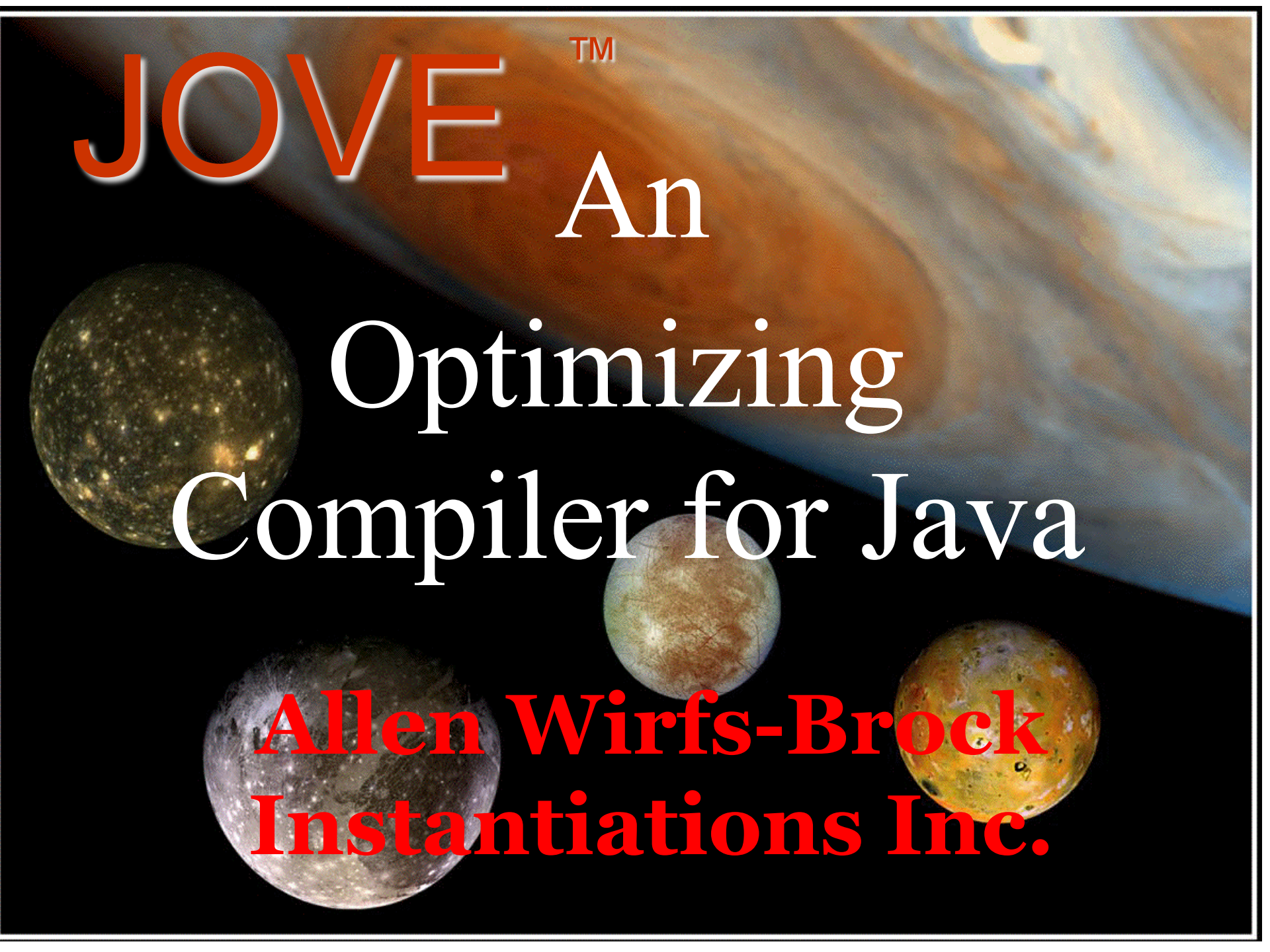

JOVE™
An
Optimizing
Compiler for Java
Allen Wirfs-Brock
Instantiations Inc.

Object-Orient Languages Provide a Breakthrough in Programmer Productivity

- Reusable software components
- Higher level abstractions
- Yield significant productivity improvements

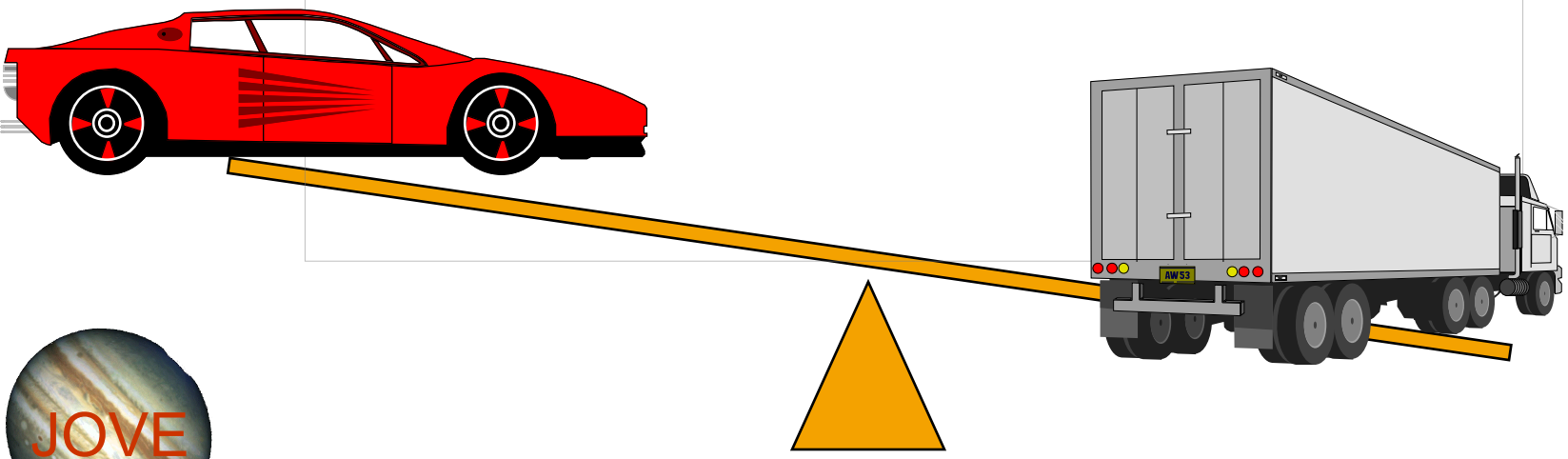
The “Rap” on Object-Oriented Languages

- They’re slow!!!
  - C++ slower than C
  - Smalltalk slower than C++
  - Java slower than C++

Why are OO Languages slow?

