Dynamic Software Updates in Java: A VM-centric approach

Suriya Subramanian
Advisor: Kathryn S. McKinley

Department of Computer Sciences
The University of Texas at Austin

July 21, 2008
Ph.D. Oral Proposal

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
- 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 [ksplice, 2008]

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
  - Thread synchronization
  - Garbage collection
  - On-stack replacement
- No DSU-related overhead during normal execution
- Support updates to real world applications

Thesis

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

Thesis

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

We propose to affirm this claim by building a JVM with DSU support and studying updates to real world applications.

Thesis

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

We propose to affirm this claim by building a JVM with DSU support and studying updates to real world applications.

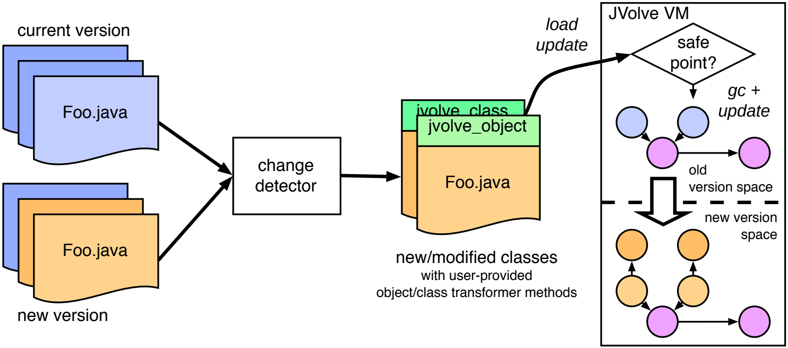
Corollary

DSU support should be a standard feature of future VMs.

Outline

- Introduction
  - Motivation
  - Solutions
  - Thesis
- 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 String username, domain, password;
  private String[] forwardAddresses;
  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 String username, domain, password;
-   private String[] forwardAddresses;
-   public String[] getForwardedAddresses() {...}
-   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 specified as part of the new version of a class
- Accepts old object and new object as parameters
- VM runs a default transformer that copies old fields and initializes new ones to `null`

