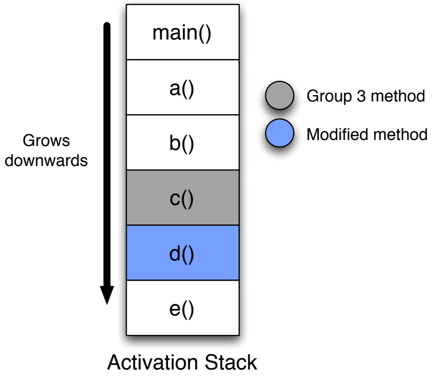
Install return barriers that trigger DSU upon unsafe method's return

(3) Methods whose bytecode is unchanged, but compiled representation is changed by the update

- Offsets of fields and methods hard-coded in machine code
- Inlined callees may have changed

Utilize on-stack replacement to recompile base-compiled methods

Reaching a safe point



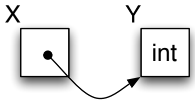
Install a return barrier for d(). Wait till it returns. On-stack replace new machine code for c().

Outline

Object Transformers Model

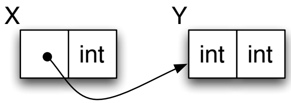
Old version

```
class X { Y y; }
class Y { int i; }
```



New version

```
class X { Y y; int z;}
class Y { int i; int j; }
```



```
Transformer for X:
    to.y = from.y;
    to.z = 0;
Transformer for Y:
    to.i = from.i;
    to.j = 0;
```

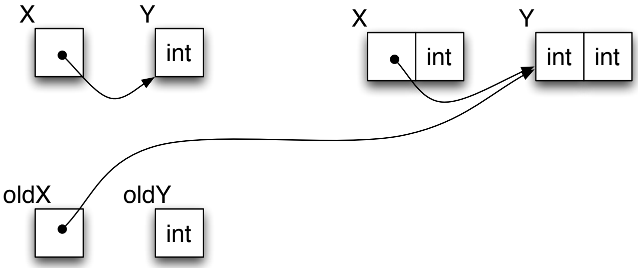
Object Transformers Model

Old version

```
class X { Y y; }
class Y { int i; }
```

New version

```
class X { Y y; int z;}
class Y { int i; int j; }
```



Old version stub

```
class oldX { Y y; }
class oldY { int i; }
```

Object Transformers Model

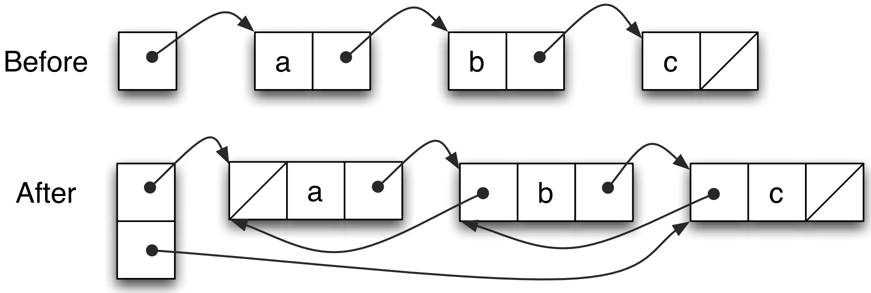
- Simple to reason about
- Transformers written in the source language
- All references are to the newest version
- Ensure that an object is transformed before reading its fields

Transforming objects in the GC

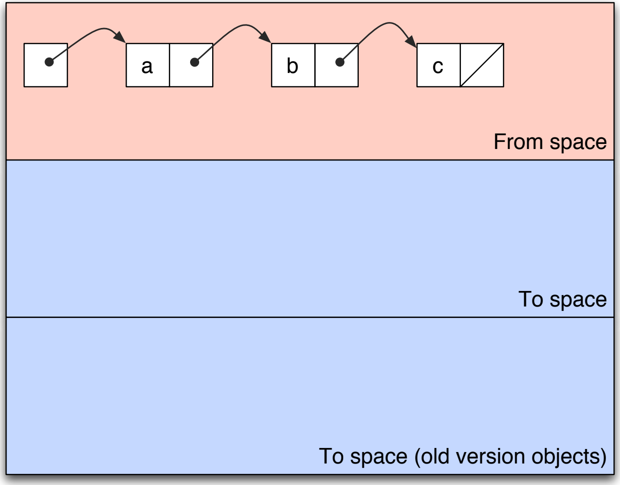
Happens in two steps

- Garbage collector creates an additional empty copy for updated objects
- Walk through and transform all these objects

