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

April 17, 2009
Parasol Seminar Series

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
- The straightforward approach is to stop and restart applications
- Stopping not desirable
  - Safety concerns
  - Revenue loss
  - Inconvenience

Applications

- Personal operating system
- High availability enterprise applications
- Even a cache with lots of state

[1] http://hurvitz.org/blog/2008/06/linkedin-architecture

Applications

- Personal operating system
- High availability enterprise applications
- Even a cache with lots of state
  - LinkedIn.com architecture[1]
    - "The Cloud": In memory representation of the LinkedIn network graph
    - Network size - 22M nodes, 120M edges
    - Rebuilding an instance takes 8 hours

[1] `http://hurvitz.org/blog/2008/06/linkedin-architecture`

Solutions to updating software

- Move state out of the process
  - State stored externally, for instance databases
  - Redundant systems: start a new process and stop this one
  - Not always possible
- Dynamic Software Updating (DSU)
  - Update process state without restarting application
  - Non-redundant systems benefit as well
  - Decouples fault-tolerance from software updating

Solutions to updating software

- Move state out of the process
  - State stored externally, for instance databases
  - Redundant systems: start a new process and stop this one
  - Not always possible
- Dynamic Software Updating (DSU)
  - Update process state without restarting application
  - Non-redundant systems benefit as well
  - Decouples fault-tolerance from 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

State of the art

Significant progress for C

- Server feature upgrades
  - Ginseng [Neamtiu et al., 2006]
  - POLUS [Chen et al., 2007]
- Security patches: OPUS [Altekar et al., 2005]
- Operating system upgrades
  - K42 [Soules et al., 2003]
  - DynAMOS [Makris and Ryu, 2007]
  - LUCOS [Chen et al., 2006]
  - Ksplice [Arnold and Kaashoek, 2009]

Opportunities for managed languages

Solutions for 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

Existing solutions for managed languages

- VM-based solutions
  - JDrums [Ritzau and Andersson, 2000], DVM [Malabarba et al., 2000]
  - Not well evaluated
  - Provide an interface similar to JVOLVE
  - Perform lazy updates
  - Overheads during normal execution
- Standard VM with DSU support
  - DJVCS [Barr and Eisenbach, 2003], DUSC [Orso et al., 2002], [Milazzo et al., 2005]
  - Special classloaders, compilers
  - Very restrictive
  - Space/time overheads

Our solution

- JVOLVE - a Java Virtual Machine with DSU support
- Key insight: Naturally extend existing VM services
  - Classloading
  - Bytecode verification[2]
  - Thread synchronization
  - JIT Compilation
  - On-stack replacement
  - Garbage collection
- No DSU-related overhead during normal execution
- Support updates to real world applications

[2] 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.

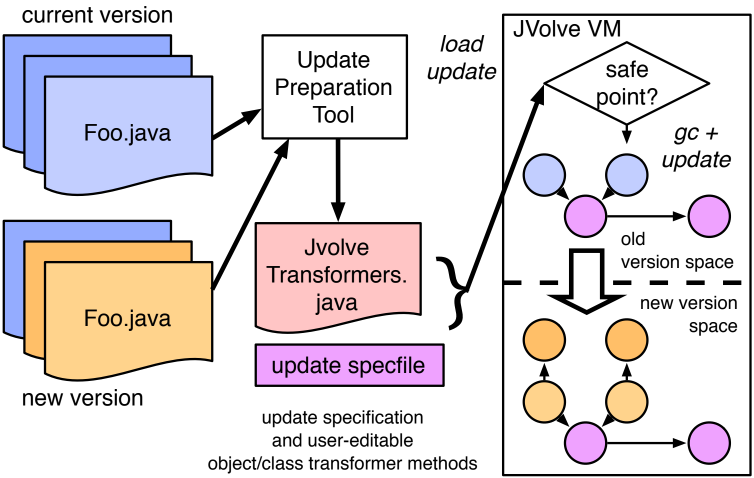
Corollary

DSU support should be a standard feature of future VMs.

Outline

- Introduction
  - Motivation
  - Solutions
- JVOLVE
  - Developer's view
  - Implementation
  - Experience
- Conclusion

Developer's view of JVOLVE



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

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;
-   public String[] getForwardedAddresses() {...}
    public User(...) ...
-   public void setForwardedAddresses(String[] f) {...}
+   private EmailAddress[] forwardAddresses;
+   public EmailAddress[] getForwardedAddresses() {...}
+   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
- A function generated by the Update Preparation Tool (UPT)
- Accepts old object and new object as parameters
- Default transformer copies old fields and initializes new ones to `null`
- User can optionally modify this function

Object Transformers

```
public class v131_User {
  private final String username, domain, pas
  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++) {
      String[] parts = from.forwardAddresses[i].split("@", 2);
      to.forwardAddresses[i] = new EmailAddress(parts[0], parts[1]);
}}}
```