Object Transformers

```
public class old_User {
  public String username, domain, password;
  public String[] forwardAddresses;
}
public class User {
  ...
  public static void jvolve_class() { ... }
  public static void jvolve_object(User to, old_User from) {
    to.username = from.username;
    to.domain = from.domain;
    to.password = from.password;
    to.forwardAddresses = null;
    int l = from.forwardAddresses.length;
    to.forwardAddresses = new EmailAddress[l];
    for (int i = 0; i < l; 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 old_User {
  public String username, domain, password;
  public String[] forwardAddresses;
}
public class User {
  ...
  public static void jvolve_class() { ... }
  public static void jvolve_object(User to, old_User from) {
    to.username = from.username;
    to.domain = from.domain;
    to.password = from.password;
    to.forwardAddresses = null;
    int l = from.forwardAddresses.length;
    to.forwardAddresses = new EmailAddress[l];
    for (int i = 0; i < l; 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`

Outline

- Introduction
  - Motivation
  - Solutions
  - Thesis
- 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 the new type

Update process

- Offline Update Preparation Tool (UPT)
- JVOLVE VM
  - Reach a safe point in the VM (thread synchronization)
  - Load new classes (classloader)
  - Transform objects to new definition (garbage collector)
  - Resume execution
  - Update active methods on stack (On-stack replacement)

Update process

- Offline Update Preparation Tool (UPT)
- JVOLVE VM
    - Reach a safe point in the VM (thread synchronization)
    - Load new classes (classloader)
    - Transform objects to new definition (garbage collector)
    - Resume execution
    - Update active methods on stack (On-stack replacement)

Update Preparation Tool

- Uses jclasslib[2], a bytecode library
- Compares bytecode of the two versions
