

Dynamic Software Updates: A VM-centric Approach

Suriya Subramanian¹ Michael Hicks²
Kathryn S. McKinley¹

¹Department of Computer Sciences
The University of Texas at Austin

²Department of Computer Science
University of Maryland

April 24, 2009
PL Lunch Series



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

Updating \Rightarrow Downtime

Downtime

- 75% of downtime in high-availability applications is for planned maintenance

¹<http://hurvitz.org/blog/2008/06/linkedin-architecture>

Downtime

- 75% of downtime in high-availability applications is for planned maintenance
- Personal operating system
- High availability enterprise applications
- Less critical applications like Twitter, Youtube

¹<http://hurvitz.org/blog/2008/06/linkedin-architecture>

Downtime

- 75% of downtime in high-availability applications is for planned maintenance
- Personal operating system
- High availability enterprise applications
- Less critical applications like Twitter, Youtube
- Even a cache with lots of state
 - LinkedIn.com architecture¹
 - “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

DSU Opportunities for managed languages

DSU 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

Possible DSU solutions

Achieve DSU support by

- Making the application DSU-aware
- Special recompilation
- A class loader based solution
- DSU support in the VM

Our solution

- JVOLVE - a Java Virtual Machine with DSU support
- Key insight: Naturally extend existing VM services
 - Classloading
 - Bytecode verification²
 - Thread synchronization
 - JIT Compilation
 - On-stack replacement
 - Garbage collection
- No DSU-related overhead during normal execution
- Support updates to real world applications

²Jikes 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.

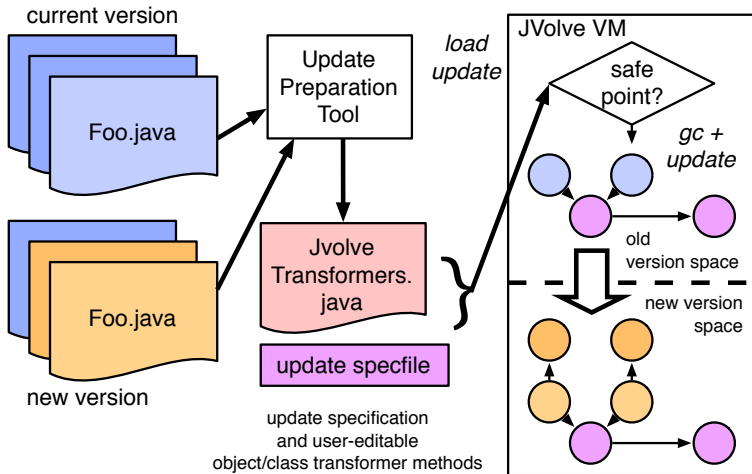
Corollary

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

Stub generated by UPT for
the old version

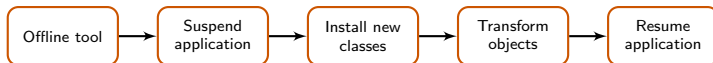
Outline

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

Update model

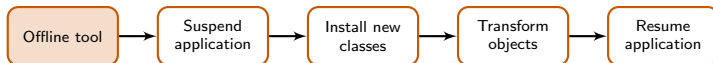
- Update happens in one fell swoop
- Simple to reason about
- Code
 - Old code before the update
 - New code after the update
- Data
 - Representation consistency (all values of a type correspond to the latest version)
 - Support a transformation function to convert objects to conform to their new definition

Update process



- Offline Update Preparation Tool (UPT)
- JVM 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



- Uses jclasslib³, a bytecode library
- Compares bytecodes of the two versions
- 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

