Dynamic Software Updates: A VM-centric Approach

Suriya Subramanian[1] Michael Hicks[2]
Kathryn S. McKinley[1]

[1]Department of Computer Sciences
The University of Texas at Austin

[2]Department of Computer Science
University of Maryland

May 1, 2009
DaCapo Research Meeting

The only thing that is constant is change.

Heraclitus of Ephesus

Motivation

- Software applications change all the time
- Deployed systems must be updated with bug fixes, new features

Motivation

- Software applications change all the time
- Deployed systems must be updated with bug fixes, new features

$$\text{Updating} \Rightarrow \text{Downtime}$$

Motivation

- Software applications change all the time
- Deployed systems must be updated with bug fixes, new features

$$\text{Updating} \Rightarrow \text{Downtime}$$

- Safety concerns
- Revenue loss
- Inconvenience

Motivation

- 75% of downtime in high-availablity applications is for planned maintenance
- Personal operating system
- Enterprise applications
- Telecommunication, transportation systems

Motivation

- 75% of downtime in high-availablity applications is for planned maintenance
- Personal operating system
- Enterprise applications
- Telecommunication, transportation systems

Conventional solutions to avoid downtime

- Move state out of the process, for instance databases
- Use multiple processes, and do a rolling update

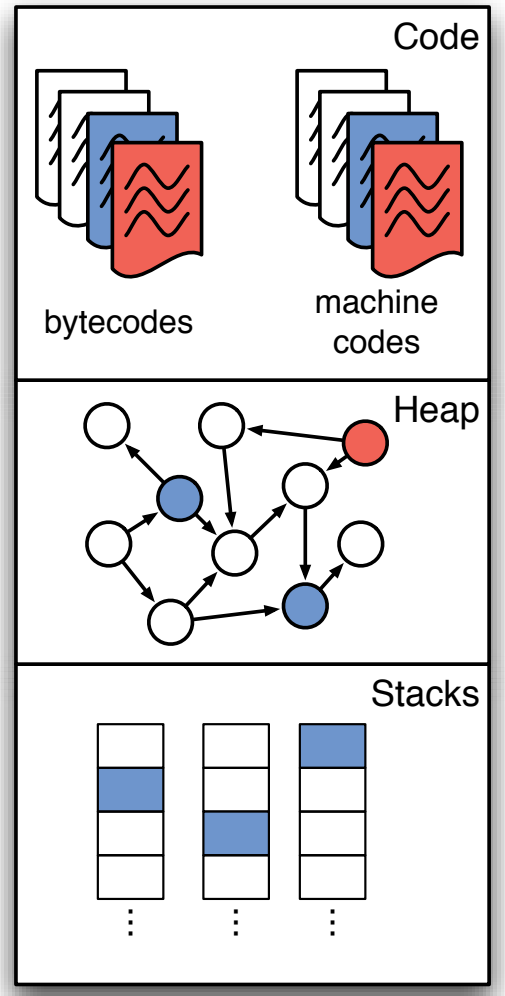
- Not always possible
- Restricted to specific application domains

Conventional solutions to avoid downtime

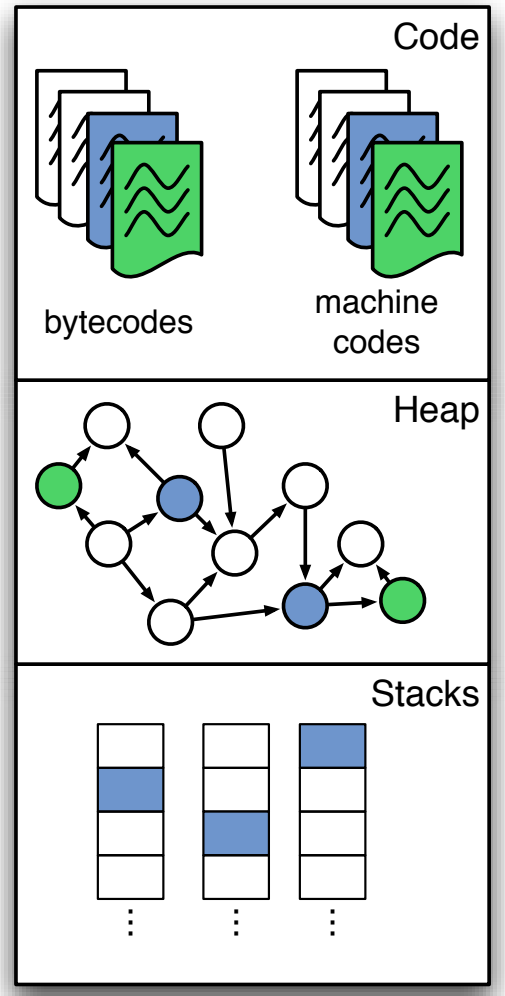
- Move state out of the process, for instance databases
- Use multiple processes, and do a rolling update

- Not always possible
- Restricted to specific application domains

Dynamic software updating

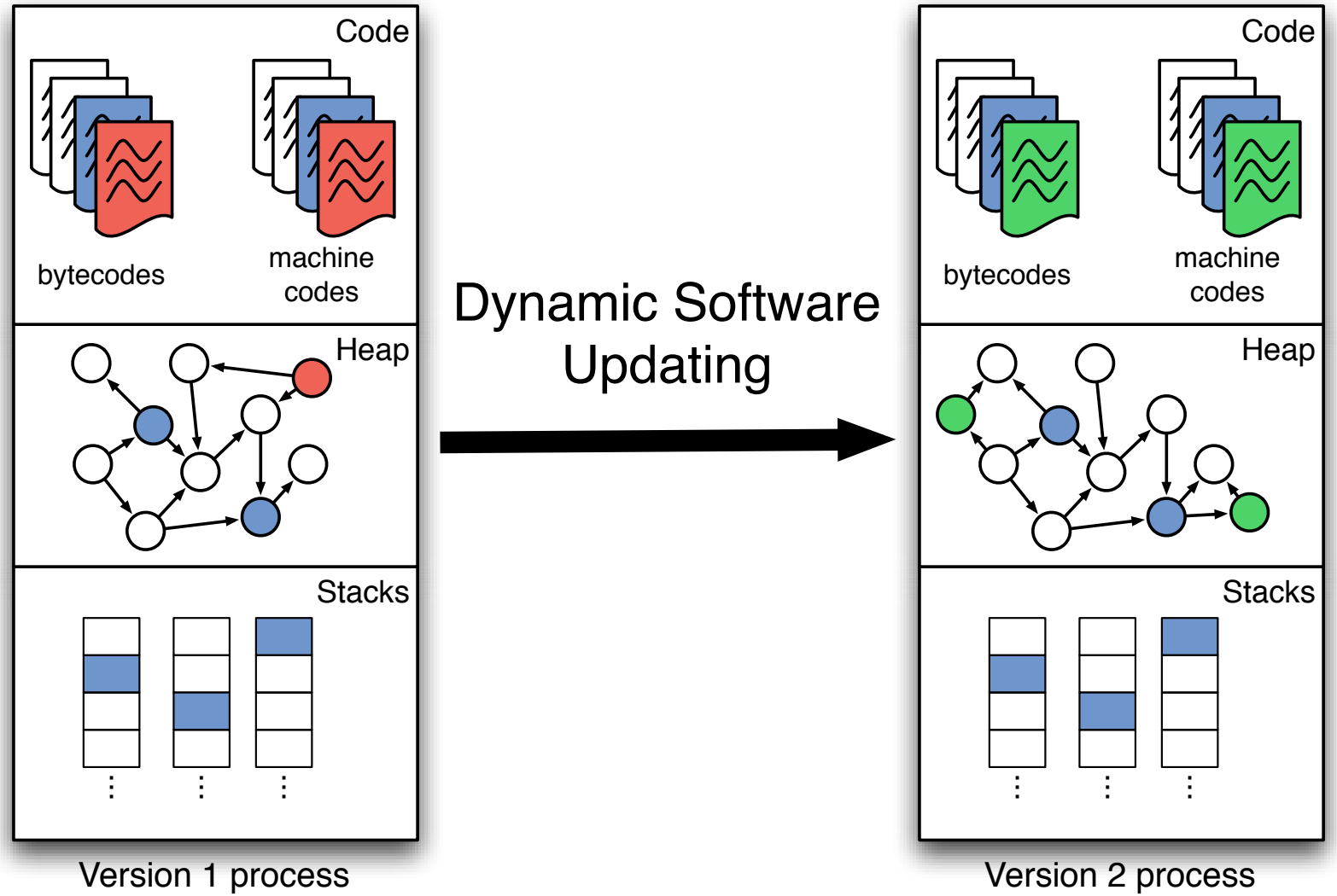


Version 1 process



Version 2 process

Dynamic software updating



DSU requirements

A Dynamic Software Updating solution should *ideally* be

Safe Updating is as correct as starting from scratch

Flexible Be able to support changes encountered in practice

Efficient No performance impact on the original application

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

Related work

- Custom class loader solutions:
  Eisenbach and Barr, Milazzo et al.
- Source-to-source translation: Orso et al.
- VM-based solutions: JDrums, Dynamic Virtual Machine (DVM)
- In a persistent object store: Boyapati et al.

- Limited, not flexible
- Restricted data-transformation model (like requiring *encapsulation* based on *ownership types*)
- Overhead during normal execution

Related work

- Custom class loader solutions:
  Eisenbach and Barr, Milazzo et al.
- Source-to-source translation: Orso et al.
- VM-based solutions: JDrums, Dynamic Virtual Machine (DVM)
- In a persistent object store: Boyapati et al.