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, pas
  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++) {
      String[] parts = from.forwardAddresses[i].split("@", 2);
      to.forwardAddresses[i] = new EmailAddress(parts[0], parts[1]);
}}}
```

Stub generated by UPT for the old version

Outline

- Introduction
  - Motivation
  - Solutions
- JVOLVE
  - Developer's view
  - Implementation
  - Experience
- Conclusion

Update model

- 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

Update process

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

Update Preparation Tool

Offline tool → Suspend application → Install new classes → Transform objects → Resume application

- Uses jclasslib[3], a bytecode library
- Compares bytecodes of the two versions
- Categorizes changes into

  Updated classes: Classes that add, remove, change signature of fields or methods

  Updated methods: Changes within a method body. Only the method has to be loaded/updated

  Indirect updates: No change to method, but refers to changed classes
- Generates old version stubs and default object transformers

[3] 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

Offline tool → Suspend application → Install new classes → Transform objects → Resume application

- 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

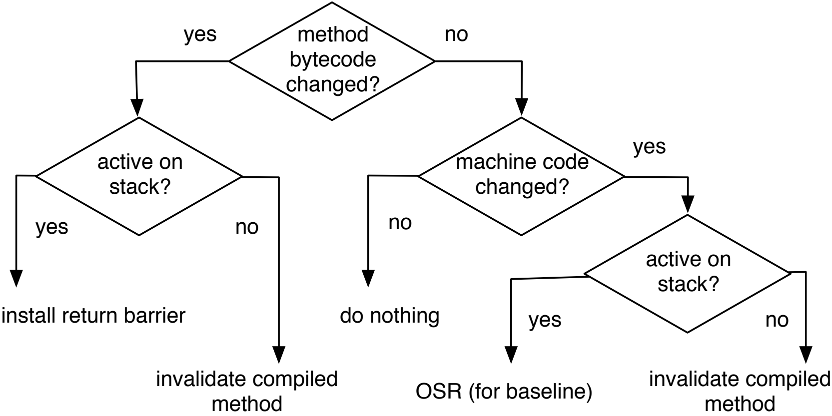
Restricted methods

- (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



On stack replacement in JVOLVE

- Used in Jikes RVM to optimize long running methods
- JVOLVE utilizes OSR for DSU
- Currently only support baseline-compiled methods
- Can OSR any method on stack
- Extract the state of the stack
- Construct a new method with a specialized prologue (at the bytecode level) that reconstructs the stack
- Last instruction of prologue jumps to bytecode where execution should resume from
- Overwrite the return address to point to the special method

On stack replacement in JVOLVE

- Used in Jikes RVM to optimize long running methods
- JVOLVE utilizes OSR for DSU
- Currently only support baseline-compiled methods
- Can OSR any method on stack
- Extract the state of the stack
- Construct a new method with a specialized prologue (at the bytecode level) that reconstructs the stack
- Last instruction of prologue jumps to bytecode where execution should resume from
- Overwrite the return address to point to the special method

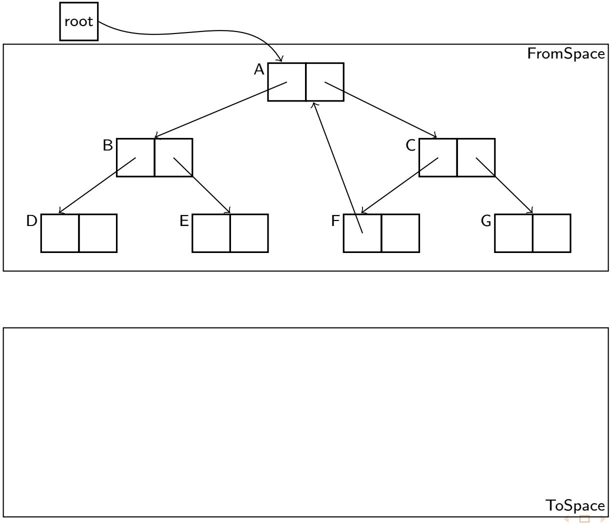