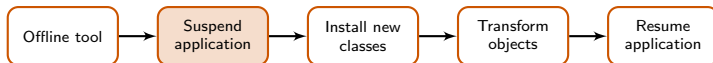
```
jvolveClass(ClassName unused)
```

- 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



- 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

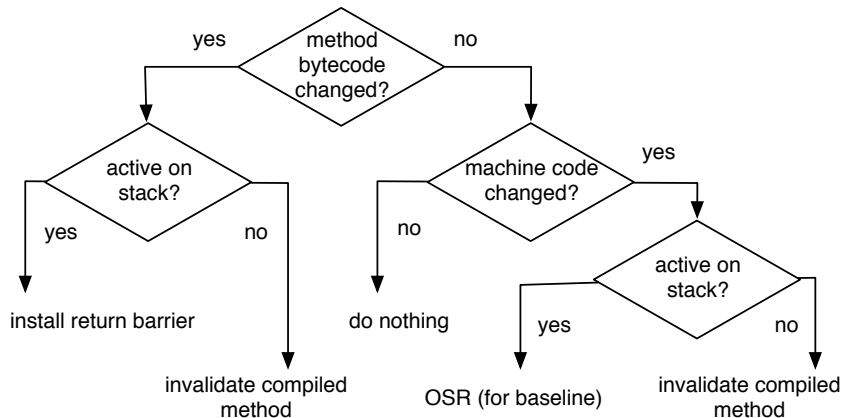
Restricted methods

- (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

OSR Example

```
public class A {  
    public int foo(int i, B b) {  
        i = i + 1;  
        i = b.bar(i);  
        i = i + 1;  
        return i;  
    }  
}
```

```
0 iload_1  
1 iconst_1  
2 iadd  
3 istore_1  
4 aload_2  
5 iload_1  
6 invokevirtual <B.bar>  
9 istore_1  
10 iload_1  
11 iconst_1  
12 iadd  
13 istore_1  
14 iload_1  
15 ireturn
```

OSR Example

```
public class A {  
    public int foo(int i, B b) {  
        i = i + 1;  
        i = b.bar(i);  
        i = i + 1;  
        return i;  
    }  
}
```

State:

Thread: #5

FP: 0x4ad33e40

PC: 9

Locals: i = 5, b = 0x52ae34a0

Stack vars: S0, S1, ...

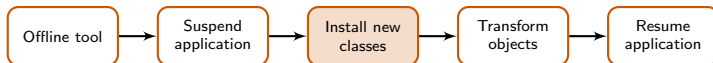
```
0 iload_1  
1 iconst_1  
2 iadd  
3 istore_1  
4 aload_2  
5 iload_1  
6 invokevirtual <B.bar>  
9 istore_1  
10 iload_1  
11 iconst_1  
12 iadd  
13 istore_1  
14 iload_1  
15 ireturn
```

OSR Example

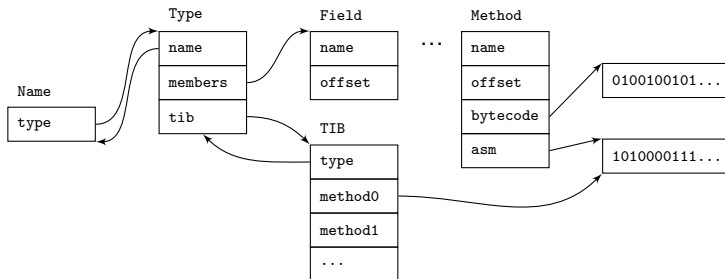
```
0 iload_1
1 iconst_1
2 iadd
3 istore_1
4 aload_2
5 iload_1
6 invokevirtual <B.bar>
9 istore_1
10 iload_1
11 iconst_1
12 iadd
13 istore_1
14 iload_1
15 ireturn
```

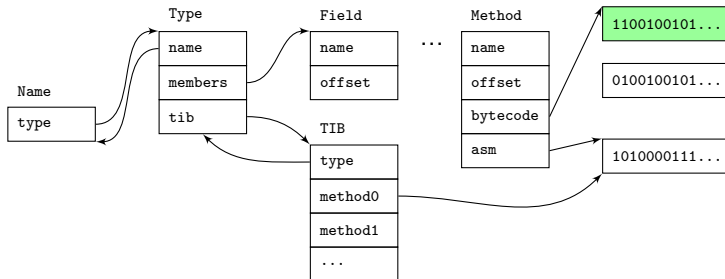
```
ldc 5
istore_0
ldc 0x52ae34a0
astore_1
goto 9
0 iload_1
1 iconst_1
2 iadd
3 istore_1
4 aload_2
5 iload_1
6 invokevirtual <B.bar>
9 istore_1
10 iload_1
11 iconst_1
12 iadd
13 istore_1
14 iload_1
15 ireturn
```