- Unconventional implementation techniques
- New, expensive fundamental operations
- Object-oriented design and programming techniques

There seems to be a delicate balancing act between OO productivity and execution performance



Why?

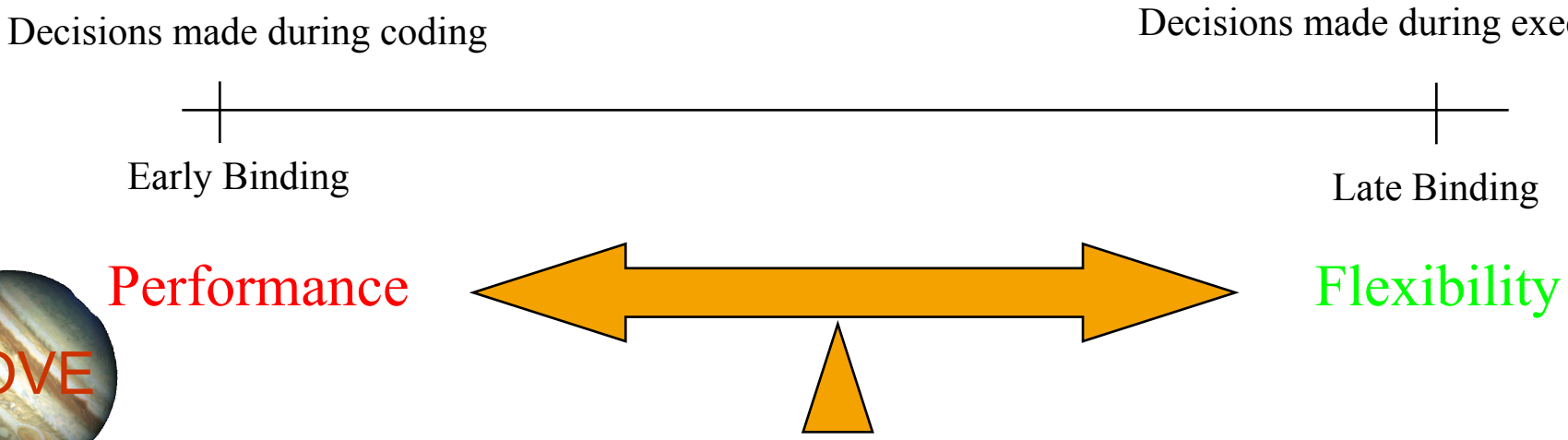
- Productivity comes from generalization
  - Reusable class libraries
  - Extensible frameworks
  - Plugable components
  - Abstract algorithms
- Implemented using
  - Subclass refinement
  - Polymorphic typing
  - Polymorphic methods
  - Accessor Methods
  - Chains of delegation

Generality = Inefficiency

- Extra functionality that is present but not used
- Many calls of tiny methods
- Accessors methods
- Generic algorithms depend upon dynamic (polymorphic) method dispatch
- Late binding via polymorphism precludes static optimizations

It's an issue of binding time.

- Procedural designs and languages are fast, but inflexible because most decisions are bound very early, during coding.
- Object oriented designs are slow, but flexible because many decisions are bound very late, during program execution.



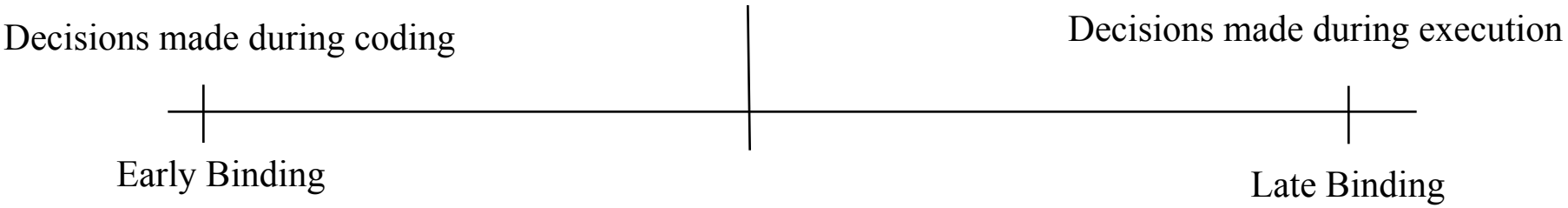
The Dilemma

- Late bound OO code is highly reusable, making programmer more productive over the course of developing several applications.
- Late bound OO code is inefficient.
- Early bound procedural code is generally not very reusable.
- Early bound procedural code can be very efficient.

Why can't we have both?

- Write program using, highly reusable object-oriented code.
- Execute them as very efficient, procedural code.

Move object-oriented binding decisions earlier by making decisions at compilation time instead of during execution



Don't OO Language Compilers already do this?

- No
- Conventional C++, Eiffel, Java, etc. compilers generally don't perform compile-time bindings of object-oriented constructed.
- They don't know enough about a program to make these decisions so…
- They implement object-oriented constructs via late binding during execution

What does a compiler need to know

- Complete class hierarchy
- All possible control paths through the program
- Initial values of variables

The compiler needs to "understand" the *whole program*.