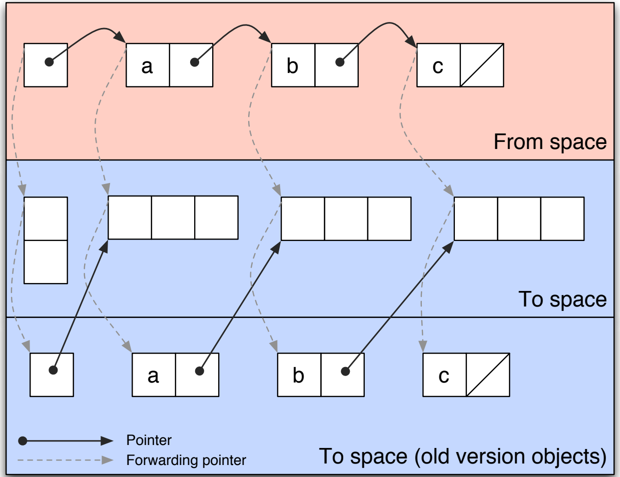
Example



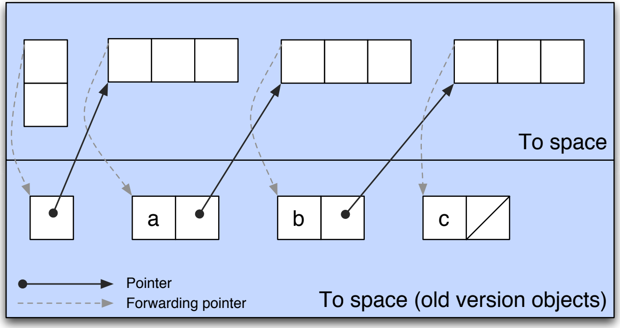
Singly-linked to doubly-linked list



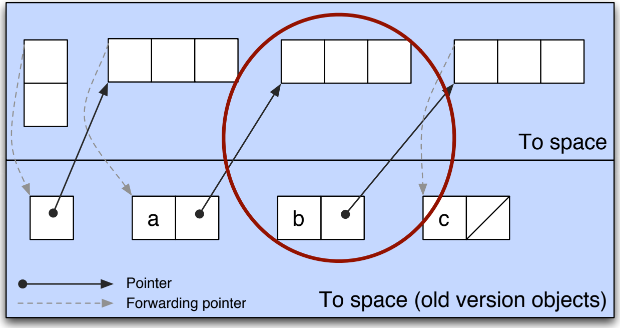
Singly-linked to doubly-linked list



Singly-linked to doubly-linked list

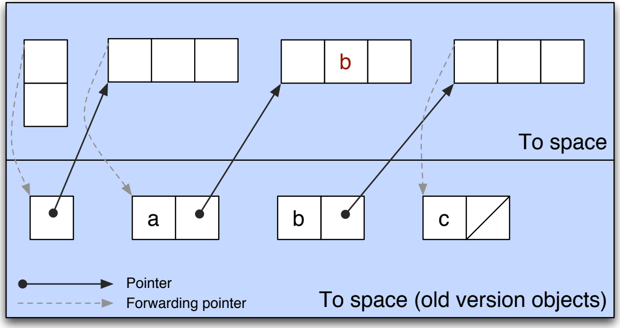


Singly-linked to doubly-linked list



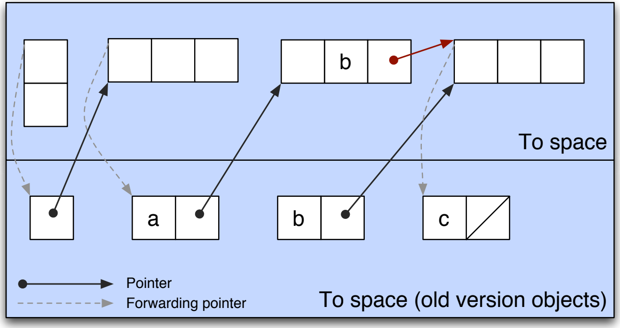
```
jvolveObject(Node to,
    old_Node from) {
  to.data = from.data;
  to.next = from.next;
  if (to.next != null)
    to.next.prev = to;
}
```