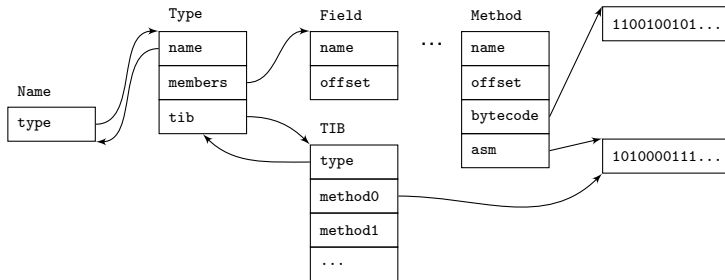
Installing new classes

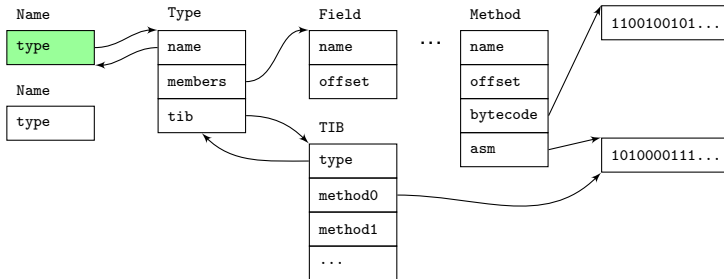


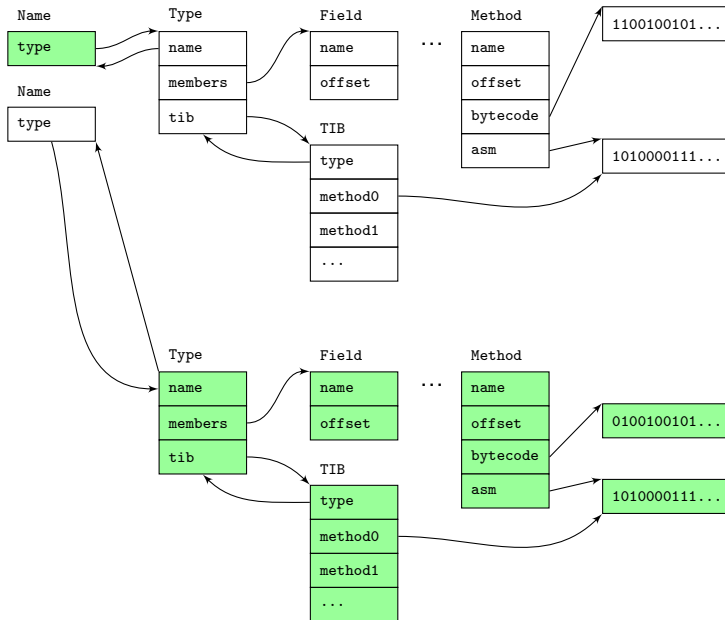
- The VM maintains Class, Method and Field data structures
- For Method updates: Only load the new method's bytecodes
- For Class updates: Rename the old class and load the entire class file (equivalent to have loaded two different class)



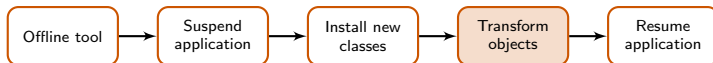








Transforming objects

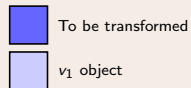
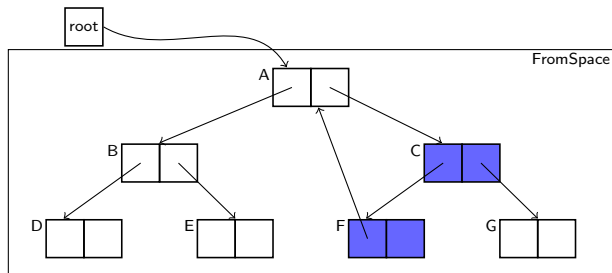


- Built on top of a semi-space copying collector
- As part of collector's visit allocate additional space for updated objects
- After GC, run class and object transformers