Whole Program Optimization

- By examining a whole program a compiler can:
  - Determine exactly which classes, methods, and fields are actually used by the program.
  - Which classes are actually instantiated
  - The specific classes of objects that will be used at each expression within the program.
  - The possible "receivers" for every method invocation
  - Whether a method invocation actually needs to be polymorphic
  - Determine which objects may be shared between threads
  - etc...

Why is the time right for Whole Program Optimizing compilers?

- Very fast processors
- With very large main memories
- Wide spread use of object-oriented languages

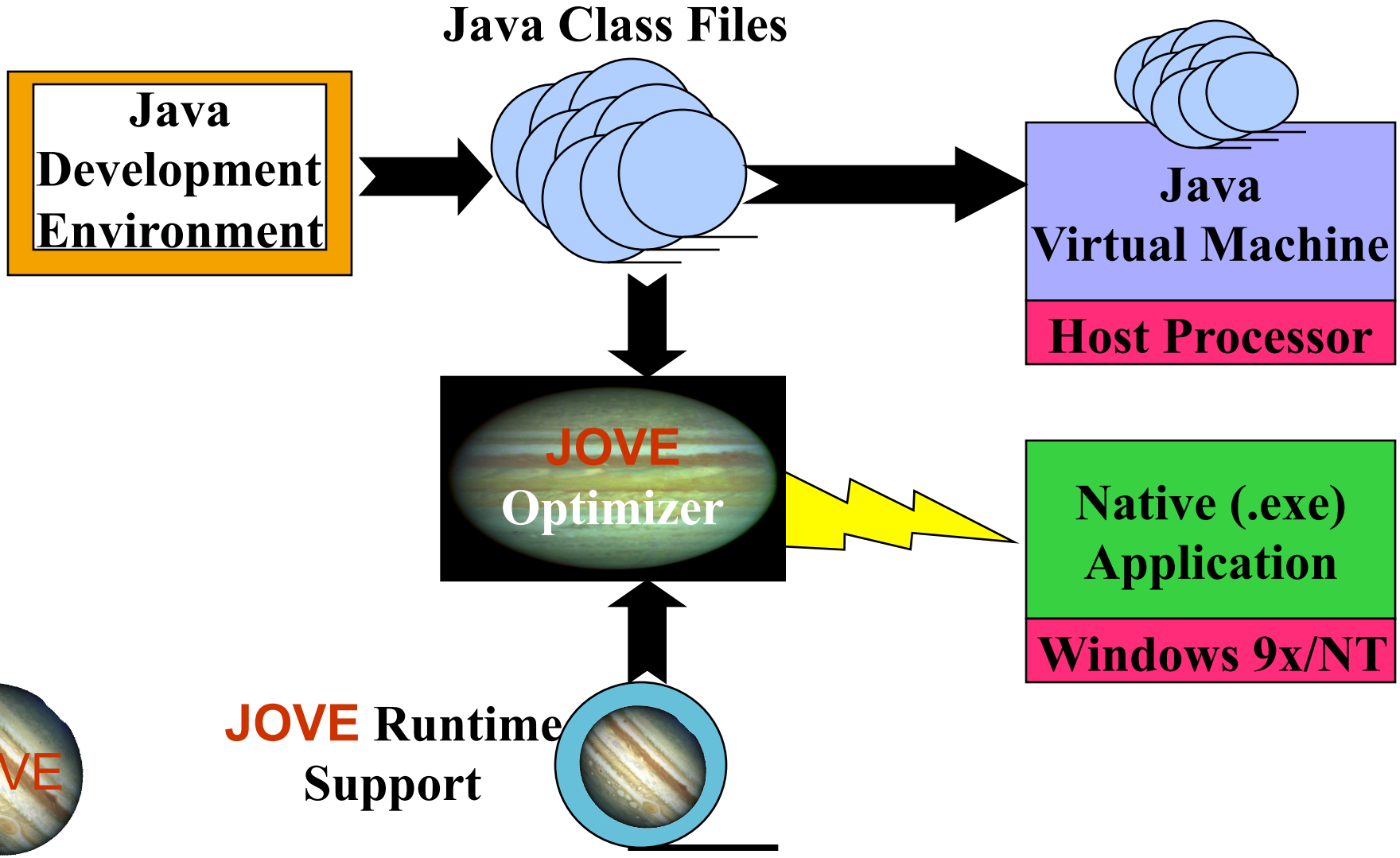
Experimental Whole Program Optimizing Compilers for Object-oriented language

- Vortex Compiler
  - University of Washington - Craig Chambers, Jeffrey Dean, et al
- Marmot
  - Microsoft Research - Robert Fitzgerald, et al

