JavaSplit

A Portable Distributed Runtime for Java

Michael Factor	**IBM**
Assaf Schuster	**Technion**
Konstantin Shagin	**Technion**

Outline

- The goal & features
- Related work
- Implementation overview
- Performance evaluation
- Conclusion & future work

The Goal

Create a runtime environment that executes a multithreaded Java application on a set of interconnected machines

- Why Java?
  - Built-in multithreading and synchronization
  - Shared memory abstraction
  - Popular

JavaSplit Features

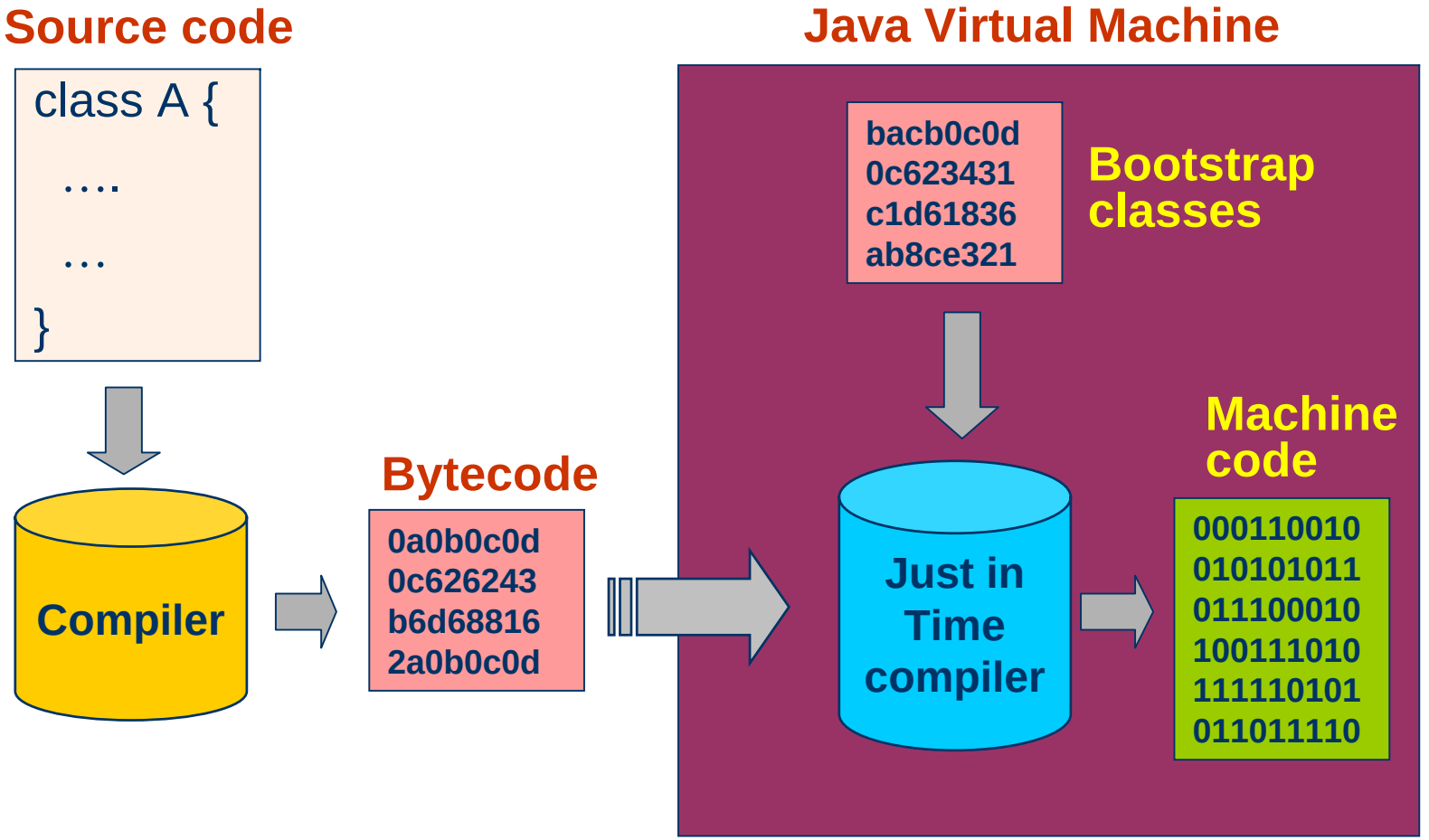
- **Transparency**
  - Executes standard, possibly pre-existing, Java applications
  - The programmer is completely unaware of the distributed nature of the runtime
- **Portability**
  - Any machine can participate without regard of its operating system and hardware architecture
- **Scalability**
  - Designed to efficiently support a large number of nodes