JVOLVE Garbage collector

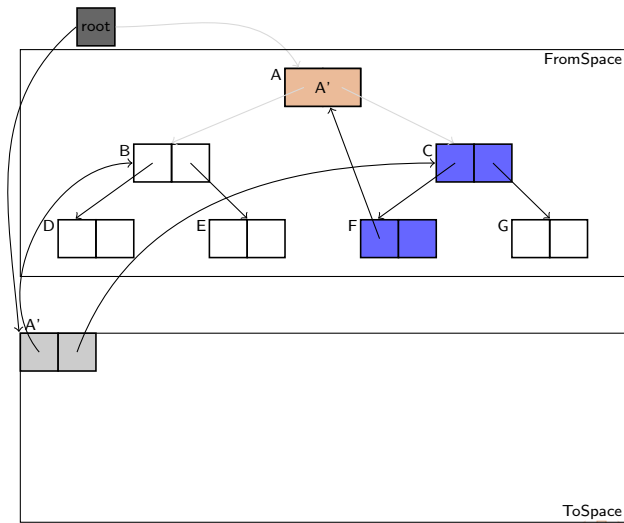
- Identical to Semispace for “regular” objects
- For objects to be transformed
 - Copy the object to ToSpace (like Semispace)
 - Also, allocate an empty object in ToSpace for the new version
- Forwarding pointers point to the “new version” object
- No field can point to an “old version” object



JVOLVE garbage collector



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

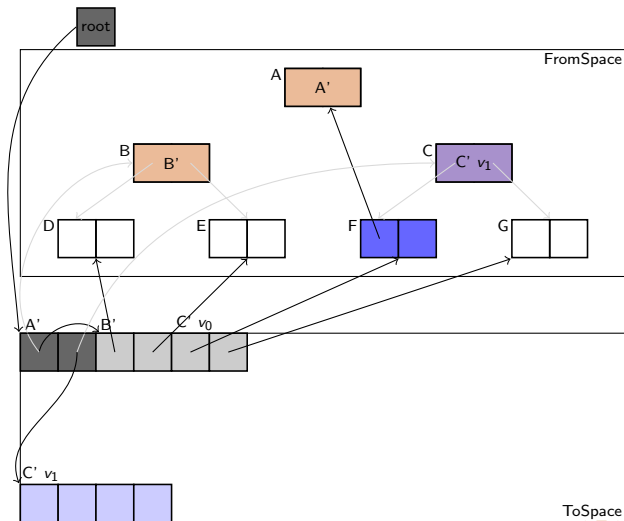
JVOLVE garbage collector



 To be transformed
 v₁ object

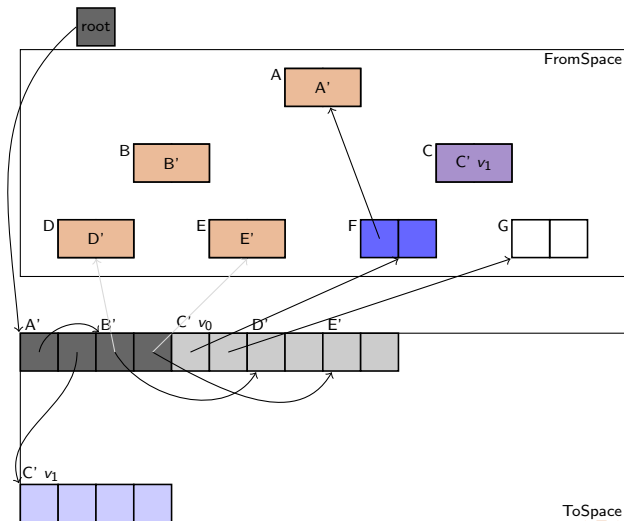
Copy A.

JVOLVE garbage collector

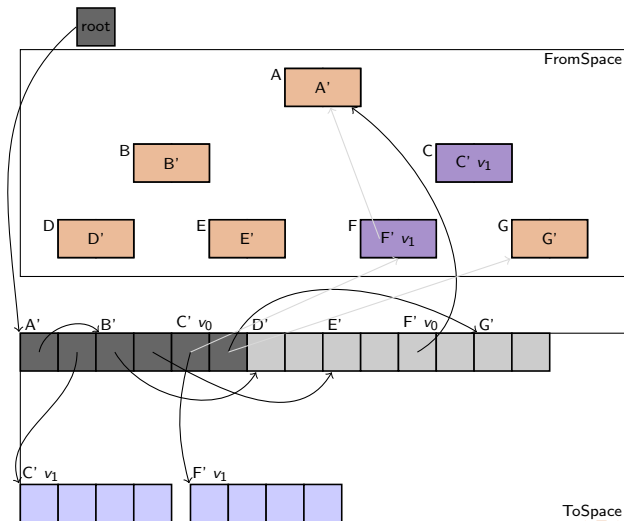


Scan A'. Copy B and C. In addition an empty object C' v₁ is allocated. **A' points to this copy and not the old one.**

