Language Tools for Distributed Computing (II)

J-Orchestra:
Automatic Java Application Partitioning

Yannis Smaragdakis
University of Oregon
These Lectures

- NRMI: middleware offering a natural programming model for distributed computing
  - solves a long standing, well known open problem!

- J-Orchestra: execute unsuspecting programs over a network, using program rewriting
  - led to key enhancements of a major open source software project (JBoss)

- Morphing: a high-level language facility for safe program transformation
  - “bringing discipline to meta programming”
Partitioning: Start with a Centralized Application
Convert it to a Distributed Application

GUI

Network

Computation

DB
Automatic Program Partitioning

- How can we do this with tools instead of manually?
  - write a centralized program
  - select elements (at some granularity) and assign them to network locations
  - let an automatic tool (compiler) transform the program so that it runs over a network, using a general purpose run-time system
    - correctness and efficiency concerns addressed by compiler—though not always possible
J-Orchestra

- For the past 5 years, J-Orchestra has been one of my major research projects
  - an automatic partitioning system for Java
  - works as a bytecode compiler
  - think of result as “applets on steroids”
    - “code near resource”
J-Orchestra

- For the past 5 years, J-Orchestra has been one of my major research projects
  - an automatic partitioning system for Java
  - works as a bytecode compiler
  - think of result as “applets on steroids”
    - “code near resource”
J-Orchestra Executive Summary

- Partitioned program is *equivalent* to the original centralized program for a very large subset of Java.
  - we handle synchronization, all OO language features, object construction, ...
  - nice analysis and compilation technique for dealing with native code
  - result: *most scalable automatic partitioning system in existence*
  - have partitioned many unsuspecting applications
    - including 8MB third party bytecode only (JBits)
Example Partitioning
Example Partitioning

Network
Example Partitioning

Benefit: 3.4MB + 1.8MB + 3.5MB transfers eliminated for view updates!

Network

Benefit: 1.28MB vs, 1.68MB per simulation step!
J-Orchestra Techniques Summary

- Program generation and program transformation at the bytecode level
  - “virtualizing” execution through bytecode transformation
    - creating a “virtual” virtual machine
  - existing classes get transformed into RMI remote objects
  - client code is redirected through proxies
  - for each class, about 8 different proxy types (for mobility, access to native code, etc.) may need to be generated
  - user input is at class level, but how objects are passed around determines where code executes
J-Orchestra Program Transformation Techniques

Neo: Programs hacking programs. Why?

[Matrix Reloaded]
The Problem Technically

- Emulate a shared memory abstraction for unsuspecting applications without changing the runtime system.
  - Complicating assumption: a pointer-based language.
  - Resembles DSM but different in objectives.
    - DSM – distribution for parallelism.
    - Auto Partitioning – functional distribution.
The Approach: User Level Indirection

- We cannot change the VM to change the notion of “pointer”/“reference”
- Can we do it by careful rewriting of the entire program?
  - any reference, method call, etc. is through a proxy
    - where an original program reference would be to an object of type A, the same reference will now be to a proxy for As
  - For example:
    - “new A()” creates proxy for A instead of instance of original class A
    - a.field becomes a.getField() or a.putField()
User Indirection (Proxy) Approach

- All clients (aliases) should view the same object regardless of location
- Change all references from direct to indirect
The Proxy Approach

- Changing all references from direct to indirect ensures correct behavior in the presence of aliases.
- A remote object can have several proxies on different network sites.
The Proxy Approach

- Proxies hide the location of the actual object: objects can move at will to exploit locality
J-Orchestra Sample Transformations

For each original class A

- class A becomes a proxy
- Remote class A__remote
- Local class A__local
- Interface A__iface
- class A__static_delegator
- Interface A__static_iface
For each original class $A$:

```java
class A {
    java.io.File _file;

    public void foo(A p) {
        _file.read();
        p._file.read();
    }
}
```

A\_interface is generated:

```java
interface A\_iface
    extends java.rmi.Remote
{
    public void foo(A p) throws Remote Exception;
    public proxy.io.File get\_file() throws RemoteException;
}
```
Generated Code

For each original class A:

class A {
    java.io.File _file;

    public void foo(A p) {
        _file.read();
        p._file.read();
    }
}

proxy is generated:

class A {
    A__iface _ref;

    public void foo(A p) {
        _ref.foo(p);
    }
}
For each original class A:

```java
class A {
    java.io.File _file;

    public void foo(A p) {
        _file.read();
        p._file.read();
    }
}
```

class A is binary-modified:

```java
class A__remote extends UnicastRemoteObject implements A__iface {
    public proxy.java.io.File get_file() { return _file; }
}
```
Complexities

Overheads, Grouping Objects, System Code
## Proxy Indirection Overhead

<table>
<thead>
<tr>
<th>Work (test, multiply, increment)</th>
<th>Original Time</th>
<th>Rewritten Time</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>35.17s</td>
<td>47.52s</td>
<td>35%</td>
</tr>
<tr>
<td>4</td>
<td>42.06s</td>
<td>51.30s</td>
<td>22%</td>
</tr>
<tr>
<td>10</td>
<td>62.50s</td>
<td>73.32s</td>
<td>17%</td>
</tr>
</tbody>
</table>