Related Work

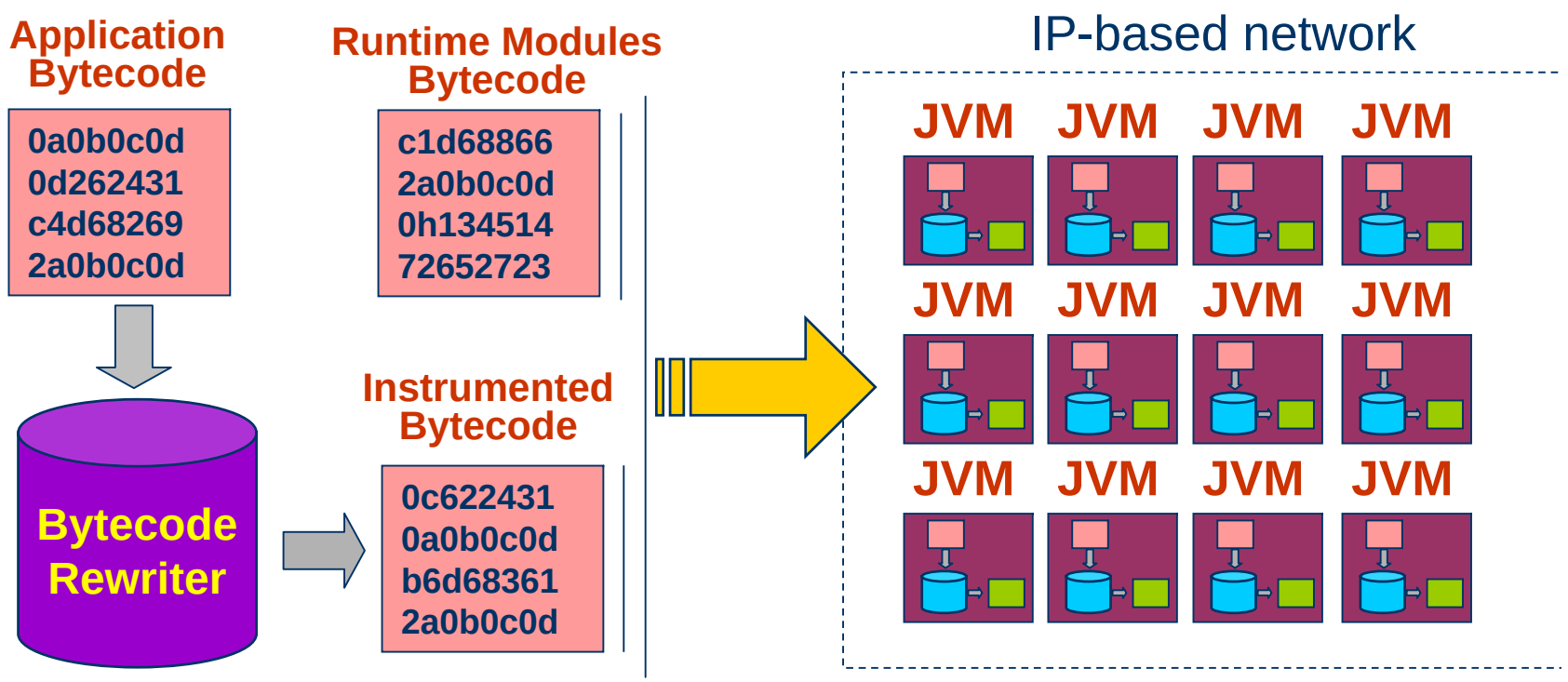
- Distributed (non standard) JVMs
  - Java/DSM (1997), Cluster VM for Java (former cJVM) (1999), JESSICA(1999)
  - Sacrifice portability
- Translation to native code combined with a DSM
  - Hyperion (2001), Jackal (2001)
  - Also not portable
  - Perform compiler optimizations
- Systems built on top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

