Dynamic Software Updates: A VM-centric Approach

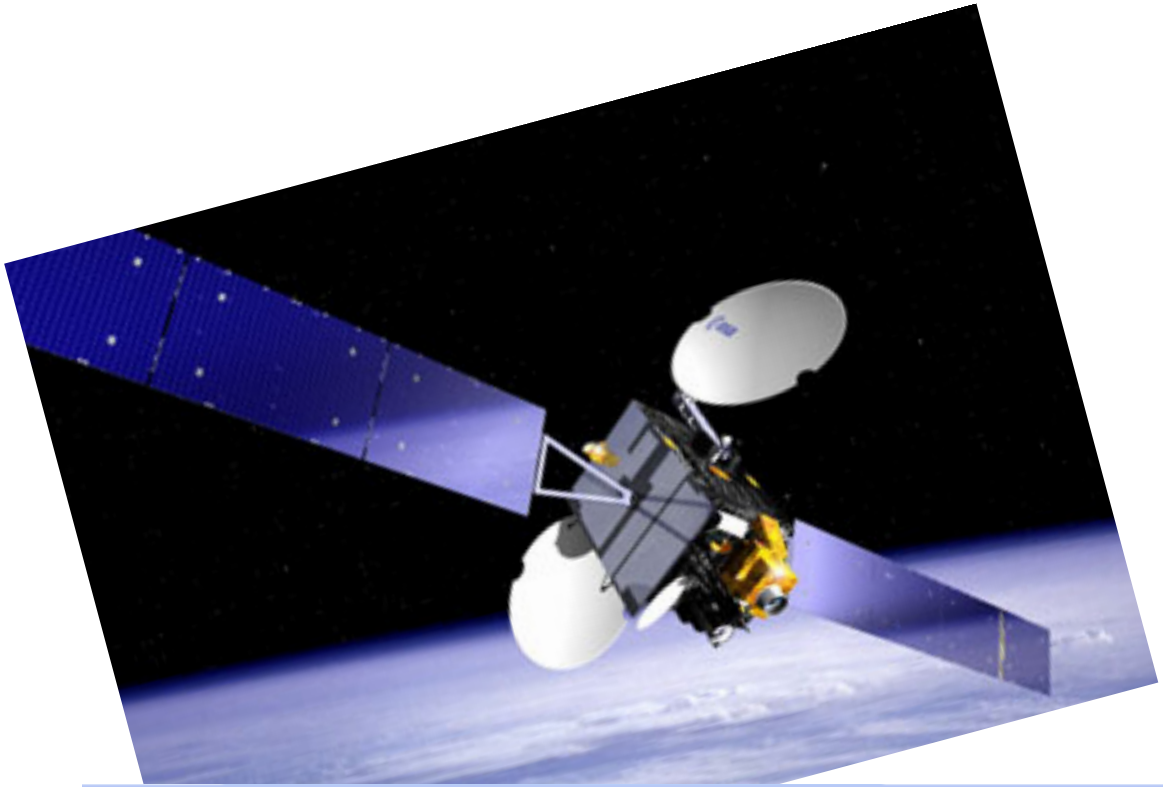
Suriya Subramanian

Department of Computer Sciences
The University of Texas at Austin

July 14, 2009

Motivation

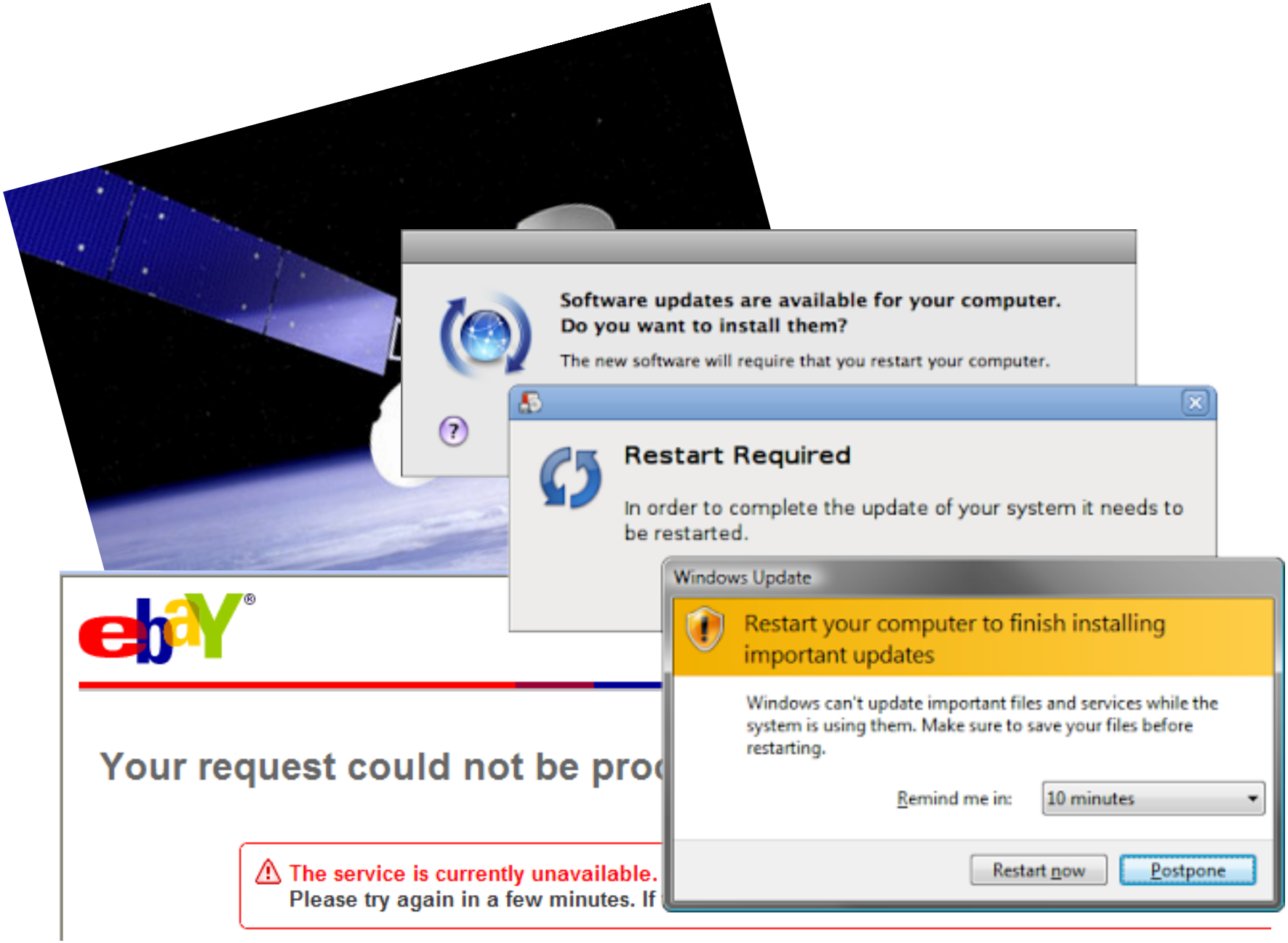
- Software applications change all the time
- Deployed systems must be updated with bug fixes, new features
- Updating typically involves: stop, apply patch, restart



ebay

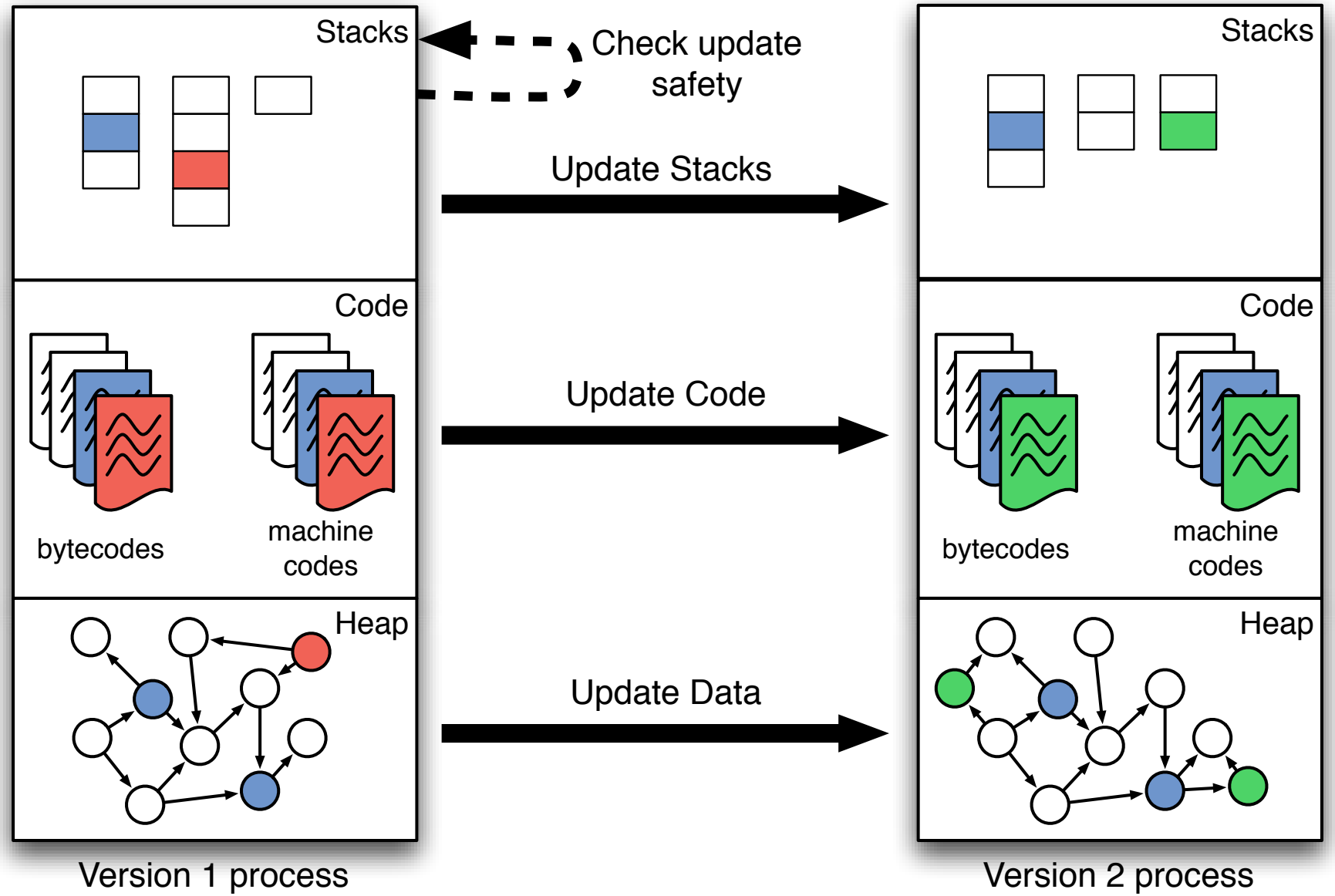
Your request could not be processed at this time

⚠ 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
Restart now
Postpone
ebay
Your request could not be proc
The service is currently unavailable.
Please try again in a few minutes. If

