A Distributed Runtime for Java: Yesterday and Today

Combining transparency with portability

Michael Factor
Assaf Schuster
Konstantin Shagin

IBM
Technion
Technion
Outline

- Background and motivation
- Existing systems
  - Cluster-aware virtual machines
  - Compiler-based DSM systems
  - Systems using standard JVMs
- The JavaSplit runtime
  - Features
  - Implementation overview
  - Performance evaluation
Background and Motivation

- Need a convenient programming paradigm for distributed systems
  - The average programmer tends to prefer shared memory over message passing
- Java is popular, comfortable, and “well defined”
  - Can extend it with libraries for efficient support for parallel computing (e.g., barriers)
- Servers are written in Java (development cycle! Portability! Efficiency?)
- Existing base of multithreaded code

The goal: create a runtime environment for distributed execution of multithreaded Java programs Portability, Transparency, Efficiency.
1. Distributed nonstandard JVMs

- Require that each node install a custom JVM
  - Not portable
  - Cannot use a native just-in-time compiler (JIT)
- Can access the system resources, *e.g.*, memory and network interface directly
- Since there is no need to preserve portability, allow use of nonstandard but more efficient networking hardware

  - Java/DSM (1997)
  - Cluster VM for Java (formerly cJVM) (1999)
  - JESSICA (1999)
2. Compiler-based DSM systems

- Translate Java sources or bytecode to machine dependent assembler while adding DSM capabilities
  - Do not need a JIT
- The use of a dedicated compiler allows performing various compiler optimizations
  - Access check elimination and batching
- Have the same portability issues as cluster-aware JVMs
  - Hyperion (2001)
  - Jackal (2001)
3. Systems using standard JVMs

- Each node carries out its part of the execution using a standard JVM
  - Portable across any Java-enabled platform
  - Can use a native JIT
  - Given certain security permissions, nodes can join the system using a Java-enabled browser
- Access machine resources through the JVM
- Not transparent
  - Introduce unorthodox programming constructs and style: JSDM 2001 (SPMD), JavaParty 1997 (remote, RMI hooks)
  - Require non-standard libraries: ProActive 1998
  - Need user assistance: Addistant 2001 (class-based distribution, non/modifiable), jOrchestra 2002 (user-defined class partition, must co-locate a system class with its referenced code)
The JavaSplit runtime

- **Combines transparency with portability**
- **Executes standard** Java applications
  - Can automatically execute preexisting applications
    - No nonstandard libraries
    - No unconventional programming constructs or style
  - Does not require user hints
- **Uses standard JVMs and therefore:**
  - Portable across any Java-enabled platform
  - Can use a native JIT
  - Given certain security permissions, nodes can join the system using a Java-enabled browser
- **Designed to efficiently support a large number of nodes**
Employment of JavaSplit

- High-performance computing
- Rapid development of low-priced distributed systems composed of commodity hardware
- Cycle stealing in large non-dedicated environments
  - Need to augment the system with fault tolerance
Java Basics

Source code

```java
class A {
    ...
    ...
}
```

Bytecode

Machine code

Java Virtual Machine

System classes

Just in Time compiler
JavaSplit Overview

- Rewriting intercepts synchronization, accesses to shared data etc.
- Threads and objects are distributed among the machines
class Producer extends Thread {
    public int field = 0;

    public void run() {
        synchronized(this) {
            field = 1;
        }
    }
}

class Consumer extends Thread {
    public Producer p;

    public void run() {
        synchronized(p) {
            System.out.println(p.field);
        }
    }
}
Distributed Shared Memory (DSM)

- Object-based
  - More suitable for Java
  - Few false-sharing
- Designed to be scalable
  - No global operations
  - Allows multiple simultaneous writers
- Implements Lazy Release Consistency
  - Consistent with the proposed revisions to the Java Memory Model (JMM)
- Only objects accessed by more than one thread are managed by the DSM
  - Detected at runtime
Treatment of system classes

- jOrchestra and Addistant treat system classes as *unmodifiable code*
- This restricts the placement of data
- requires *user intervention* to find a correct and efficient placement

```java
java.util.Calendar

public Object clone() {
    ...
    other.zone = (java.util.TimeZone) zone.clone();
    return other;
}
```

The code of Calendar cannot be updated to access the distribution-aware proxy of the *zone* object, therefore the zone object and the calendar instance pointing to it must be located on the same node.
Bytecode Instrumentation