➢ Unlike previous works, JavaSplit combines portability with transparency

Java Basics


Source code
class A {
….
…
}
Compiler
Bytecode
0a0b0c0d
0c626243
b6d68816
2a0b0c0d
Java Virtual Machine
bacb0c0d
0c623431
c1d61836
ab8ce321
Bootstrap classes
Just in Time compiler
Machine code
000110010
010101011
011100010
100111010
111110101
011011110

JavaSplit Overview



- Rewriting intercepts synchronization, accesses to shared data etc.
- Threads and objects are distributed among the machines
- Each node uses a standard Java Virtual Machine (JVM)

Advantages of using standard JVMs

- Any machine with a JVM can be utilized
- Nodes can join the system using a Java-enabled browser
- No need to install any software
- Can use just-in-time compiler (JIT)
  - Unlike most distributed JVMs
- Can employ a standard garbage collector

Distributed Shared Memory (DSM)

- Object-based
  - More suitable for Java
  - Few false-sharing
- Designed to be scalable
  - Never requires global cooperation
  - 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

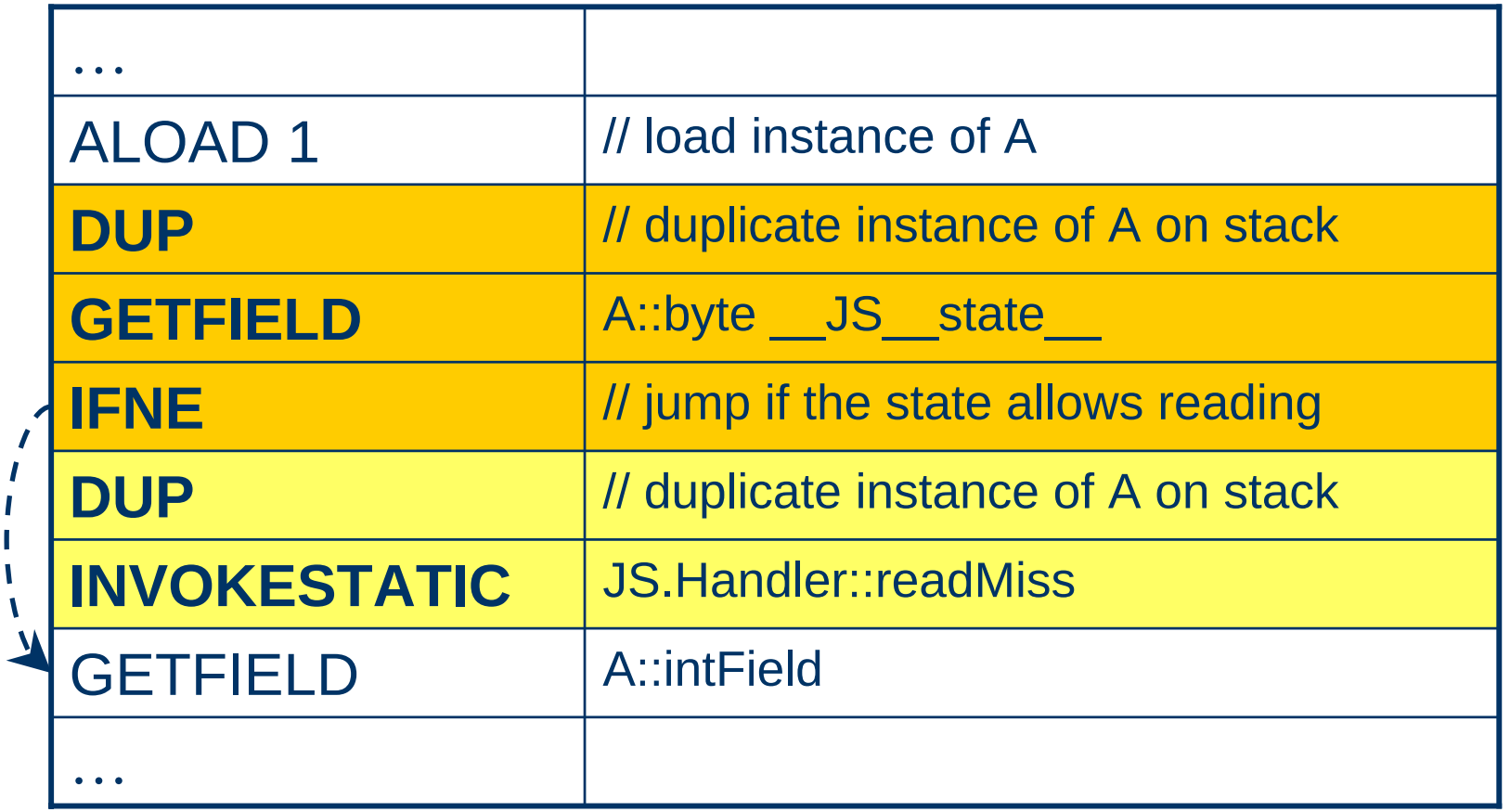
Instrumentation Details

- Classes are augmented with utility fields and methods
- The fields indicate the state and 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

```
class A extends C {
  …
  // inherited utility fields
  public byte  __JS__state;
  public int   __JS__version;
  public long  __JS__globalID;

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

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	A::intField
…	

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
}
```

Access Checks Overhead

[nanosec.]	Original	Rewritten	Slowdown
Field read	**0.84**	**1.82**	**2.17**
Field write	**0.97**	**2.48**	**2.56**
Static write	**0.84**	**1.84**	**2.2**
Static read	**0.97**	**2.97**	**3.1**
Array read	**0.98**	**5.45**	**5.57**
Array write	**1.23**	**5.05**	**4.1**