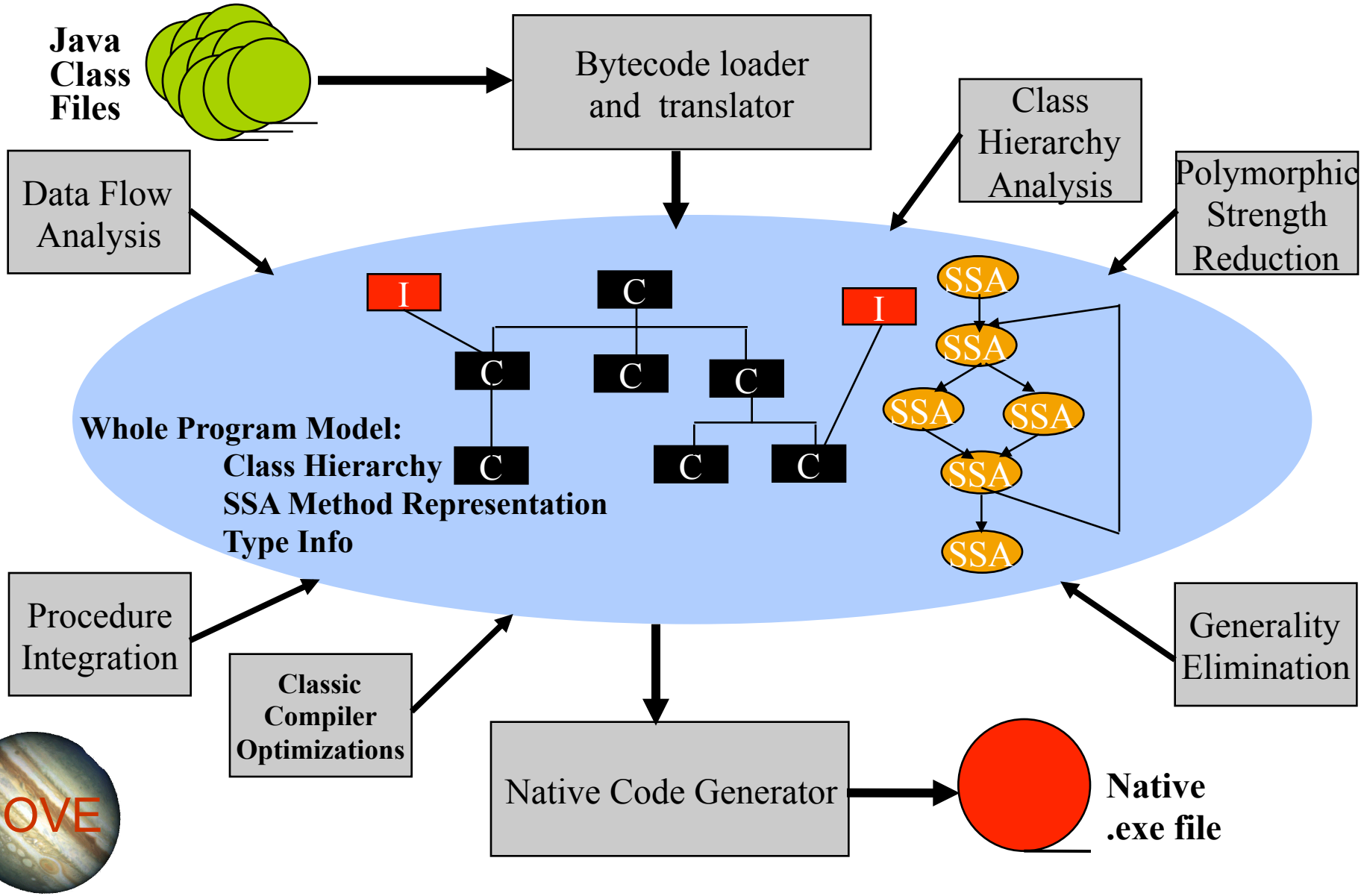
JOVE

- First production-quality whole program optimizing compiler for Java
  - Static deployment compiler for Java
  - Aggressive, whole-program optimizer
  - Creates single-file, native executables for deployment
  - Supports JDK 1.1, 1.2, and 1.3 client or server Java applications

Java
Development & Deployment



Jove Conceptual Architecture



Whole Program Object-Oriented Optimizations

- **Generality elimination**
- **Class hierarchy analysis**
- **Polymorphic type elimination**
- **Polymorphic strength reduction**
- **Type check elimination, etc.**

Generality Elimination

```
class Customer refines Person{
        public static Customer nextCustomer{…}
        protected String name;
        protected String address;
        protected int id;
        public String name() {return name;}
        public String address() {return address;}
        public String id() {return id;}}
```

```
while ((c= Customer.nextCustomer())  != null)
        System.out.println(c.name());
```

Generality Elimination

```
class Customer refines Person{
        public static Customer nextCustomer{…}
        protected String name;
        protected String address;
        protected int id;
        public String name() {return name;}
        public String address() {return address;}
        public String id() {return id;}}
```

```
… while ((c= Customer.nextCustomer())  != null)
        System.out.println(c.name());...
```

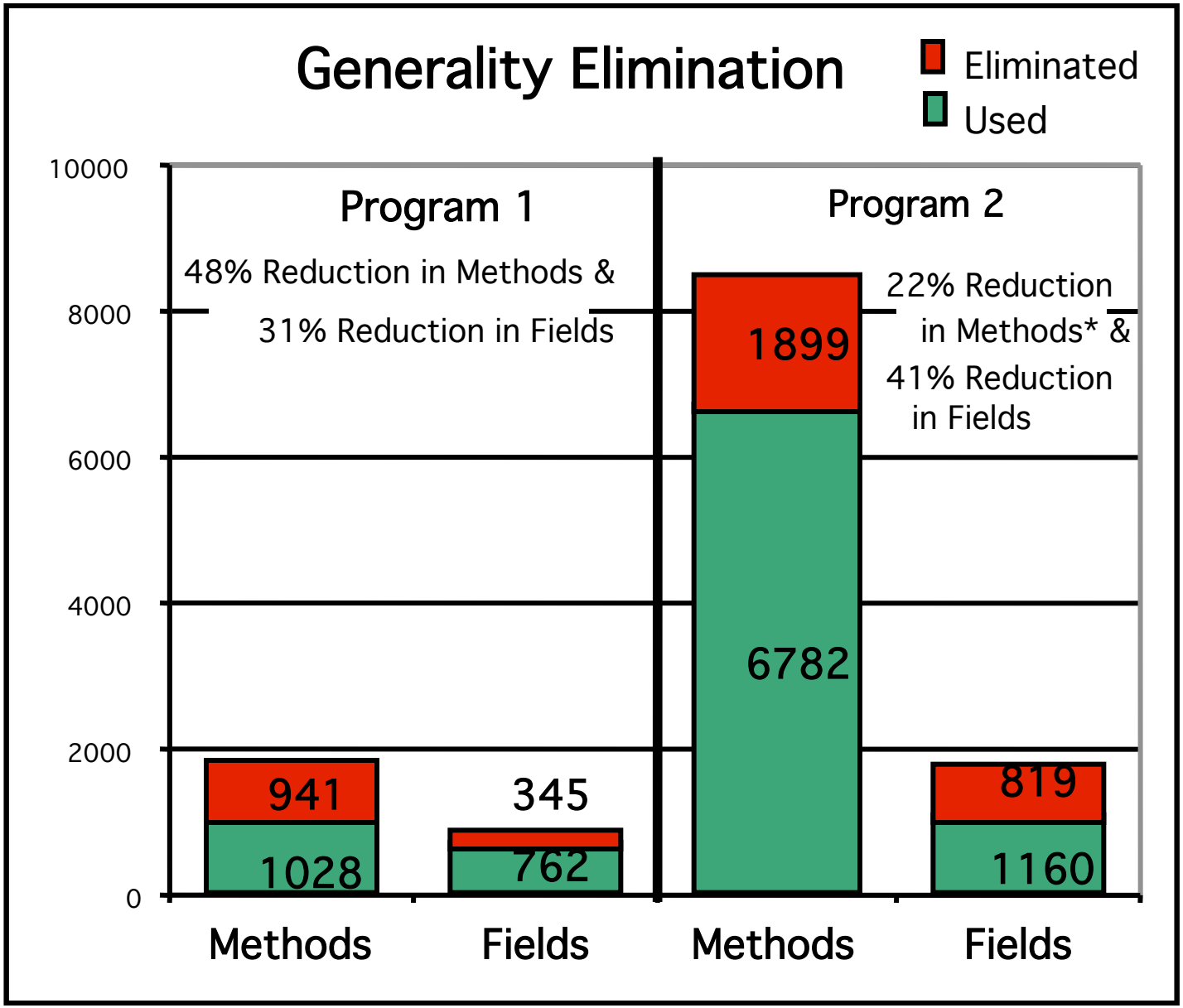
Assume that some program only uses the nextCustomer *and* name *methods of* Customer *as shown above.*

