Ramblings on Object Models

David Grove

IBM Research

Jikes RVM project

January 11, 2006

Overview

- What is the object model?
  - VM’s internal, universal representation of objects
  - Cross-cuts almost every component of the VM
- Impact on Space
  - Per-object overhead
- Impact on Time
  - Access costs
  - Cache locality (related to space cost)

Goals for an Object Model

- Perform well for common case
    - Lots of ways to trade time/space
    - What exactly is the “common” case you are optimizing for?
    - JIT generates inline code for many (most) basic operations
    - Space efficient object models have less redundancy (impacts debugging)
- Desirable to support multiple object models
    - Rapid prototyping
    - Tune object model to GC and other aspects of system
    - “Common case” may differ depending on platform/application
    - Supporting radically different object models may not actually be desirable

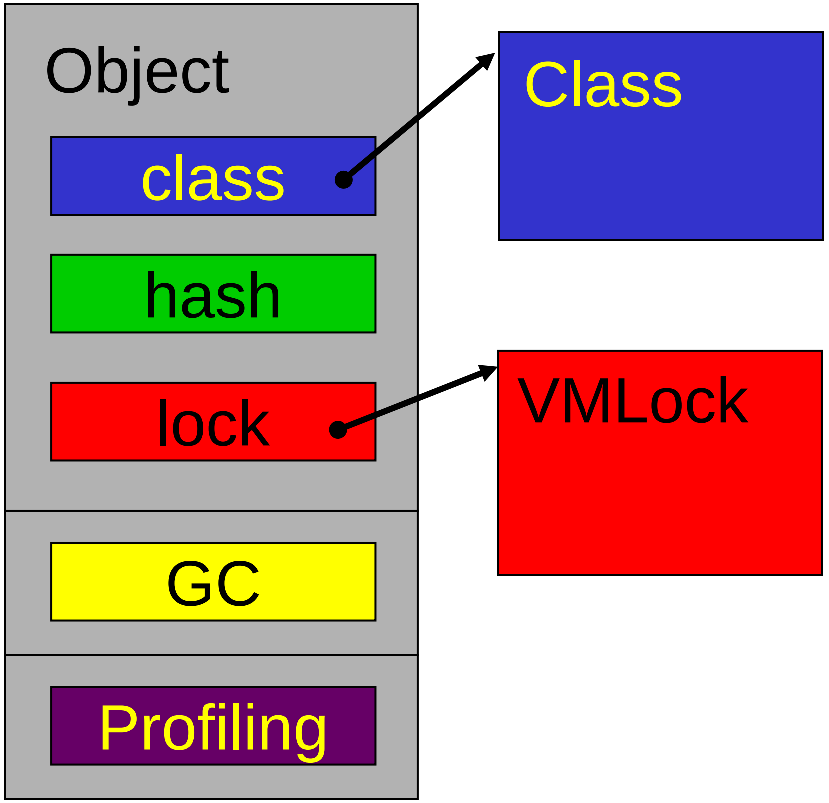
- Engineering, not rocket science (mostly…)

Topics for Today’s Rambling

- Object header
- Dynamic Typechecking & Dispatching

Abstract Java Object Model

```
class Object {
    Class getClass();
    int hashCode();
    void wait();
    void wait(long);
    void wait(long,int);
    void notify();
    void notifyAll();
    Object clone();
    boolean equals();
    void finalize();
}
```



Compression Techniques: Hashing

(Agesen ’97, Bacon et al.’98)

- Observations
  - Objects usually die before they move
  - Objects usually are not hashed
  - The address of an object is a good hash code (or seed)
- Use 3-state encoding
  - *unhashed*, *hashed*, *hashed&moved*
  - In states *unhashed* and *hashed* hash code is address
  - On GC, *hashed* object has address copied to new object
  - In state *hashed&moved*, hash code is retrieved from end