Installing new classes

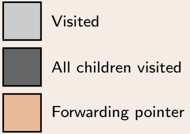
- The VM maintains Class, Method and Field data structures
- For Method updates: Only load the new method's bytecodes
- For Class updates: Rename the old class and load the entire class file (equivalent to have loaded two different class)

Transforming objects

- Built 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

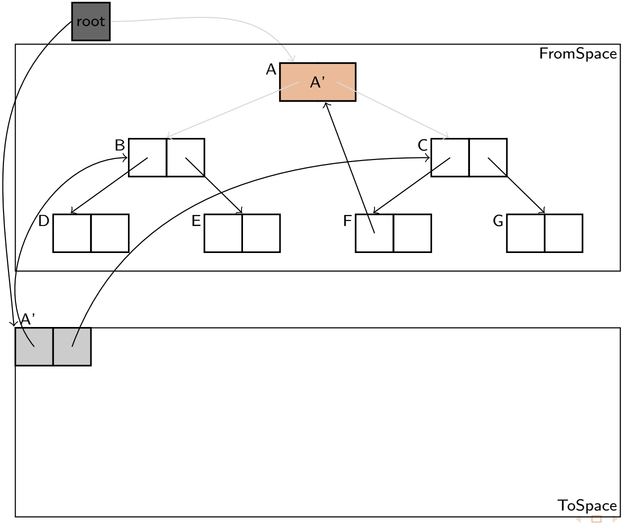
Semi-space copying collector





The heap is divided into two spaces. Only one space is used by the application. The garbage collector copies objects from *FromSpace* to *ToSpace*.

Semi-space copying collector



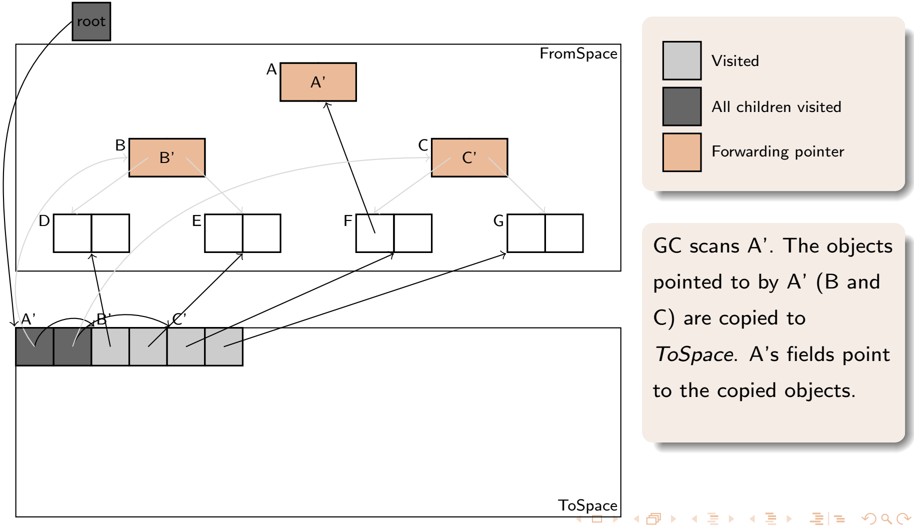
Visited

All children visited

Forwarding pointer

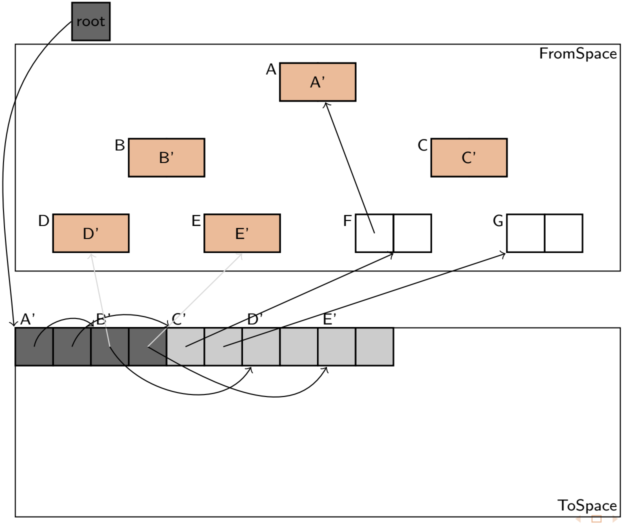
GC copies A to *ToSpace*, leaves a forwarding pointer pointing to the new copy A'.

Semi-space copying collector



GC scans A'. The objects pointed to by A' (B and C) are copied to *ToSpace*. A's fields point to the copied objects.

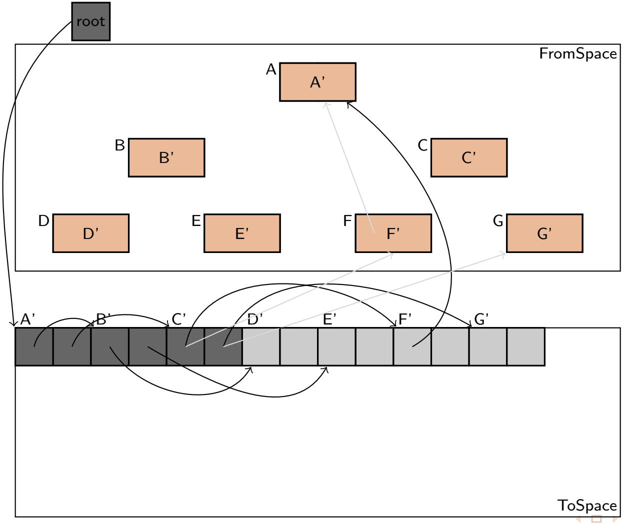
Semi-space copying collector



- Visited
- All children visited
- Forwarding pointer

Next, the GC scans B', and copies objects D and E.