- Categorizes changes into
  - Class updates: Classes that add, remove, change signature of fields or methods
  - Method updates: 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

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

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)
- Uses a simple, non-deterministic timer retry
- Supported only on a single processor
- On-stack replacement support to allow more methods to remain on stack

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)
- Uses a simple, non-deterministic timer retry
- Supported only on a single processor
- On-stack replacement support to allow more methods to remain on stack

Restricted methods

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

- Identified by UPT
  - All methods in updated classes
  - Methods with new implementations
  - Methods that refer to updated classes (have to be recompiled since field/method offsets might have changed)
- Identified by the VM
  - With inlining, transitive closure of callers of methods identified above

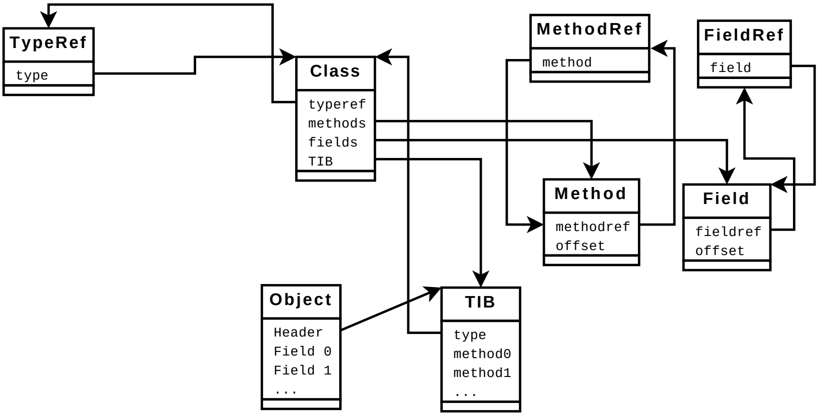
Loading new classes

- Modifications to the classloader
- Involved a lot of engineering effort

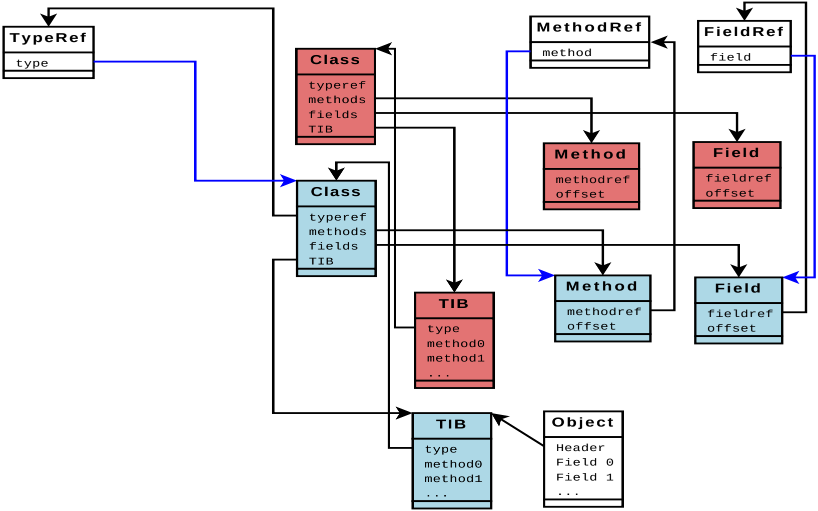
Loading new classes

- Modifications to the classloader
- Involved a lot of engineering effort
- Correctly update metadata maintained by the VM
  - Classes
  - Type information blocks
  - Methods
  - Fields
  - Subclass, superclass relations
  - Innerclasses
  - Exceptions
  - Annotations

VM Datastructures



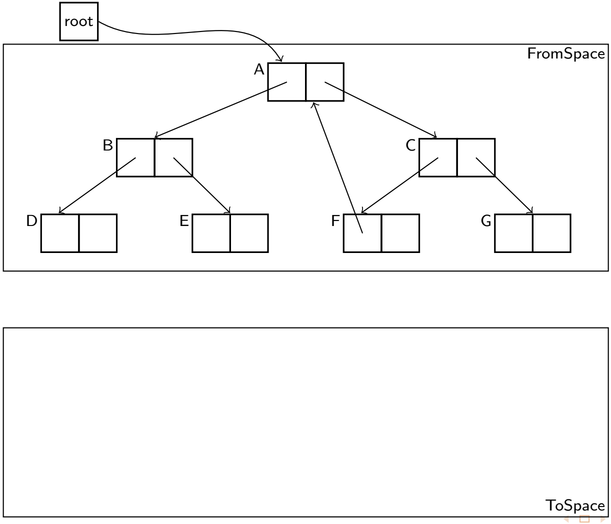
VM Datastructures

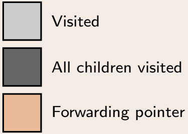


Transforming objects

- Built on top of a semi-space copying collector
- Allocate additional space and run object transformers as part of the collector's visit

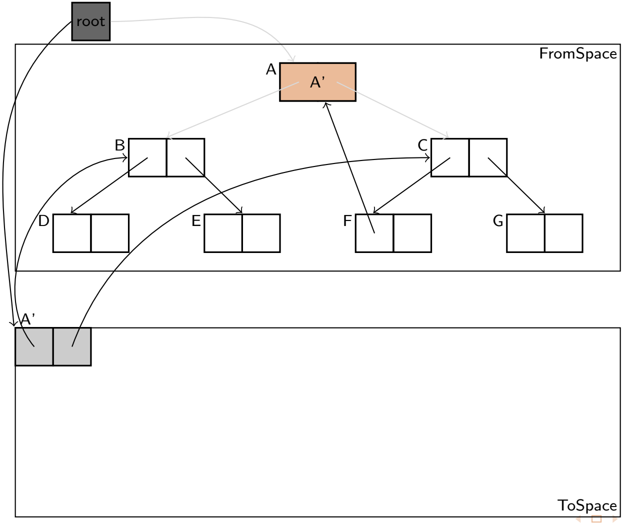
Semi-space copying collector