- Limited, not flexible
- Restricted data-transformation model (like requiring *encapsulation* based on *ownership types*)
- Overhead during normal execution

Our solution

- JVOLVE - a Java Virtual Machine with DSU support
- Key insight: Naturally extend existing VM services
  - Classloading
  - Bytecode verification[1]
  - Thread synchronization
  - JIT Compilation
  - On-stack replacement
  - Garbage collection
- No DSU-related overhead during normal execution
- Support updates to real world applications

[1] Jikes RVM does not have a bytecode verifier

Claim

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

Claim

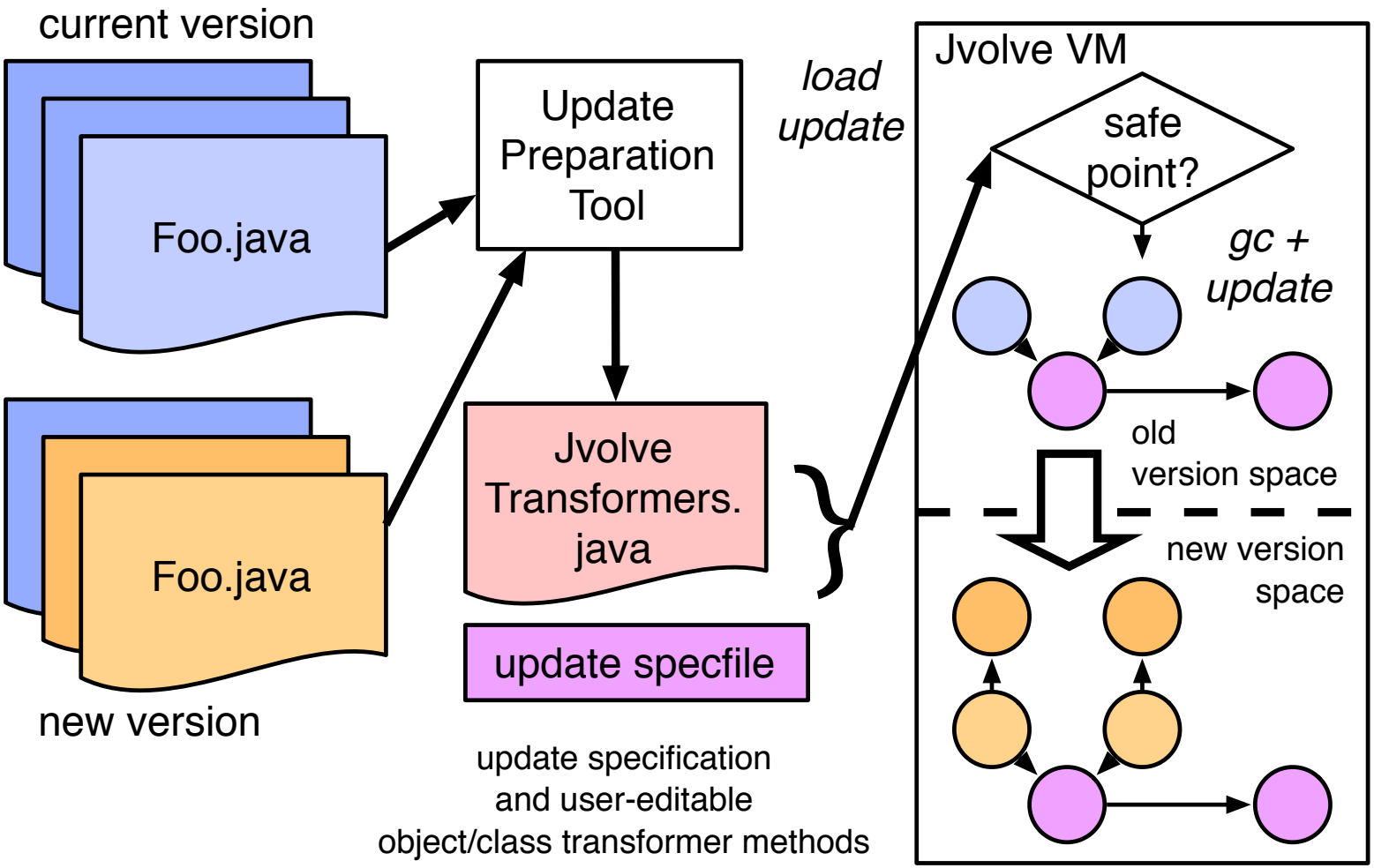
Dynamic software updating in managed languages can be achieved in a safe, flexible and efficient manner by naturally extending existing VM services.

Corollary

DSU support should be a standard feature of future VMs.

Outline

Developer's view of Jvolve



VM's view of DSU

- Update happens in one fell swoop
- Simple to reason about
- Code
  - Old code before the update
  - New code after the update
- Data
  - *Representation consistency* (all values of a type correspond to the latest version)
  - Support a transformation function to convert objects to conform to their new definition

Division of Labor

- Developer
  - Write the old and new versions
  - Write class/object transformation functions for classes that changed (optional)
  - Testing (both the application and the update)
- JVOLVE system
  - Update Preparation Tool (UPT) compares versions and presents the update to the JVOLVE VM.
  - JVOLVE VM handles the update

Supported updates

- Changes within the body of a method
- Class signature updates
  - Add, remove, change the type signature of fields and methods
- Changes can occur at any level of the class hierarchy

Outline

Example of an update (JavaEmailServer)

```
public class User {
  private final String username, domain, password;
  private String[] 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;
    }
  }
```

Object Transformers

- "Transform" objects to correspond to the new version
- Functions generated by the Update Preparation Tool (UPT)
- Accept old object and new objects as parameters
- Default transformer copies old fields and initializes new ones to `null`
- User can optionally modify these functions

Object Transformers

```
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`

Object Transformers

```
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

Outline

Update process

- Offline Update Preparation Tool (UPT)
- JVOLVE VM
  - Reach a safe point in the VM
  - Install new classes
  - Transform objects to new definition
  - Resume execution

Update Preparation Tool

- Uses jclasslib[2], a bytecode library
- Compares bytecodes of the two versions
- Generates old version stubs and default object transformers

[2] `http://jclasslib.sourceforge.net`

Compiling transformation functions

- All transformers specified in a separate source file
- Class transformers

```
jvolveClass(ClassName unused)
```

- Object transformers

```
jvolveObject(old_ClassName from, ClassName to)
```

- Compiled specially by a JastAddJ extension to the Java language
- Ignores access protection and allows assigning to `final` fields

Safe point for the update

- Update must be atomic
- Updates happen at "safe points"
  (VM yield points with restriction on what methods can be on stack)
- Extend the thread scheduler to suspend all application threads
- Examine all stacks, ensure no restricted methods on stack and perform the update

Restricted methods

(1) Methods changed by the update

(2) 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

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

Handling restricted methods
- *On-stack replace* baseline-compiled category (2) methods
- Do not allow (1) and (3) to be active on stack, install a return barrier for such methods