Singly-linked to doubly-linked list



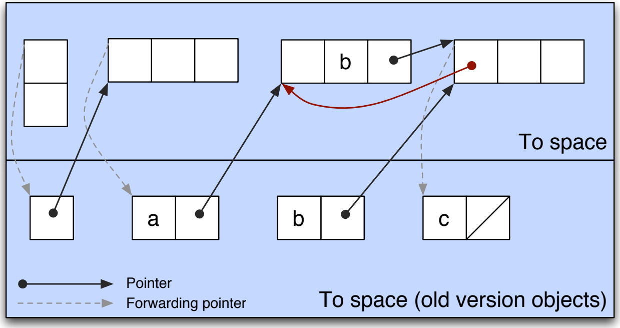
```
jvolveObject(Node to,
    old_Node from) {
  to.data = from.data;
  to.next = from.next;
  if (to.next != null)
    to.next.prev = to;
}
```

Singly-linked to doubly-linked list



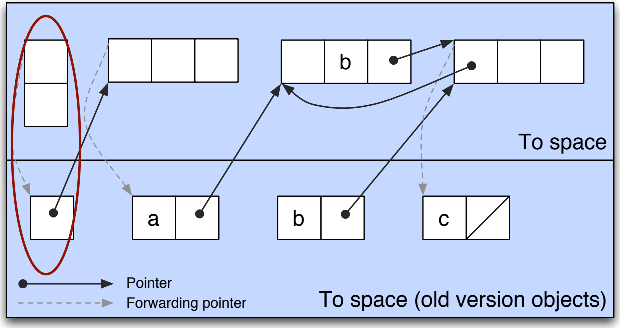
```
jvolveObject(Node to,
    old_Node from) {
  to.data = from.data;
  to.next = from.next;
  if (to.next != null)
    to.next.prev = to;
}
```

Singly-linked to doubly-linked list

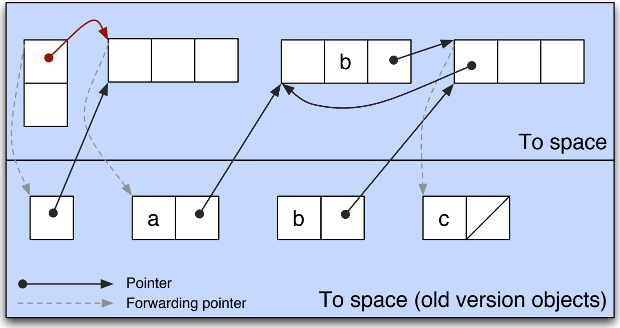


```
jvolveObject(Node to,
    old_Node from) {
  to.data = from.data;
  to.next = from.next;
  if (to.next != null)
    to.next.prev = to;
}
```