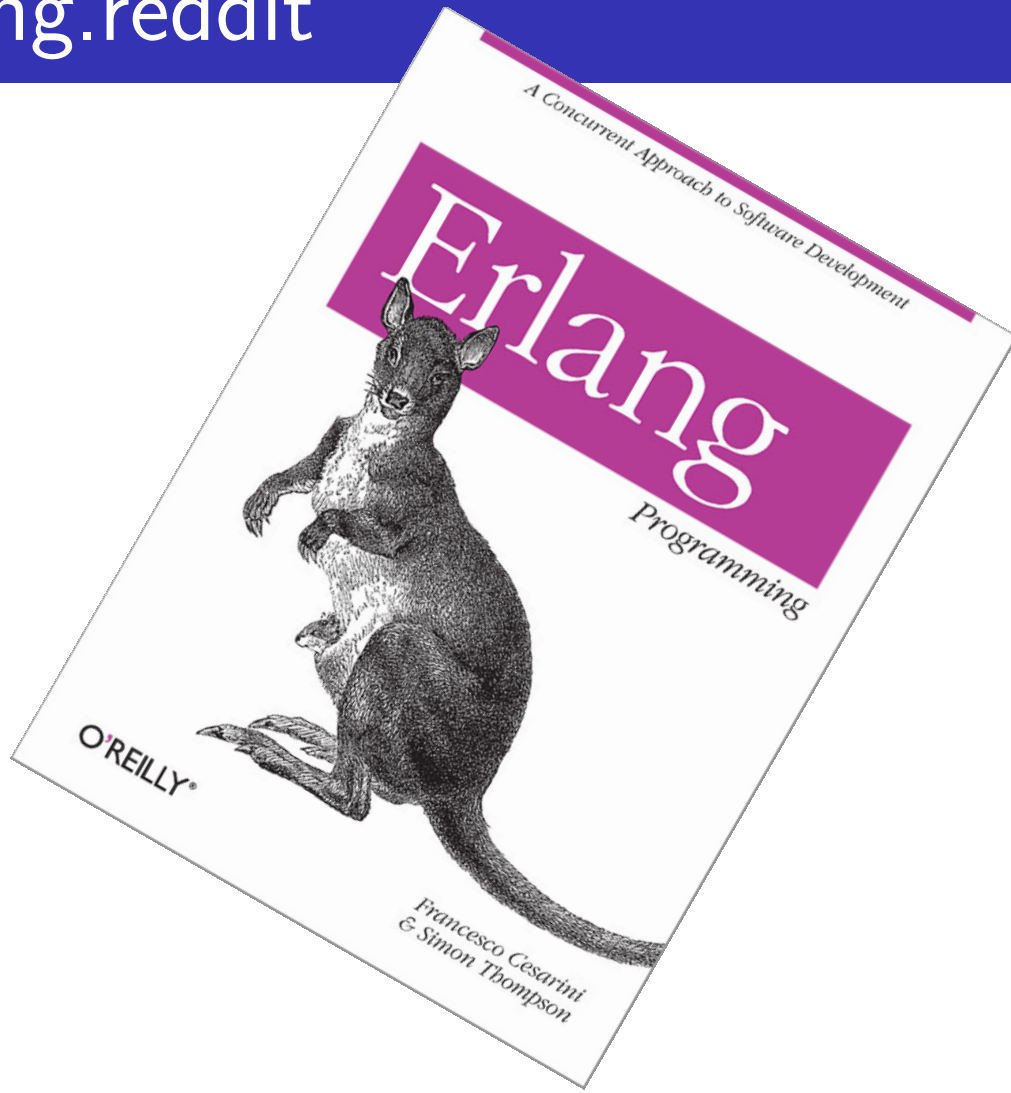
The fundamental problem is losing state because of downtime.

Dynamic software updating



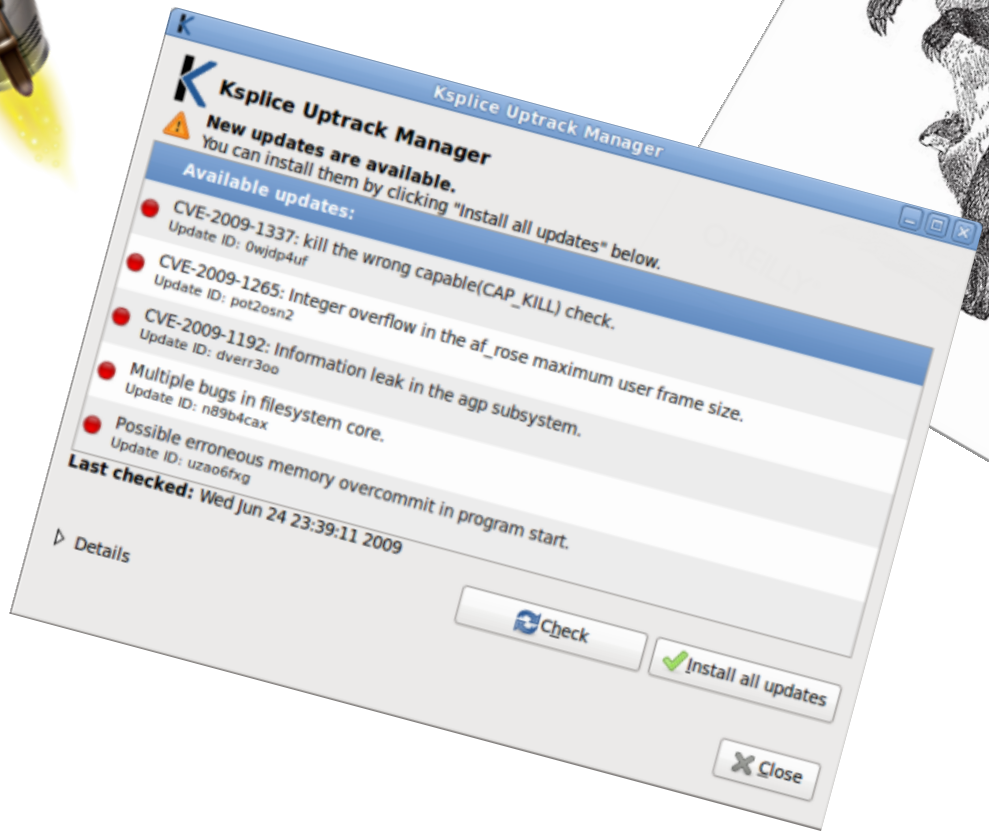
Overheard on programming.reddit

Overheard on programming.reddit



Overheard on programming.reddit

mozilla LABS
Jetpack



Dynamic updating systems

- Special-purpose architectures, application-specific solutions exist
- General-purpose solutions gaining strength
  - K42, Ksplice for OS updates
  - Polus, Ginseng for C applications
- Not for managed languages

DSU opportunity for managed languages

DSU Solutions for C/C++ typically

- Require special compilation
- Statically/dynamically insert indirection for function calls
- Restrict structure updates, require extra allocation
- Impose space/time overheads on normal execution
- Make type-safety for updates difficult
- Not multi-threaded

Our solution

- JVOLVE - a Java Virtual Machine with DSU support
- Key insight: Extend existing VM services
- No DSU-related overhead during normal execution
- Support updates to real world applications

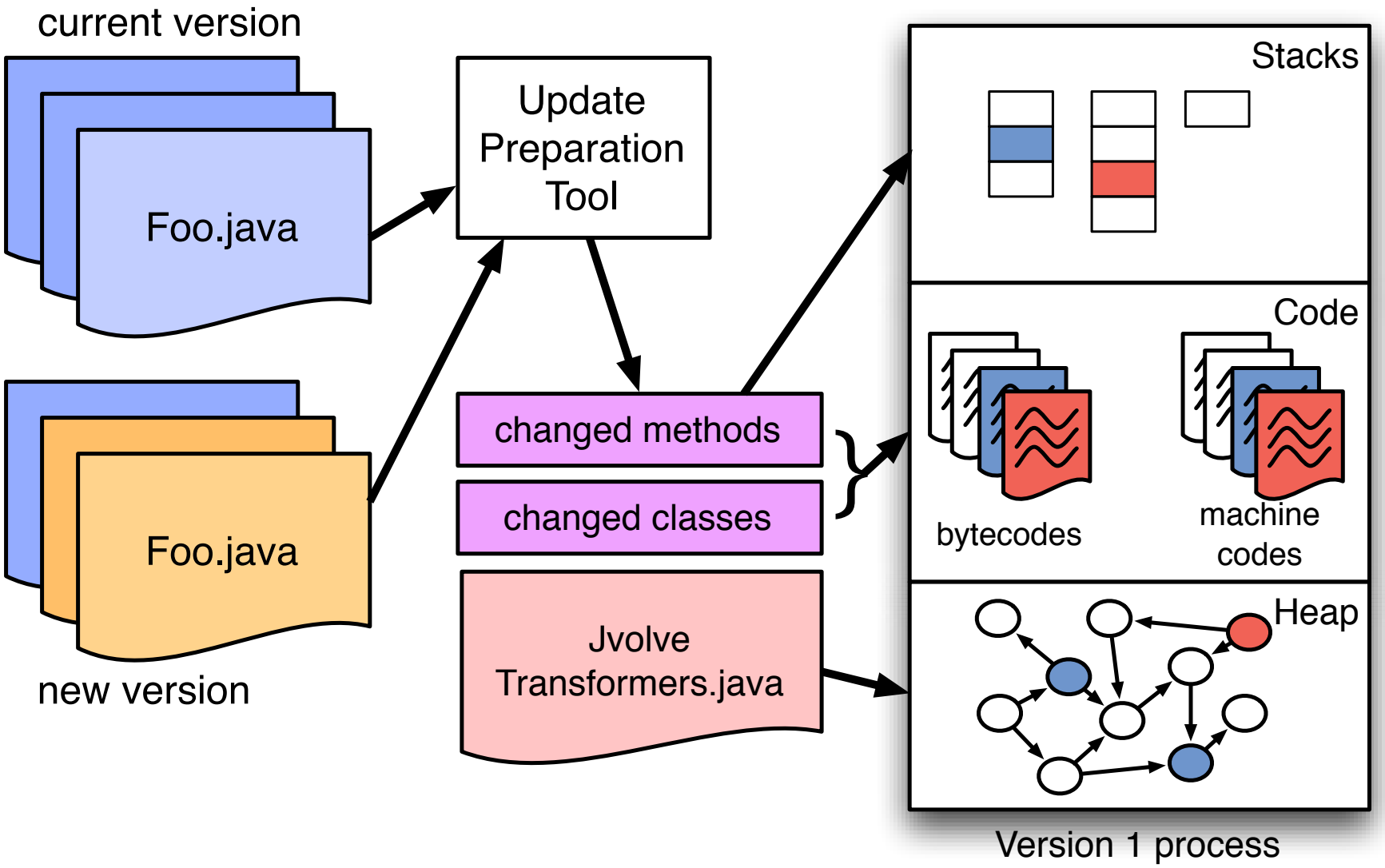
Dynamic software updating in managed languages can be achieved in a **safe**, **flexible** *and* **efficient** *manner by naturally extending existing VM services.*

DSU support should be a standard feature of future VMs.

Outline

- Introduction
- JVOLVE
  - VM Implementation
  - Evaluation
- Future Work
- Conclusion

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

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]);
}}}
```

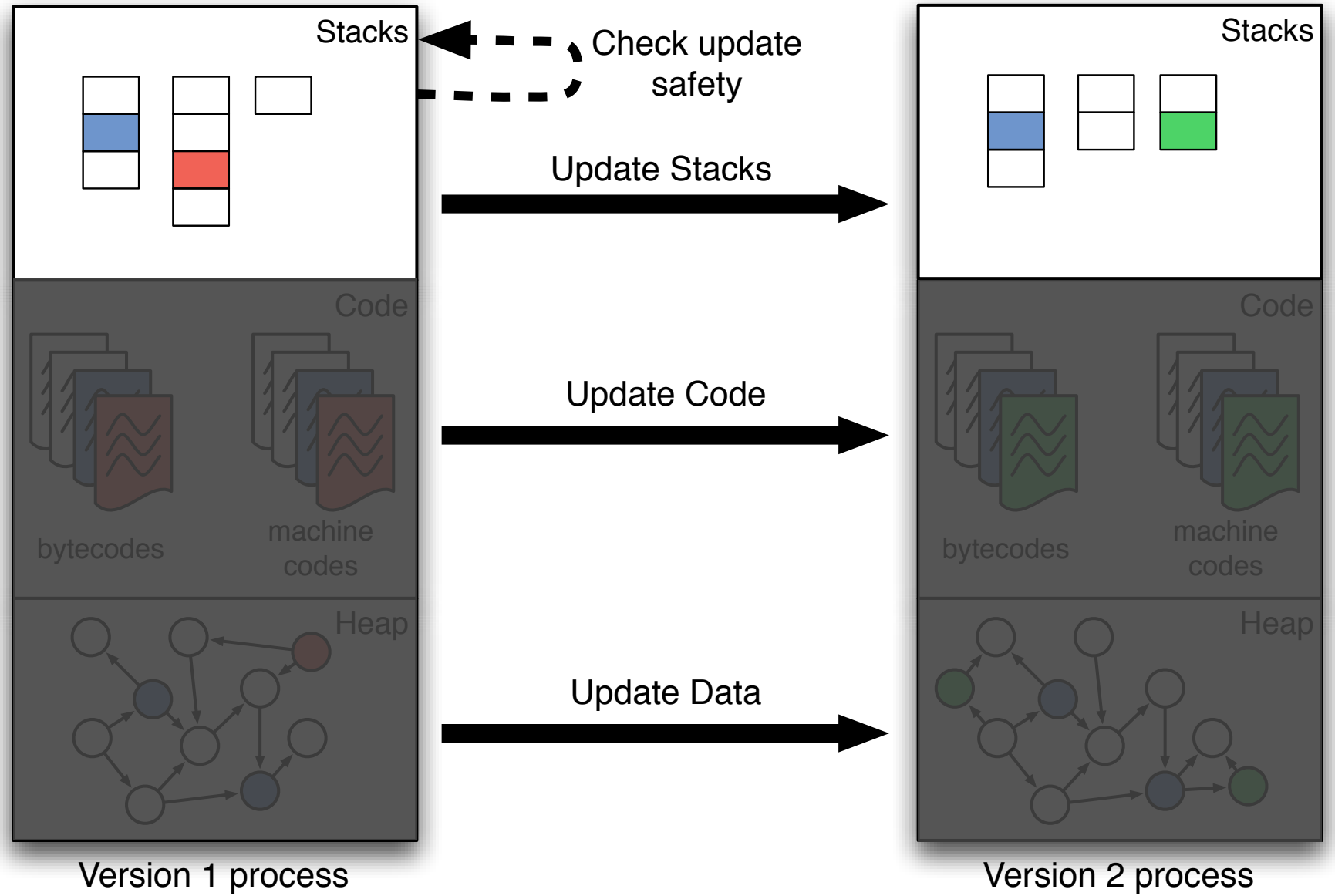
Stub generated by UPT for the old version

Compiling transformation functions