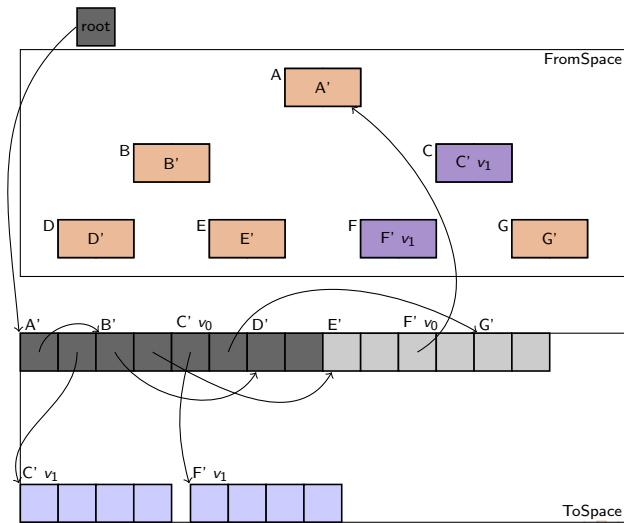
JVOLVE garbage collector





JVOLVE garbage collector



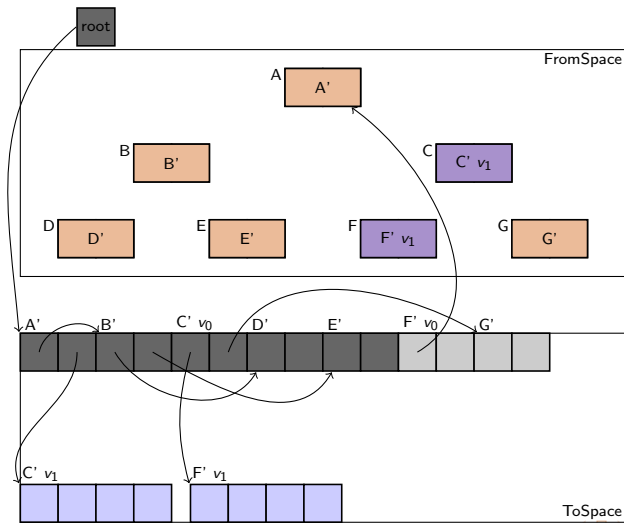
JOLVE garbage collector



 To be transformed
 v_1 object

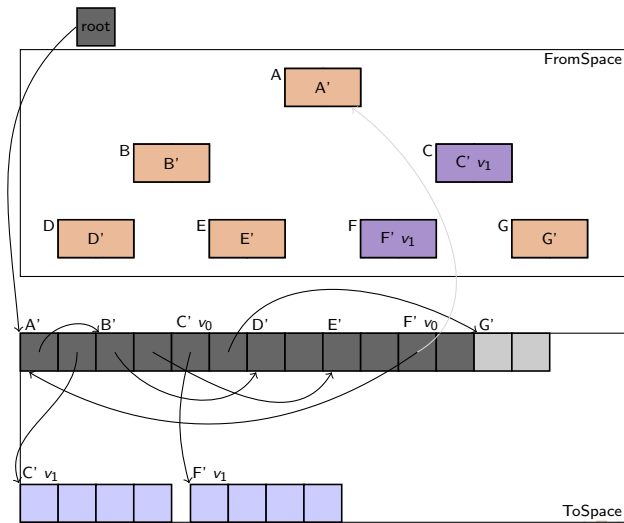
Scan D'.

JOLVE garbage collector

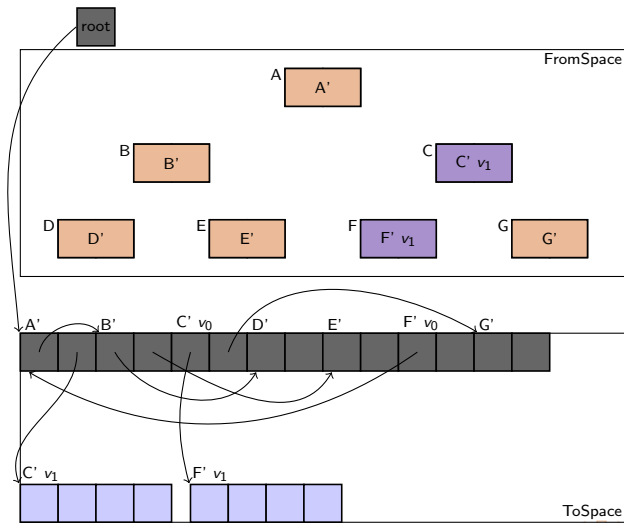


Scan E'.