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



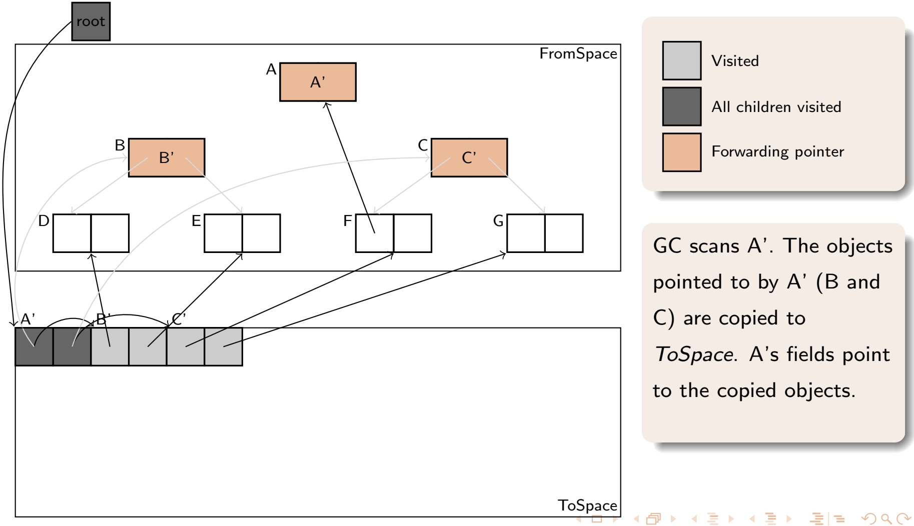
Visited

All children visited

Forwarding pointer

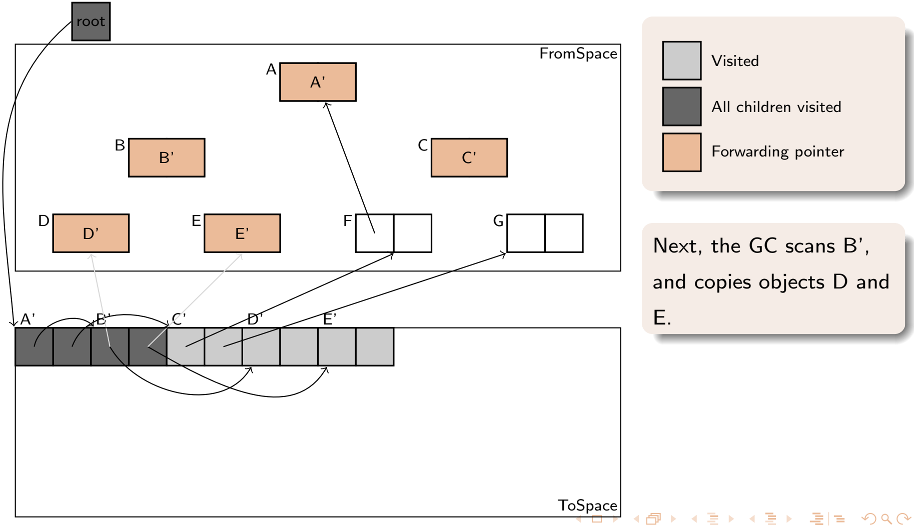
GC copies A to *ToSpace*, leaves a forwarding pointer pointing to the new copy A'.

Semi-space copying collector



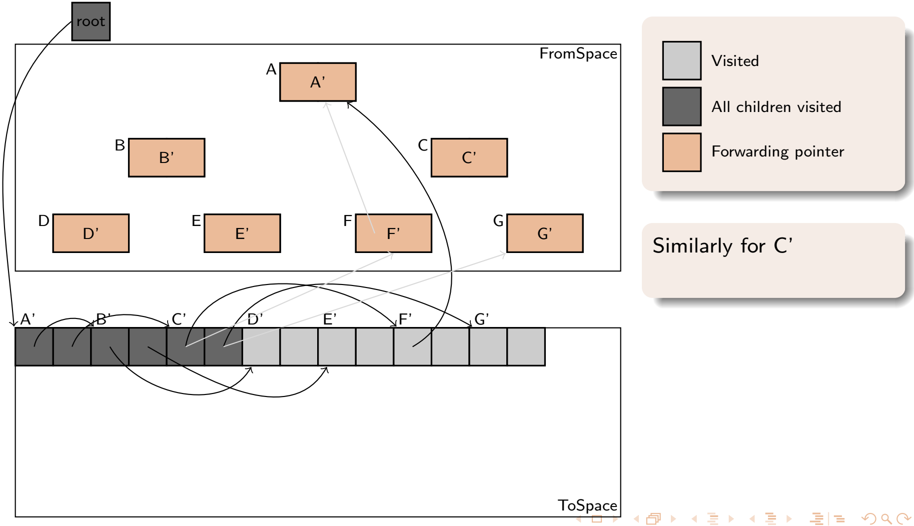
GC scans A'. The objects pointed to by A' (B and C) are copied to *ToSpace*. A's fields point to the copied objects.

Semi-space copying collector



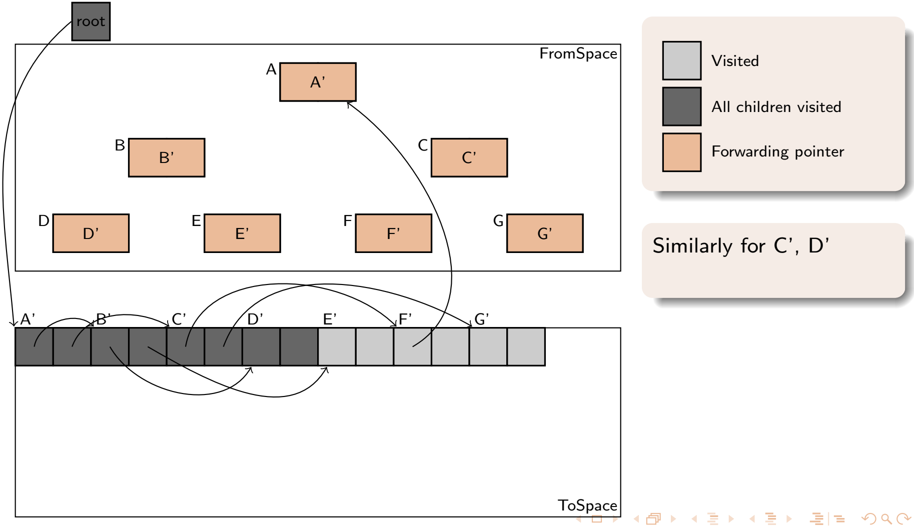
Next, the GC scans B', and copies objects D and E.

Semi-space copying collector



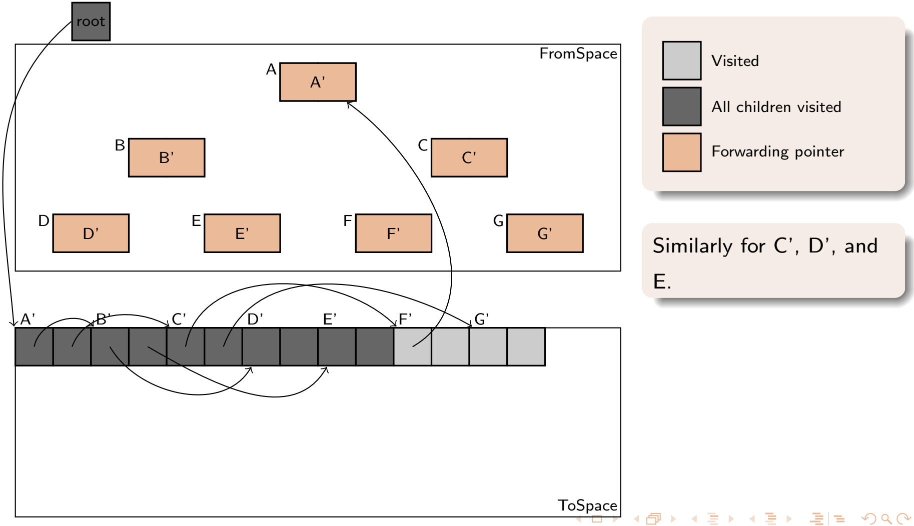
Similarly for C'

Semi-space copying collector



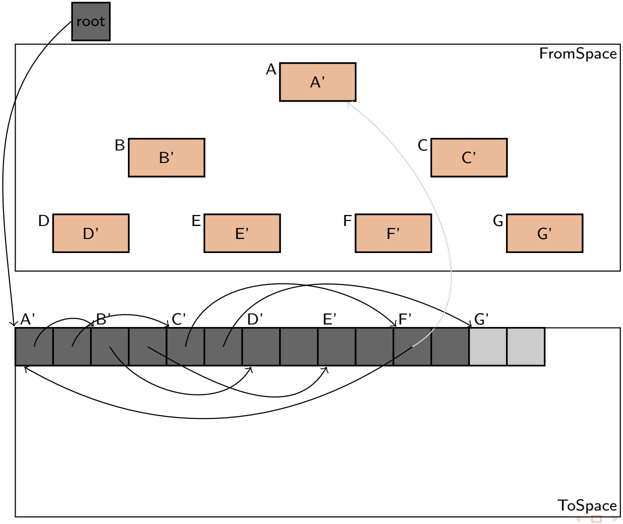
Similarly for C', D'

Semi-space copying collector



Similarly for C', D', and E.

Semi-space copying collector



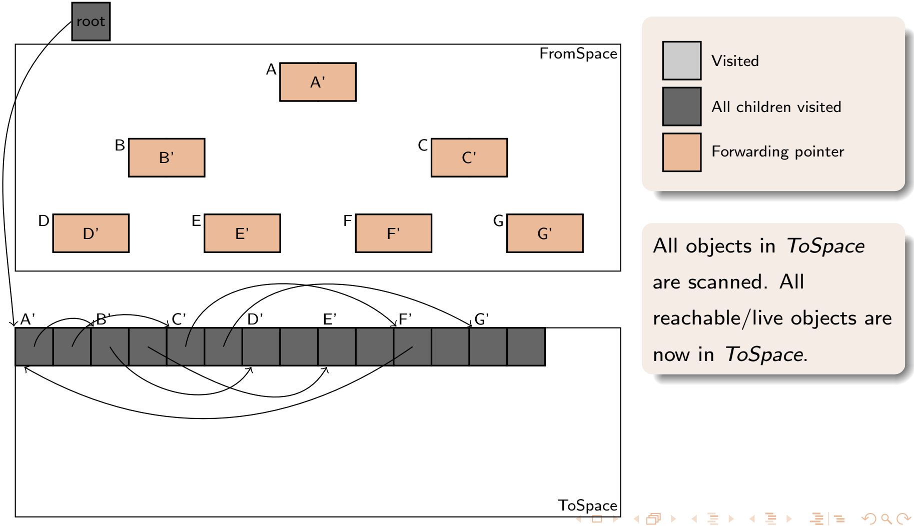
Visited

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



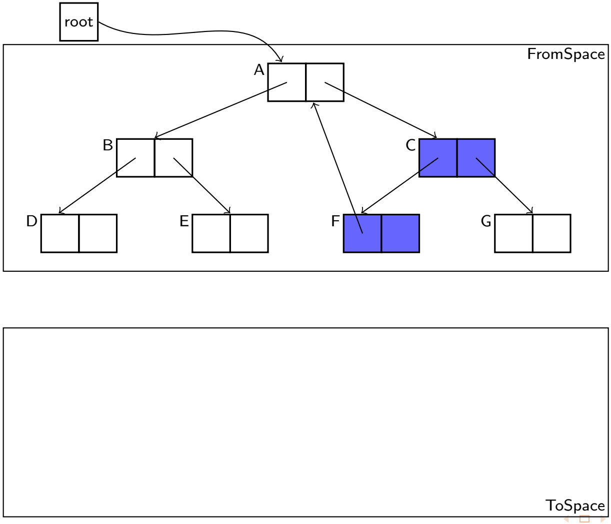
All objects in *ToSpace* are scanned. All reachable/live objects are now in *ToSpace*.

JVOLVE Garbage collector

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

- 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

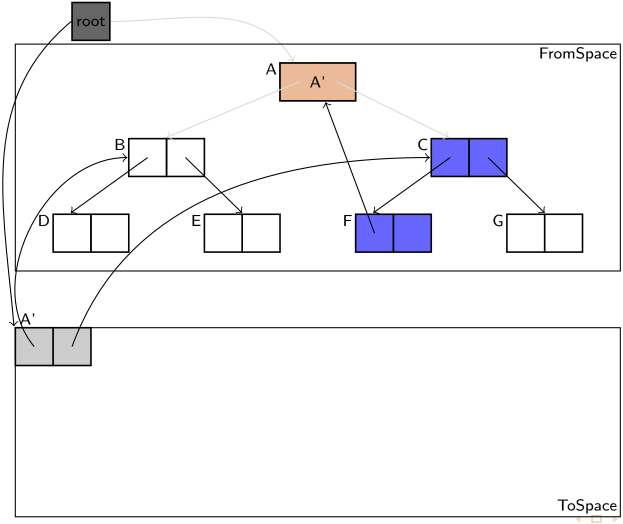


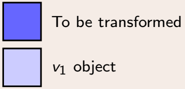
To be transformed

v_1 object

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

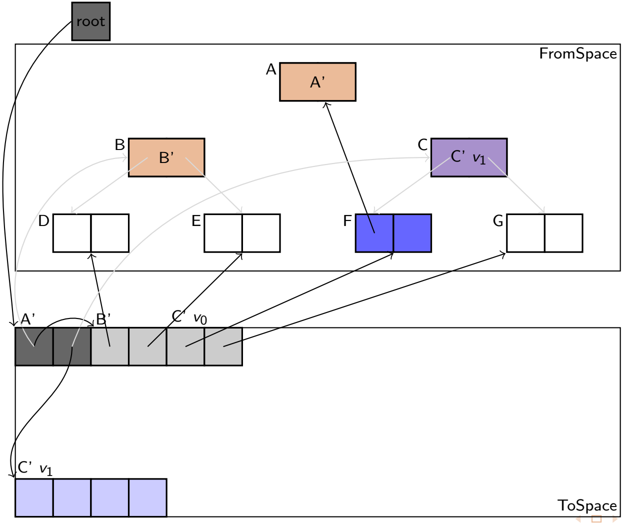
JVOLVE garbage collector

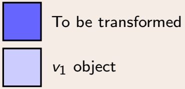




Copy A.

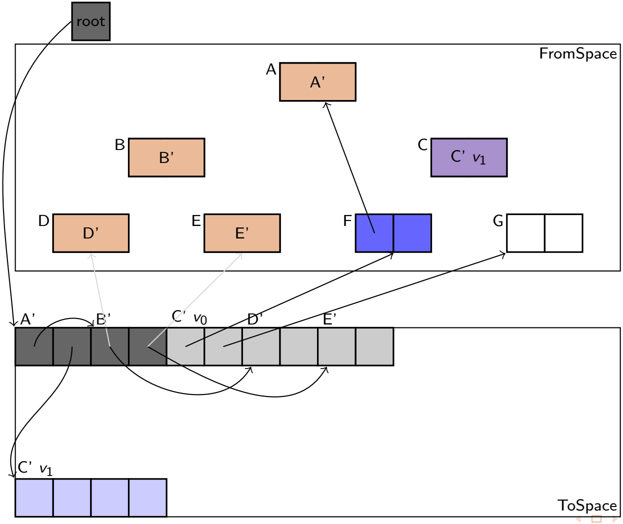
JVOLVE garbage collector

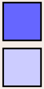




Scan A'. Copy B and C. In addition an empty object C'v_1 is allocated. A' points to this copy and not the old one.

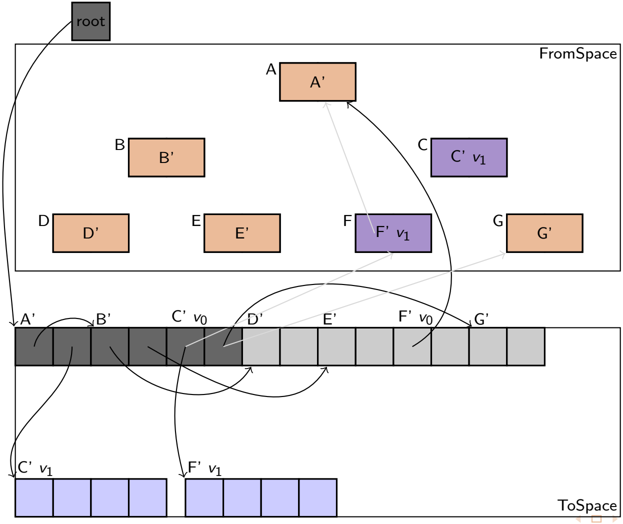
JVOLVE garbage collector

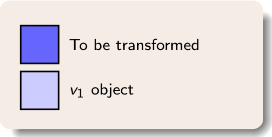




Scan B'.

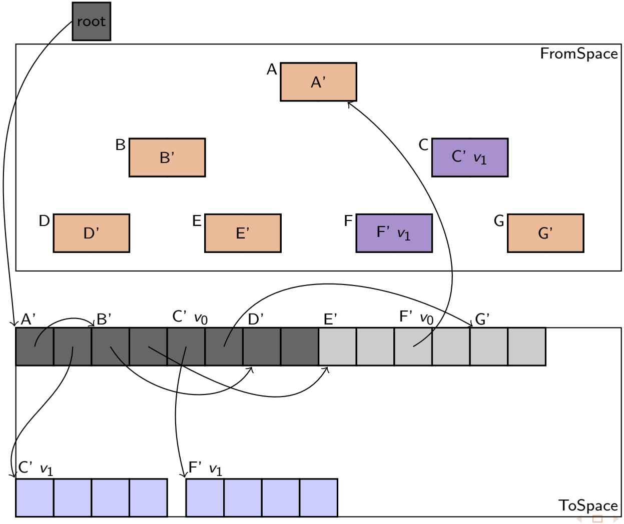
