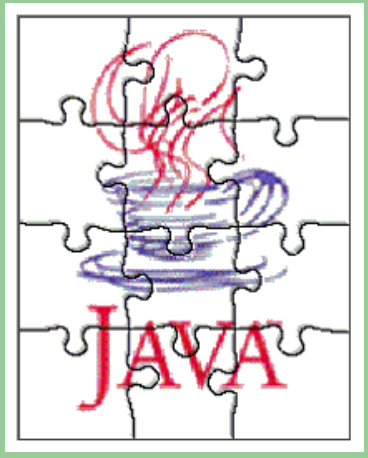
JavaSplit

A Portable Distributed Runtime for Java



Michael Factor
Assaf Schuster
Konstantin Shagin

Motivation

- Interconnected commodity workstations can yield a cost-effective substitute to super-computers.
  - Suitable for computation intensive applications
- Need a comfortable programming paradigm
  - Message passing programming is complex
  - Shared memory abstraction is desirable
- Java is popular and comfortable
  - Supports multithreading and synchronization

The Goal

Create a runtime environment that utilizes a set of interconnected machines to execute a standard multithreaded Java application

- Required features:
  - Transparency (implied by the goal)
  - Cross-platform portability
  - Scalability

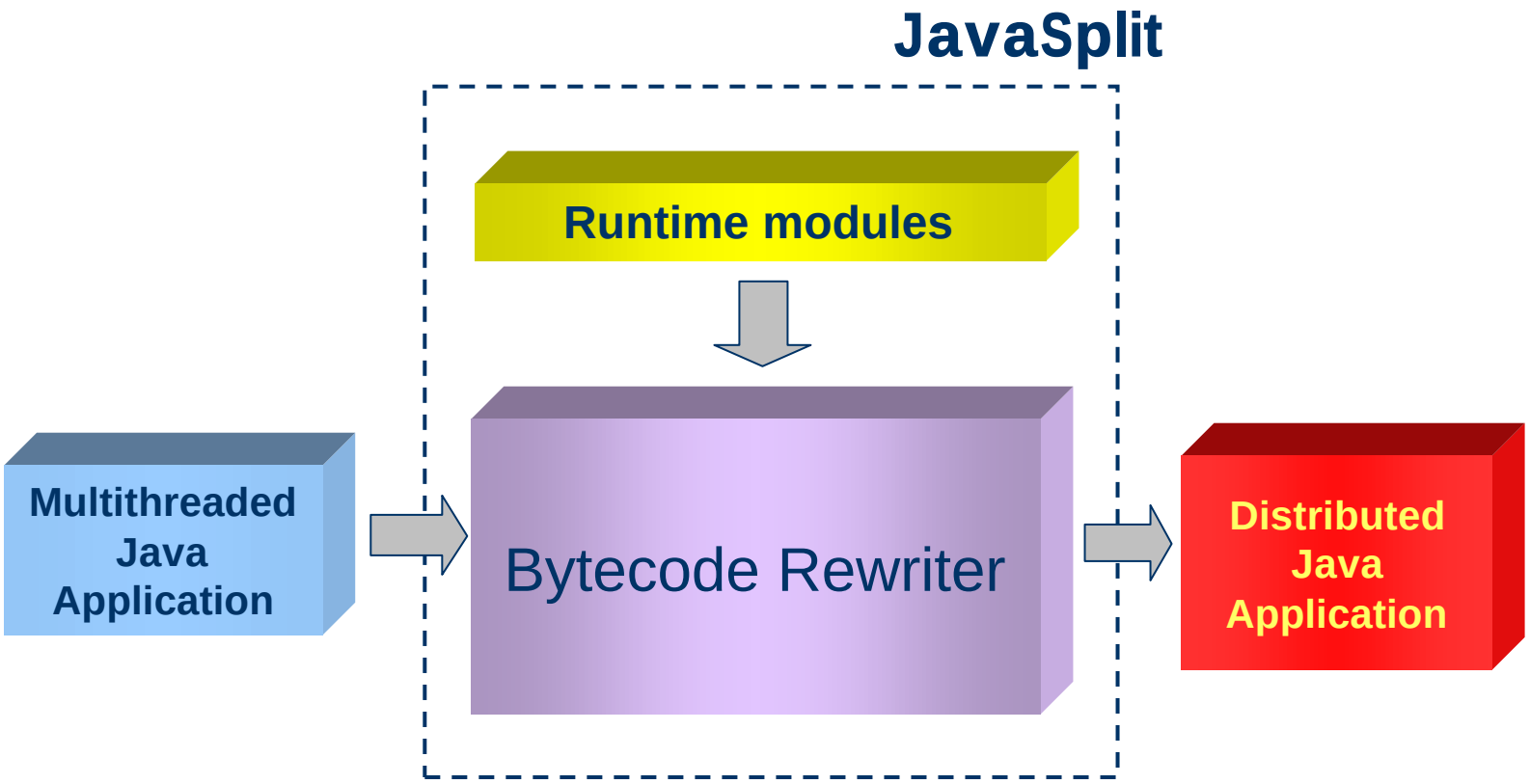
Related Work

- Distributed (non standard) JVMs
  - Java/DSM (1997), Cluster VM for Java (former cJVM) (1999), JESSICA (1999)
  - Sacrifice portability
- Translation to native code combined with a DSM
  - Hyperion (2001), Jackal (2001)
  - Also not portable
  - Perform compiler optimizations
- Systems built on top of standard JVMs
  - JavaParty(1997), ProActive (1998), JSDM (2001)
  - Introduce unorthodox programming constructs and style

Our Approach - JavaSplit

- Works through bytecode instrumentation