JOLVE garbage collector

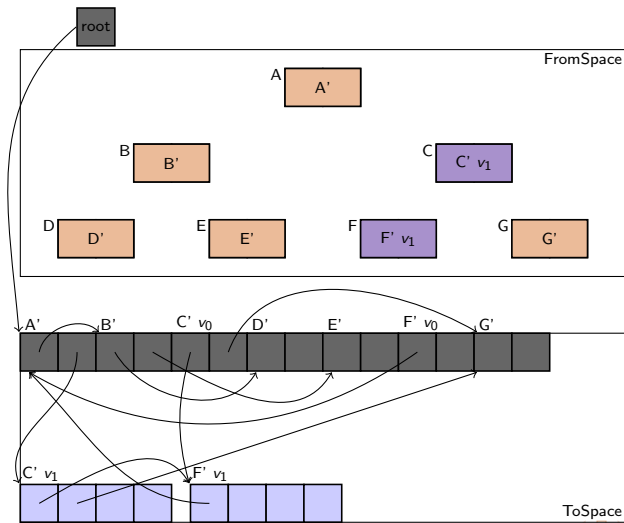


JVOLVE garbage collector

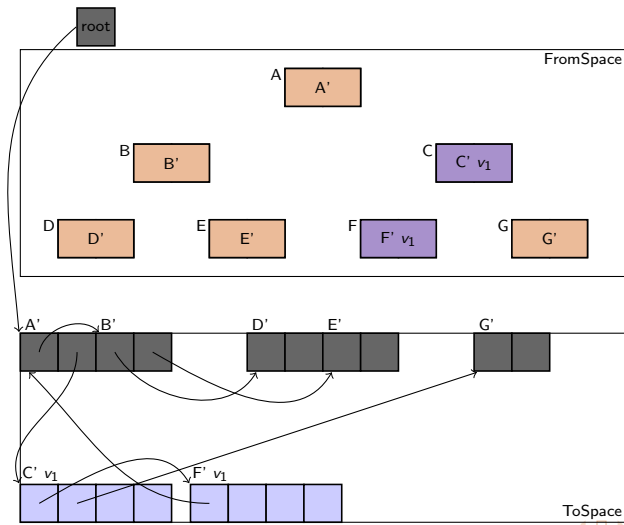



GC is now complete.
No field can point to C' v₀ or F' v₀. Pointers to C and F point to v₁ (empty) objects.
`memcpy(v1, v0);`
will give us a valid heap.


JVOLVE garbage collector



JVOLVE garbage collector

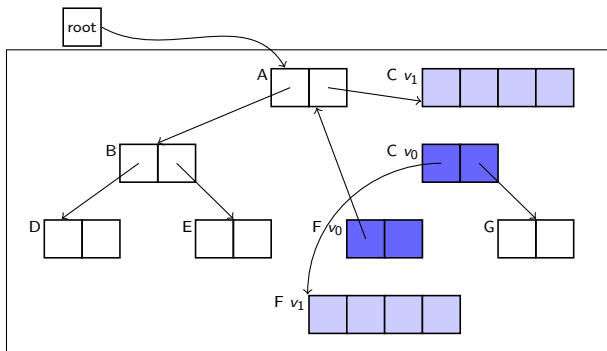


 To be transformed

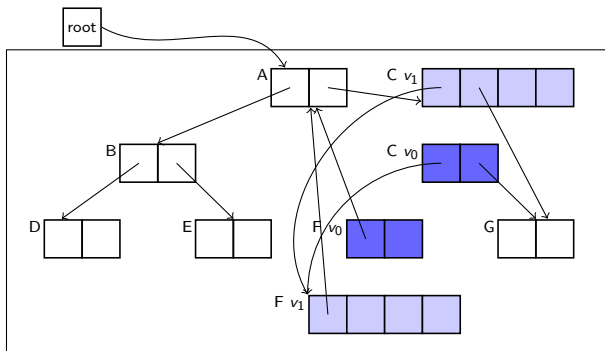
 v₁ object

C'v₀ and F'v₀ can be reclaimed.