```
public static void jvolveObject(User to, v131_User from) {
    to.username = from.username;
    ...
}
```

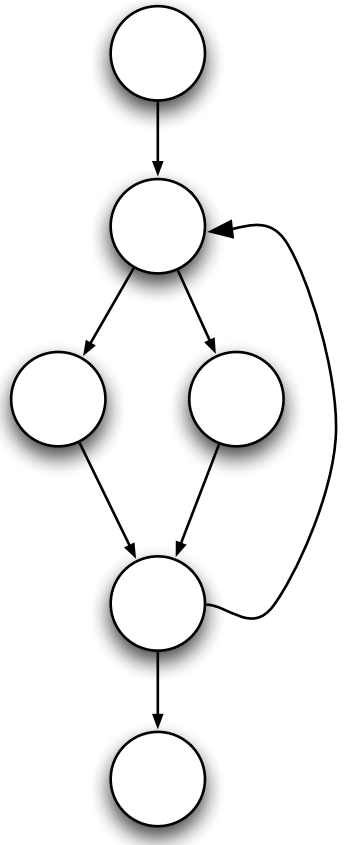
- Very close to Java semantics
- Field `username` is `private` and `final`
- Functions compiled specially by a JastAddJ extension to the Java language
- Ignores access protection and allows assigning to `final` fields

Check for update safety



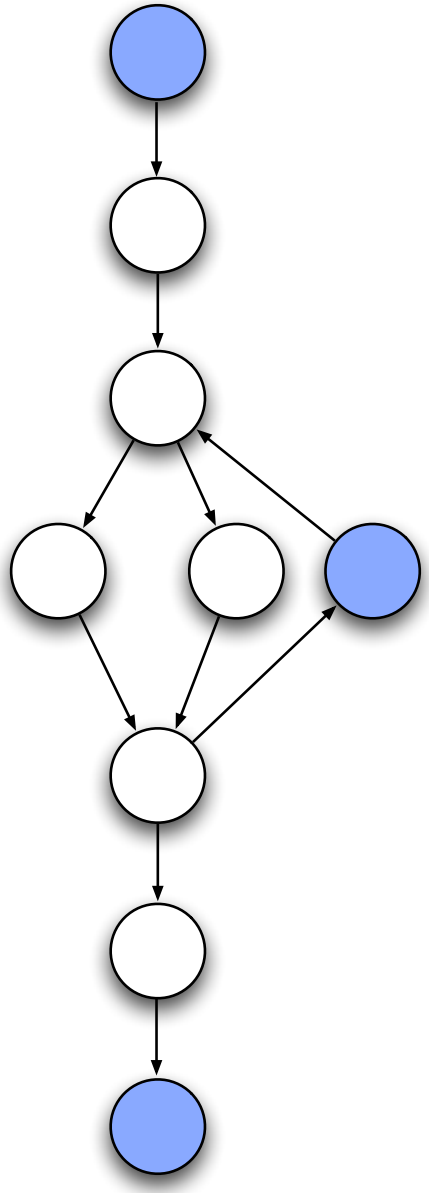
Safe point for the update

- Update must be atomic
- 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



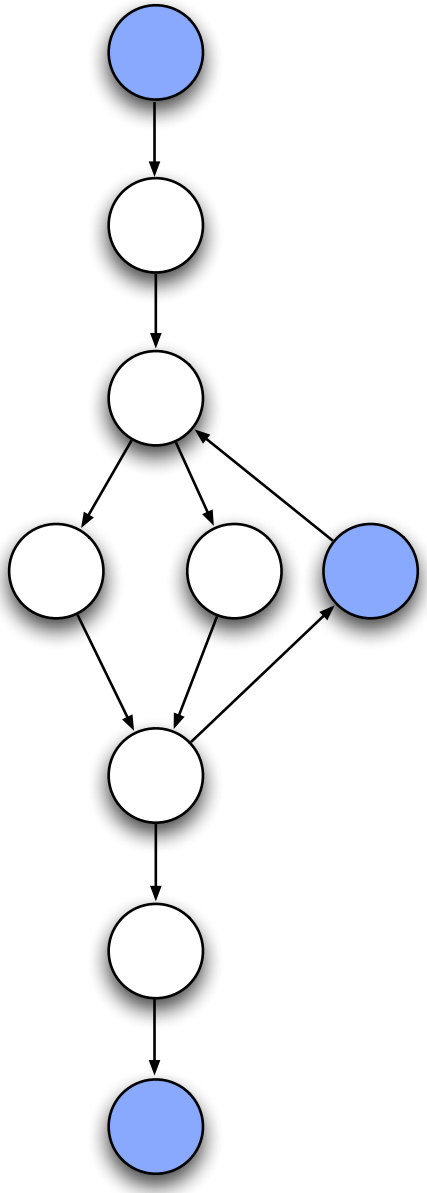
Safe point for the update

- Update must be atomic
- 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

- Update must be atomic
- 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



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

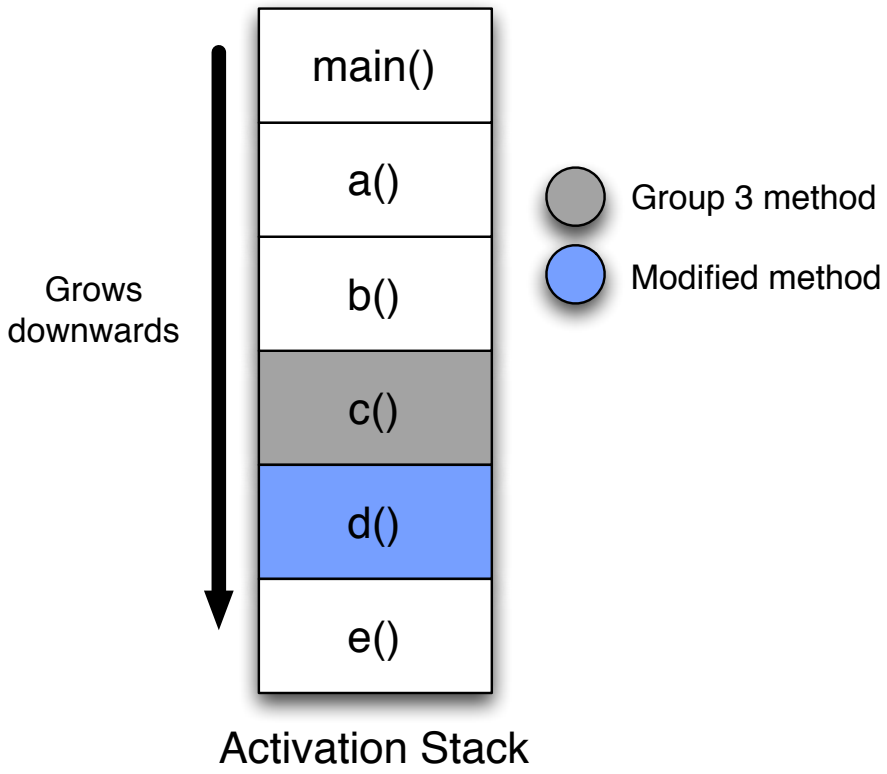
Install return barriers that trigger DSU upon unsafe method's return

(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

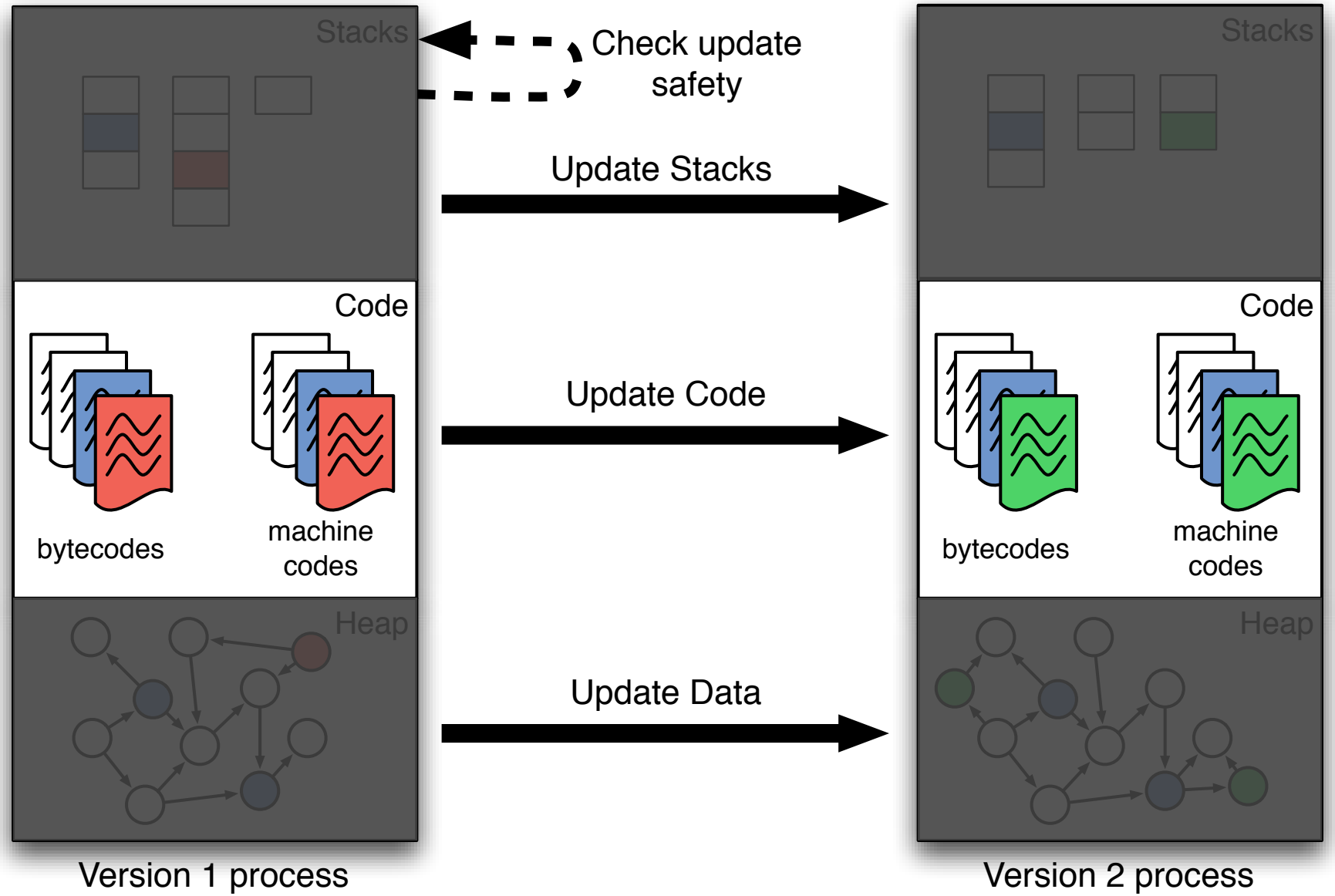
Utilize on-stack replacement to recompile base-compiled methods

Reaching a safe point



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