  - The bytecodes of the application are tied to the bytecodes of the runtime logic (written in Java)
  - Does not require the source code, which can be unavailable
- Uses standard JVMs at each node
  - ⇒ JavaSplit is a *distributed* JVM built on top of *standard* JVMs.
  - Any machine with a JVM can be utilized (portability)
  - Can use Just-in-Time compiler (JIT)
  - Able to employ the standard garbage collector
  - No need to install a specialized JVM or any other software
- Utilizes IP-based communication through Java sockets
- Incorporates a scalable object-based DSM

Bytecode Instrumentation



Intercepted Events

- Thread creation: for shipping threads to other nodes

```
someThread.start();
```

®

```
JS.Handler.startThread(someThread);
```

- Synchronization: for distributed synchronization

```
synchronized(lock){
        … … …
}
```

®

```
JS.Handler.acquire(lock);
        … … …
JS.Handler.release(lock);
```

- Field and array element accesses: for data consistency

```
var = obj.intField;
```

®

```
if(obj.__JS__state == 0)
        JS.Handler. readMiss(obj);
var = obj.intField;
```

Read Access Check Example

…	
ALOAD 1	// load instance of A
DUP	// duplicate instance of A on stack
GETFIELD	A::byte __JS__state__
IFNE	// jump if the state allows reading
DUP	// duplicate instance of A on stack
INVOKESTATIC	JS.Handler::readMiss
GETFIELD	JS.A::intField
…	