JVOLVE garbage collector

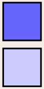




Scan C'.

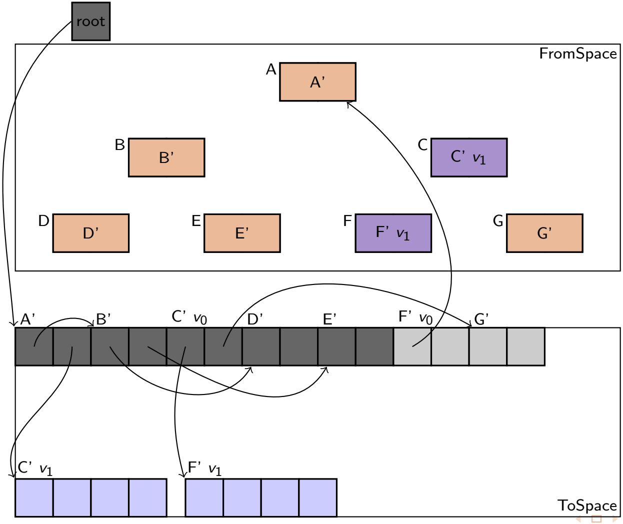
JVOLVE garbage collector

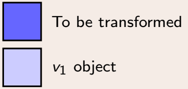




Scan D'.

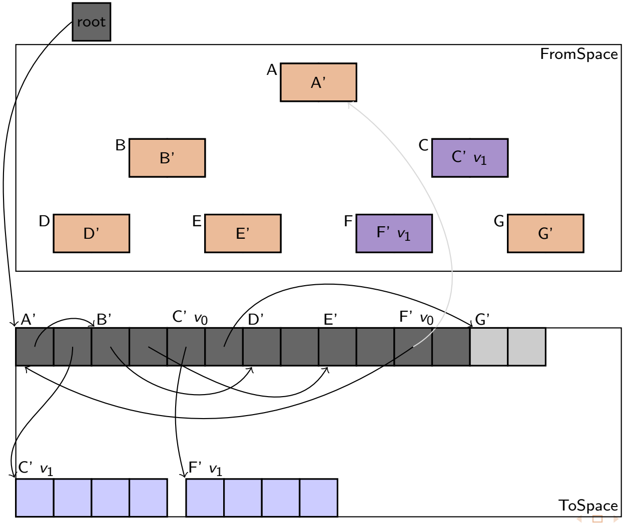
JVOLVE garbage collector





Scan E'.

JVOLVE garbage collector

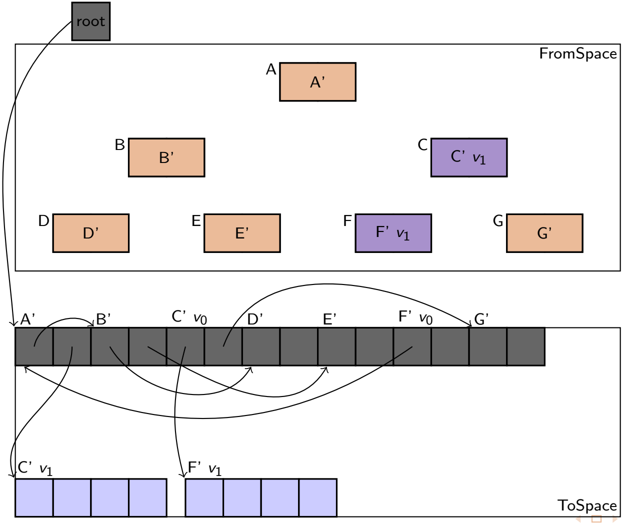


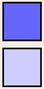
To be transformed

v_1 object

Scan F'.

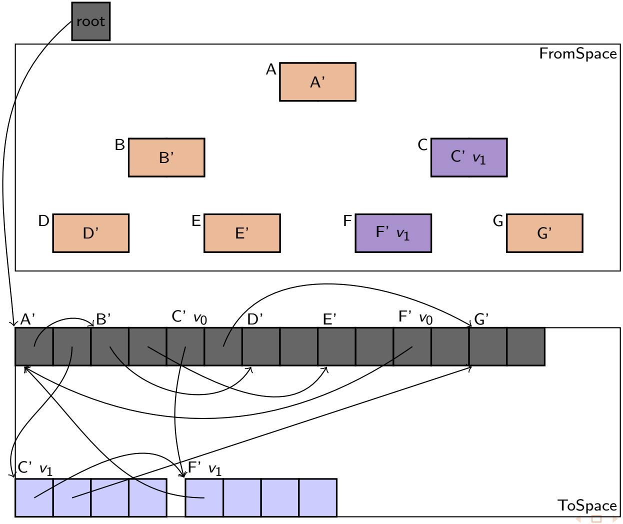
JVOLVE garbage collector





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

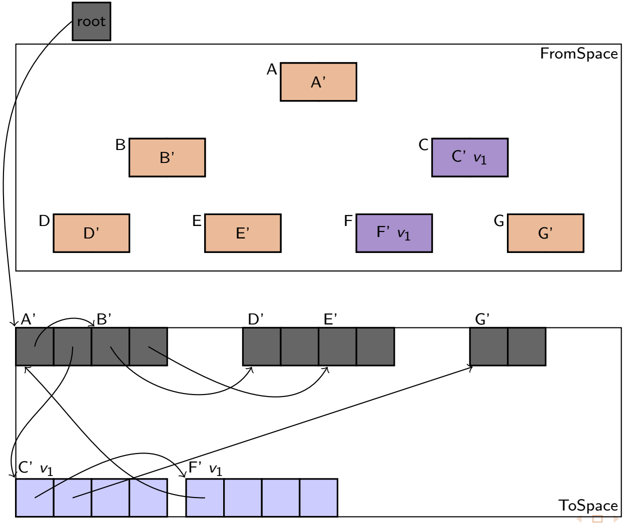


To be transformed

v_1 object

Setting fields of C'v_1 and F'v_1.

JVOLVE garbage collector

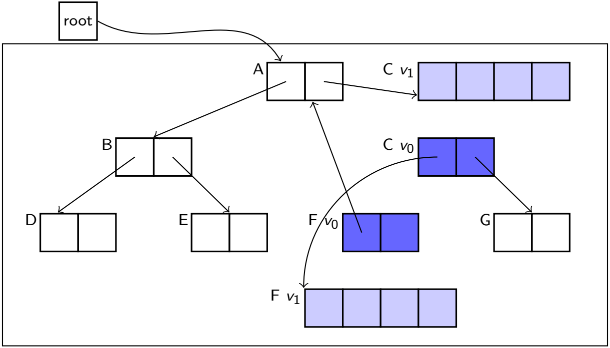


To be transformed

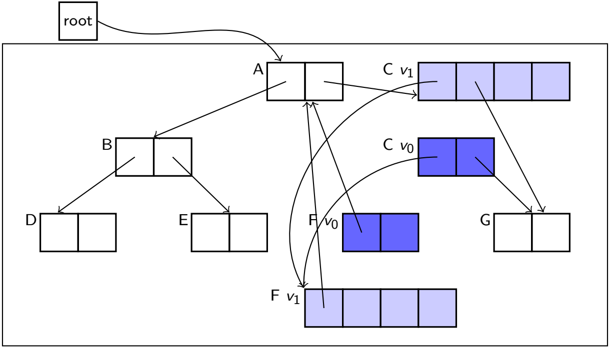
v_1 object

C'v_0 and F'v_0 can be reclaimed.

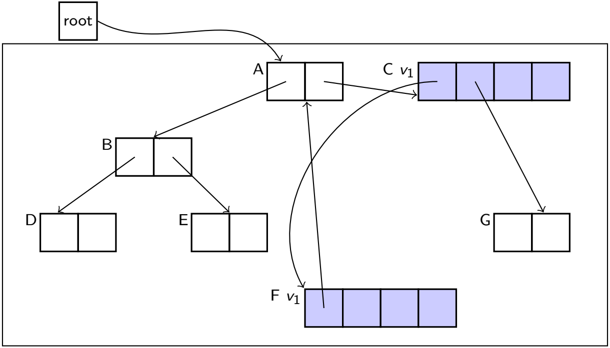
JVOLVE Garbage collector



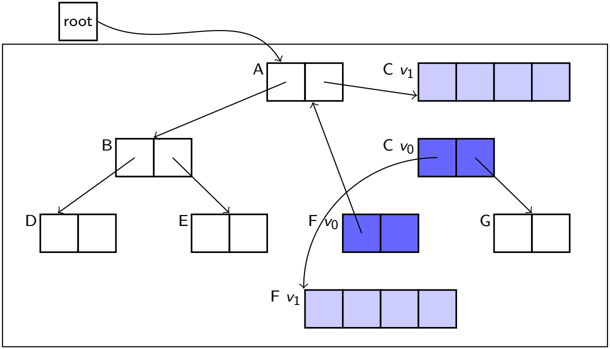
JVOLVE Garbage collector



JVOLVE Garbage collector



Revisiting transformation functions



We have an ordering problem

(C v_0).field0.field0 might be uninitialized

Revisiting transformation functions

Offline tool → Suspend application → Load new classes → **Transform objects** → Resume application

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

Proposed work

- Improving flexibility: On-stack replacement (OSR) support
- Improving efficiency: Concurrent collector support

Extending On-stack replacement (OSR)

- Some updates cannot be performed because the VM does not reach a DSU safe point
- Jikes RVM employs OSR to promote long running methods for optimization
  - Extract compiler-independent state from an activation record
  - Generate a *specialized prologue* that sets up local variables
  - Jump to corresponding program counter in optimized code
- We can utilize this functionality taking into account old and new versions

OSR issues