Generality Elimination

```
class Customer refines Person{
        public static Customer nextCustomer{…}
        protected String name;
        protected String address;
        protected int id;
        public String name() {return name;}
        public String address() {return address;}
        public String id() {return id;}}
```

```
… while ((c= Customer.nextCustomer())  != null)
        System.out.println(c.name());...
```

The program does not need the methods id() *and* address() *or the fields* address *and* id. *Generality elimination will remove these from the program model.*



*Eliminated 55% of java.lang methods

Class Hierarchy Analysis

```
abstract class Person{
        public String name() {return "John Doe"; }

class Customer refines Person{
        public static Customer nextCustomer(){…}
        protected String name;
        public String name() {return name;} }
```

Person *is an abstract class whose only subclass is* Customer. Customer.name() *overrides* Person.name() *hence* Person.name() *can be eliminated and* Customer.name() *is not polymorphic.*

Class Hierarchy Analysis

```
abstract class Person{
        public String name() {return "John Doe"; }

class Customer refines Person{
        public static Customer nextCustomer(){…}
        protected String name;
        public String name() {return name;} }
```

Person *is an abstract class whose only subclass is* Customer. Customer.name() *overrides* Person.name().

Class Hierarchy Analysis

```
abstract class Person{
        public String name() {return "John Doe"; }

class Customer refines Person{
        public static Customer nextCustomer(){…}
        protected String name;
        public String name() {return name;} }
```

Because Customer.name() *overrides* Person.name() *which is an instance method of an abstract class and because there are no other subclasses of* Person, Person.name() *can be eliminated and* Customer.name() *is not polymorphic.*

Polymorphic Type Elimination

```
String Classify (Object p) {
        if (p instanceof Customer) return "Customer";
        if (p instanceof Employee) return "Employee";
        return "Generic Person";}
```

```
...Classify(Customer.nextCustomer())...
```

Classify *is method whose parameter is declared as type* Object *but its only invocation is passing the result of a methods whose result type is Customer.*

Polymorphic Type Elimination

String Classify (~~Object~~ Customer p {
    ~~if (p instanceof Customer)~~ return "Customer";
    ~~if (p instanceof Employee) return "Employee";~~
    ~~return "Generic Person";~~}

...Classify(Customer.nextCustomer())...

The compiler can now treat the argument type of Classify *as if it was declared to be* Customer *which has been determined to be a non-polymorphic type for this program. Under this assumption the first* instanceof *test will always be* true *and it and subsequent code can be eliminated.*

Polymorphic Type Elimination

```
String Classify (Customer p {
        return "Customer";
        }
```

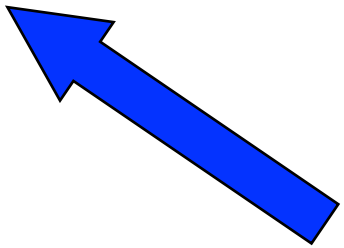
```
...Classify(Customer.nextCustomer())...

…"Customer" …                    // inlined call to Classify
```

The simplified version of Classify *can then be easily inlined at its call site. Because the result of* nextCustomer() *is not used that call is also eliminated.*