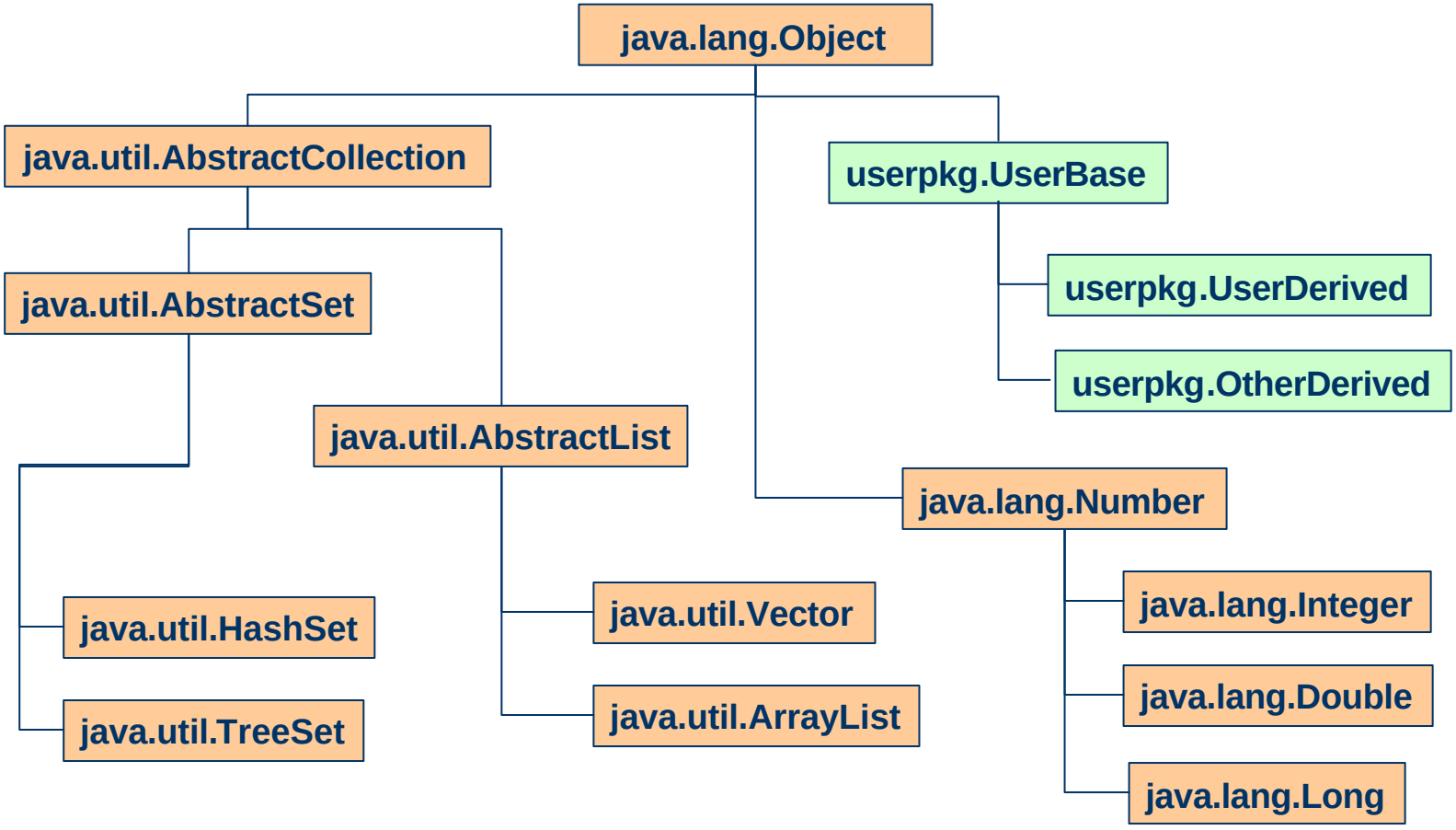
Access Check Elimination Examples

```
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
}
```