Compression Techniques: Locking

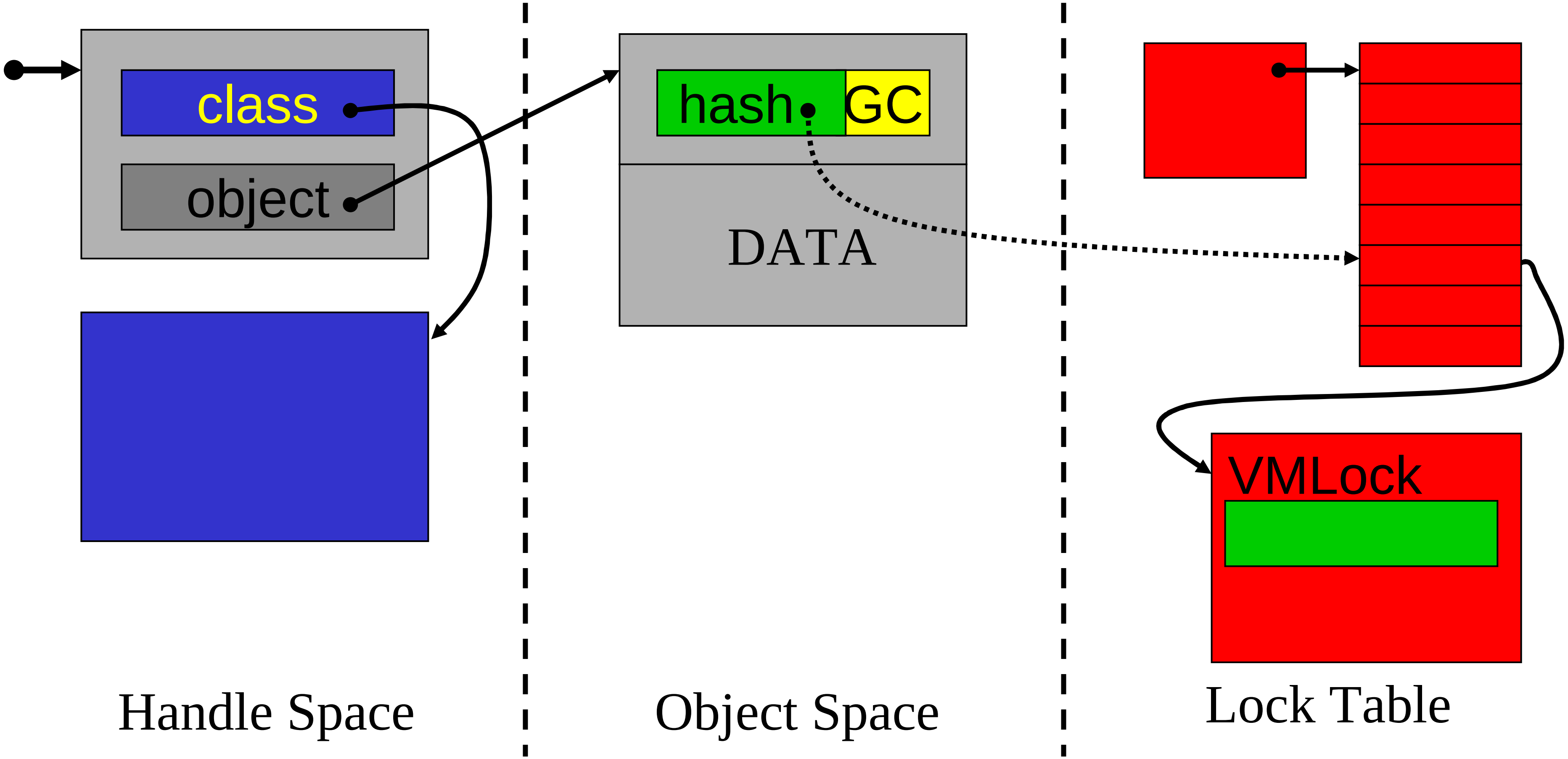
(Bacon et al.'98)

- Observations
  - Most objects are not locked
  - Nesting of locks is shallow
  - Most locked objects are not contended
- Encode as 24-bit *thin lock*
  - In thin case: *fat bit*=0, *owning thread*, *nest level*
  - In fat case, *fat bit*=1, *index* of inflated lock structure
  - In usual thin case, only 1 compare&swap needed
- Numerous variants and improvements

Compression Techniques: Locking (Bacon et al.'02)

- Observations
  - Most objects are not locked
  - Most locked objects have synchronized methods
- Treat lock as an *implicit field*
  - Defined by first synchronized method in hierarchy
  - synchronized methods will know the offset
  - synchronized blocks may need to look up offset