Restricted methods

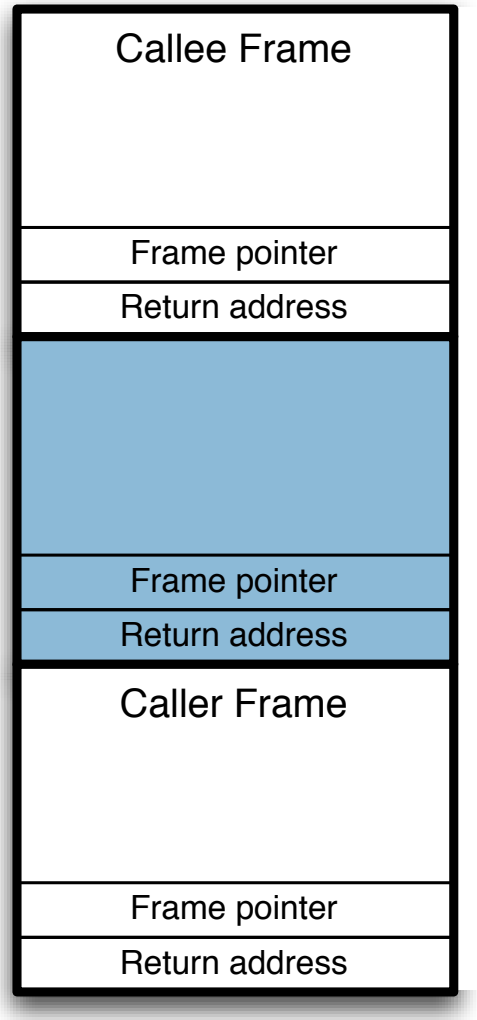
(1) Methods changed by the update

(2) 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

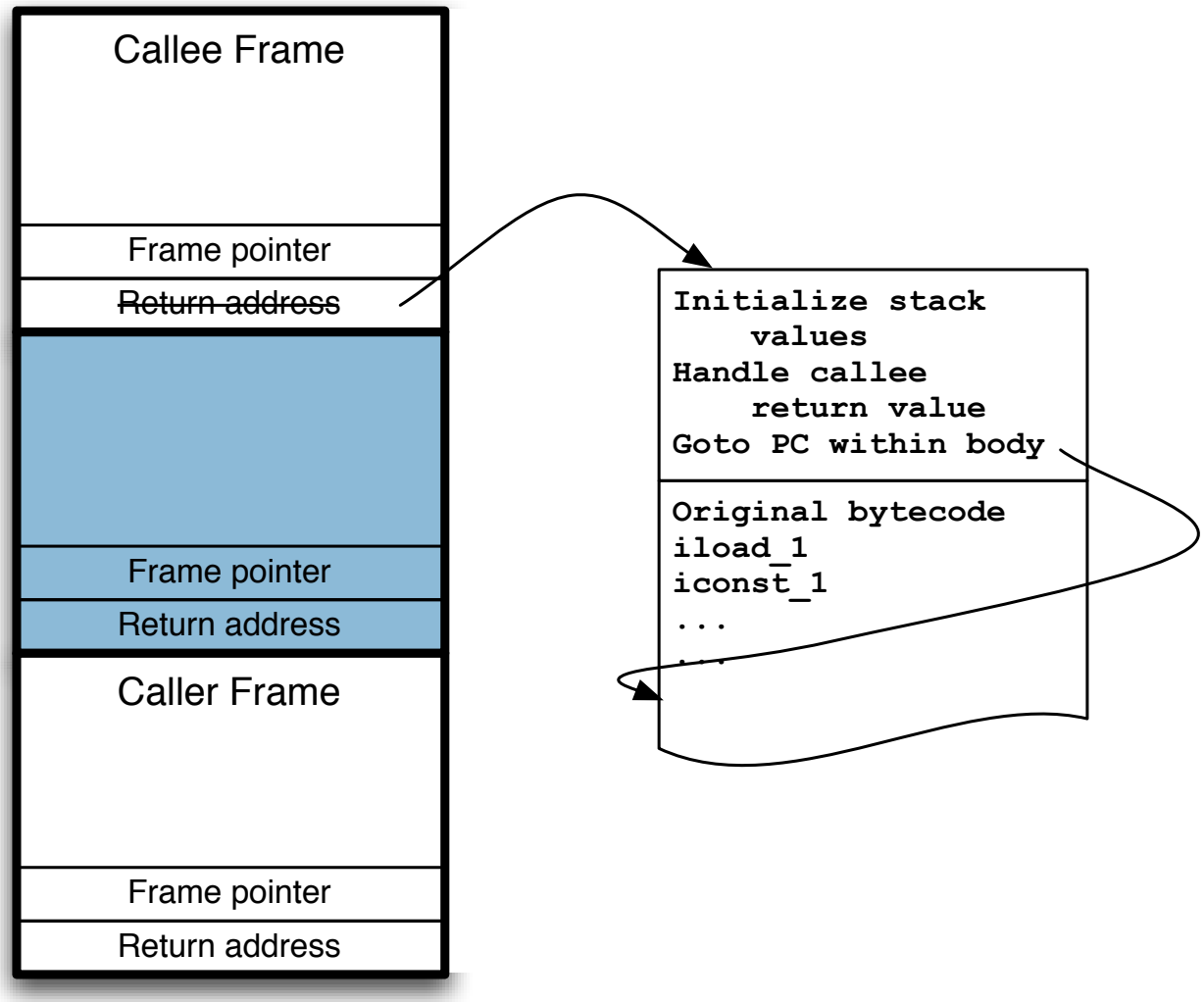
(3) Methods identified by the user as unsafe based on semantic information about the application

Handling restricted methods
- *On-stack replace* baseline-compiled category (2) methods
- Do not allow (1) and (3) to be active on stack, install a return barrier for such methods