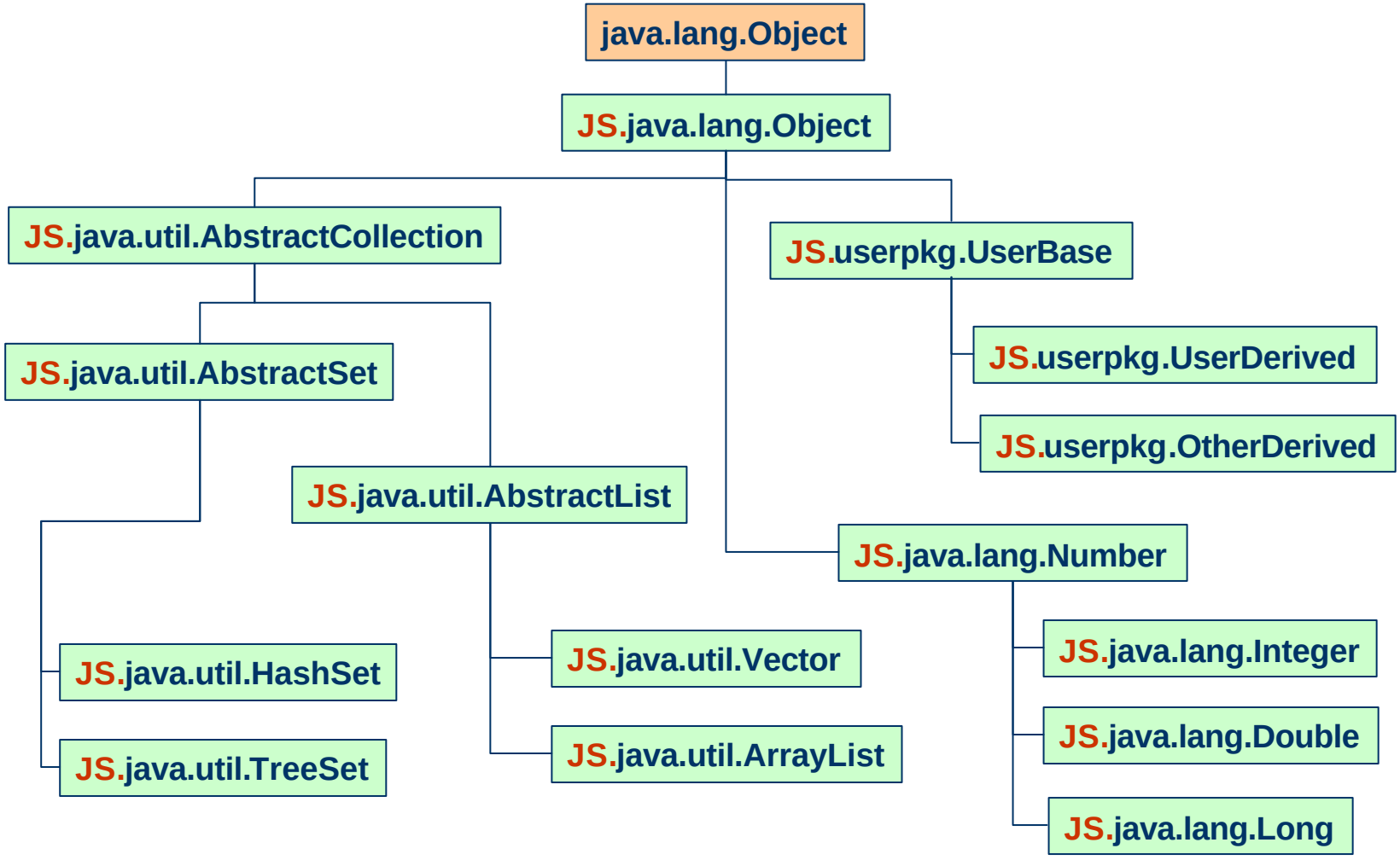
Additional Bytecode Transformations

- The rewritten classes are renamed to enable sound loading and usage of rewritten bootstrap classes
- All the references to classes in bytecode are replaced with the new names
- The classes are augmented with utility fields and methods

Original Class Hierarchy



Twin Class Hierarchy



```
class A extends somepkg.C {
    private int intField;
    public B refField;

    protected void doSomething
    (java.lang.String str, int n) {…}
}
```