Original Sun Object Model ('95)

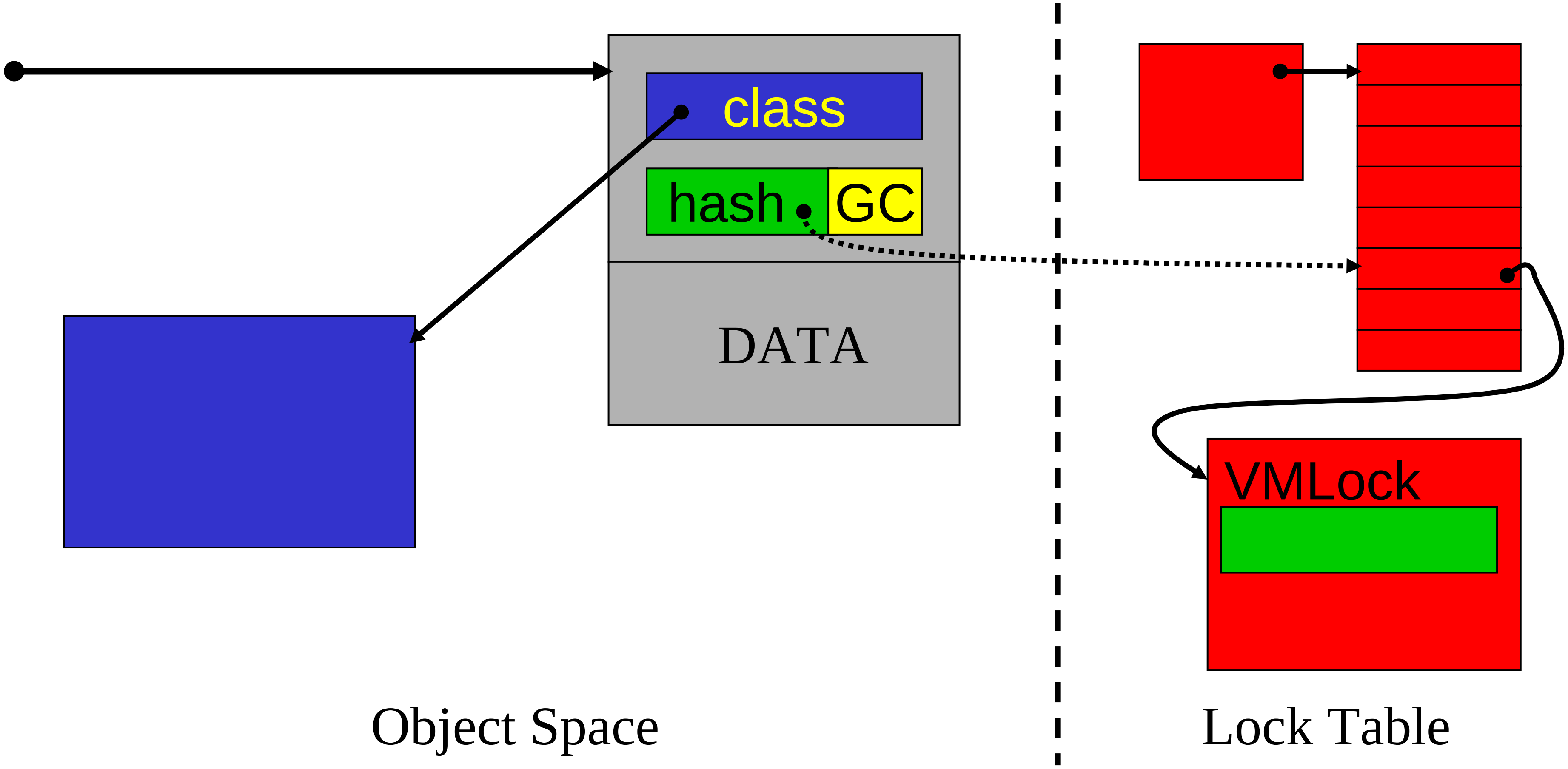


Handle Space