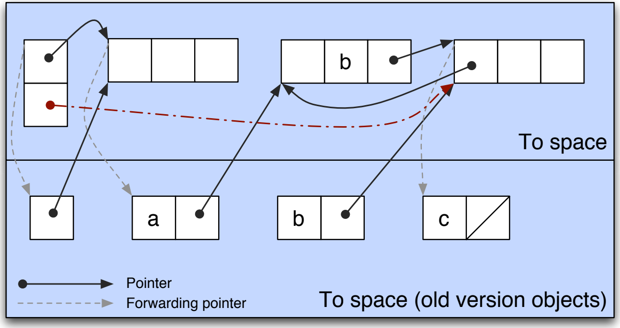
Singly-linked to doubly-linked list



Singly-linked to doubly-linked list



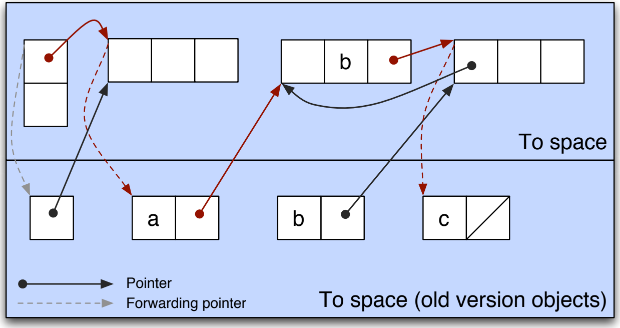
Singly-linked to doubly-linked list



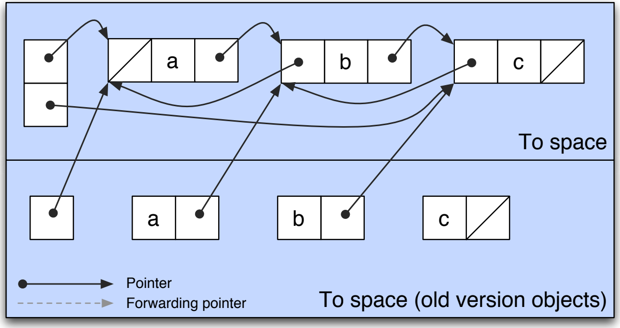
Finding the list's tail

```
Node prev = null;
Node current = from.head;
while (current != null) {
  prev = current;
  if (! VM.is_transformed(current)) {
    r0_Node current_old = VM.old_version_object(current);
    current = current_old.next;
  } else {
    current = current.next;
  }
}
to.tail = prev;
```

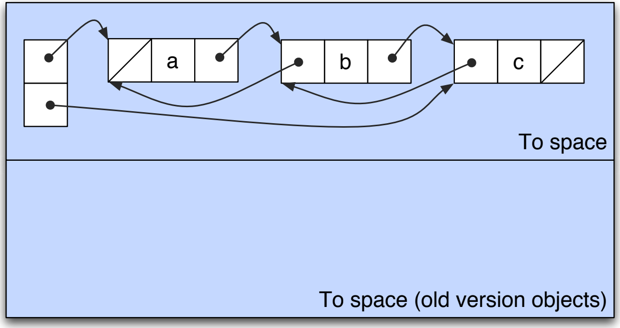
Singly-linked to doubly-linked list



Singly-linked to doubly-linked list



Singly-linked to doubly-linked list



Outline

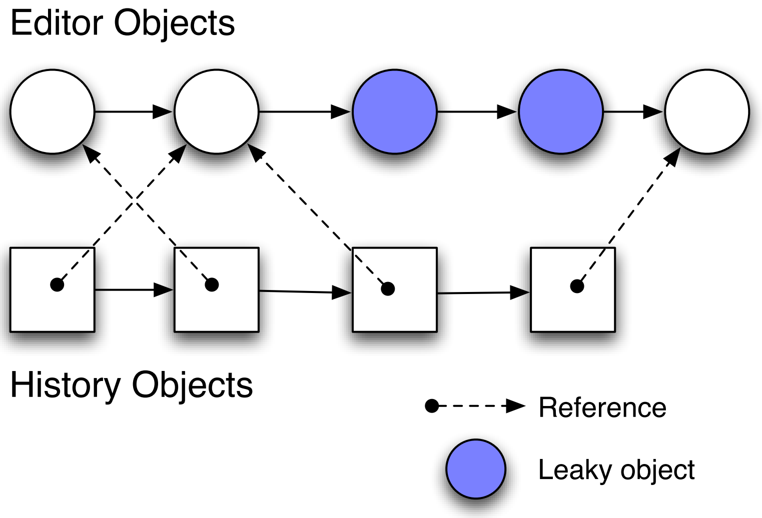
Recovering app. state with transformers

- Transformer logic depends on state of object
- Used to repair application state, for instance fix a memory leak
  - Fix code so that application does not leak memory
  - Repair data at update time to remove past leak (in a subset of objects)

    -
      ```
      if leaky-object(o):
        o.field = null
      ```

    -
      ```
      if leaky-object-in-collection(c, o):
        c.remove(o)
      ```