- What types of updates can benefit from OSR?
- How does OSR know where to resume execution?
- What about new local variables and those that need to be transformed?
- OSR in Jikes RVM can only replace the topmost method on stack.
  - Implement "return barriers"
  - Overwrite return addresses and jump to VM code that will perform OSR for the current top method
  - Some methods might be long running and always belong to some old version

JVOLVE efficiency

- JVOLVE requires a stop-the-world full-heap GC
- Update time is dominated by GC time
- Real-time and highly available applications use a concurrent GC

JVOLVE with a concurrent collector

- Application and collector run concurrently
- Guard application from accessing an object of the old version
- Collectors already use read/write barriers to guard the application from disrupting the tricolor abstraction. Piggyback on these barriers for DSU
- What is the additional overhead?
- How flexible can object transformers be?

Outline

- Introduction
  - Motivation
  - Solutions
  - Thesis
- 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

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

Jetty webserver: Reaching a DSU safe-point

Upd. to ver.	Reached safe point?	Number of methods at runtime	# methods not allowed on stack, due to *class updates*	*method body updates*	*indirect method updates*	Total	Number of restricted methods
5.1.1	always	1378 (376)	26/49	7/12	20/29	53/90 (17)	67
5.1.2	4/5†	1374 (375)	25/25	3/5	35/43	63/73 (35)	67
5.1.3	0/5*	1374 (375)	326/382	4/6	42/45	370/433 (97)	373
5.1.4	always	1384 (374)	82/82	5/6	15/16	101/104 (24)	101
5.1.5	always	1380 (372)	14/80	39/60	13/15	62/155 (17)	62
5.1.6	3/5†	1394 (378)	203/219	3/3	16/19	222/241 (40)	223
5.1.7	always	1394 (380)	186/187	1/2	53/69	239/258 (74)	243
5.1.8	always	1402 (379)	0/0	1/1	0/0	1/1 (1)	1
5.1.9	always	1402 (379)	0/0	0/1	0/0	0/1 (0)	0