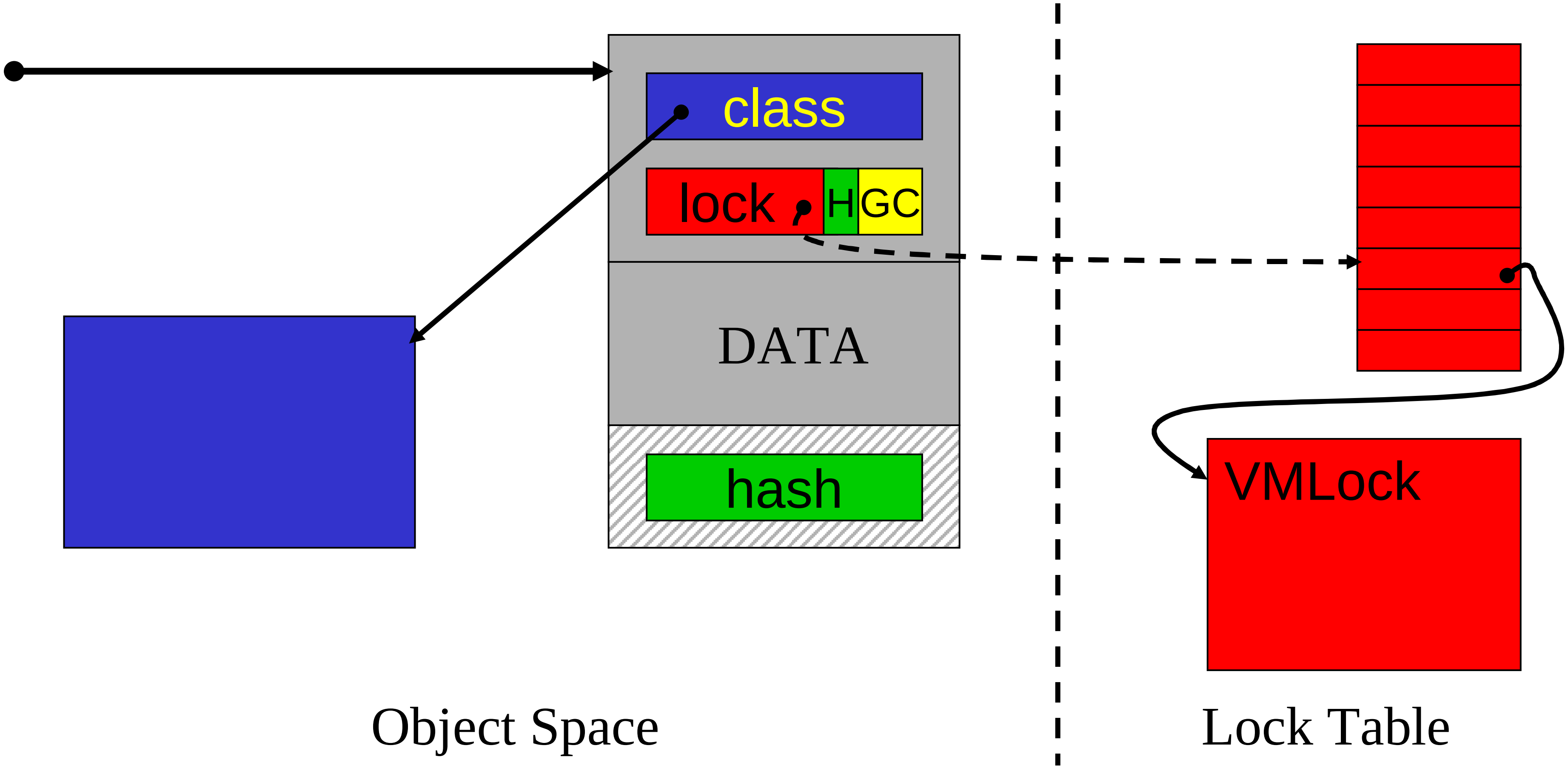
Object Space

Lock Table

IBM JVM without Handles ('97)