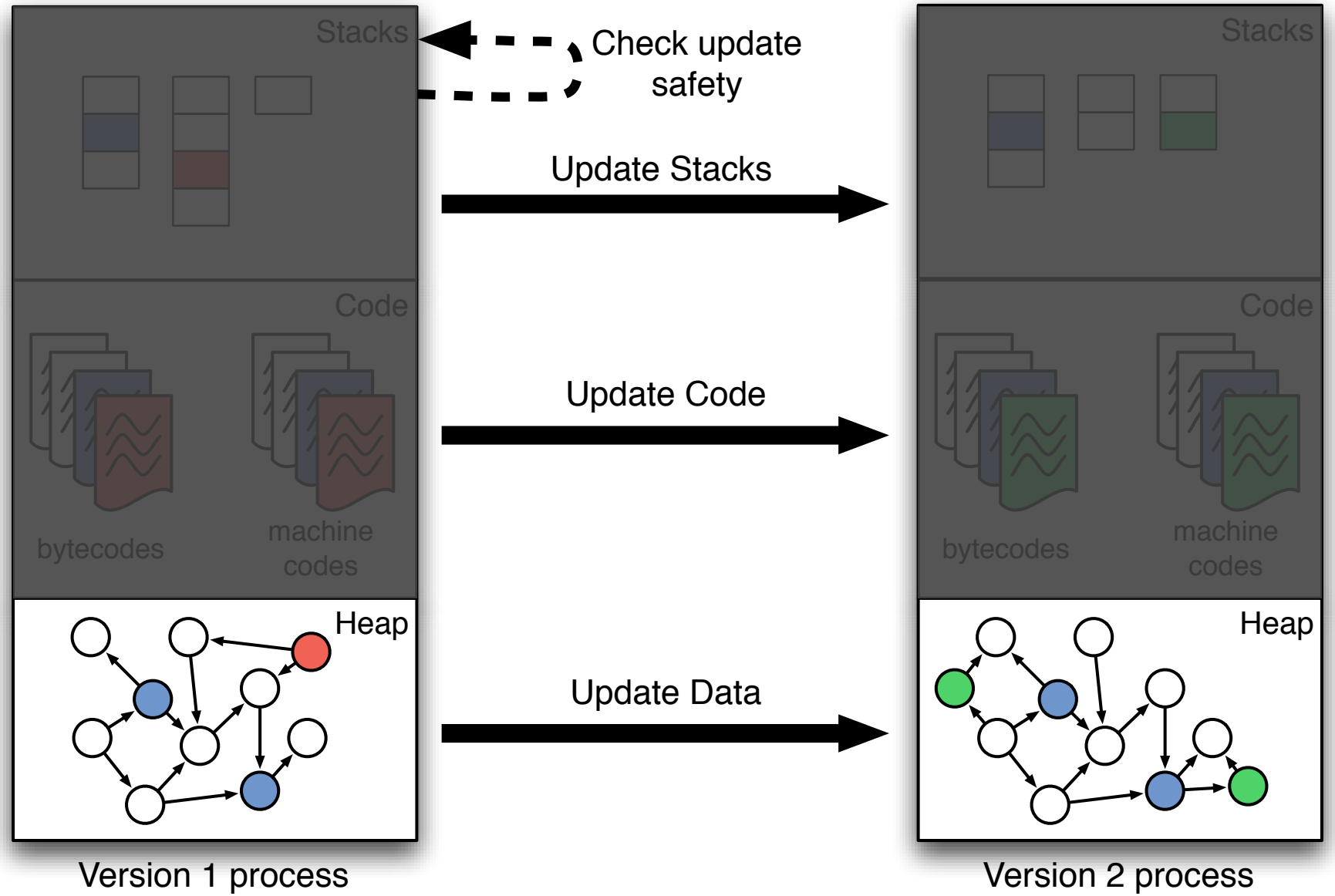
Update code


Stacks
Check update safety
Update Stacks
Code
bytecodes
machine codes
Update Code
Heap
Update Data
Version 1 process
Stacks
Code
bytecodes
machine codes
Heap
Version 2 process

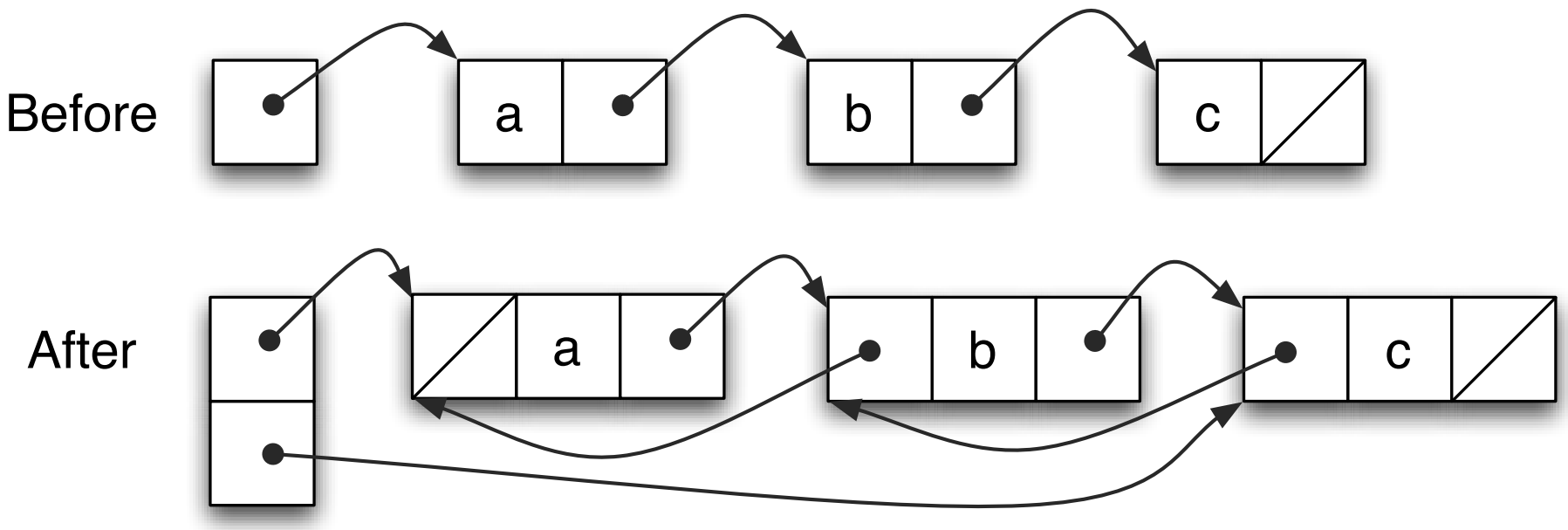
Update code

- Modify class loader to recognize new versions of classes
- Install new versions of classes and methods
- Rely on Just-in-time Compiler to compile new versions of methods on demand
- Extend On-stack replacement to update active methods

Update data


Stacks
Code
bytecodes
machine codes
Heap
Version 1 process
Check update safety
Update Stacks
Update Code
Update Data
Stacks
Code
bytecodes
machine codes
Heap
Version 2 process

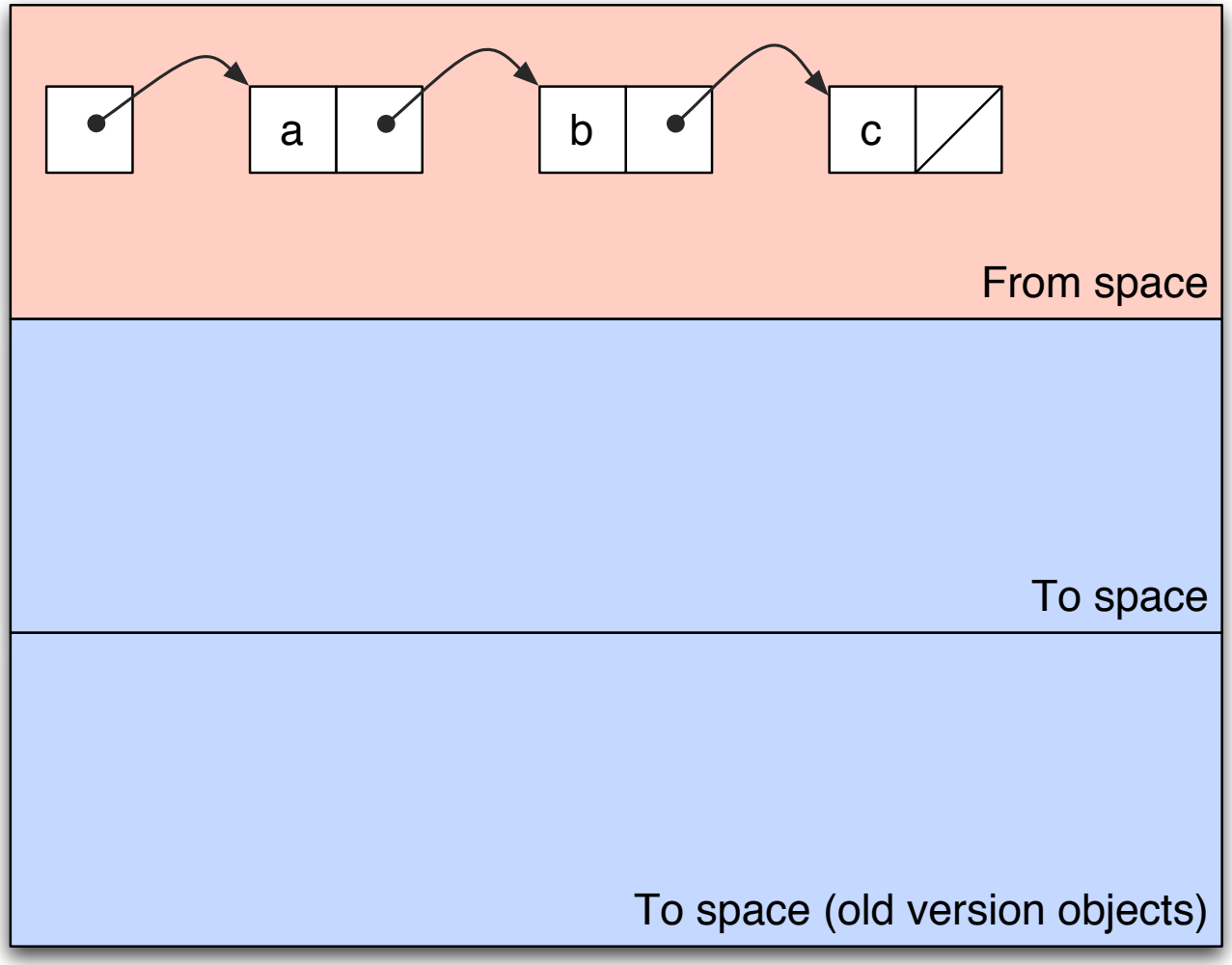
Transforming objects in the GC



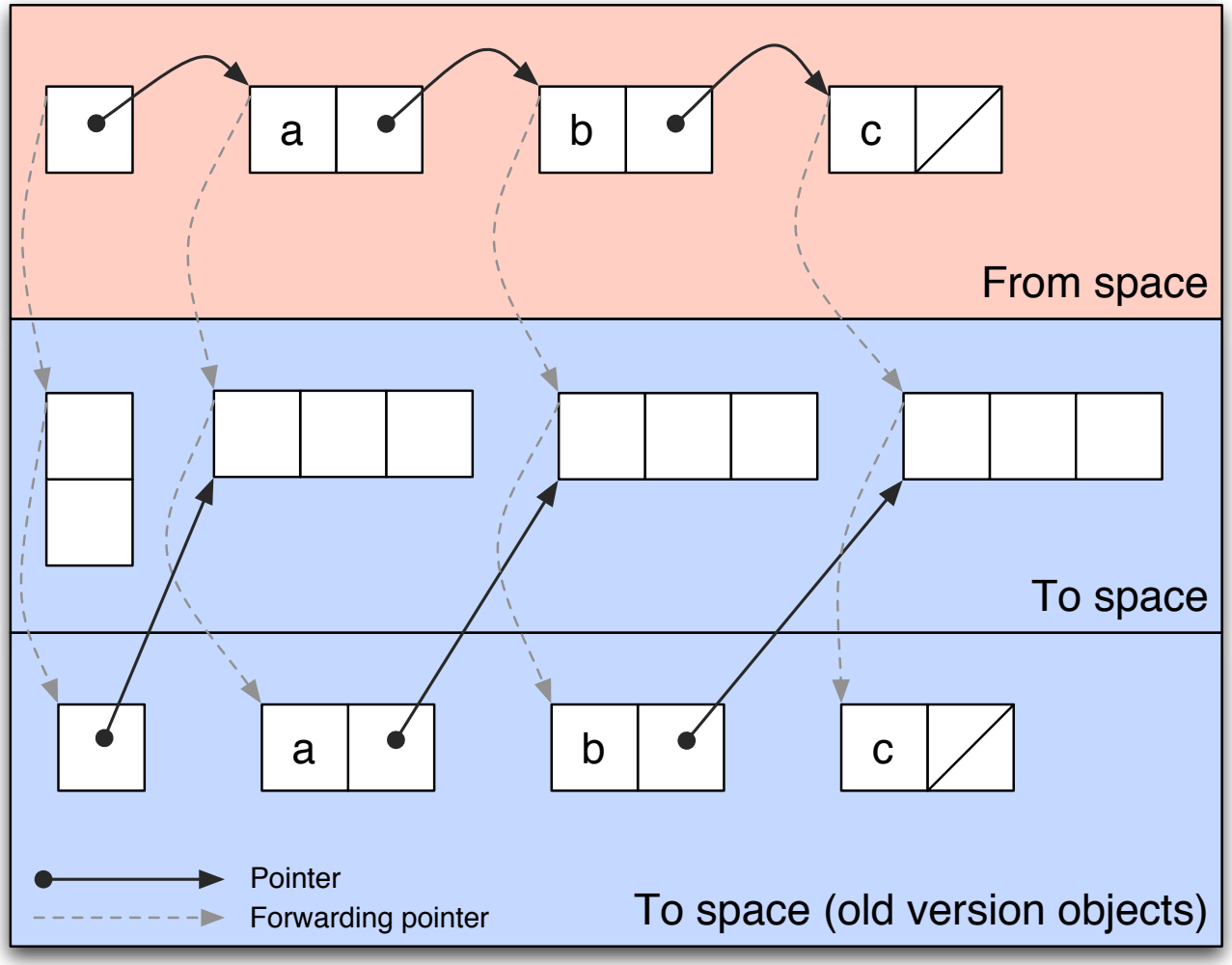
Happens in two steps

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

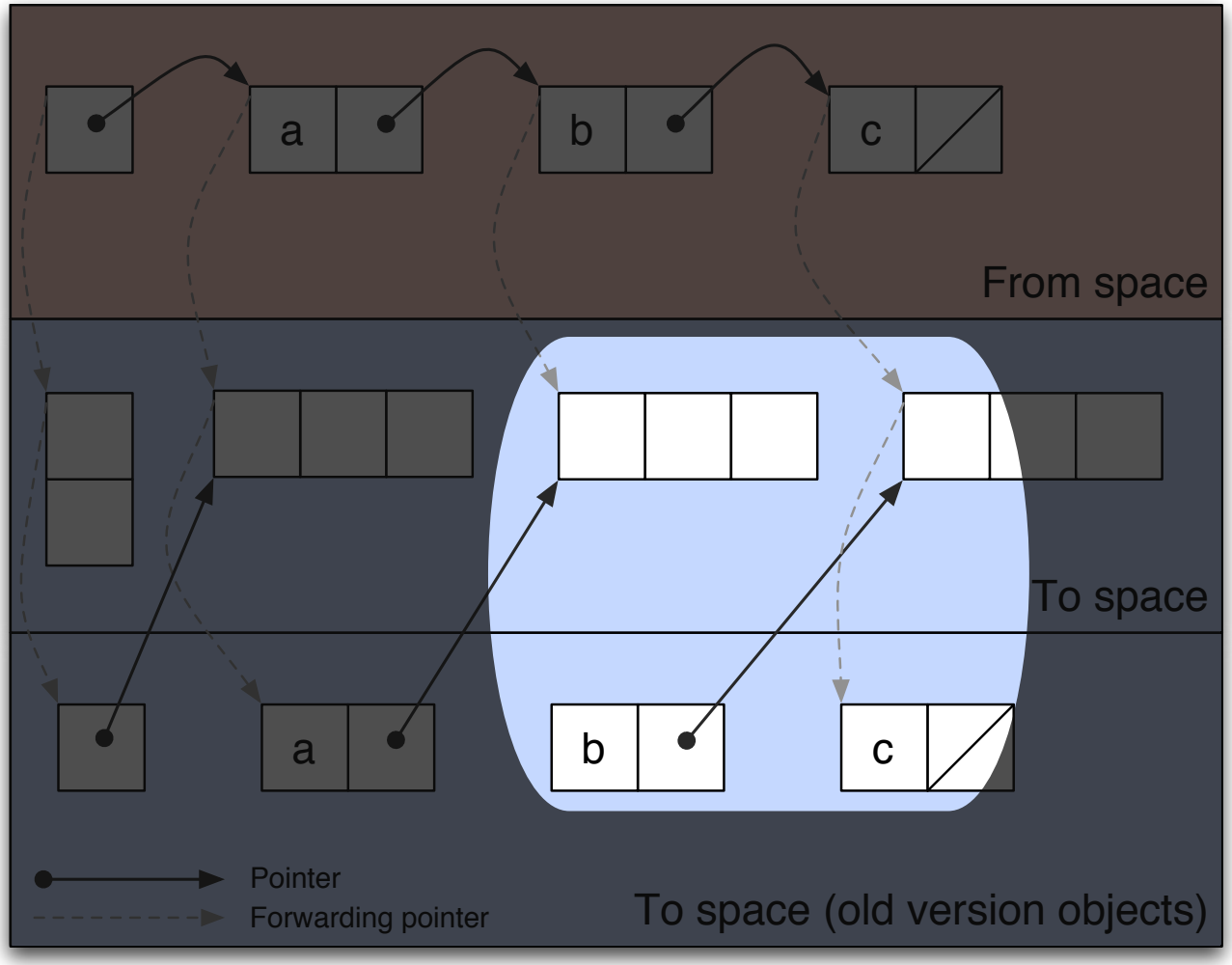
Jvolve GC



Jvolve GC

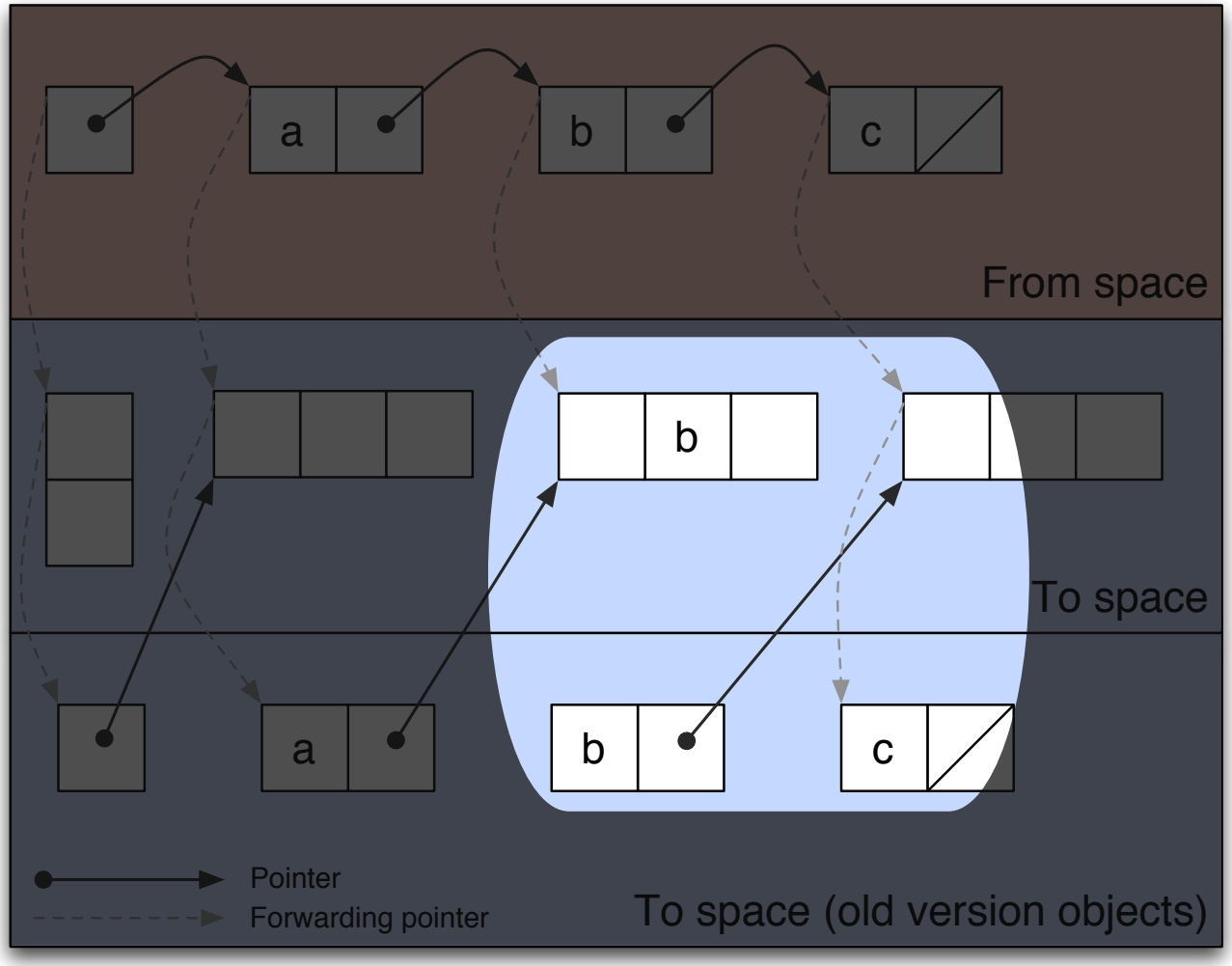

a
b
c
From space
To space
a
b
c
Pointer
Forwarding pointer
To space (old version objects)

Jvolve GC



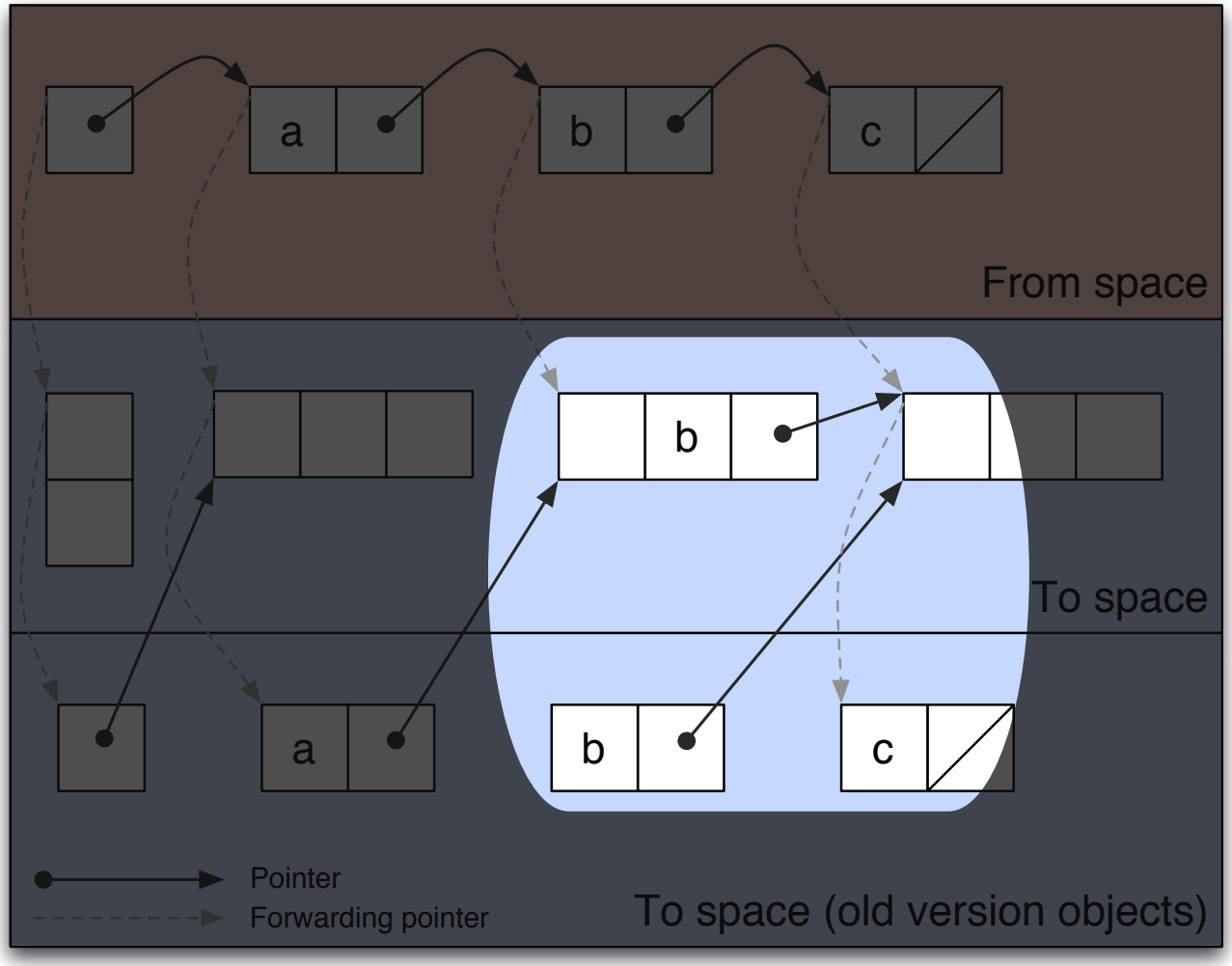