Semi-space copying collector



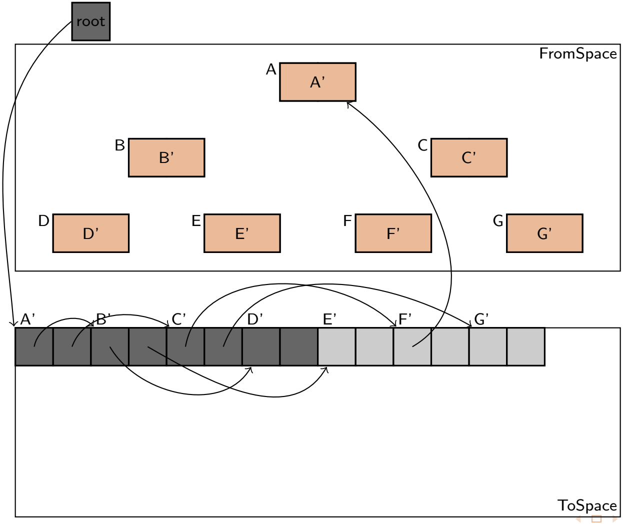
Visited

All children visited

Forwarding pointer

Similarly for C'

Semi-space copying collector



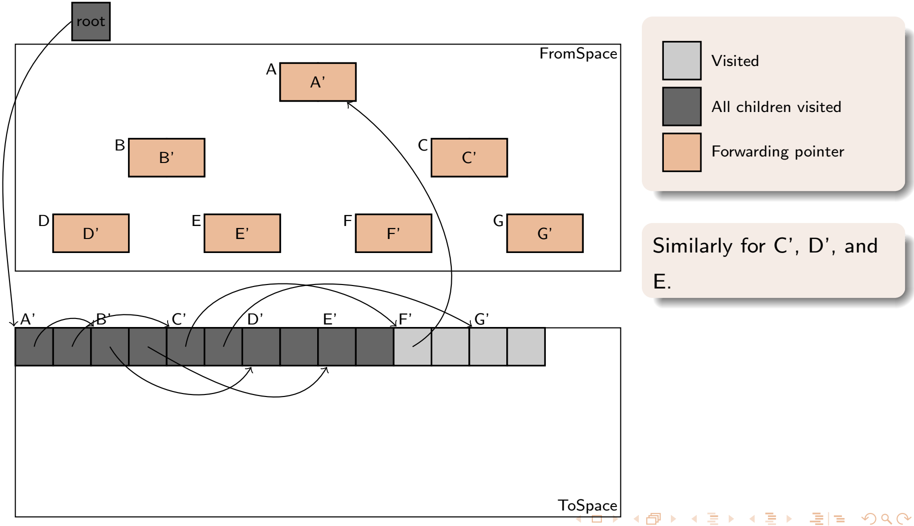
Visited

All children visited

Forwarding pointer

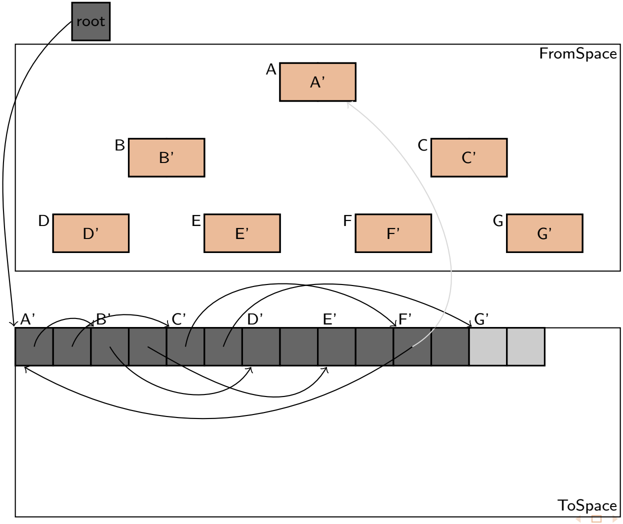
Similarly for C', D'

Semi-space copying collector



Similarly for C', D', and E.

Semi-space copying collector



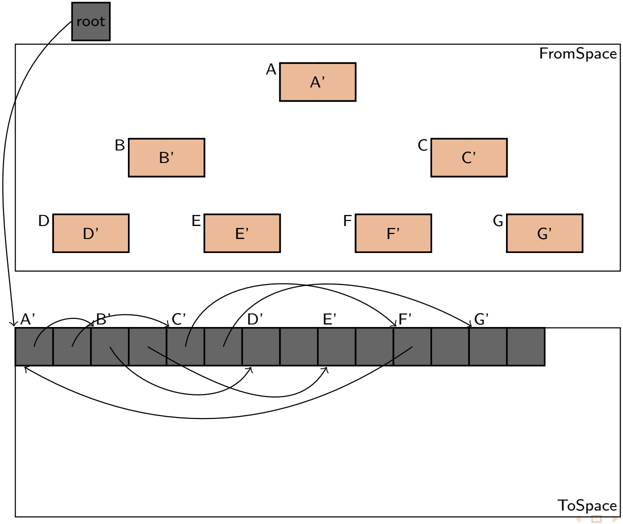
Visited

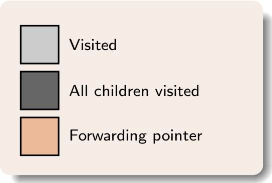
All children visited

Forwarding pointer

When scanning F', the first field points to A in *FromSpace*, which is a forwarding pointer. After the scan, this field points to A'.

Semi-space copying collector



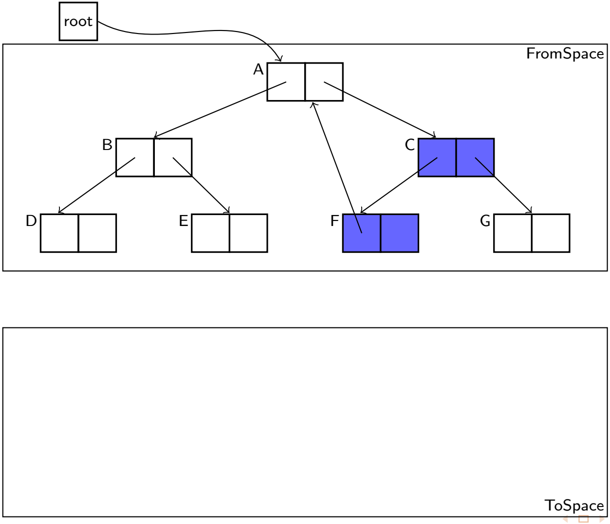


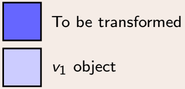
All objects in *ToSpace* are scanned. All reachable/live objects are now in *ToSpace*.

JVOLVE Garbage collector

- Identical to Semispace for "regular" objects
- For objects to be transformed
  - Copy the object to ToSpace (like Semispace)
  - Also, allocate an empty object in ToSpace for the new version
- Forwarding pointers point to the "new version" object
- No field can point to an "old version" object