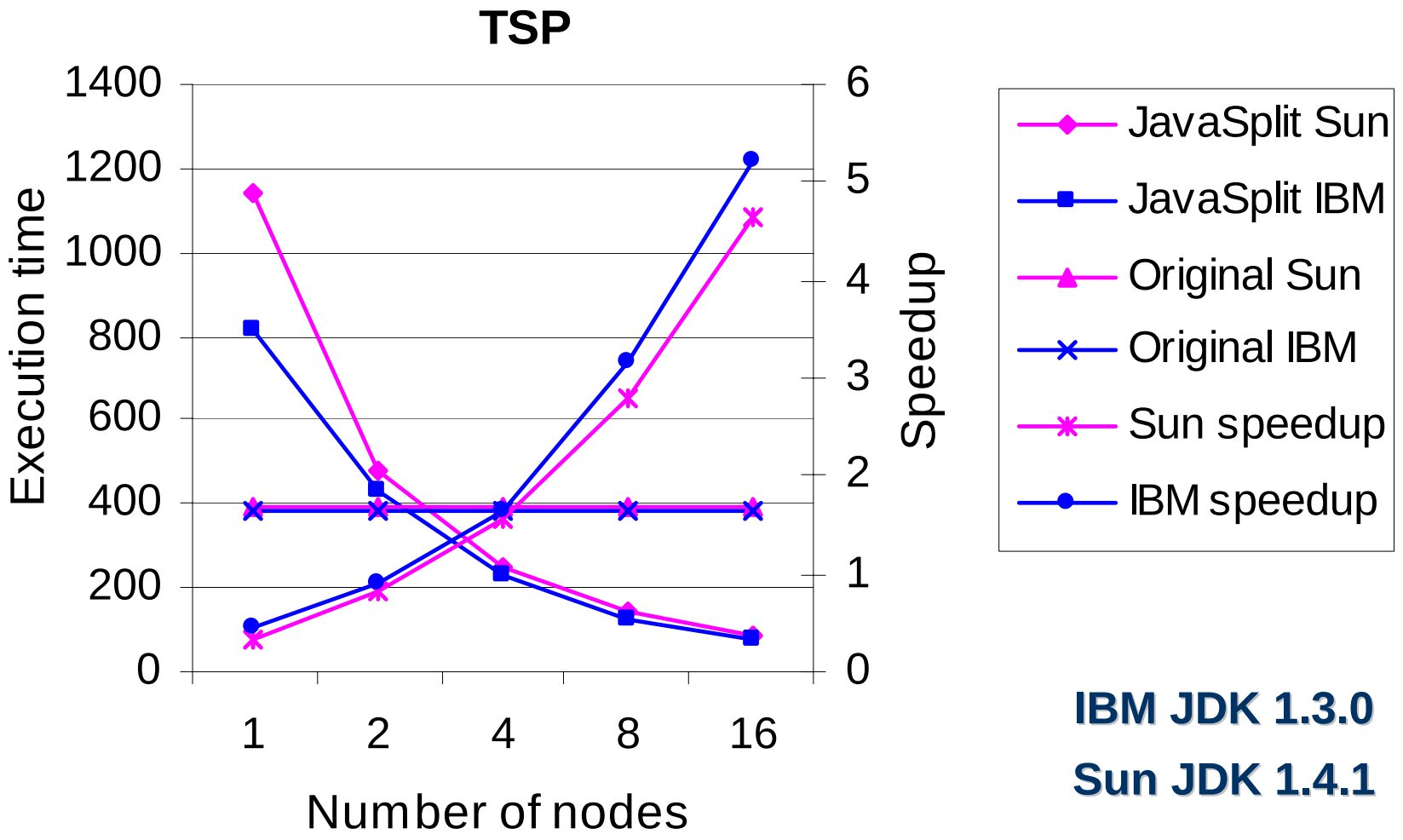
Sun JDK 1.4.1

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