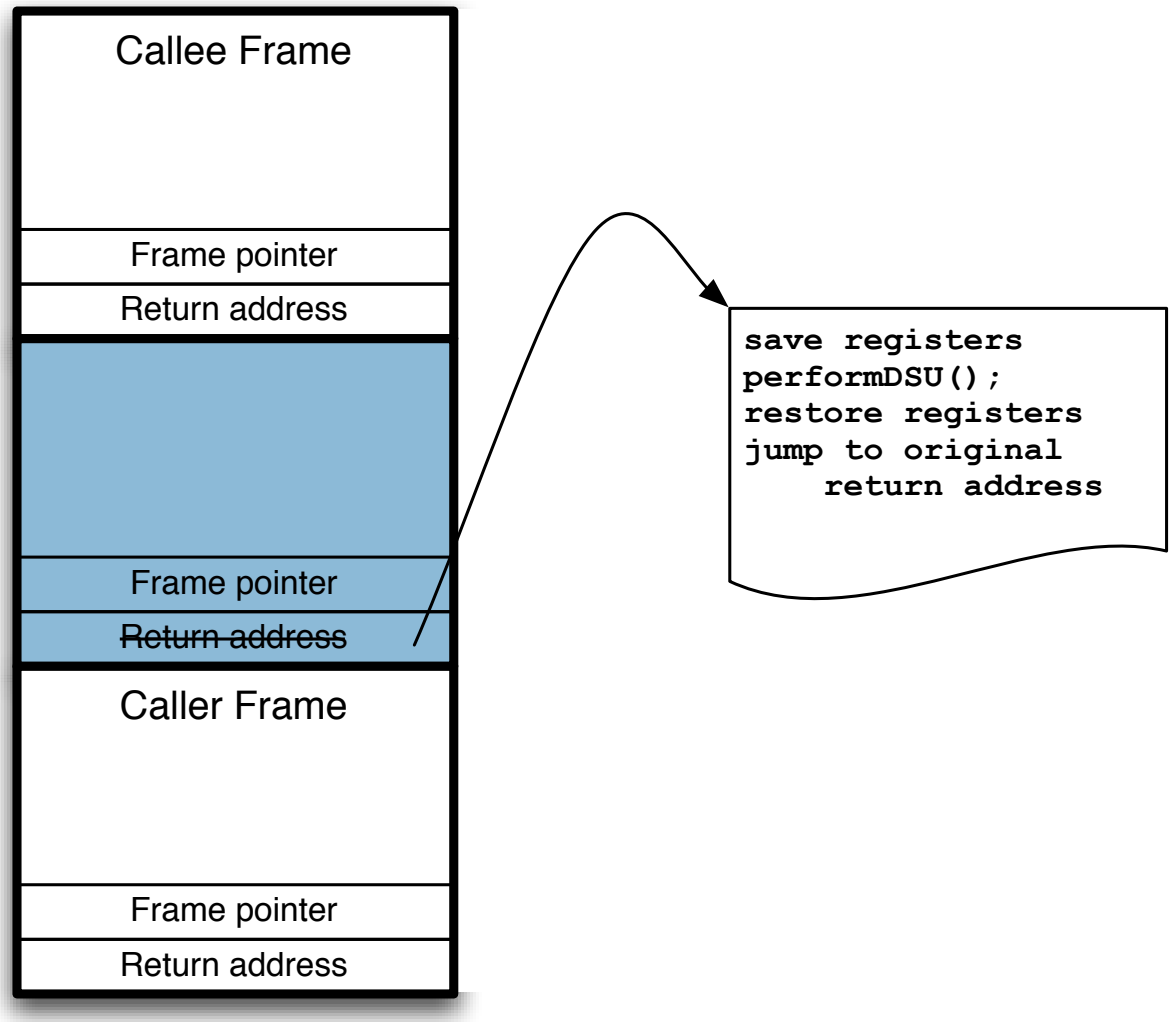
Handling restricted methods



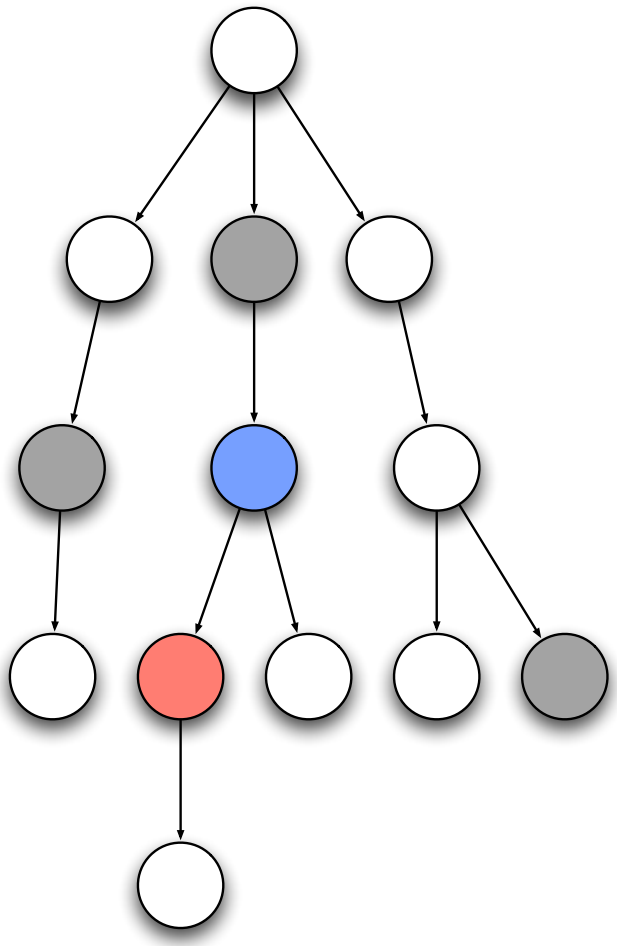
Performing On-stack replacement



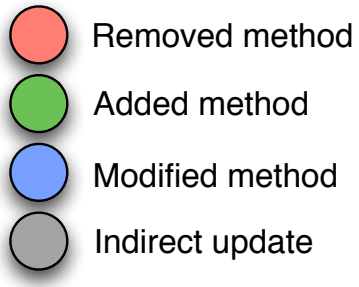
Installing return barrier for DSU

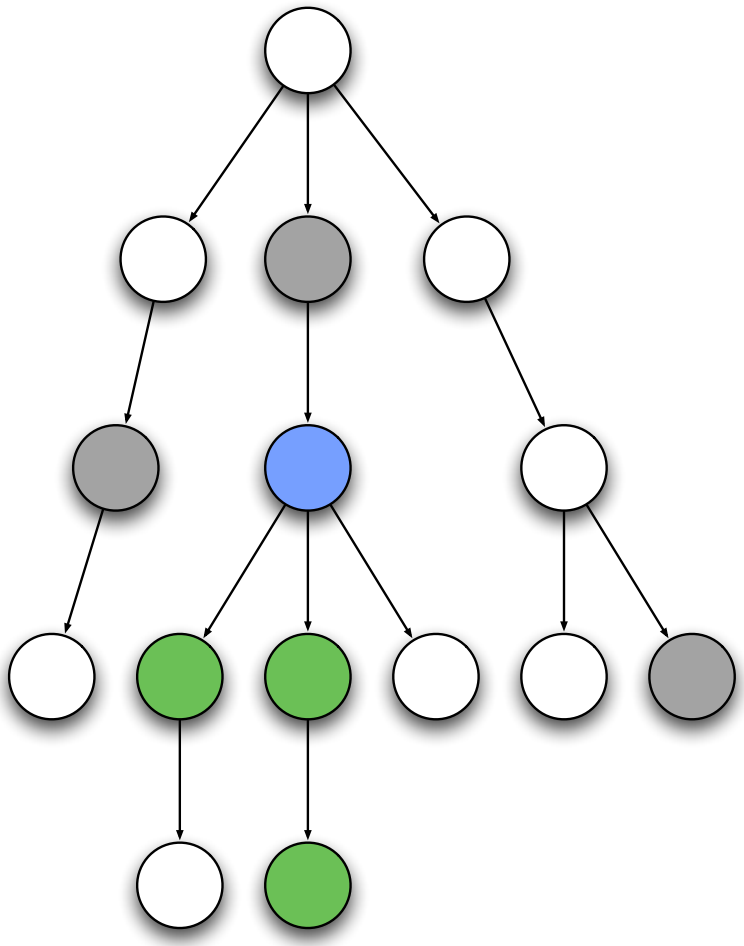


Reaching a safe point



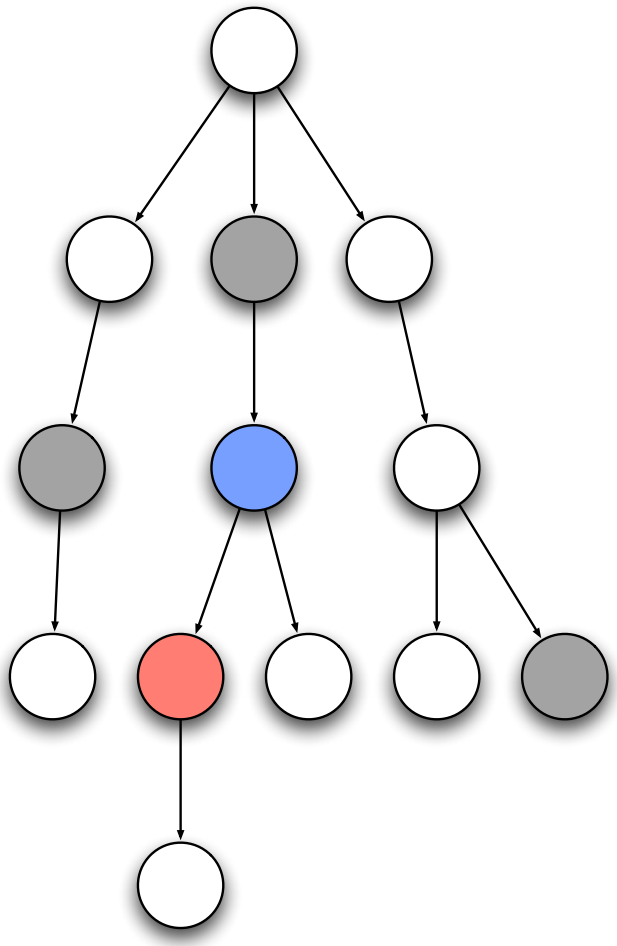
Version 1 Call graph