```
jvolveObject(Node to,
    old_Node from) {
  to.data = from.data;
  to.next = from.next;
  if (to.next != null)
    to.next.prev = to;
}
```

Jvolve GC



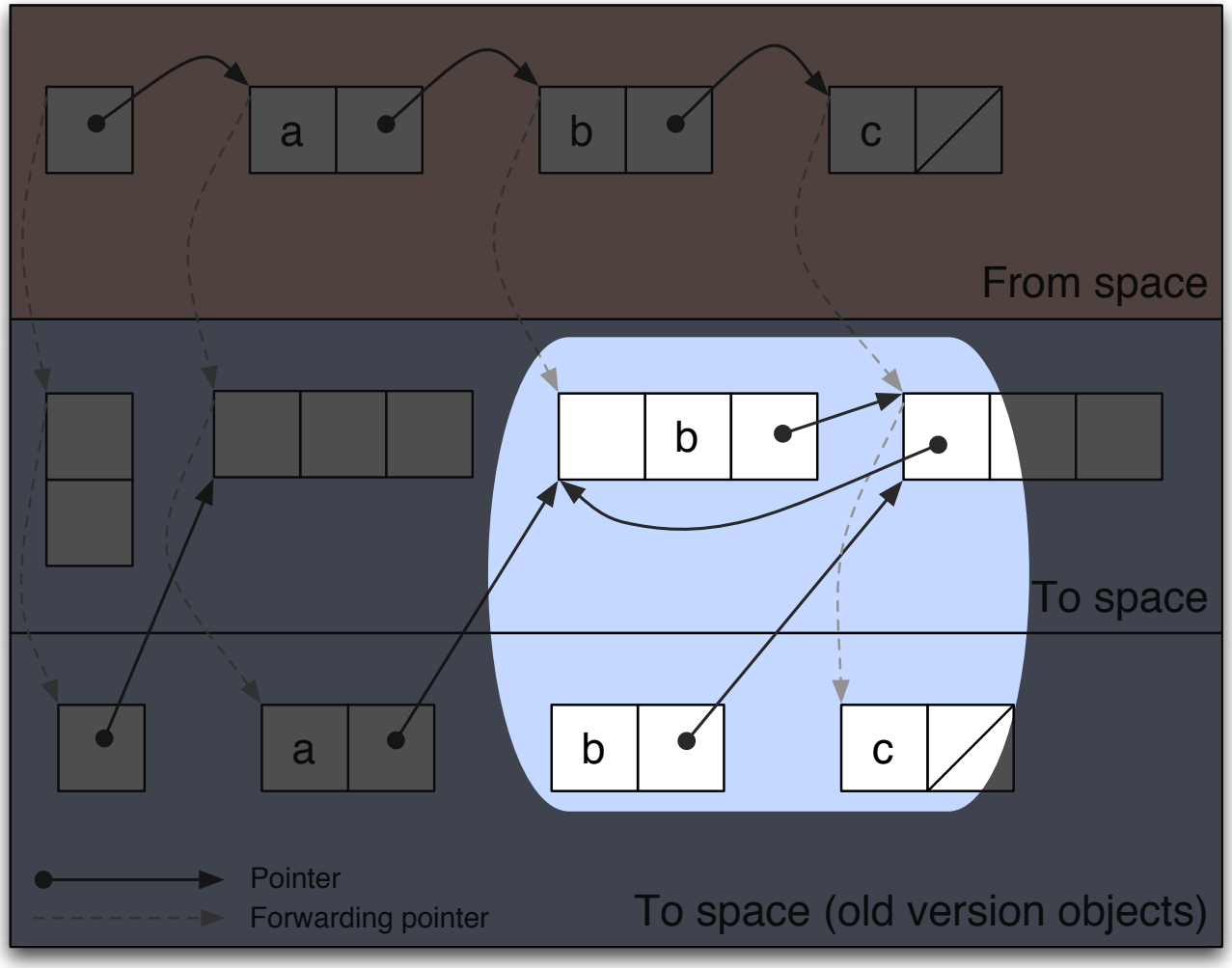