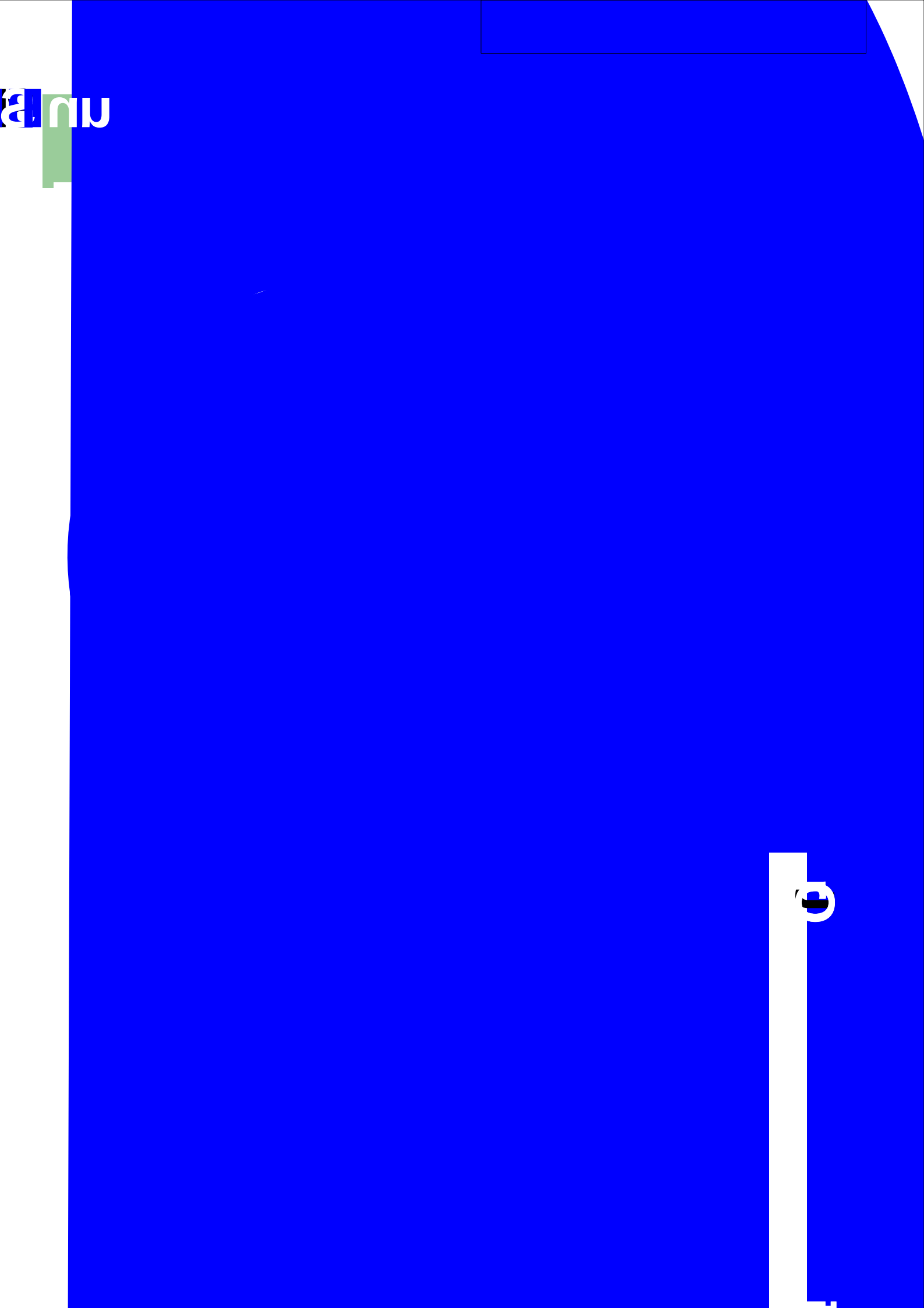
[nanosec.]	Original	Local	Shared
Sun 1.4.1	90.6	19.6	281
IBM 1.3.0	93.4	54.7	327