Eclipse Diff Leak



Eclipse Diff Leak (State transformer)

```
public static void jvolveObject(NavigationHistory to) {
  // set all refcounts to 0
  for (Editor e : to.editors)
    e.refcount = 0;

  // recompute refcounts
  for (History h : to.history)
    if (h.editor != null)
      h.editor.refcount++;

  // free leaky objects
  for (Editor e : to.editors)
    if (e.refcount == 0)
      nh.editors.remove(e);
}
```

Automatically generating fixes

```
  class Foo {
    void bar() {
      ...
      if (...) {
        ...
+       this.field = null;
        ...
      }
    }
  }
```

- Identify leaky objects from heap state, at update time?
- Dynamic analysis to discover predicate that identifies leaky objects
- Not just leaks, can be used to generate the correct state transformer

Azureus patch

- Adds a new field `BaseMdiEntry.isExpanded`
- Default transformer would set this field to `false`
- Instead, we discover the following property

```
this.isExpanded == this.soParent.paintListenerHooked
```

Outline

Application Experience

- Jetty webserver
  - 11 versions, 5.1.0 through 5.1.10, 1.5 years
  - 45 KLOC
- JavaEmailServer
  - 10 versions, 1.2.1 through 1.4, 2 years
  - 4 KLOC
- CrossFTP server
  - 4 versions, 1.05 through 1.08, more than a year
  - 18 KLOC

What works

Support 20 of 22 updates

- 13 updates change class signature by adding new fields
- Several updates require On-stack replacement support
- Two versions update an infinite loop, postponing the update indefinitely

Unsupported updates

- JavaEmailServer 1.2.4 to 1.3
    - Update reworks the configuration framework of the server
    - Many classes are modified to refer to the configuration system
    - Including infinite loops in SMTP and POP threads
- Jetty 5.1.2 to 5.1.3
    - The application would never reach a safe point
    - Modified method `ThreadedServer.acceptSocket()` that waits for connections is nearly always on stack
    - Return barrier not sufficient since the main method in other threads `PoolThread.run()` is itself modified

Future Work

- Exhaustive testing of updates to multi-threaded applications
- Language support for expressing changes
- DSU-compatible refactoring and IDEs

Conclusion

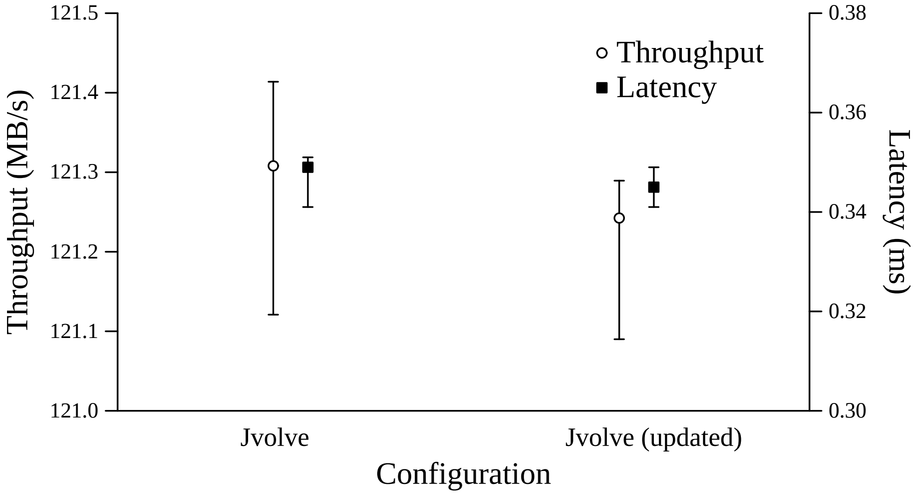
- JVOLVE, a Java VM with support for Dynamic Software Updating
- Most-featured, best-performing DSU system for Java
- Naturally extends existing VM services
- Supports about two years worth of updates

Dynamic software updating in managed languages can be achieved in a safe, flexible and efficient manner.