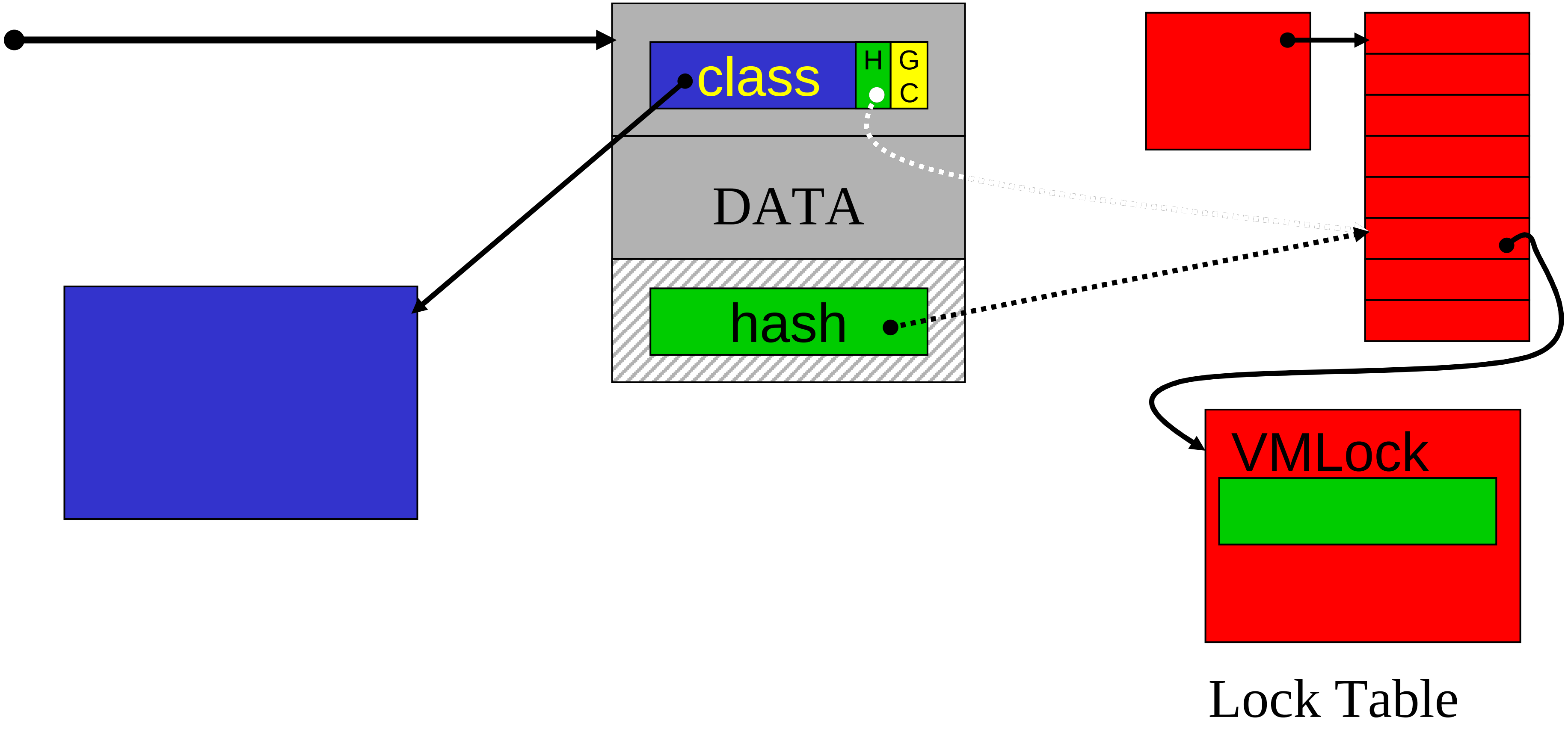
IBM JVM with Thin Locks ('98)
Jikes RVM default