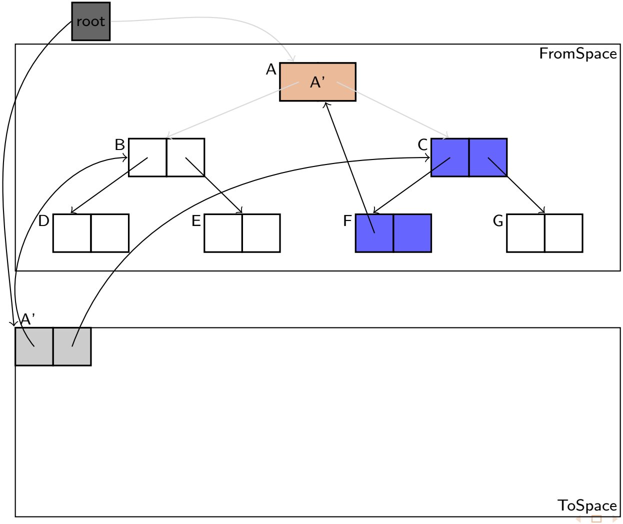
JVOLVE garbage collector





The same heap as before. Objects to be transformed are highlighted.

JVOLVE garbage collector

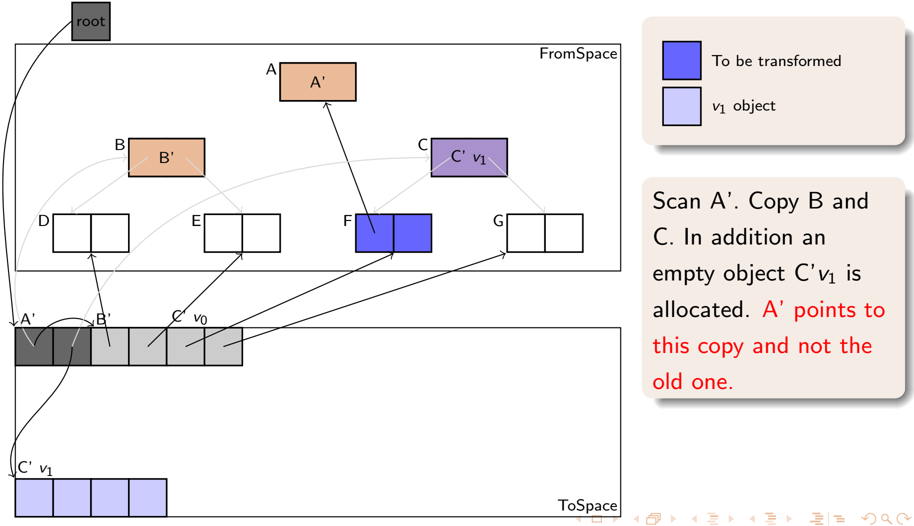


To be transformed

v_1 object

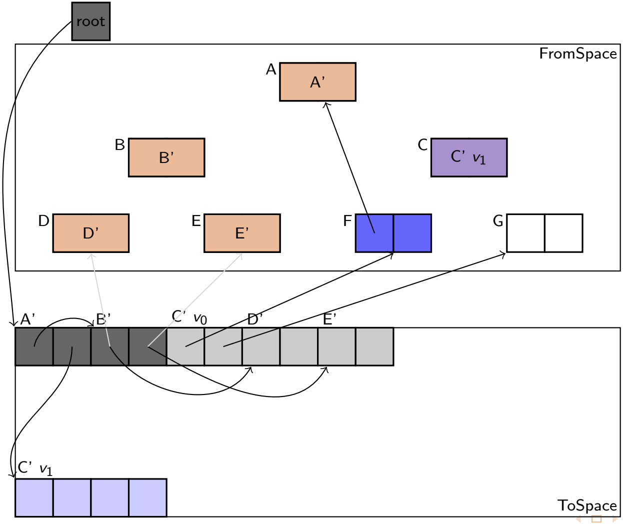
Copy A.

JVOLVE garbage collector



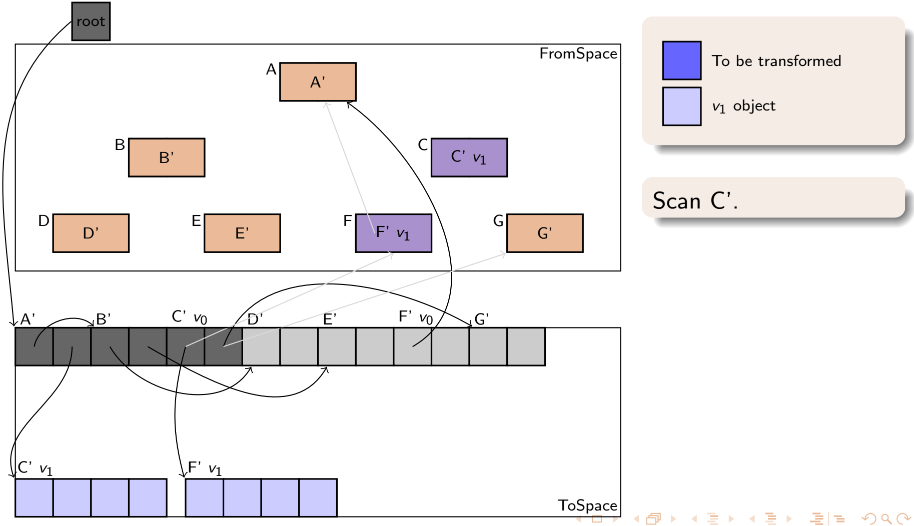
Scan A'. Copy B and C. In addition an empty object C'v_1 is allocated. A' points to this copy and not the old one.

JVOLVE garbage collector



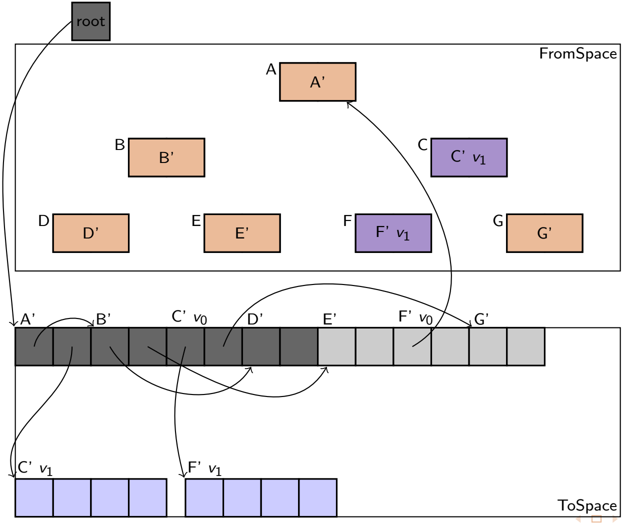
Scan B'.

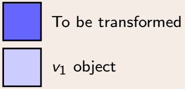
JVOLVE garbage collector



Scan C'.

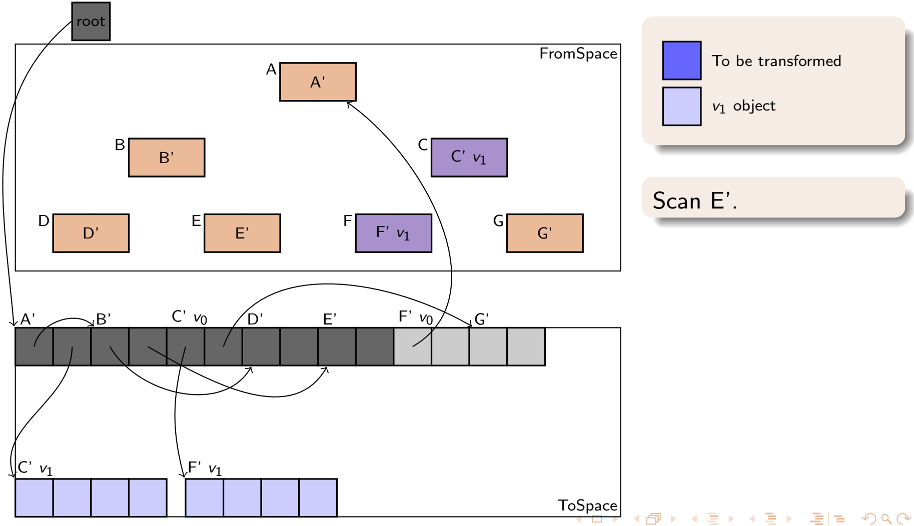
JVOLVE garbage collector





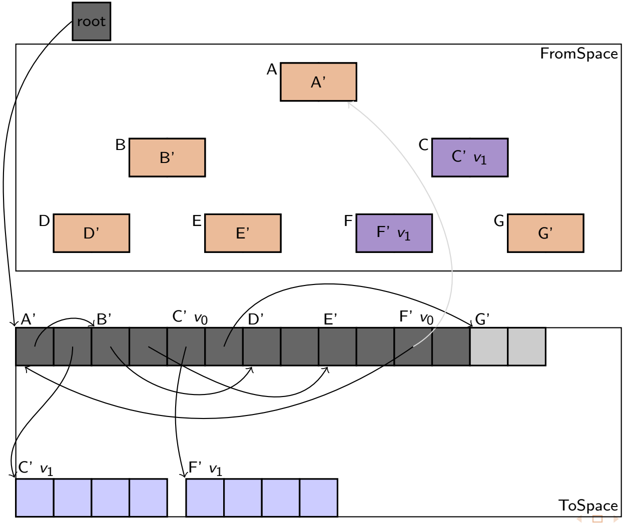
Scan D'.

JVOLVE garbage collector



Scan E'.

JVOLVE garbage collector

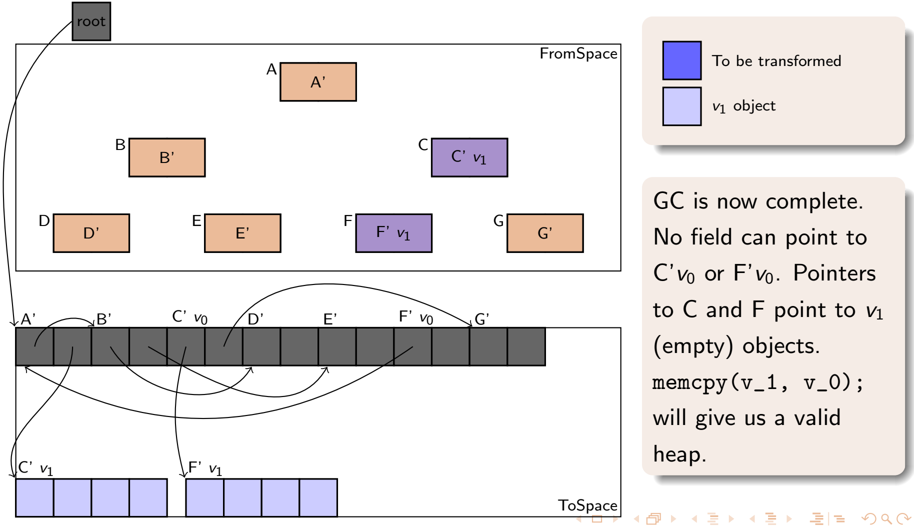


To be transformed

v_1 object

Scan F'.

JVOLVE garbage collector

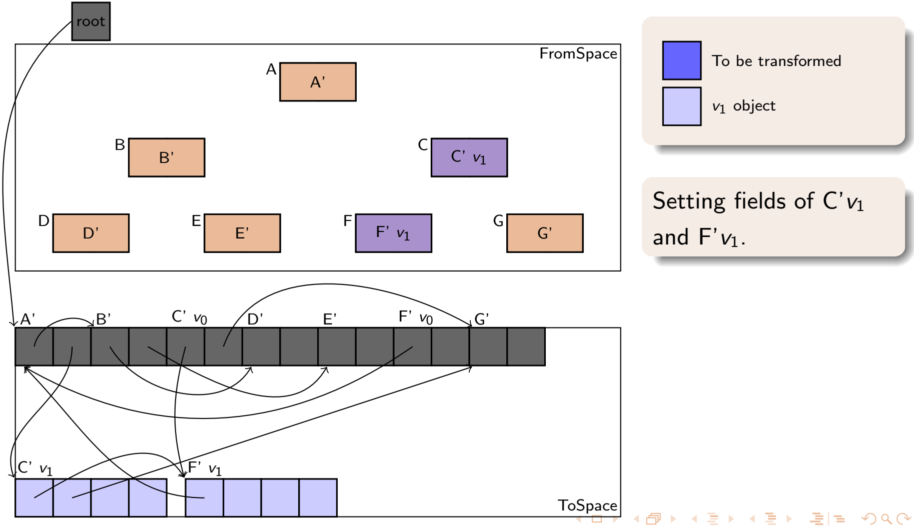


GC is now complete. No field can point to C'v_0 or F'v_0. Pointers to C and F point to v_1 (empty) objects.

`memcpy(v_1, v_0);`

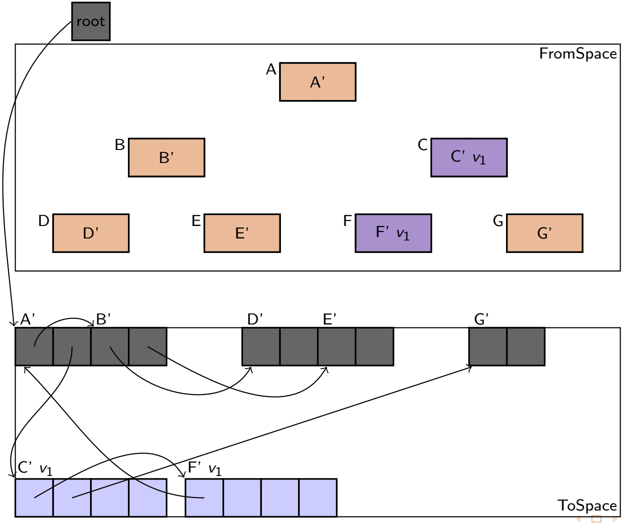
will give us a valid heap.

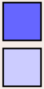
JVOLVE garbage collector