- Micro benchmark
- A function of average work per method call
- 1 billion calls total
Optimizing Proxy Indirection
Optimizing Proxy Indirection

sensor

direct call

dot object

GUI

DB
Optimizing Proxy Indirection

sensor

GUI

DB

object
direct call
proxy
proxy call

Yannis Smaragdakis
UOregon
Optimizing Proxy Indirection

- Object
- Direct call
- Proxy
- Proxy call
- Mobile object
- Opt. proxy call

Diagram showing the relationships between sensor, GUI, DB, and objects with direct and proxy calls.
Optimizing Proxy Indirection

- sensor
- GUI
- DB
- object
- direct call
- proxy
- proxy call
- mobile object
- opt. proxy call
How is This Implemented?

- Two kinds of references: direct and indirect
- Direct: for code statically guaranteed to refer to the object itself
  - i.e., object on the same site
- Indirect: maybe we are calling a method on the object, maybe on a proxy
System Code

- The same idea applies to dealing with system classes
  - system classes are split in groups
    - we assume that groups are consistent with what native code does (more later)
  - code accesses objects in the same group directly
  - other objects accessed indirectly
Wrapping / Unwrapping

- For this approach to work, we need to inject code in many places to convert direct references to indirect and vice-versa
  - dynamic “wrapping/unwrapping”
  - code injected at compile time, wrapping/unwrapping takes place at run time
Example: Pass a Reference to System Code

- What if a system object is passed from user code to system code?

```javascript
{ window.add(button); }
```

![Diagram showing interactions between objects and network](image-url)
Wrapping/Unwrapping at the Proxy

- The easy case: callee can tell wrapping is needed
  - applies to system code

```
foo (Proxy_Of_p); //unwrap
proxy_Of_p = bar (); //wrap
```

Diagram:

- **Proxy_Of_p**
- **Stub_Of_p**
- **Caller** (Mobile code)
- **Callee** (Anchored code)
Wrapping/Unwrapping at Call Site

- The harder case: sometimes we need to wrap/unwrap at call site
  - either to keep proxy simple, or because we’d end up with overloaded methods only differing in return type
    - a problem since our proxies are generated in source, although the rest of the transforms are in bytecode
  - need to reconstruct call stack, inject code
Example: “this”

//original code
class A { void foo (B b) { b.baz (this); } }
class B { void baz (A a) {...} }

//generated remote object for A
class A__remote {
    void foo (B b) { b.baz (this); }  //”this” is of type A__remote!
}

//rewritten bytecode for foo
aload_0                               //pass “this” to locateProxy method
invokestatic Runtime.locateProxy
checkcast “A”                          //locateProxy returns Object, need a cast to “A”
astore_2                               //store the located proxy object for future use
aload_1                                //load b
aload_2                                //load proxy (of type A)
invokevirtual B.baz
“How Do You Handle...?”

Native code, Synchronization
Handling Java Language Features

- Many language features need explicit handling, but most complexities are just engineering
  - static methods and fields
  - inheritance hierarchies
  - remote object creation
  - inner classes
  - System.in, System.out, System.exit, System.properties

- Some require more thought
  - native code
  - synchronization
Native Code

- Recall how we split system classes into groups
- These groups have to respect native code behavior
- But we don’t know what native code does!
- The problem: we may let a proxy escape into native code, and the native code will try to access it directly
  - e.g., read fields from the original object
Heuristic Type-Based Analysis: Group Based on Types

- class `C` extends `S` {
  `F f;`
  `public native R meth ( A a);`
}

- Conservative, but still not safe
  - nothing can be!
  - type information can be disguised at the native code interface level
    - i.e., native code can do type casts
How Safe?

- Studied native code in JDK 1.4.2 for Solaris
- Two analyses:
  - 13 applications, dynamic analysis of execution
  - code inspection of native code for **Object**, **IsInstanceOf**
- Overall, fairly safe—few violations
  - PlainSocketImp.socketGetOption casts Object to InetAddress
  - GlyphVector assumed to be StandardGlyphVector, Composite assumed to be AlphaComposite
- native code respects types more than library code!
  - JNI **IsInstanceOf** : 69 occurrences
  - Java **instanceof** : 5900 occurrences
- In practice, J-Orchestra works without (much) intervention
Synchronization

- We only handle monitor-style synchronization: synchronized blocks and methods, wait/notify/notifyAll
  - not volatile variables, concurrent data structures, atomic operations, etc.