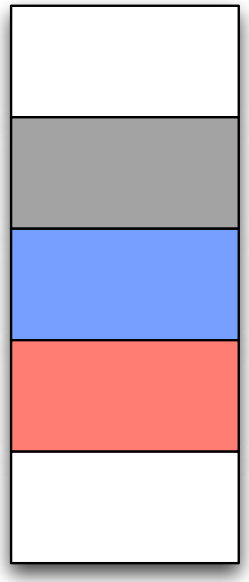


Version 2 Call graph

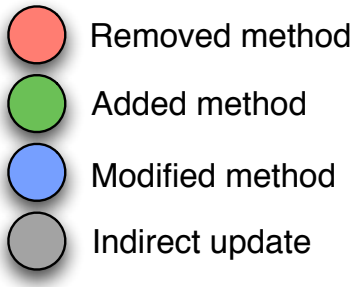
Reaching a safe point

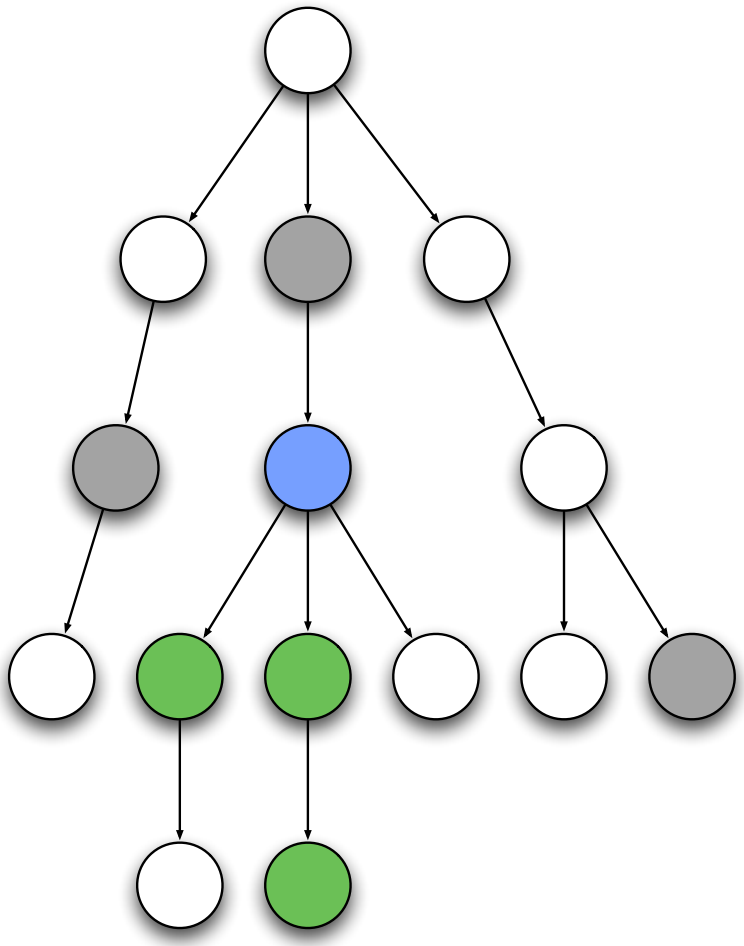


Version 1 Call graph



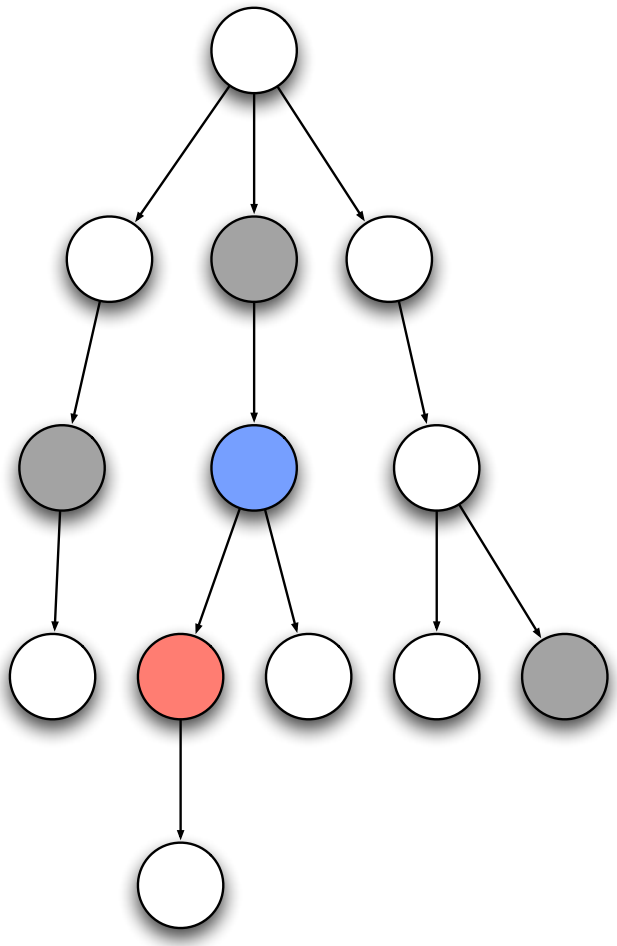
Activation Stack



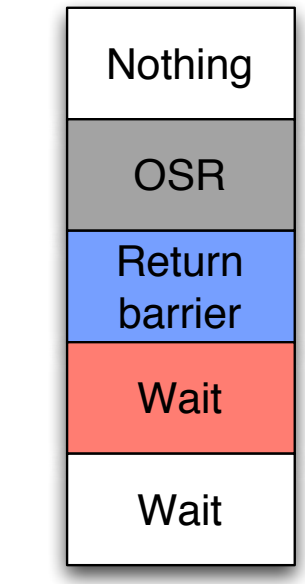


Version 2 Call graph

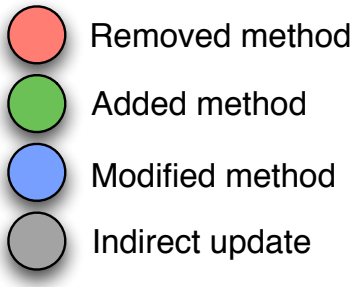
Reaching a safe point