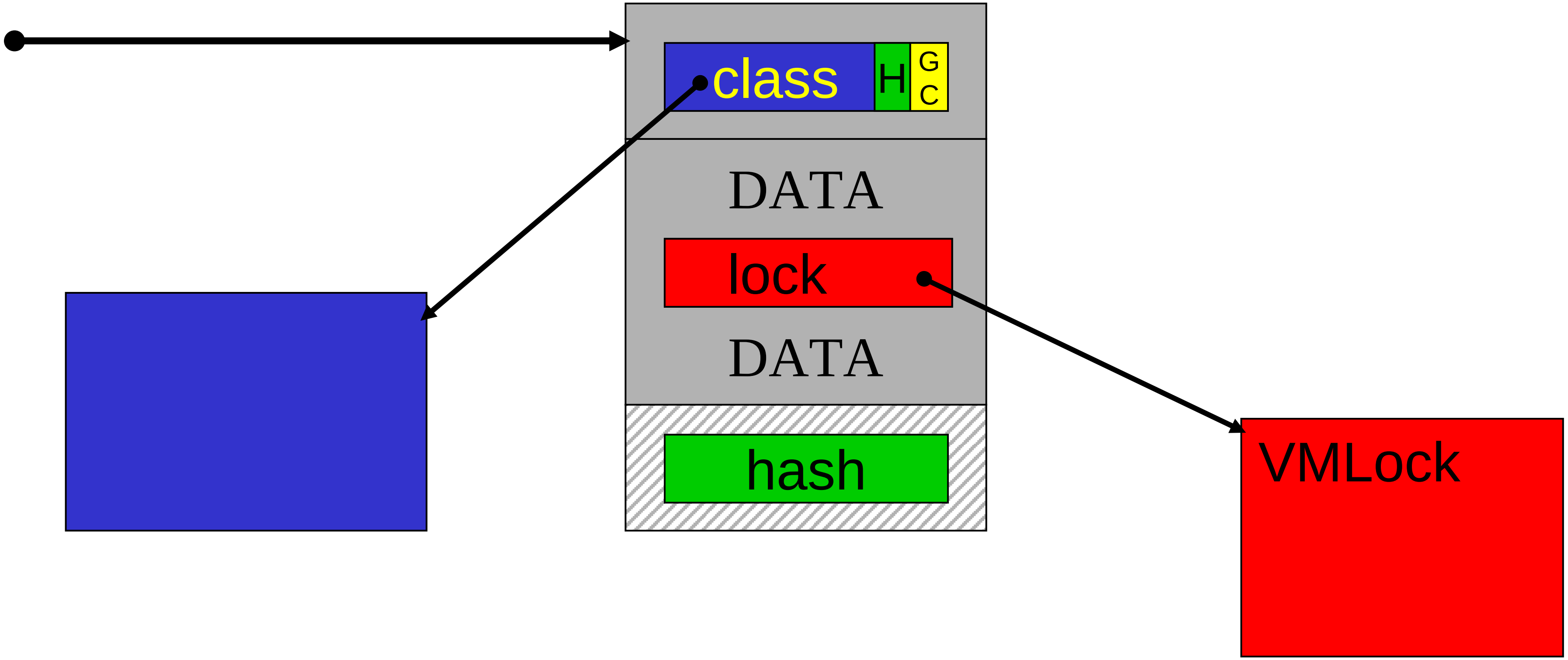
Object Space

Lock Table

Jikes RVM 1-Word Masked (no lock)



Jikes RVM Single-Word Masked (with lock)



Object Header Summary

- Object headers vary from 1-3+ words
- Many approaches to header compression
- Invariants can be quite subtle
  - Whose updating which bits? When? Atomic?

Dynamic Type Checking

- Explicit type tests (instanceof)

```
b instanceof A;
```

- Down casts (checkcast)

```
(A) b;
```

- Interface method invocation (invokeinterface)

```
I i = b; // I an interface type
i.foo();
```

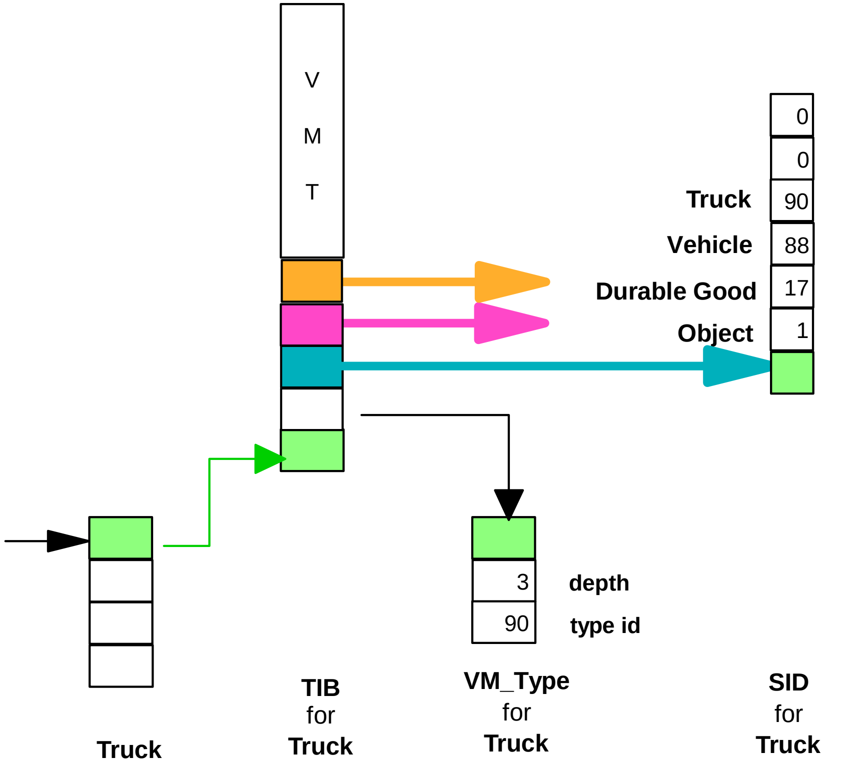
- Exception delivery (athrow)

```
try { . . .} catch (A a) { . . . }
```