```
jvolveObject(Node to,
     old_Node from) {
  to.data = from.data;
  to.next = from.next;
  if (to.next != null)
    to.next.prev = to;
}
```

Jvolve GC



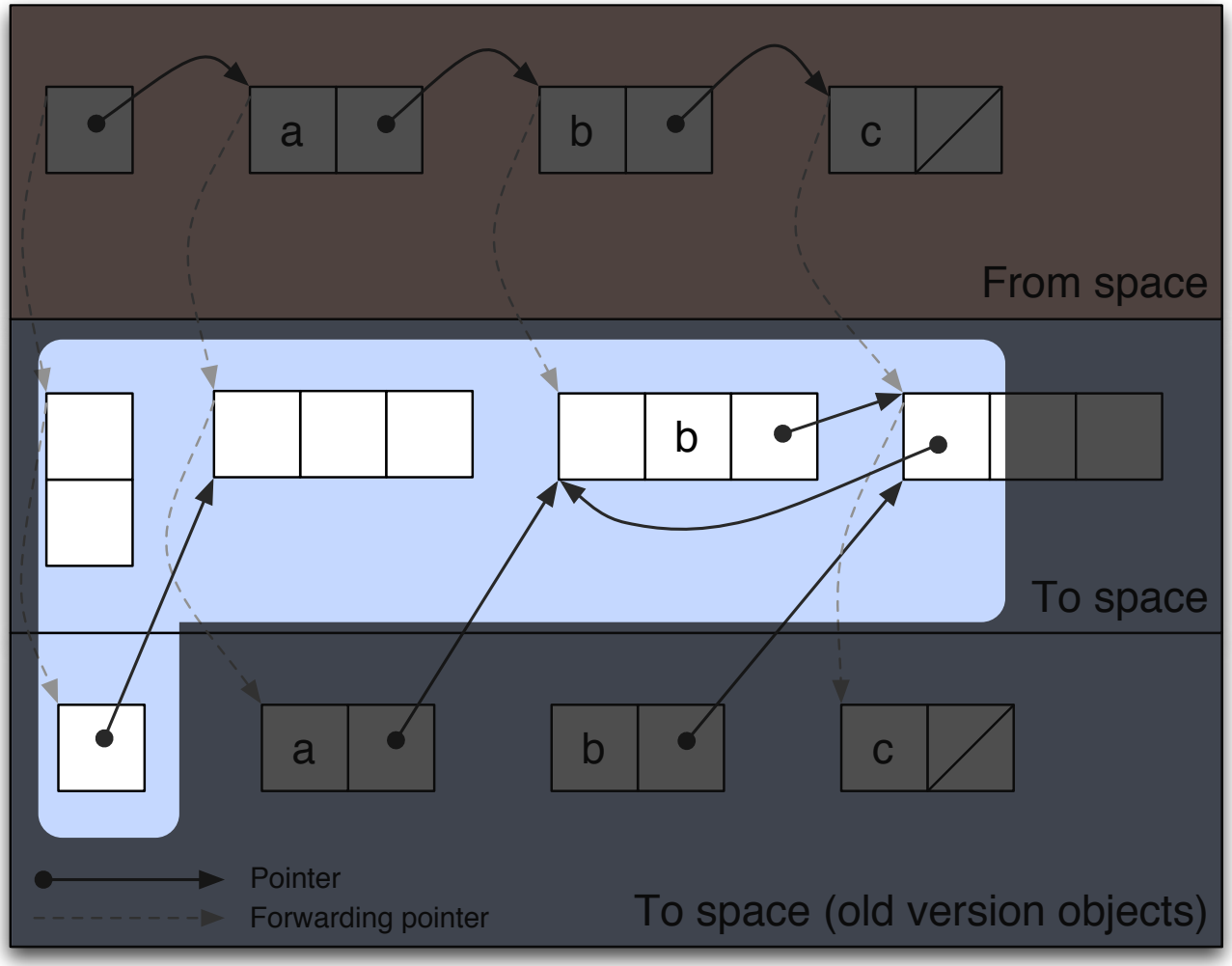