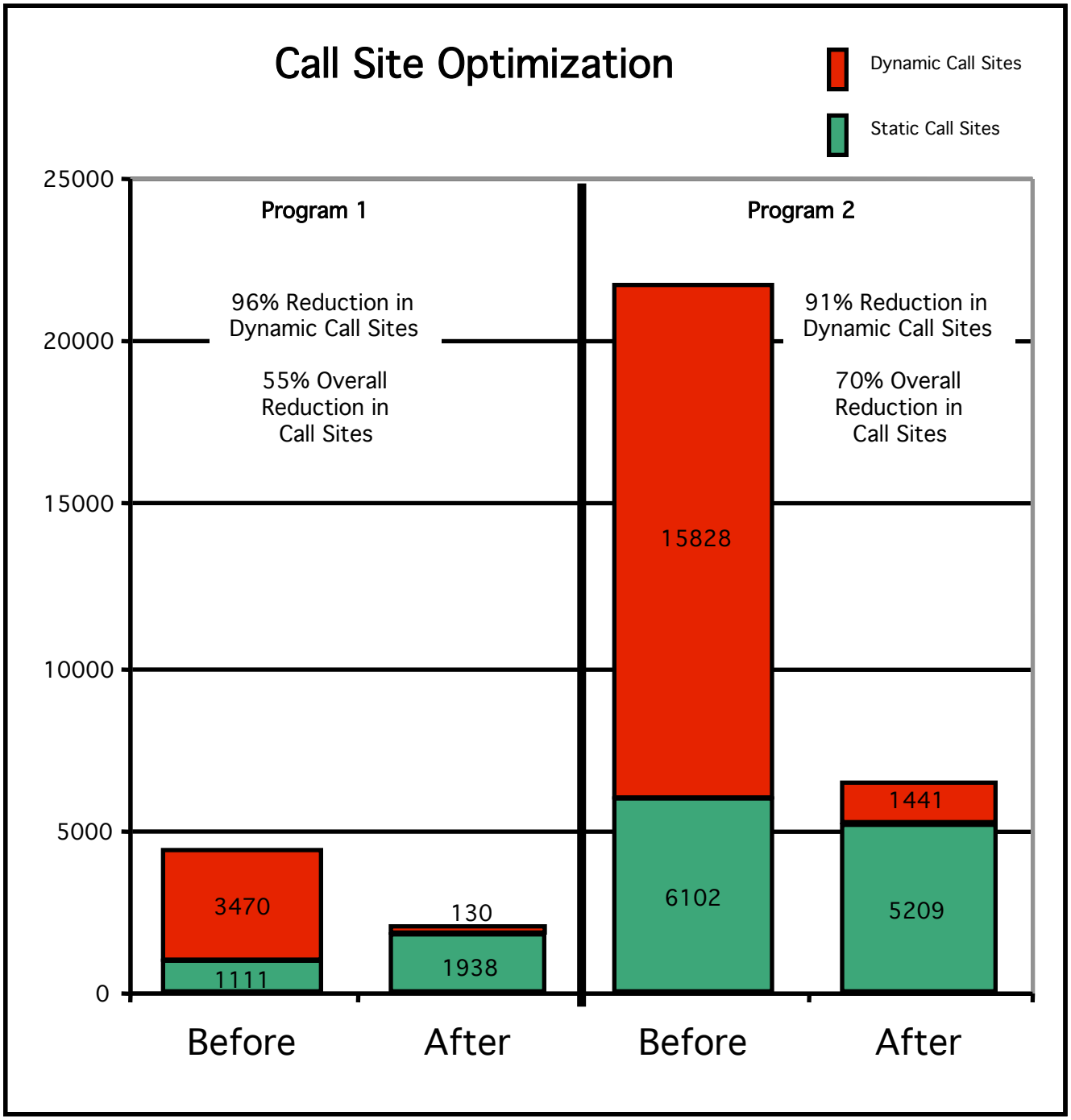
Polymorphic Strength Reduction

```
String Classify2 (~~Object~~ Customer p) {
        if (hashtable[p.hashCode()] ==p {
                …
}
```



hashCode() *can be strength reduced from a dynamic polymorphic call into a static call (and probably inlined)*

```
...Classify2(Customer.nextCustomer())...
```


Call Site Optimization
Dynamic Call Sites
Static Call Sites
25000
20000
15000
10000
5000
0
Program 1
Program 2
96% Reduction in Dynamic Call Sites
55% Overall Reduction in Call Sites
91% Reduction in Dynamic Call Sites
70% Overall Reduction in Call Sites
3470
1111
130
1938
15828
6102
1441
5209
Before
After
Before
After

The JOVE Runtime

Scalable Runtime Architecture designed to support extremely large Java programs

- Low memory overhead objects
- Precise multi-generational garbage collection
- Native multi-threading or single threaded
- Multi-threaded garbage collection
- Fast method dispatch and type checks