Setting fields of C'v_1 and F'v_1.

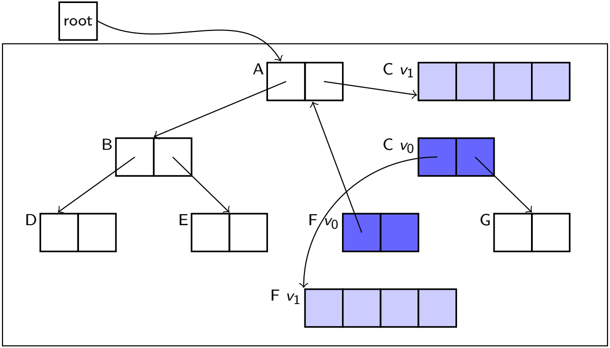
JVOLVE garbage collector



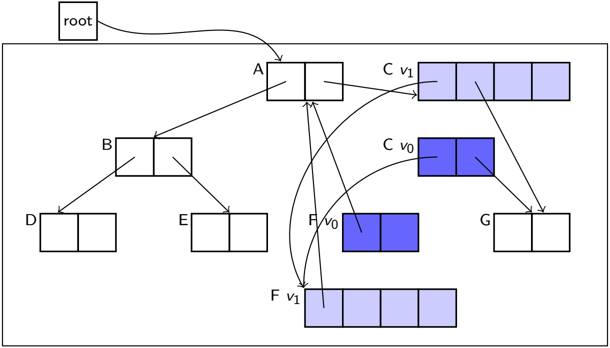


C'v_0 and F'v_0 can be reclaimed.

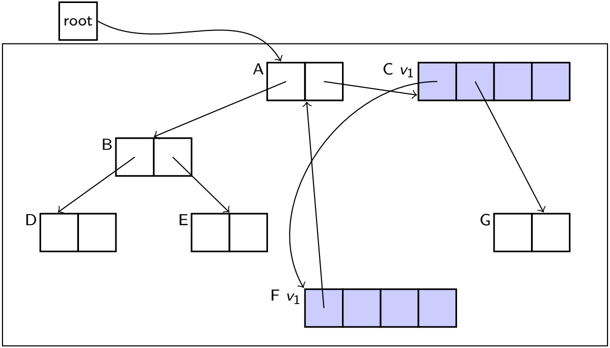
JVOLVE Garbage collector



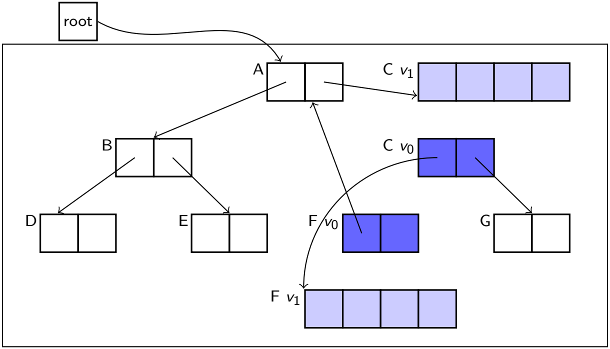
JVOLVE Garbage collector



JVOLVE Garbage collector



Revisiting transformation functions



We have an ordering problem

`(C` v_0`).field0.field0` might be uninitialized

Revisiting transformation functions

Solutions to the ordering problem

- Programmer can invoke a VM function that will transform objects on demand. Moves burden of safety to the programmer
- Insert read barrier code to perform this check when compiling the transformation function
- Perform some static analysis to determine an order to queue objects

Revisiting transformation functions

Solutions to the ordering problem

- Programmer can invoke a VM function that will transform objects on demand. Moves burden of safety to the programmer
- Insert read barrier code to perform this check when compiling the transformation function
- Perform some static analysis to determine an order to queue objects

Updating from a singly to doubly linked list