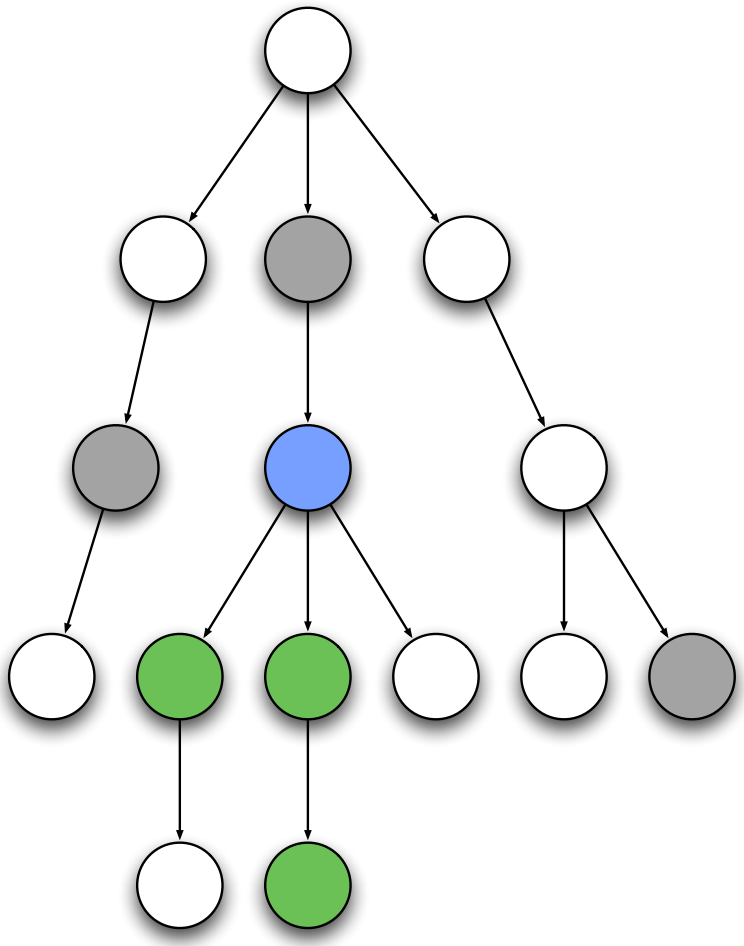


Version 1 Call graph



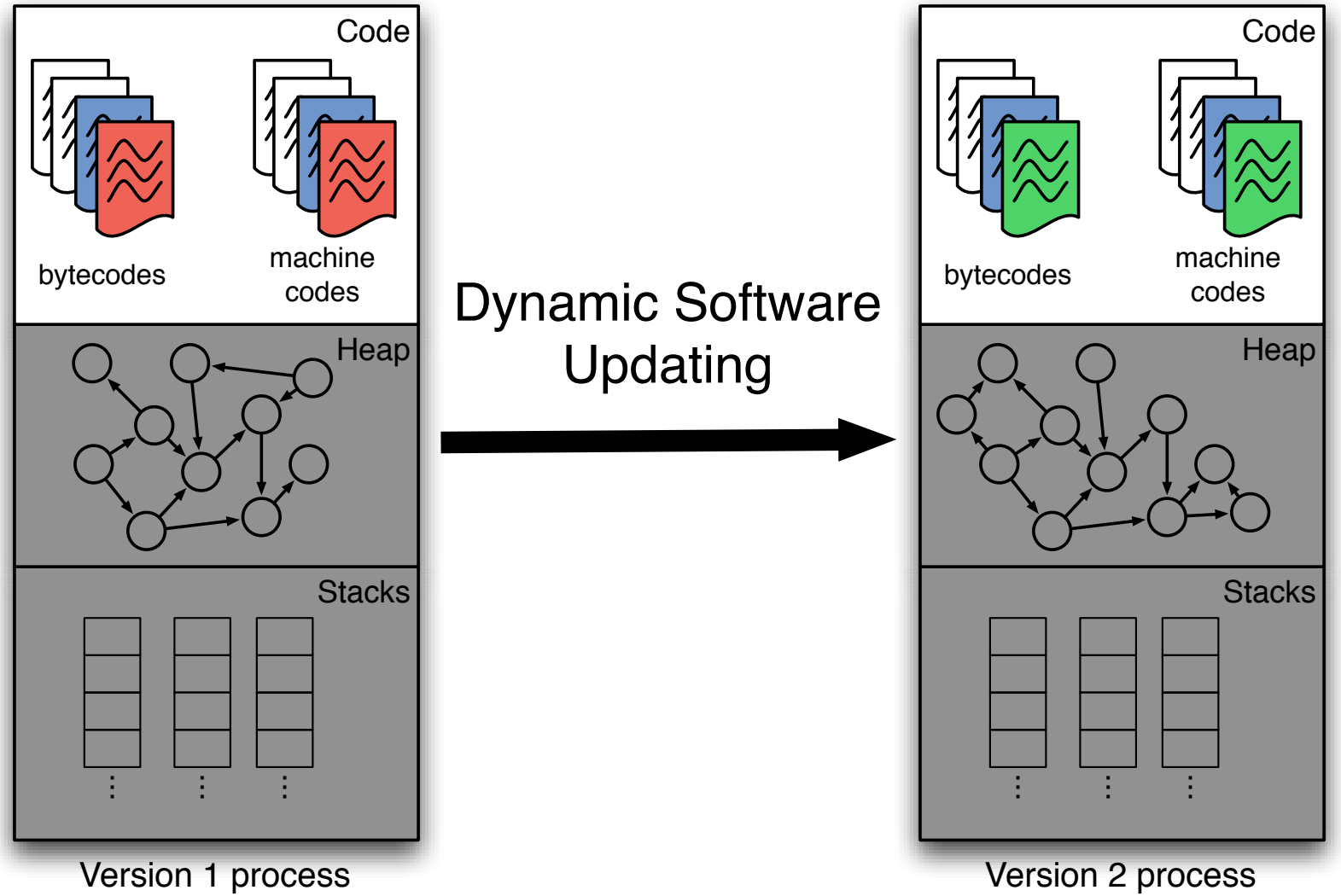
Activation Stack





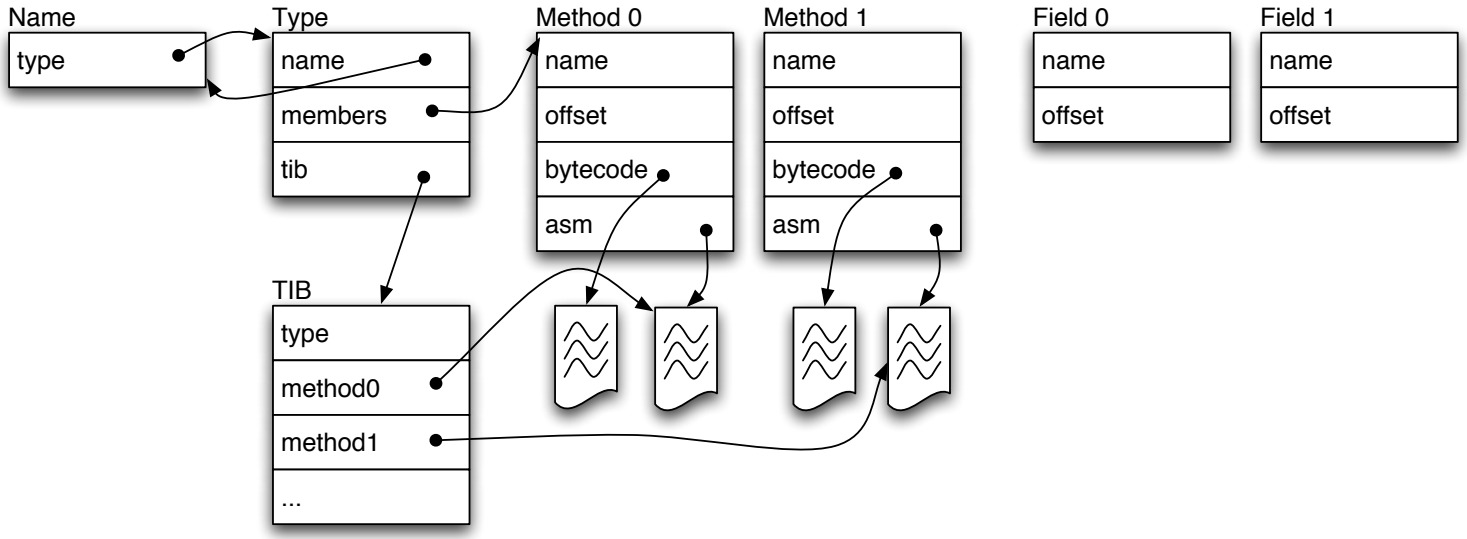
Version 2 Call graph

Updating code

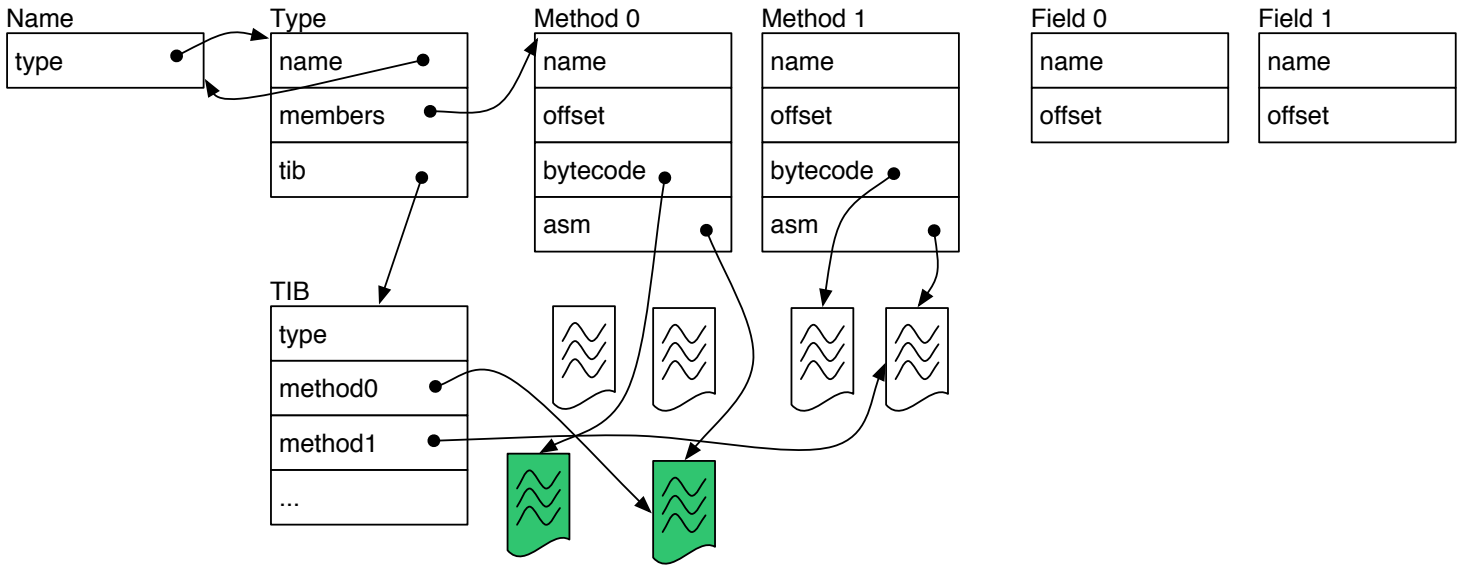


Installing new classes

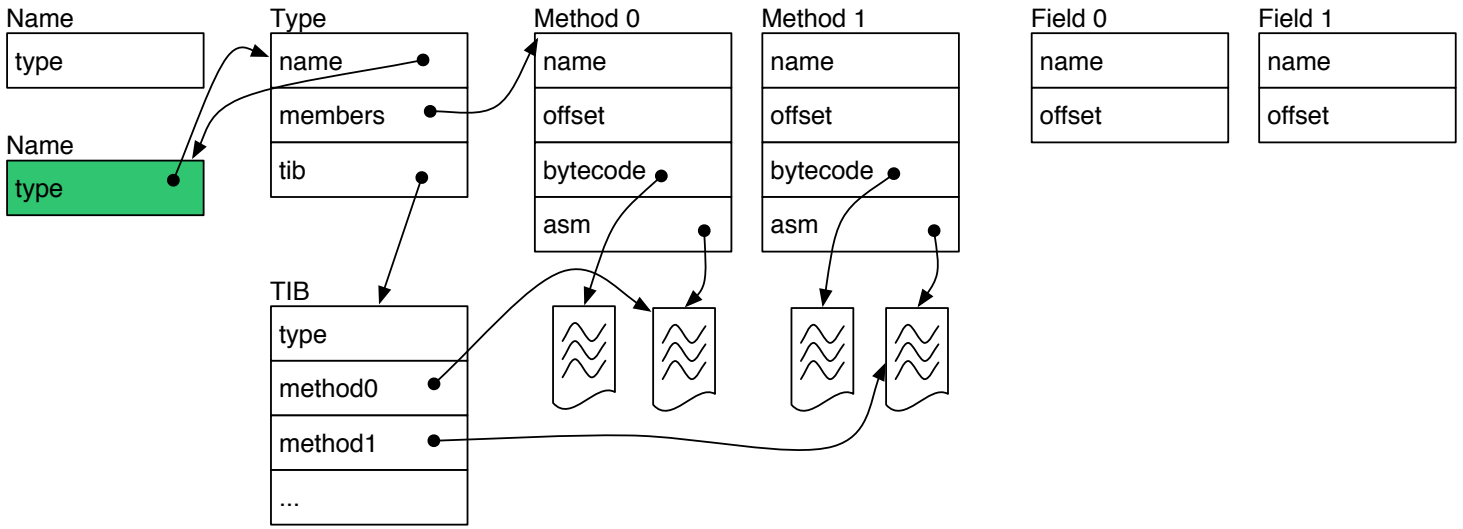
The VM maintains Class, Method and Field data structures