Message Dispatch Type Inclusion Testing

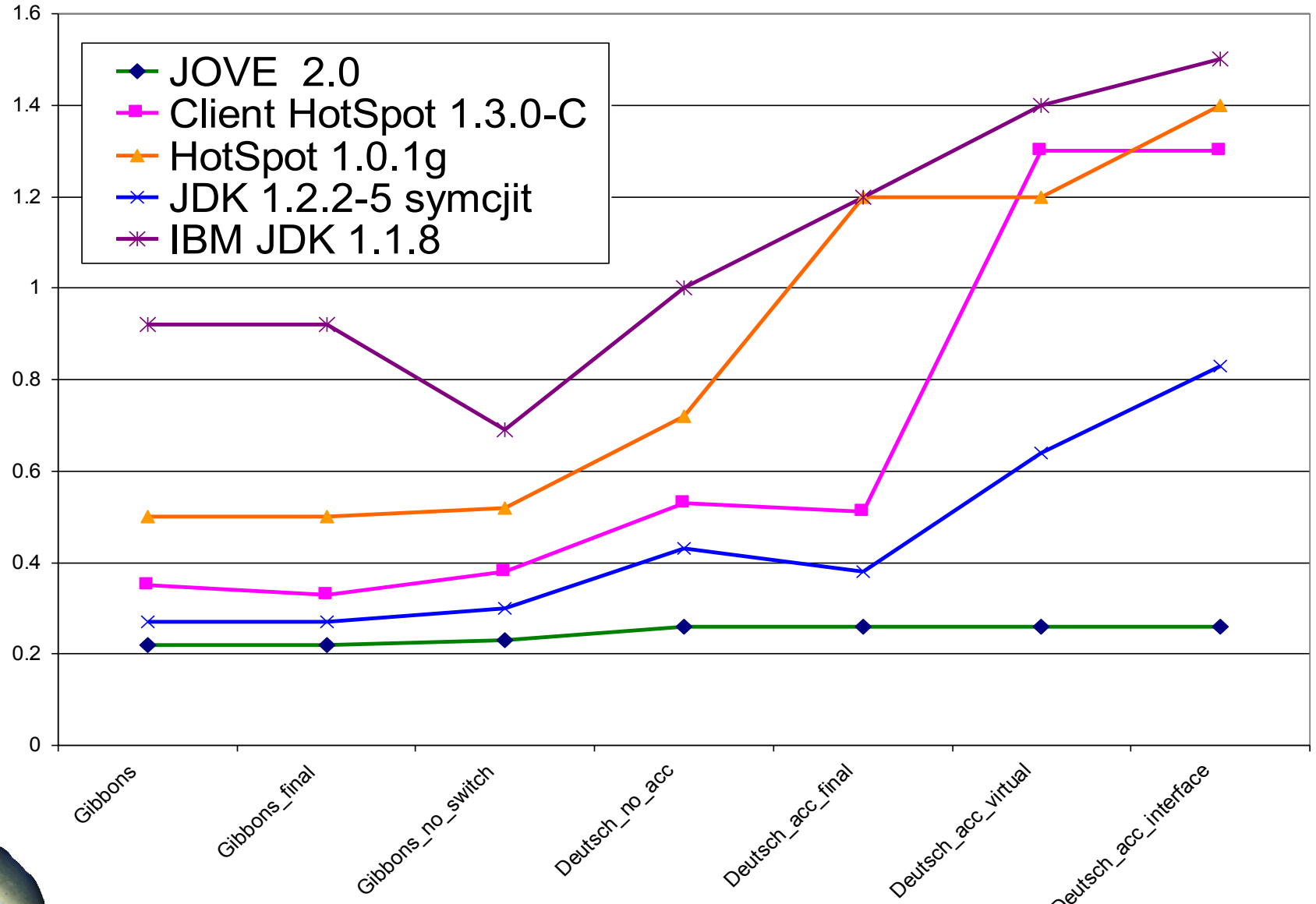
- Selector table dispatch for polymorphic calls
  - Row displacement compacting
  - No extra dispatch overhead for interfaces
- Compact encoded tables for very fast type inclusion testing (checkcast & instanceof)
  - Constant time tests
  - Test is a 4 instruction sequence on Intel architecture

~~Why are OO Languages slow?~~

Using JOVE, Java programs are fast!

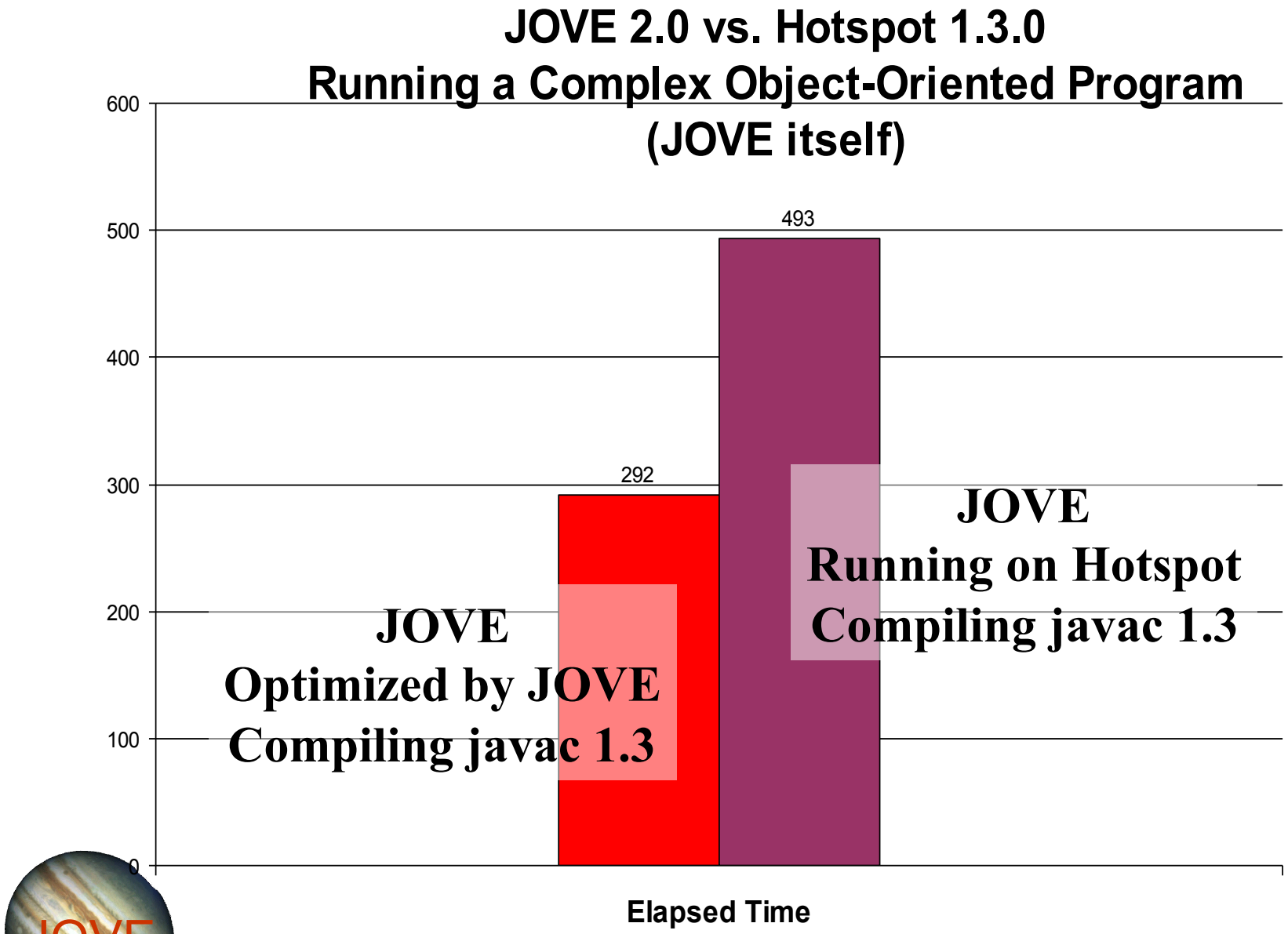
- ~~Unconventional implementation techniques~~
  - Compiled native code instead of a virtual machine
- ~~New, expensive fundamental operations~~
  - Implemented by efficient runtime system
- ~~Object-oriented design and programming techniques~~
  - Overhead eliminated using whole program optimizations.