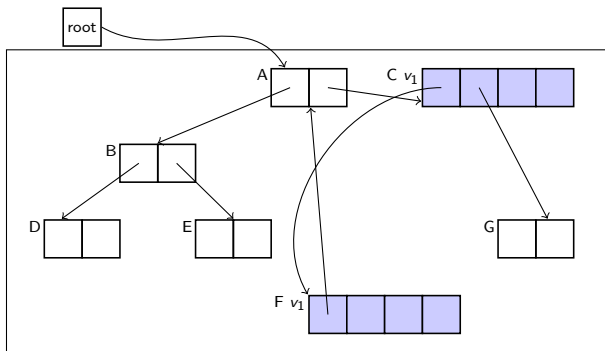
JVOLVE Garbage collector



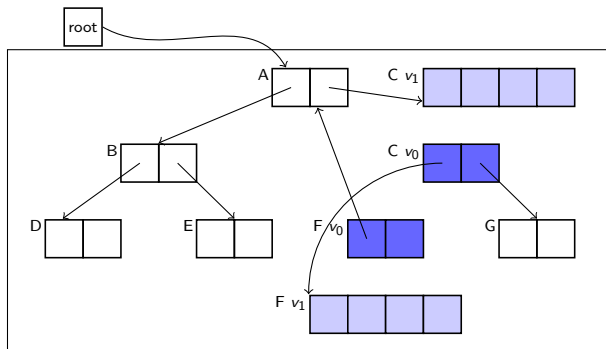
JOLVE Garbage collector



JVOLVE Garbage collector



Revisiting transformation functions



We have an ordering problem

`(C v0).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

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

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

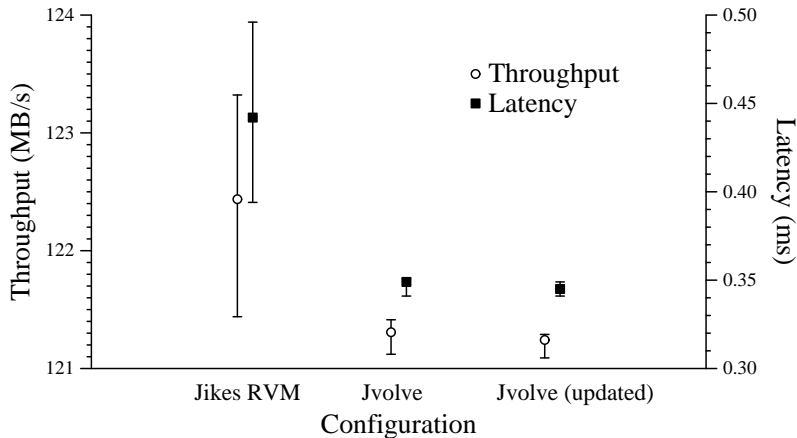
Overhead of DSU

- 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

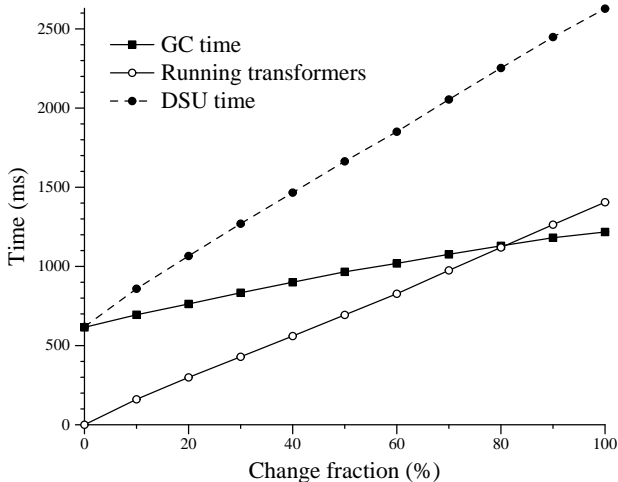
Jetty webserver: Throughput measurements



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)



Outline

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

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.

Thank you