Performance

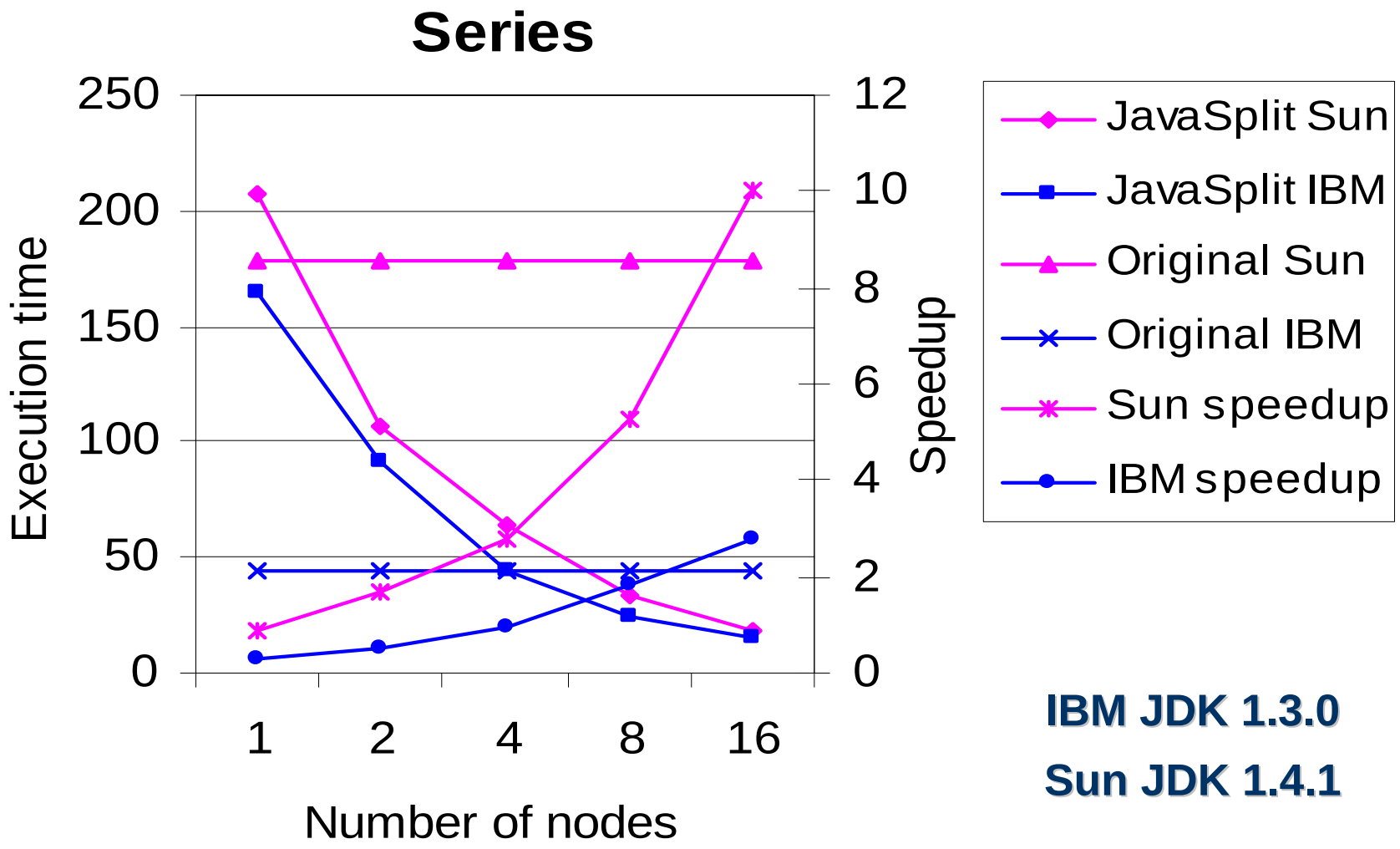


IBM JDK 1.3.0

Sun JDK 1.4.1



Performance (3)