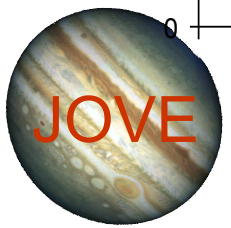
Richards Benchmarks



More object-oriented

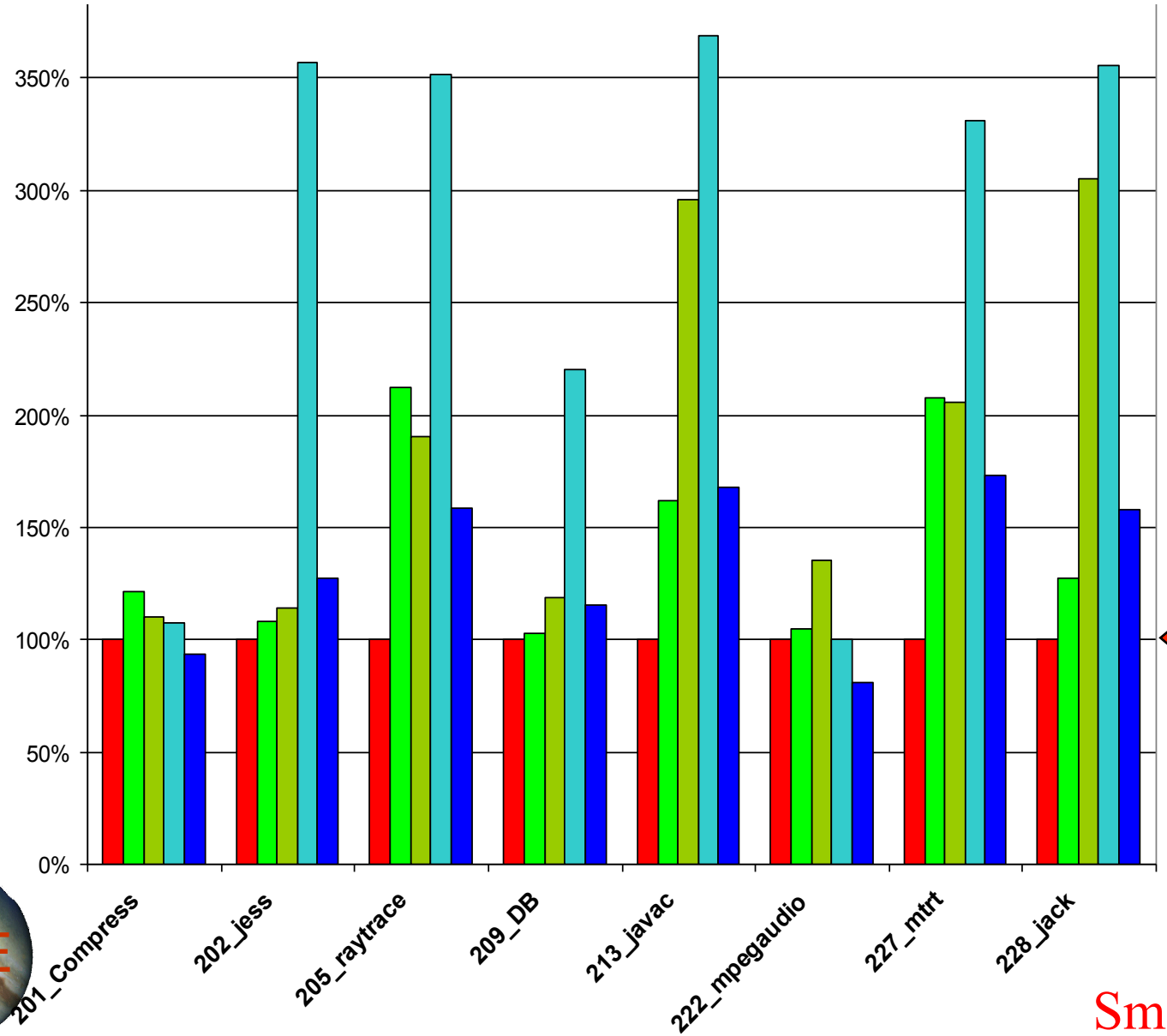
Smaller is better





Smaller is better

SPECjvm Benchmarks for Various JVMs Normalized to JOVE 2.0



JOVE 100%

Smaller is better

Applicability

- Whole program optimization requires a static program structure.
- Unconstrained reflection impedes optimizations.
- Dynamically composed programs not supported.

All classes are available at deployment time.

Use JOVE

New classes loaded from network during execution.

Use a JIT

But…Many existing Java programs use:

- Class.forName()
- Reflection
- Depend upon native methods that use reflection (via JNI)

Jove accommodates such programs by providing a declarative mechanism for:

- Identifying classes that will be "dynamically" loaded
- Identifying specific classes, methods, and fields that are accessed via reflection
- Identifying classes, methods, and fields that are accessed from native methods using JNI

Beware of "Attractive Nuisance" Features

- Don't use dynamic class loading to initialize data arrays
- Use interfaces instead of method reflection
- Don't use reflection to implement polymorphic static methods
- Don't use reflection to implement configuration flags - instead use interfaces for actual data flags.

Conclusion

Using Static, whole-program optimization techniques we can reap the full productivity benefits of Object-Oriented programming without sacrificing runtime performance

Allen Wirfs-Brock
Instantiations, Inc.
www.Instantiations.com
Allen_Wirfs-Brock@Instantiations.com