```
jvolveObject(Node to,
    old_Node from) {
  to.data = from.data;
  to.next = from.next;
  if (to.next != null)
    to.next.prev = to;
}
```

Jvolve GC

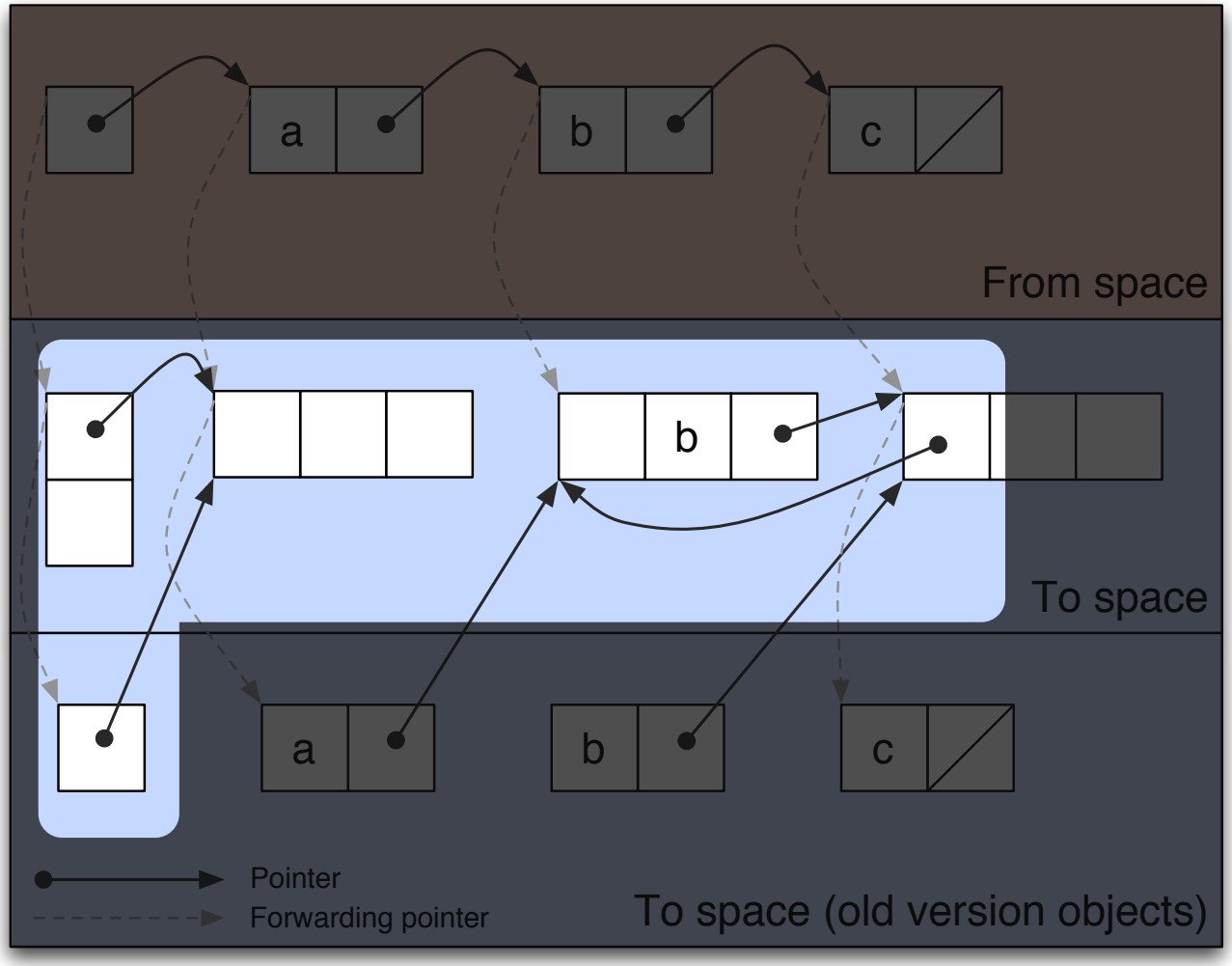