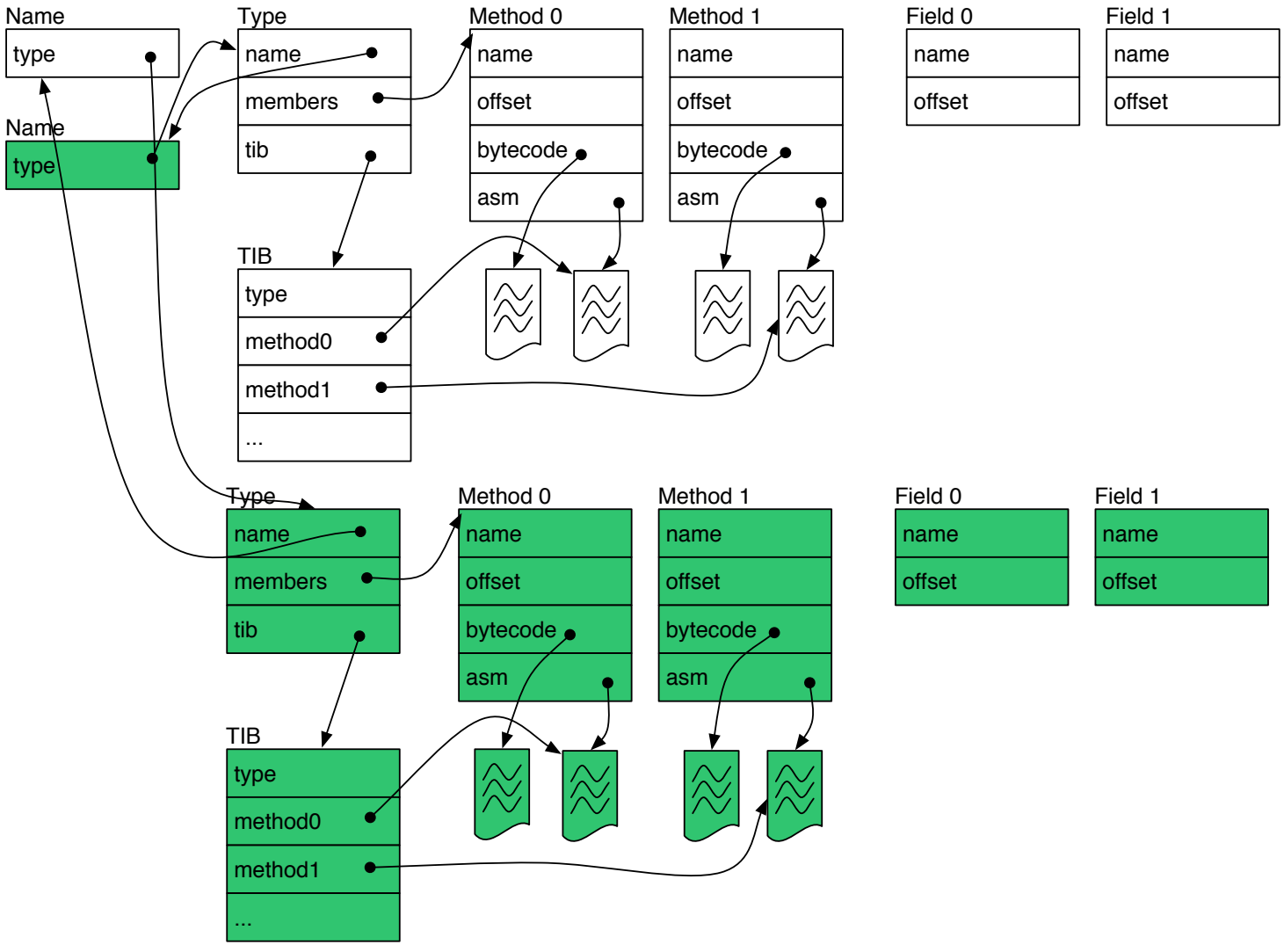
Updating a method



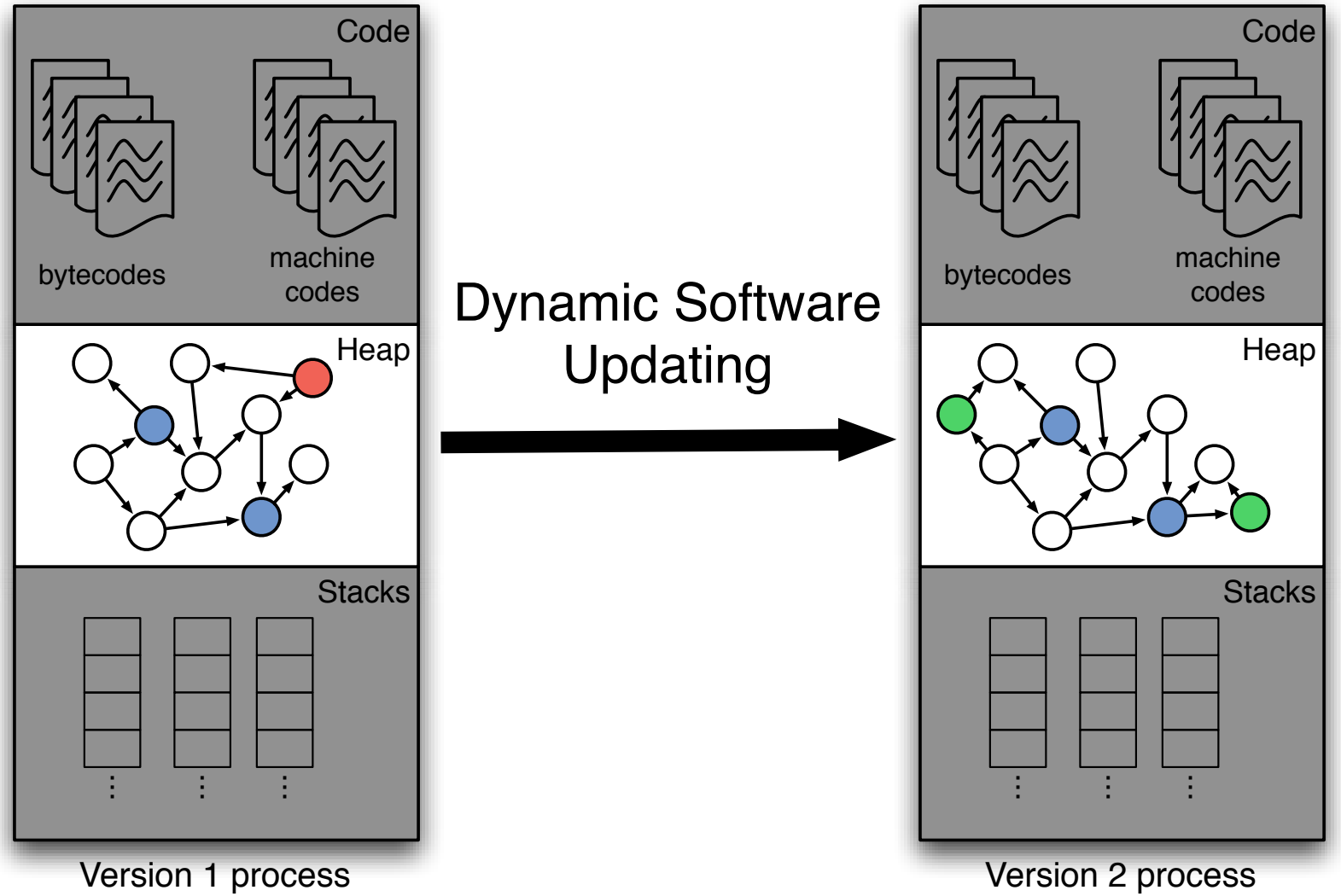
Updating a class



Updating a class



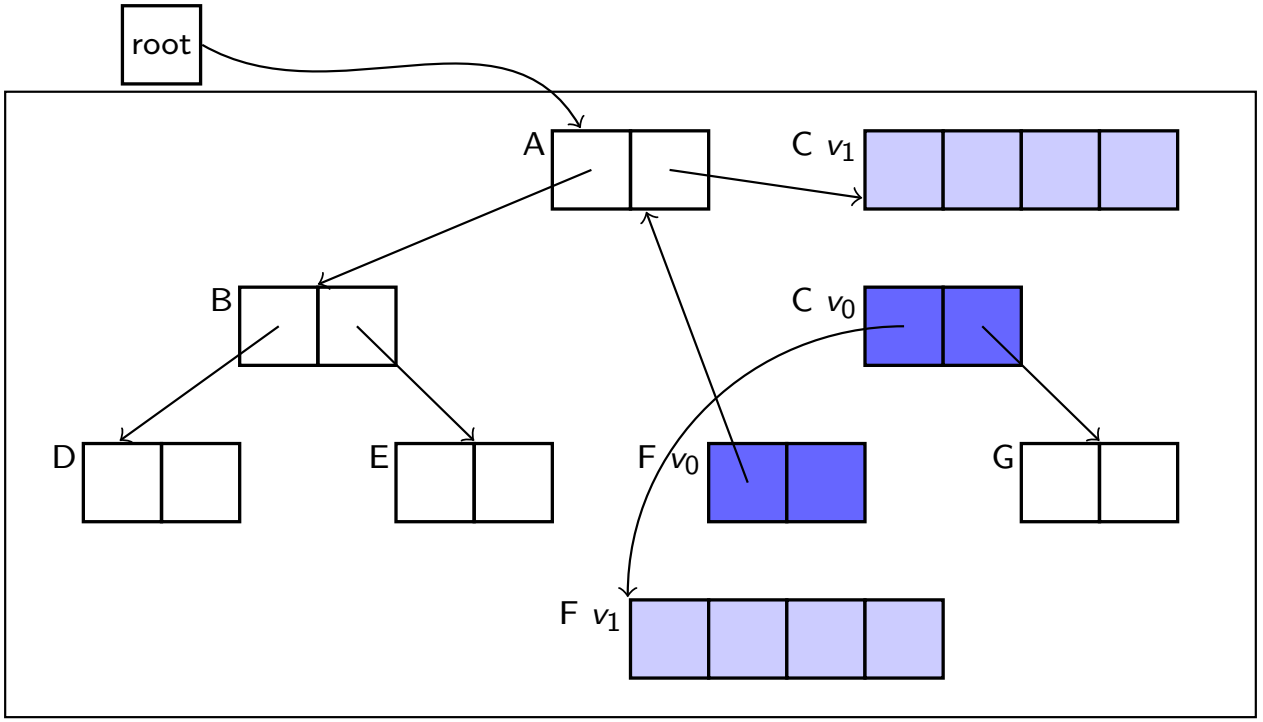
Updating heap data



Transforming objects

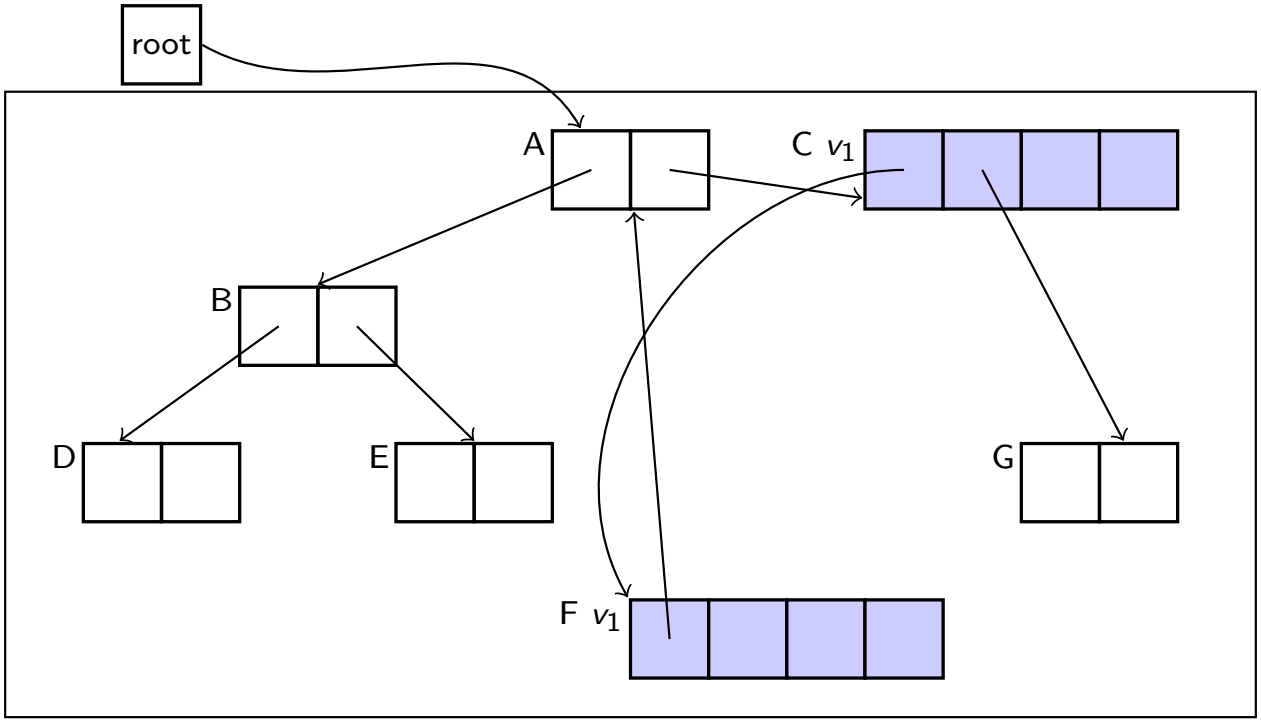
- Supported on top of a semi-space copying collector
- As part of collector's visit allocate additional space for updated objects
- After GC, run class and object transformers