IBM JDK 1.3.0

Sun JDK 1.4.1

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

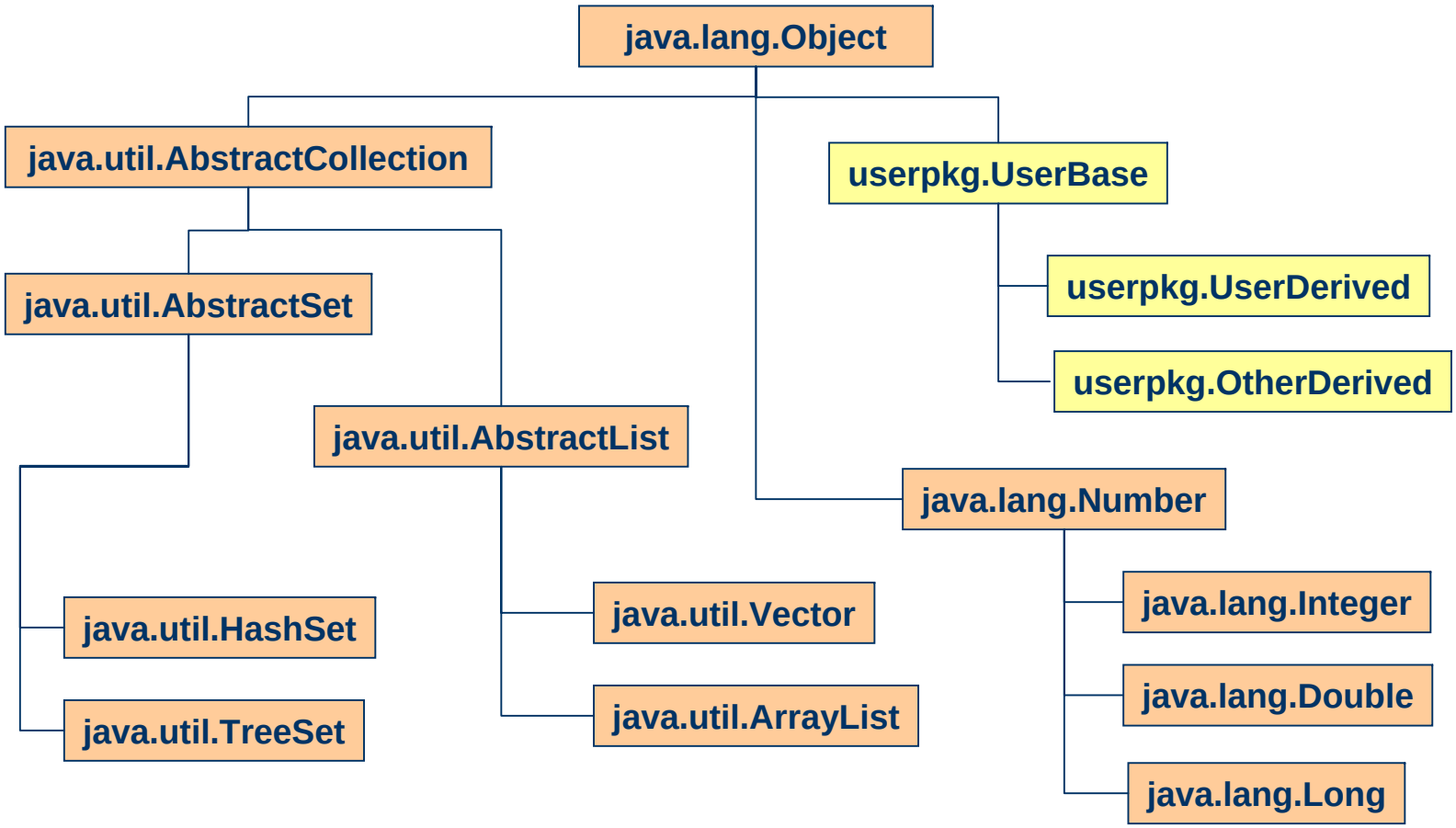
Future Work

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

The End

Questions?

Original Class Hierarchy



Twin Class Hierarchy