5.1.10	always	1402 (379)	0/0	4/5	0/0	4/5 (2)	6

† Restricted method `HttpConnection.handleNext()` was active
* Restricted method `ThreadedServer.acceptSocket()` was (always) active

We propose to extend On-stack replacement to support changes to active methods on stack

Jetty webserver: Reaching a DSU safe-point

Upd. to ver.	Reached safe point?	Number of methods at runtime	# methods not allowed on stack, due to *class updates*	*method body updates*	*indirect method updates*	Total	Number of restricted methods
5.1.1	always	1378 (376)	26/49	7/12	20/29	53/90 (17)	67
5.1.2	4/5†	1374 (375)	25/25	3/5	35/43	63/73 (35)	67
5.1.3	0/5*	1374 (375)	326/382	4/6	42/45	370/433 (97)	373
5.1.4	always	1384 (374)	82/82	5/6	15/16	101/104 (24)	101
5.1.5	always	1380 (372)	14/80	39/60	13/15	62/155 (17)	62
5.1.6	3/5†	1394 (378)	203/219	3/3	16/19	222/241 (40)	223
5.1.7	always	1394 (380)	186/187	1/2	53/69	239/258 (74)	243
5.1.8	always	1402 (379)	0/0	1/1	0/0	1/1 (1)	1
5.1.9	always	1402 (379)	0/0	0/1	0/0	0/1 (0)	0
5.1.10	always	1402 (379)	0/0	4/5	0/0	4/5 (2)	6

† Restricted method `HttpConnection.handleNext()` was active
* Restricted method `ThreadedServer.acceptSocket()` was (always) active

We propose to extend On-stack replacement to support changes to active methods on stack

Overhead of DSU

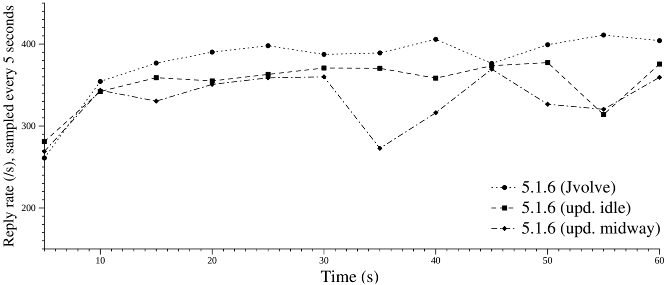
- No discernible overhead for normal execution (before and after the update)
- Only effect on execution time is the update pause time
  - Comparable to GC pause time

Jetty webserver performance

- Used `httperf` to issue requests
- Create 100 new connections/second (saturation rate)
- 5 serial requests to 10KB file per connection
- Compared versions 5.1.5 and 5.1.6
- Experiments on Dual-P4, 3 GHz, 2GB RAM, running JikesRVM 2.9.1 on Linux 2.6.22.8

Jetty webserver: Throughput measurements

Config.	Req. rate (/s)
5.1.5 (JVOLVE)	361.3 +/- 33.2
5.1.6 (Jikes RVM)	352.8 +/- 28.5
5.1.6 (JVOLVE)	366.2 +/- 26.0