- Two problems:
  - thread identity is not maintained over the network
  - synchronization operations (synchronized, wait, notify, etc.) do not get propagated by RMI
Thread Identity Is Not Maintained
(The Zigzag Deadlock Problem)

synchronized void foo()
{
    obj2.bar();
}

synchronized void baz()
{
    ...
}

void bar()
{
    thread-1
}

obj1

thread-1

obj2

obj1.baz();
Thread Identity Is Not Maintained
(The Zigzag Deadlock Problem)

synchronized void foo() {
    obj2.bar();
}
synchronized void baz() {
    ...
}
Thread Identity Is Not Maintained
(The Zigzag Deadlock Problem)

```java
synchronized void foo() {
    obj2.bar();
}

synchronized void baz() {
    ...;
}

void bar() {
    obj1.baz();
}
```
Synchronization Operations Don’t Get Propagated Over the Network

- **obj** – a remote object, implementing interface $RI$ and remotely accessible through it
- **$RI$ ri** – points to a local RMI “stub” object
- `ri.foo();` //will be invoked on **obj** on a remote machine
- The stub serves as an intermediary, propagating method calls to the **obj** object
- Only *synchronized* methods are propagated correctly
- *Synchronized* blocks might not work correctly
Synchronized Blocks

- Even if `obj` and `ri` point to the same object, synchronization will be on stub vs. true object.
Synchronization Operations Don’t Get Propagated Over the Network

- Monitor operations: `Object.wait`, `Object.notify`, `Object.notifyAll` don’t work correctly
- They are declared `final` in class `Object` and cannot be overridden in subclasses
- Calling any of them on an RMI stub does not get propagated over the network
J-Orchestra Synchronization

- Maintain per-site “thread id equivalence classes”
- Replace all the standard synchronization constructs (\texttt{monitorenter, Object.wait, Object.notify}) with the corresponding calls to a per-site synchronization library
Thread Identity Is Not Maintained
(The Zigzag Deadlock Problem)

synchronized void foo() {
    obj2.bar();
    {thread-1}
} {thread-1, thread-2, thread-3}

synchronized void baz() {
    {thread-1, thread-2}
    obj1.baz();
    {thread-1, thread-2}
}
Maintaining Thread Id Equivalence Classes \textit{Efficiently}

- Updating thread equivalence classes \textit{only} when the execution of a program crosses the network boundary
- This happens only after it enters a method in an RMI stub
- Use bytecode instrumentation on standard RMI stubs
- Equivalence classes’ representation is very compact (encoded into a \textit{long int}). Imposes virtually no overhead on remote calls
A Specialized Application

“Appletizing”
Java Applets

- Execute on the client.
- Transfer all code to client.
- Provide “sandbox” secure execution environment.
Java Servlets

- Execute on the server.
- Thin GUI through Web Forms.
Appletizing

- A hybrid between Applets and Servlets.
- Rich GUI client; full access to server resources.
- Safe and secure execution model.
- Ease of development and deployment.
Sanitizing GUI Code

- Some code inside GUI classes is rejected by the Applet Security Manager.
- E.g., `System.exit`, read/write graphical files from the local hard drive, closing a frame.
- Two approaches to replacing unsafe code:
  1. With different code.
  2. With semantically similar (identical) code.
Sanitizing: Reading Image From File

//Creates an ImageIcon from
//the specified file
//will cause a security exception when
//a file on disk is accessed

javax.swing.ImageIcon icon =
    new javax.swing.ImageIcon ("AnIconFile.gif");
Sanitizing: Reading Image From File

//Sanitize by replacing with the
//following safe code

javax.swing.ImageIcon icon =
    new jorch.rt.ImageIcon("AnIconFile.gif");

//will safely read the image from
//the applet's jar file
Sanitizing: JFrame.setDefaultCloseOperation

- Method `setDefaultCloseOperation` in system class `javax.swing.JFrame`.

- Applet Security Manager prevents it from taking `EXIT_ON_CLOSE` parameter.

```java
invokevirtual
    JFrame.setDefaultCloseOperation
```
Sanitizing:

```java
JFrame.setDefaultCloseOperation

pop //pop value on top of the stack

push 0 //param 0 is DO NOTHING ON CLOSE

invokevirtual
   JFrame.setDefaultCloseOperation
```

Yannis Smaragdakis
UOregon

60
Wrap up
J-Orchestra Impact

- Although the J-Orchestra work is well-cited, its greatest impact was unconventional
  - in late 2002, we gave a demo to Marc Fleury, head of the JBoss Group
    - JBoss: probably the world’s most popular J2EE Application Server—millions of downloads (open source)
    - Application Server: OS for server-side computing
      - handles persistence, communication, authentication, ...
      - imagine a web store, bank, auction site, etc.
  - great excitement about using bytecode engineering to generate and transform code, to turn Java classes into EJBs
  - J2EE middleware has strict conventions (e.g., “each session bean needs to implement local and remote interfaces, such that…”)
Program Transformation and Generation in JBoss

- JBoss engineers had little expertise
  - my M.Sc. student Austin Chau did the first implementation
  - we fixed the bytecode generation platform (Javassist)
  - JBoss contributors then took over
- Radical innovation in version 4: can use plain Java objects as Enterprise Java Beans
  - a general mechanism: “Aspect-Oriented Programming in JBoss”
  - JBoss can now produce automatically much of the tedious J2EE code
    - given plain Java code (together with user annotations)
    - annotation mechanism in Java 5 largely motivated by program generation tasks for J2EE code
Publications

- Main paper: ECOOP’02
- Synchronization: Middleware ’04
- Appletizing: ICSM’05
- Dealing with native code: ECOOP’02 + GPCE’06