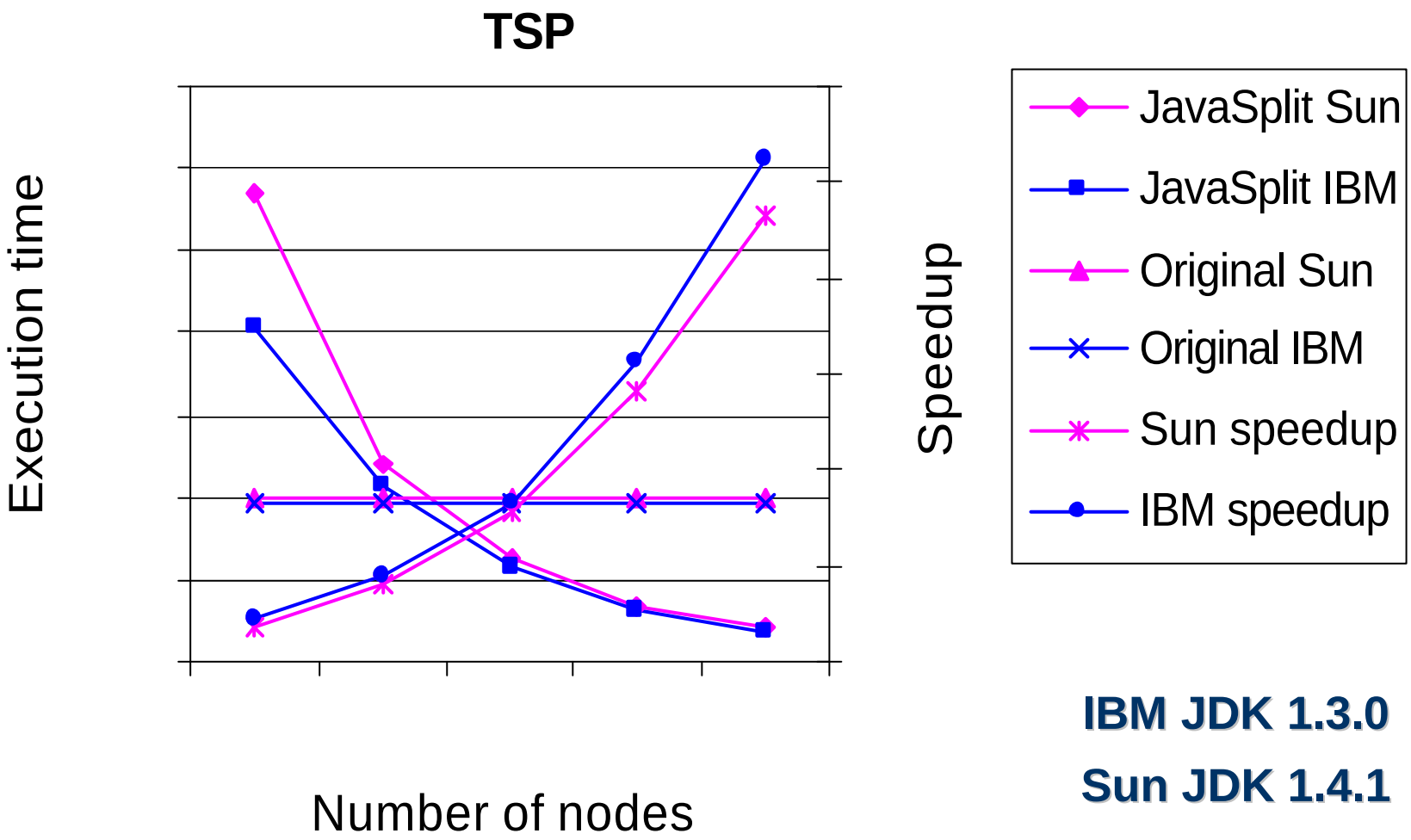
```
class JS.A extends JS.somepkg.C {
    private int intField;
    public JS.B refField;

    protected void doSomething
    (JS.java.lang.String str, int n) {…}

    // inherited utility fields
    public byte  __JS__state;
    public int   __JS__version;
    public long  __JS__globalID;

    public void _JS_serialize
    (OutputStream out)
    {…}
    public void _JS_deserialize
    (InputStream in)
    {…}
    public _JS_Diff _JS_compare
    (JS.A other)
    {…}
}
```

```
_JS_diff diff =
        super._JS_compare(other);
if(this.intField != other.intField) {
    …       // add to diff
}
if(this.refField != other.refField) {
    …       // add to diff
}
return diff;
```

Efficient Synchronization

- Java application contain a great amount of unneeded synchronization
  - May cause significant performance degradation in instrumented classes
- We distinguish between synchronization operations on local and shared objects
  - A handler is invoked for shared objects
  - Local objects avoid calling the handler (but rather manage a variable indicating whether the object is locked or not)
- As a result, synchronization of local objects is 4.5 times cheaper than in original Java

Performance



IBM JDK 1.3.0

Sun JDK 1.4.1

Future Work

- DSM optimizations
  - Load balancing
  - Data distribution
- Compiler optimizations
  - Inter-procedural access check elimination
  - Static detection of local objects and classes by the means of escape analysis
- High availability
  - For wide-area cycle stealing
  - An algorithm is ready, need only to implement