Backup slides

JVOLVE performance

No overhead during steady-state execution



Jetty webserver performance

- Used `httperf` to issue requests
- Both client and server on a the same machine, an Intel Core 2 Quad
- Report throughput and latency, median of 21 runs

DSU pause times

- JVOLVE performs a GC to transform objects
- Pause time determined by
  - Heap size
  - # of objects transformed
- Simple microbenchmark varying the fraction of objects transformed in a 1GB heap

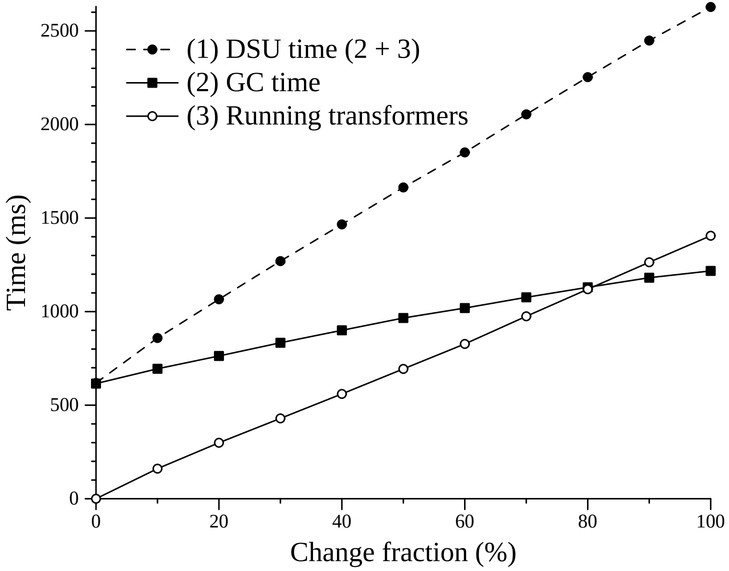
Update pause time

- No apriori overhead during normal execution (before and after the update)
- Only effect on execution time is the update pause time
    - Comparable to GC pause time

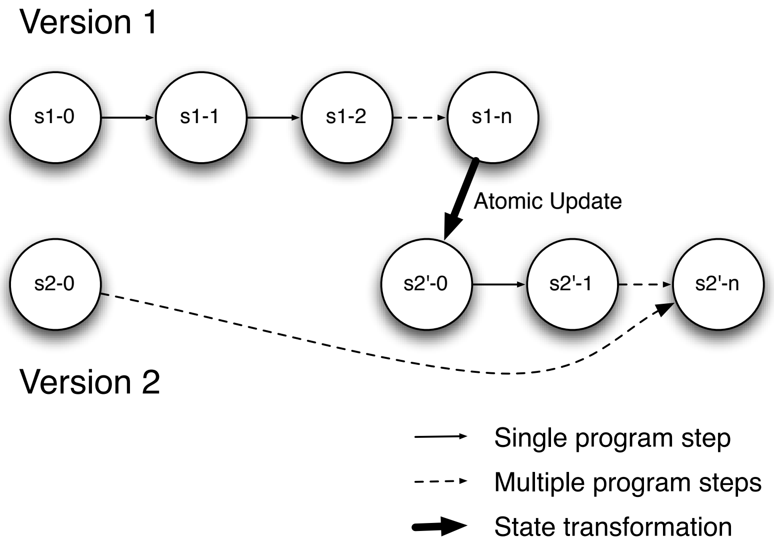
Update pause time

$$\begin{aligned}\text{DSU Pause Time} &\approxeq \text{Regular GC Time} + \\ &\quad\ \text{Time to allocate upd. objects} + \\ &\quad\ \text{Time to transform objects} \\ &\propto \text{Upd. objects fraction} \\ &\quad\ \text{Heap size}\end{aligned}$$

DSU pause times (microbenchmark)



Update correctness



Correct update semantics

Update correctness depends on semantics of application and state transformers

- Transformer that initializes all variables to "unknown" is equivalent to restarting the program. Not useful.
- When going from an 8-bit to 16-bit counter, no correct transformer exists

Safety Guarantees

- Correctness
    - Showing an update correct is undecidable
    - Rely on programmer and testing process for semantics and safety of update
- Well-formed updates
    - All data accesses and method calls in the program respect language semantics.
    - Type safety