```
jvolveObject(Node to,
     old_Node from) {
  to.data = from.data;
  to.next = from.next;
  if (to.next != null)
    to.next.prev = to;
}
```

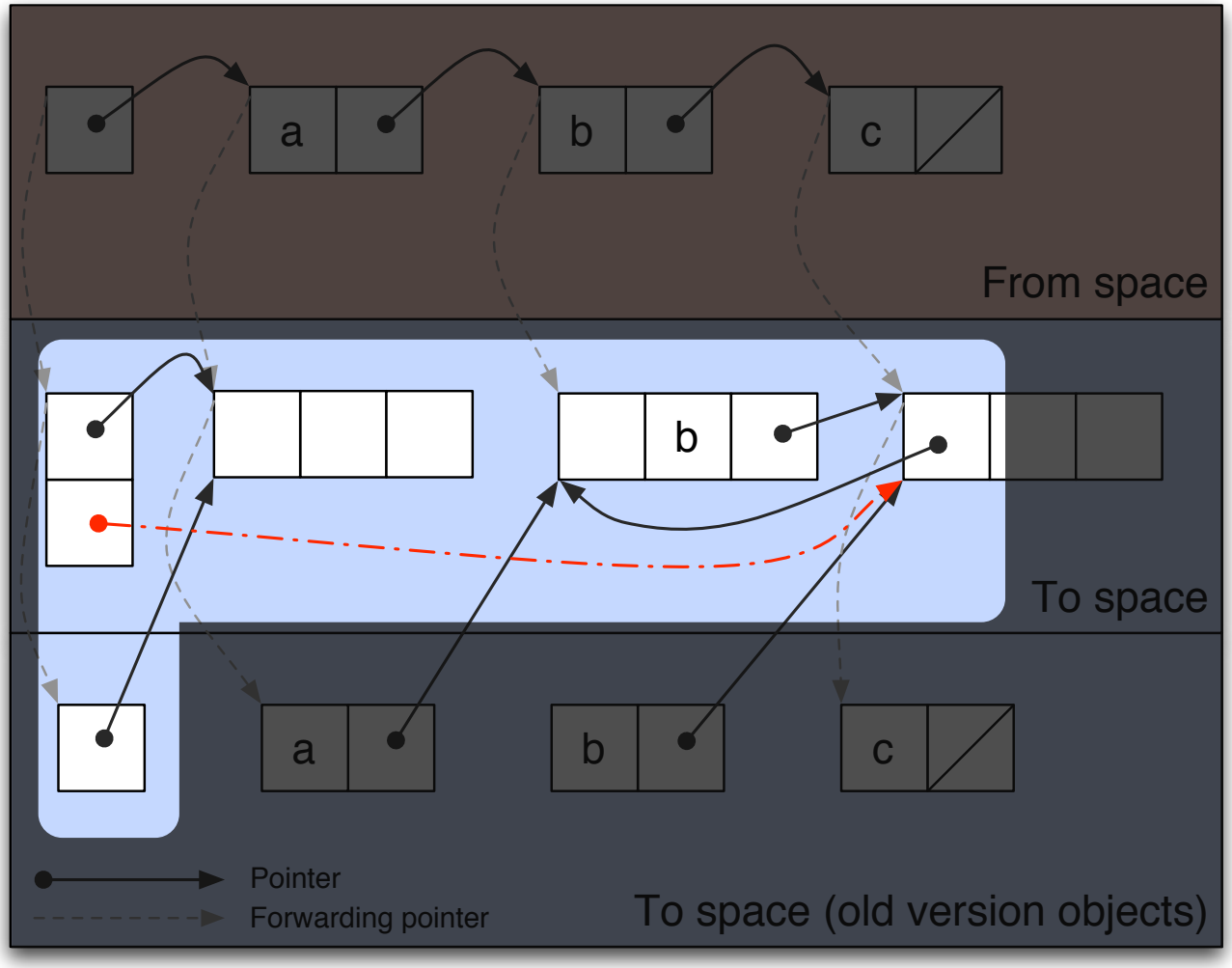
Jvolve GC


a
b
c
From space
b
To space
a
b
c
Pointer
Forwarding pointer
To space (old version objects)

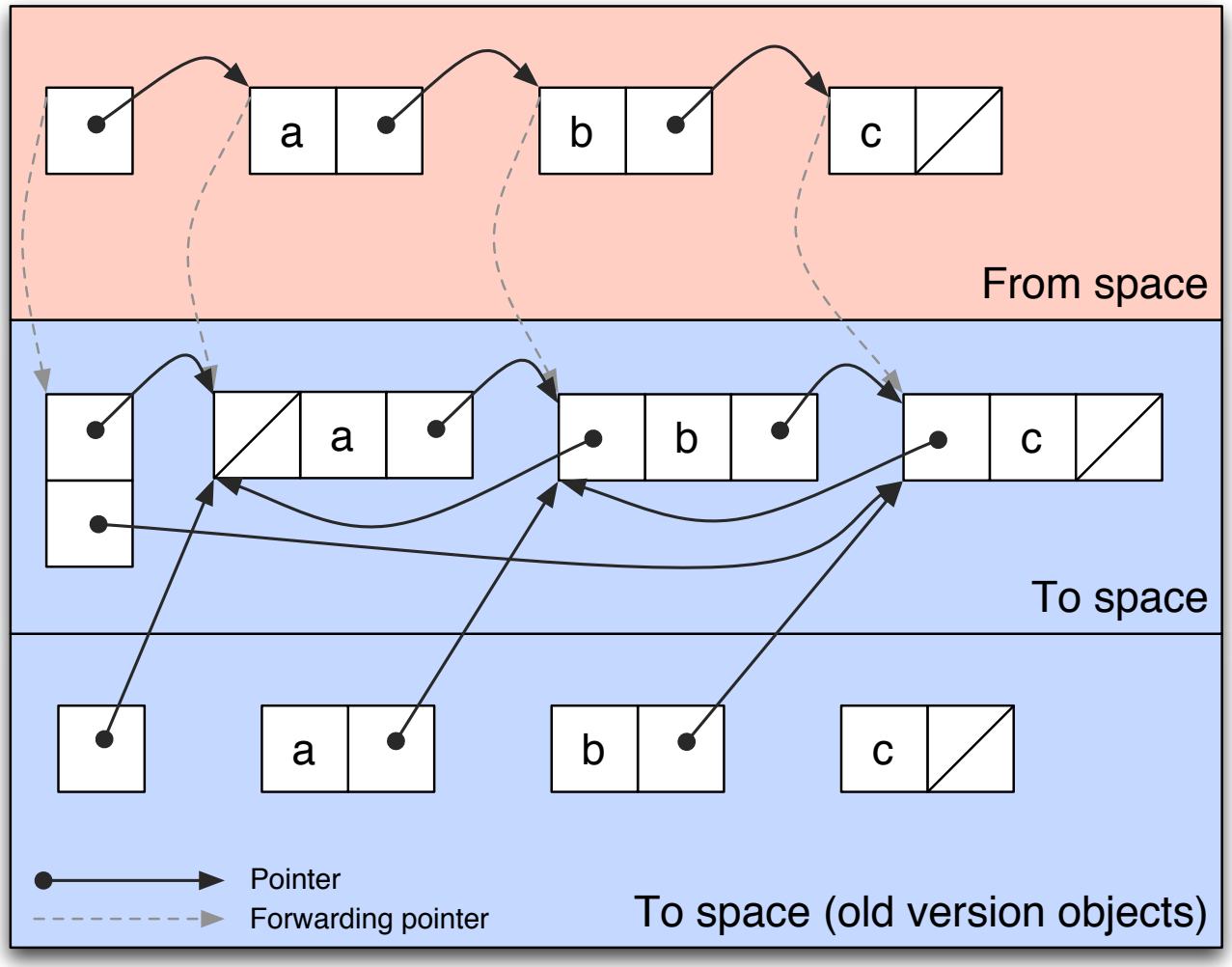
Jvolve GC


a
b
c
From space
b
To space
a
b
c
Pointer
Forwarding pointer
To space (old version objects)

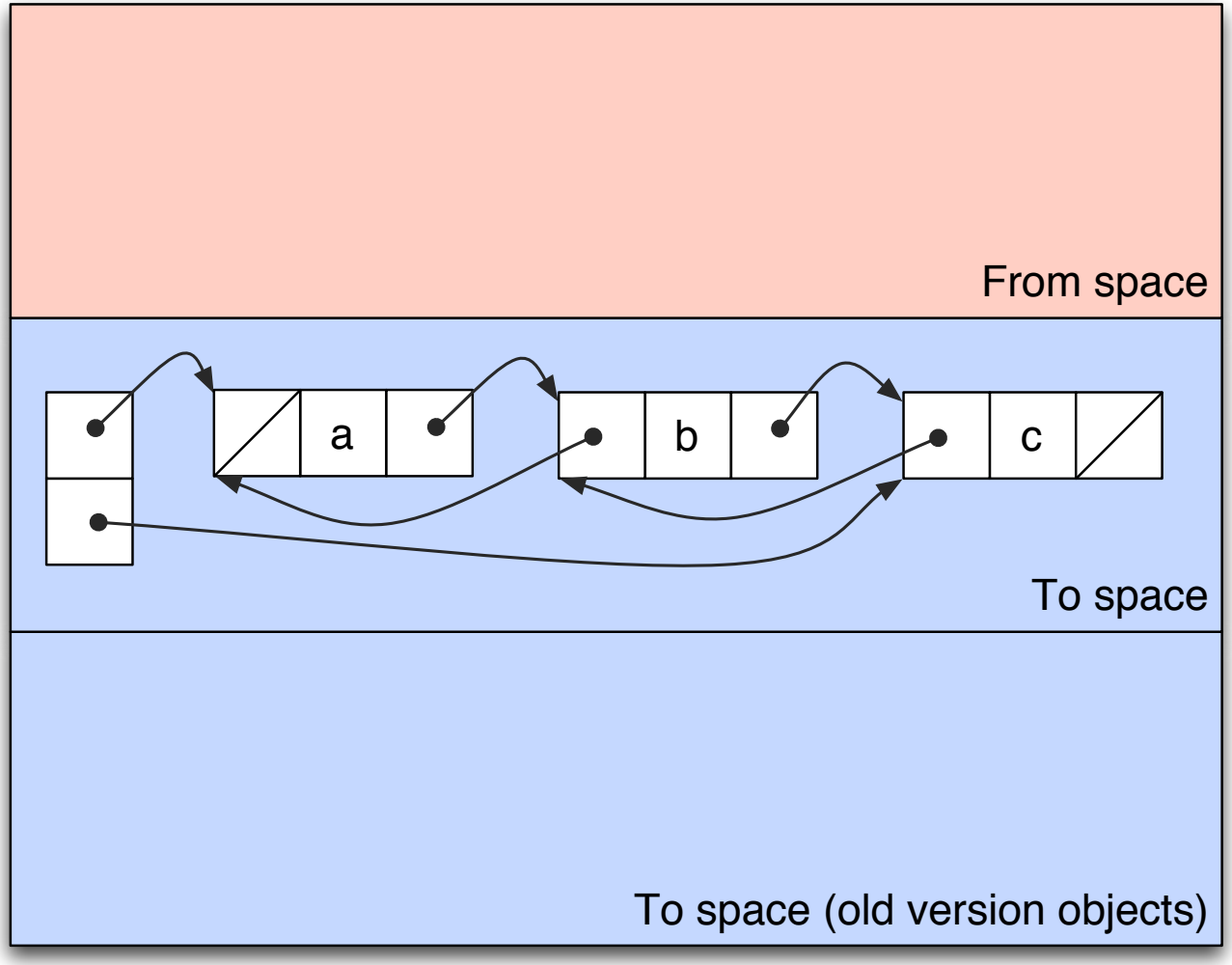
Jvolve GC


a
b
c
From space
b
To space
a
b
c
Pointer
Forwarding pointer
To space (old version objects)

Jvolve GC


a
b
c
From space
a
b
c
To space
a
b
c
Pointer
Forwarding pointer
To space (old version objects)

Jvolve GC



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

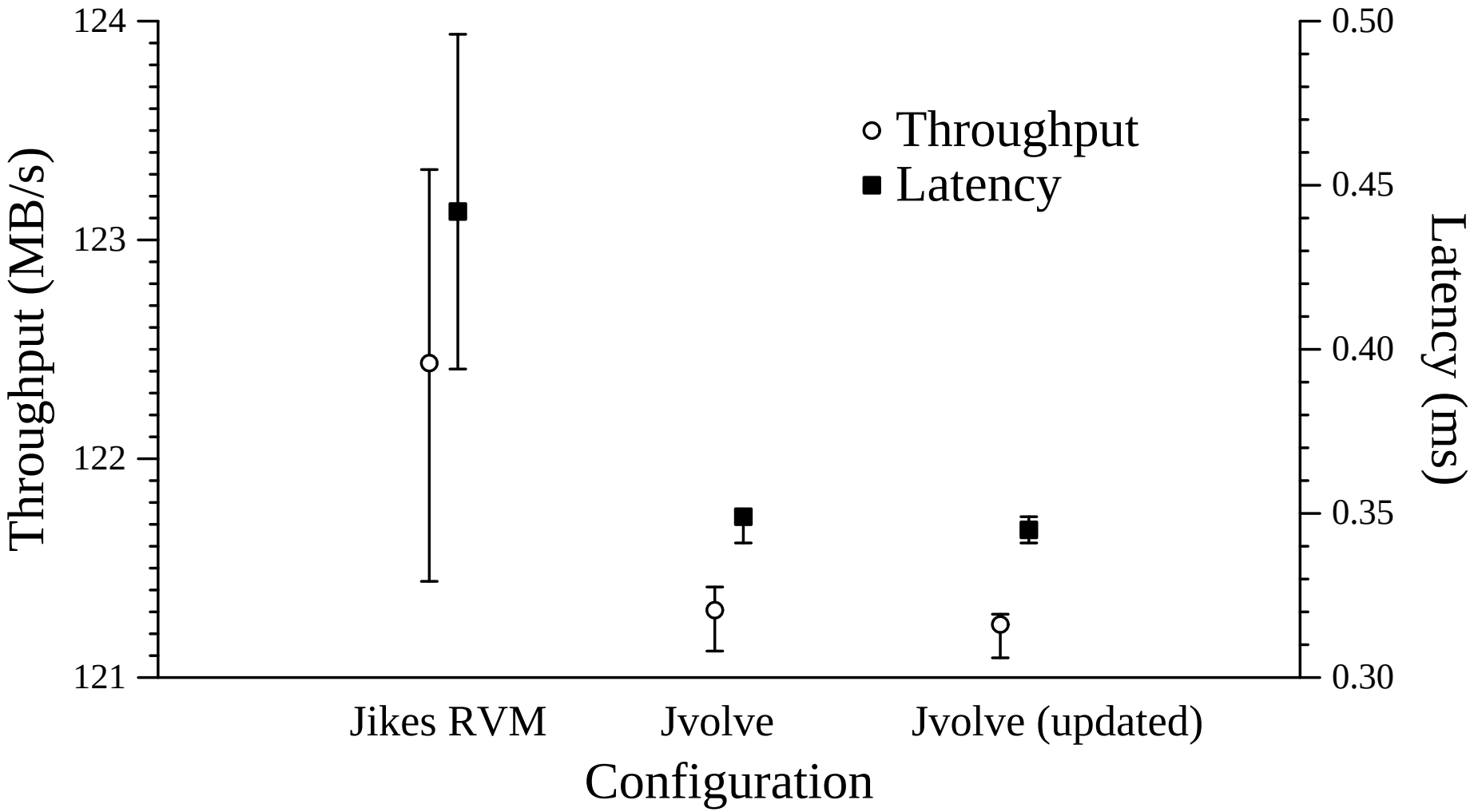
- 13 updates change class signature by adding new fields
- Several updates require On-stack replacement support
- Two versions update an infinite loop, postponing the update indefinitely

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

JVOLVE performance

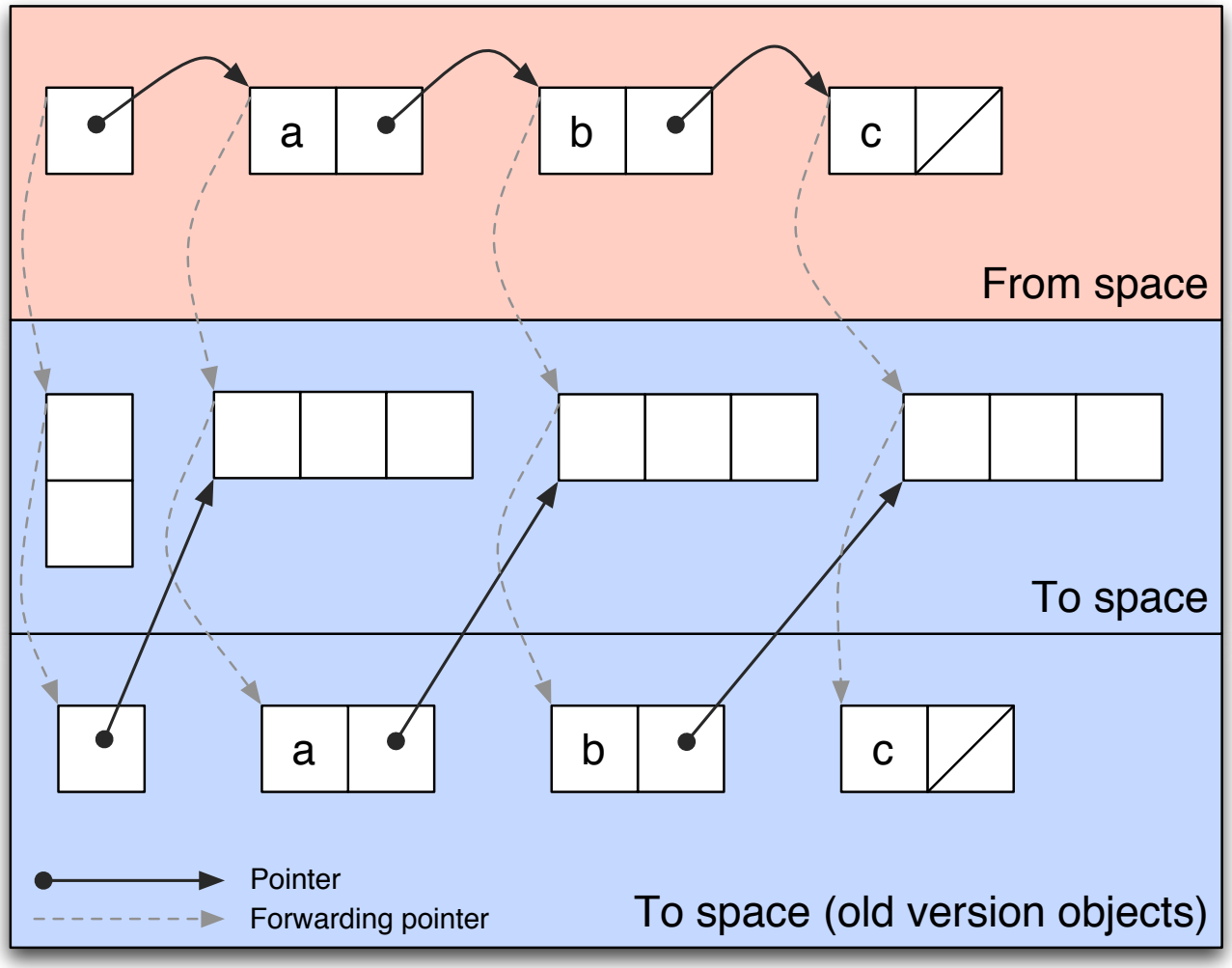
No overhead during steady-state execution



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

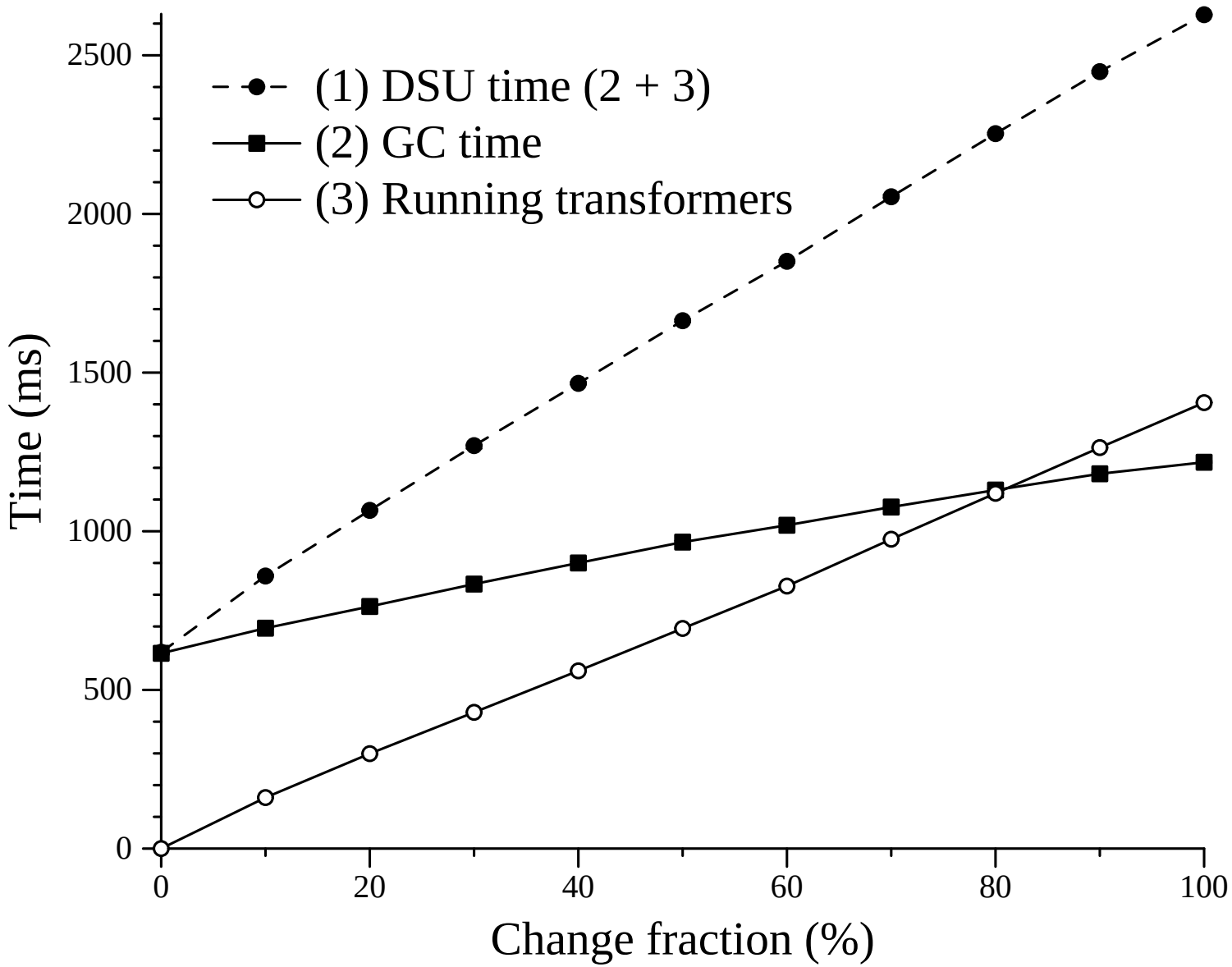
Update pause time



Update pause time

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

DSU pause times (microbenchmark)



Outline

- Introduction
- Jvolve
  - VM Implementation
  - Evaluation
- Future Work
- Conclusion

Looking ahead

What's in the path towards mainstream adoption?

Safety

- Currently, we guarantee type safety. Can we do more?
- Safe behavior in a general context is undecidable [Gupta et al., 1996]
- Divide program into transactions, ensure they are version consistent [Neamtiu et al., 2008]
- Exhaustively testing updates

Flexibility

- Supporting every possible update is not a goal
- Support most of the updates that occur in practice
- Large and complex applications and updates
- Understand update semantics from refactorings

Efficiency

- Update time roughly proportional to GC time
- Semi-space GC is not practical
- We have two requirements
  - Copying objects
  - Full-heap collection
- Other collectors? Concurrent collectors?

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.

Source code and other information:
`http://www.cs.utexas.edu/~suriya/jvolve`

References I

Gupta, D., Jalote, P., and Barua, G. (1996).
A formal framework for on-line software version change.
IEEE Trans. Softw. Eng., 22(2):120–131.

Neamtiu, I., Hicks, M., Foster, J. S., and Pratikakis, P. (2008).
Contextual effects for version-consistent dynamic software updating and safe concurrent programming.
In *Proc. POPL*.