Jvolve Garbage collector



- No field can point to an "old version" object
- "new version" objects are all empty
- Run transformation functions after GC to fill in fields

Jvolve Garbage collector



- No field can point to an "old version" object
- "new version" objects are all empty
- Run transformation functions after GC to fill in fields

Outline

Applications

- 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

Yes we can

Support 20 of 22 updates

Yes we will

- 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

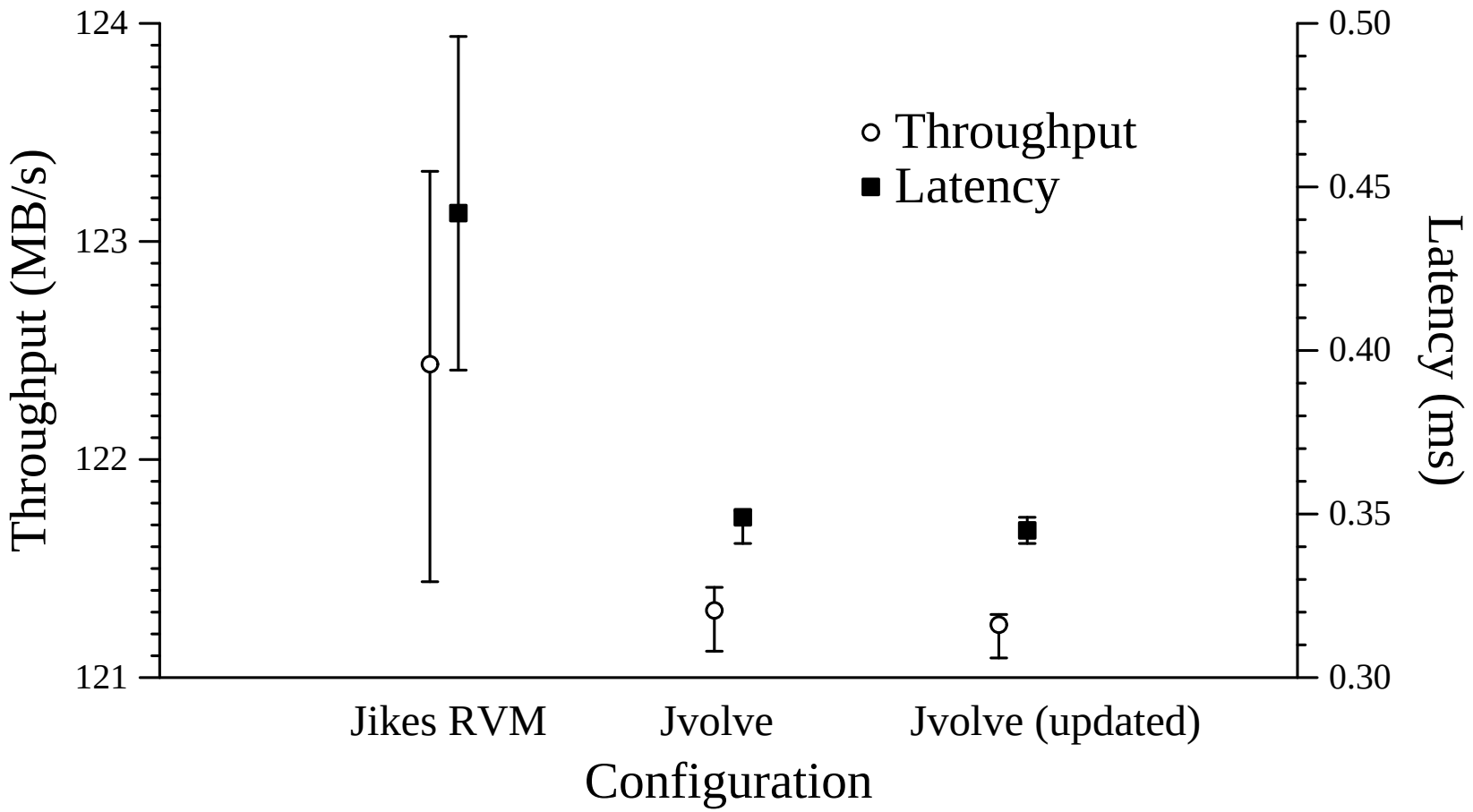
Overhead of DSU

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

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

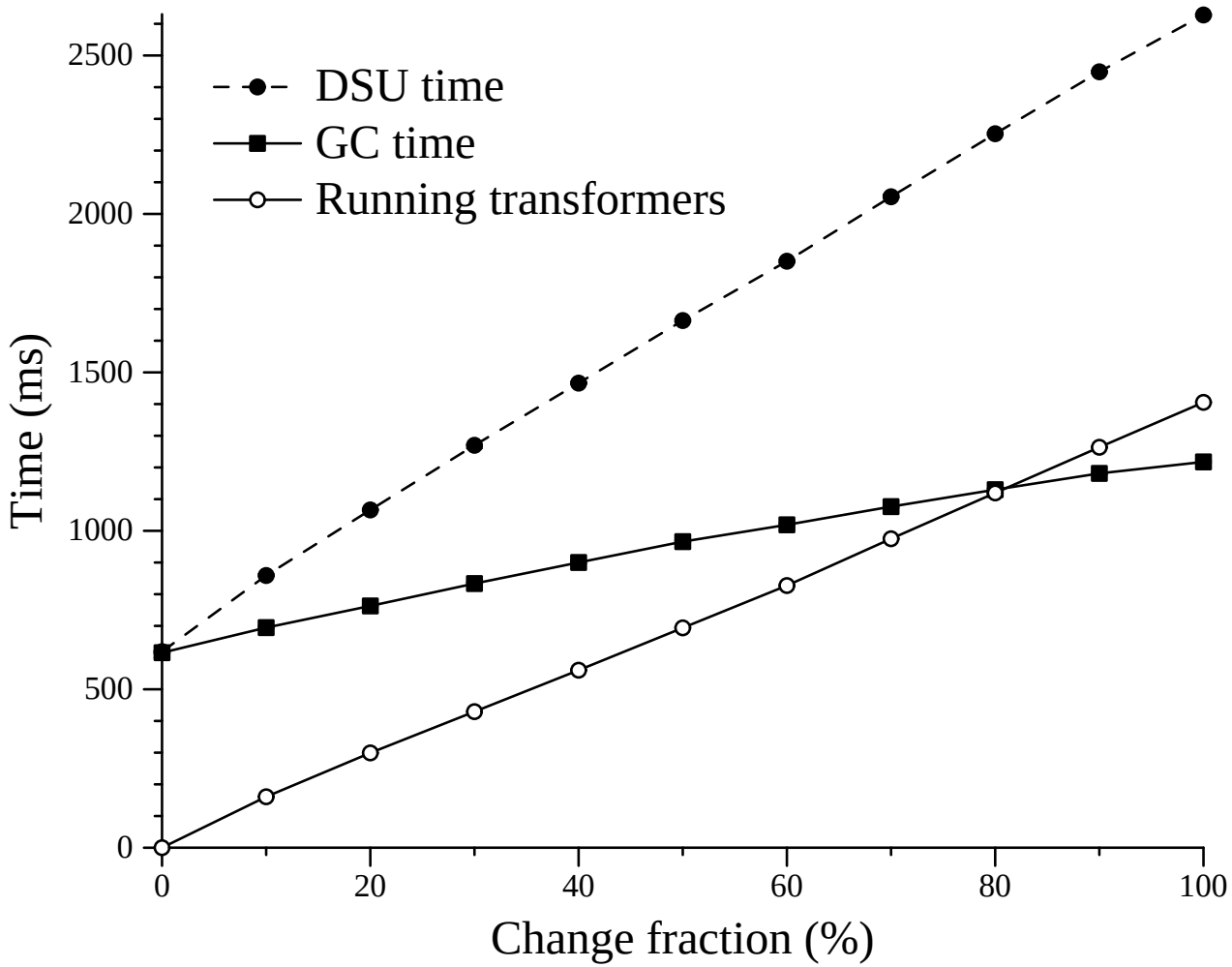
Jetty webserver: Throughput and latency measurements



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

DSU pause times (microbenchmark)



Outline

Future work

- Baby steps towards using static analysis towards asserting safety of updates
- Restricting update points based on semantics of the update
- How can we show that these update points are indeed safe?

Conclusion

- JVOLVE, a Java VM with support for Dynamic Software Updating
- Most-featured, best-performing DSU system for Java
- 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 by naturally extending existing VM services.

Acknowledgements

- Iulian Neamtiu
- Jikes RVM community, Mike Bond, David Grove, Filip Pizlo
- NSF CNS-0346989, NSF CCF-0811524, NSF CNS-0719966, NSF CCF-0429859, Intel, IBM, CISCO, and Microsoft

Thank you