5.1.6 (upd. idle)	357.4 +/- 34.9
5.1.6 (upd. midway)	357.5 +/- 41.6



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)

# objects	Fraction of Change objects		
	0.00	0.50	1.00
Total pause (ms)			
1000	381.21	391.21	400.77
10000	382.62	461.82	544.46
50000	382.73	779.67	2152.32
100000	383.64	1276.26	7903.21
Running transformation functions (ms)			
1000	0.22	4.62	8.89
10000	0.22	44.05	85.39
50000	0.22	215.94	1423.34
100000	0.22	511.02	6809.91
Garbage collection time (ms)			
1000	376.83	382.47	387.78
10000	378.25	413.64	454.95
50000	378.36	559.62	723.43
100000	379.29	756.01	1089.16

Outline

- Introduction
  - Motivation
  - Solutions
  - Thesis
- 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.

Schedule

- Proof-of-concept implementation that supports two years worth of updates
- On-stack replacement support (by September), a PLDI submission (November)
- DSU in a concurrent collector (January 2009 - May 2009)
- More applications (Summer 2009)
- Dissertation (Fall 2009)

Schedule

- Proof-of-concept implementation that supports two years worth of updates
- On-stack replacement support (by September), a PLDI submission (November)
- DSU in a concurrent collector (January 2009 - May 2009)
- More applications (Summer 2009)
- Dissertation (Fall 2009)

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

References II

ksplice (2008).
Ksplice: Rebootless Linux kernel security updates.
http://web.mit.edu/ksplice/.

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

Malabarba, S., Pandey, R., Gragg, J., Barr, E., and Barnes, J. F. (2000).
Runtime Support for Type-Safe Dynamic Java Classes.
In *Proc. ECOOP*.

References III

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*, pages 72–83.

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 *Java for Embedded Systems Workshop*, London.

References IV

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 ATC*.