- JavaSplit instruments all classes used by the original application, including system classes
- To instrument system classes we use a novel technique, called Twin Class Hierarchy (TCH)
  - All instrumented classes are renamed
  - In a rewritten class, all referenced class names are replaced with the new names
  - All rewritten system classes become user classes, which are much easier to instrument
class A extends somepackage.C {
    // fields
    private int myIntField;
    public B myRefField;
    public java.util.Vector myVectorField;
    // methods
    protected void doSomething(B b, int n) {
        if(b instanceof java.util.List) { ... }
        java.lang.Class vecClass = java.lang.Class.forName("java.util.Vector");
        ...}
    
    public B doSomethingElse(java.lang.String str) { 
        java.lang.System.out.println(str);
        java.io.File f = new java.io.File(str);
        ...}
    }
}
class JS.A extends JS.somepackage.C {
    // fields
    private int myIntField;
    public JS.B myRefField;
    public JS.java.util.Vector myVectorField;
    // methods
    protected void doSomething(JS.B b, int n) {
        if(b instanceof JS.java.util.List){ ... }
        JS.java.lang.Class vecClass =
            JS.java.lang.Class.forName("java.util.Vector");
        ...
    }
    public JS.B doSomethingElse(JS.java.lang.String str) {
        JS.java.lang.System.out.println(str);
        JS.java.io.File f = new JS.java.io.File(str);
        ...
    }
}
Implementing native methods

**JS.java.lang.System**

```java
public static long currentTimeMillis()
{
    return java.lang.System.currentTimeMillis();
}
```

**JS.java.net.Inet4AddressImpl**

```java
public JS.java.lang.String getLocalHostName() {
    // origImpl_ is a private field of type java.net.Inet4AddressImpl
    java.lang.String name = origImpl_.getLocalHostName();
    // convert the name into a JS.java.lang.String and return it
    return JS.java.lang.String.__JS__convertFromJavaString(name);
}
```
Implementing native methods (2)

**JS.java.math.StrictMath**

```java
public static strictfp double sqrt(double arg){
    return java.lang.StrictMath.sqrt(arg);
}
```

**JS.java.lang.Inet4AddressImpl**

```java
public boolean isPrimitive(){
    // origImpl_ is a private field of type java.lang.Class
    return origImpl_.isPrimitive();
}
```
Performance (2)

Series

Execution time

Speedup

Number of CPUs

IBM JDK 1.3.0
Sun JDK 1.4.2

JavaSplit Sun
JavaSplit IBM
Original Sun
Original IBM
Sun Speedup
IBM Speedup

IBM JDK 1.3.0
Sun JDK 1.4.2
Performance (3)

RayTracer

Number of CPUs

Execution time

Speedup

- JavaSplit Sun
- JavaSplit IBM
- Original Sun
- Original IBM
- Sun Speedup
- IBM Speedup

IBM JDK 1.3.0
Sun JDK 1.4.2
Conclusion

- JavaSplit is a first step towards creating a convenient and portable infrastructure for distributed computing
  - Provides shared memory abstraction
  - Can be used by any Java developer
  - Any platform with a JVM can participate
- Achieves scalable speedups executing computation-intensive applications
  - Despite relatively slow access to the network
  - With few simple optimizations
The End

Questions?
● Classes are augmented with utility fields and methods
● The fields indicate the state of an object
  – Inserted at the topmost hierarchy classes
  – The state data can be quickly accessed and easily disposed.
● Implementation of the utility methods is class-specific
  – Perform the same operation on each field of the class
## Read Access Check Example

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALOAD 1</td>
<td>// load instance of A</td>
</tr>
<tr>
<td>DUP</td>
<td>// duplicate instance of A on stack</td>
</tr>
<tr>
<td>GETFIELD</td>
<td>A::byte <strong>JS__state</strong></td>
</tr>
<tr>
<td>IFNE</td>
<td>// jump if the state allows reading</td>
</tr>
<tr>
<td>DUP</td>
<td>// duplicate instance of A on stack</td>
</tr>
<tr>
<td>INVOKESTATIC</td>
<td>JS.Handler::readMiss</td>
</tr>
<tr>
<td>GETFIELD</td>
<td>A::intField</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Access Check Elimination

A aObject = new A();
...

<WRITE ACCESS CHECK of aObject>
aObject.intField = 2003;
... // no lock acquires

<READ ACCESS CHECK of aObject>
for(int k=0; k<N; k++){
    ... // no lock acquires
    <READ ACCESS CHECK of aObject>
    System.out.println(k+aObject.intField);
    ... // no lock acquires
}


Efficient Synchronization

- Java applications contain a great amount of unneeded synchronization [Choi et. al., OOPSLA’99]
  - May cause significant performance degradation in instrumented classes
- We distinguish between synchronization operations on local and shared objects
  - Lightweight synchronization for local objects (similar to [Bacon et. al., PLDI’98])
  - Synchronization of local objects is cheaper than in original Java

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Local</th>
<th>Shared</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sun 1.4.1</td>
<td>90.6</td>
<td>19.6</td>
<td>281</td>
</tr>
<tr>
<td>IBM 1.3.0</td>
<td>93.4</td>
<td>54.7</td>
<td>327</td>
</tr>
</tbody>
</table>
Classification of existing systems

1. Distributed nonstandard JVMs

2. Compilation to native code combined with a distributed shared memory

3. Systems built on top of standard JVMs