```
public static void jvolveObject(LinkedList.Node to,
                                r0_LinkedList.Node from) {
  to.next = from.next;
  to.data = from.data;
  if (to.next != null) to.next.prev = to;
}

public static void jvolveObject(LinkedList to, r0_LinkedList from) {
  Jvolve.transformReferences(from);
  to.head = from.head;
  LinkedList.Node n0 = null;
  LinkedList.Node n1 = to.head;
  while (n1 != null) {
    n0 = n1;
    n1 = n1.next;
  }
  to.tail = n0;
}
```

Outline

- Introduction
  - Motivation
  - Solutions
- JVOLVE
  - Developer's view
  - Implementation
  - Experience
- Conclusion

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

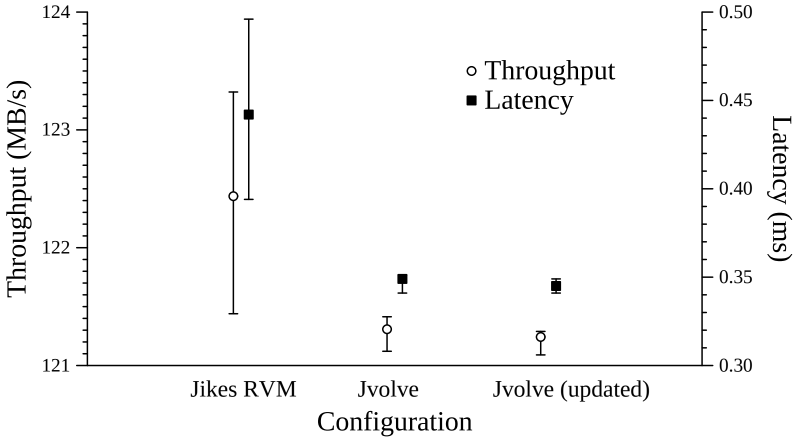
Overhead of DSU

- 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
- Create 800 new connections/second (saturation rate)
- 5 serial requests to 40KB file per connection
- Compared versions 5.1.5 and 5.1.6
- Experiments on Intel Core 2 Quad, Linux 2.6.22, JikesRVM SVN r15532

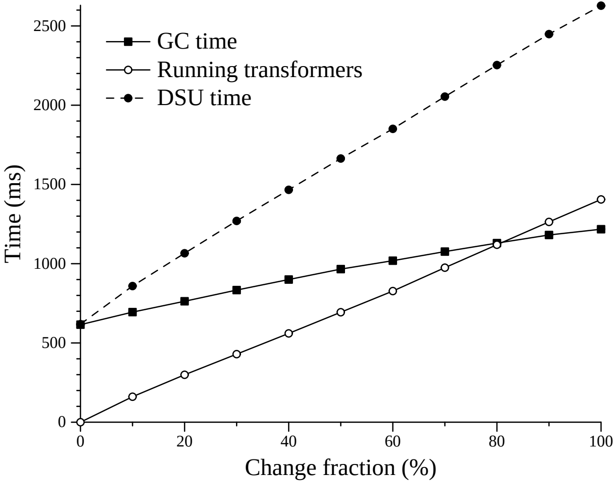
Jetty webserver: Throughput measurements



DSU pause times

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

DSU pause times (microbenchmark)



Jetty webserver: Summary of changes

Ver.	# classes added	# changed classes	methods add	methods del	methods chg	fields add	fields del
5.1.1	0	14	4	1	38/0	0	0
5.1.2	1	5	0	0	12/1	0	0
5.1.3	3	15	19	2	59/0	10	1
5.1.4	0	6	0	4	9/6	0	2
5.1.5	0	54	21	4	112/8	5	0
5.1.6	0	4	0	0	20/0	5	6
5.1.7	0	7	8	0	11/2	9	3
5.1.8	0	1	0	0	1/0	0	0
5.1.9	0	1	0	0	1/0	0	0
5.1.10	0	4	0	0	4/0	0	0

Unsupported updates

- 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
- 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

Outline

- Introduction
  - Motivation
  - Solutions
- JVOLVE
  - Developer's view
  - Implementation
  - Experience
- Conclusion

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.

Thank you

References I

Altekar, G., Bagrak, I., Burstein, P., and Schultz, A. (2005).
OPUS: Online patches and updates for security.
In *Proc. USENIX Security*.

Arnold, J. and Kaashoek, F. (2009).
Ksplice: Automatic rebootless kernel updates.
In *Proc. EuroSys*.

Barr, M. and Eisenbach, S. (2003).
Safe upgrading without restarting.
In *Proc. ICSM*.

References II

Chen, H., Chen, R., Zhang, F., Zang, B., and Yew, P.-C. (2006).
Live updating operating systems using virtualization.
In *Proc. VEE*.

Chen, H., Yu, J., Chen, R., Zang, B., and Yew, P.-C. (2007).
POLUS: A POwerful Live Updating System.
In *Proc. ICSE*.

Makris, K. and Ryu, K. D. (2007).
Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels.
In *Proc. EuroSys*.

References III

Malabarba, S., Pandey, R., Gragg, J., Barr, E., and Barnes, J. F. (2000).
Runtime support for type-safe dynamic Java classes.
In *Proc. ECOOP*.

Milazzo, M., Pappalardo, G., Tramontana, E., and Ursino, G. (2005).
Handling run-time updates in distributed applications.
In *Proc. SAC*.

Neamtiu, I., Hicks, M., Stoyle, G., and Oriol, M. (2006).
Practical dynamic software updating for C.
In *Proc. PLDI*.

References IV

Orso, A., Rao, A., and Harrold, M. J. (2002).
A technique for dynamic updating of Java software.
In *Proc. ICSM*.

Ritzau, T. and Andersson, J. (2000).
Dynamic deployment of Java applications.
In *Proc. Java for Embedded Systems Workshop*.

Soules, C., Appavoo, J., Hui, K., Silva, D. D., Ganger, G., Krieger, O., Stumm, M., Wisniewski, R., Auslander, M., Ostrowski, M., Rosenburg, B., and Xenidis, J. (2003).
System support for online reconfiguration.
In *Proc. USENIX Annual Technical Conference*.