- Object array stores (aastore)

```
DeclaredType [] X = ......
X[3] = b;
```

Superclass Identifier Display



Dynamic type check:

```
l       r1, TIBoffset(b)
l       r1, SIDoffset(r1)
l       r1, A_depth<<1(r1)
cmpi r1, A_id
bne   NoMatch
```

Dynamic type check:

```
l       r1, TIBoffset(b)
l       r1, SIDoffset(r1)
l       r2, lengthOffset(r1)
cmpi r2, A_depth
bge   NoMatch
l       r1, A_depth<<1(r1)
cmpi r1, A_id
bne   NoMatch
```

Each type has a depth and a type id

SID for a type is an array of shorts

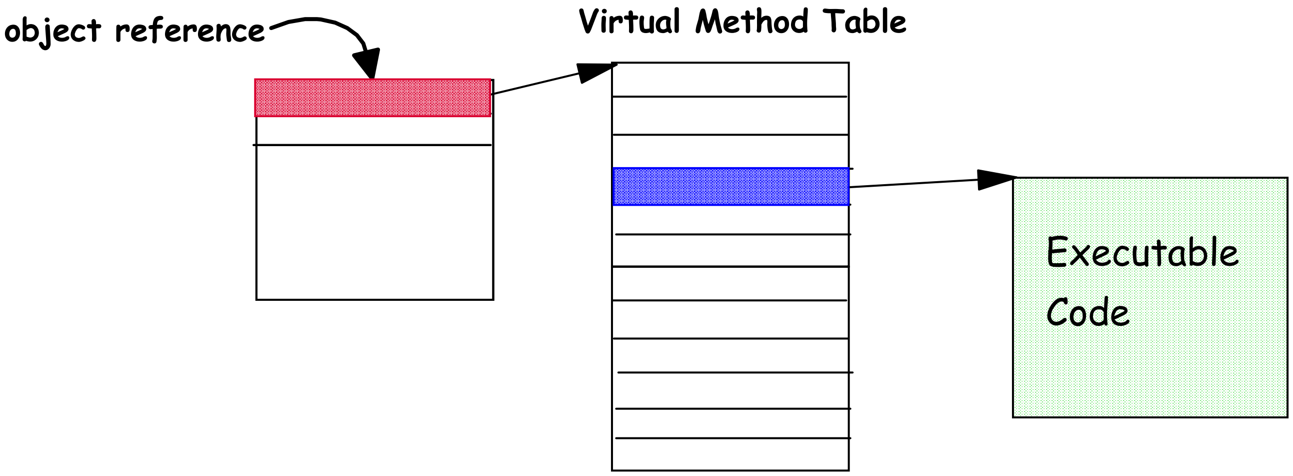
- Maps superclass depth to superclass type id
- Padded to a minimum depth with invalid ids

Method Dispatch

- invokevirtual
  - Single inheritance, statically typed
  - VFTs (Virtual Function Tables) most common
  - PICs (Polymorphic Inline Caches) also used
- invokeinterface
  - dynamic type check
  - Effectively multiple inheritance
  - Large number of schemes

Virtual Method Dispatch

- Single inheritance: virtual method dispatch tables

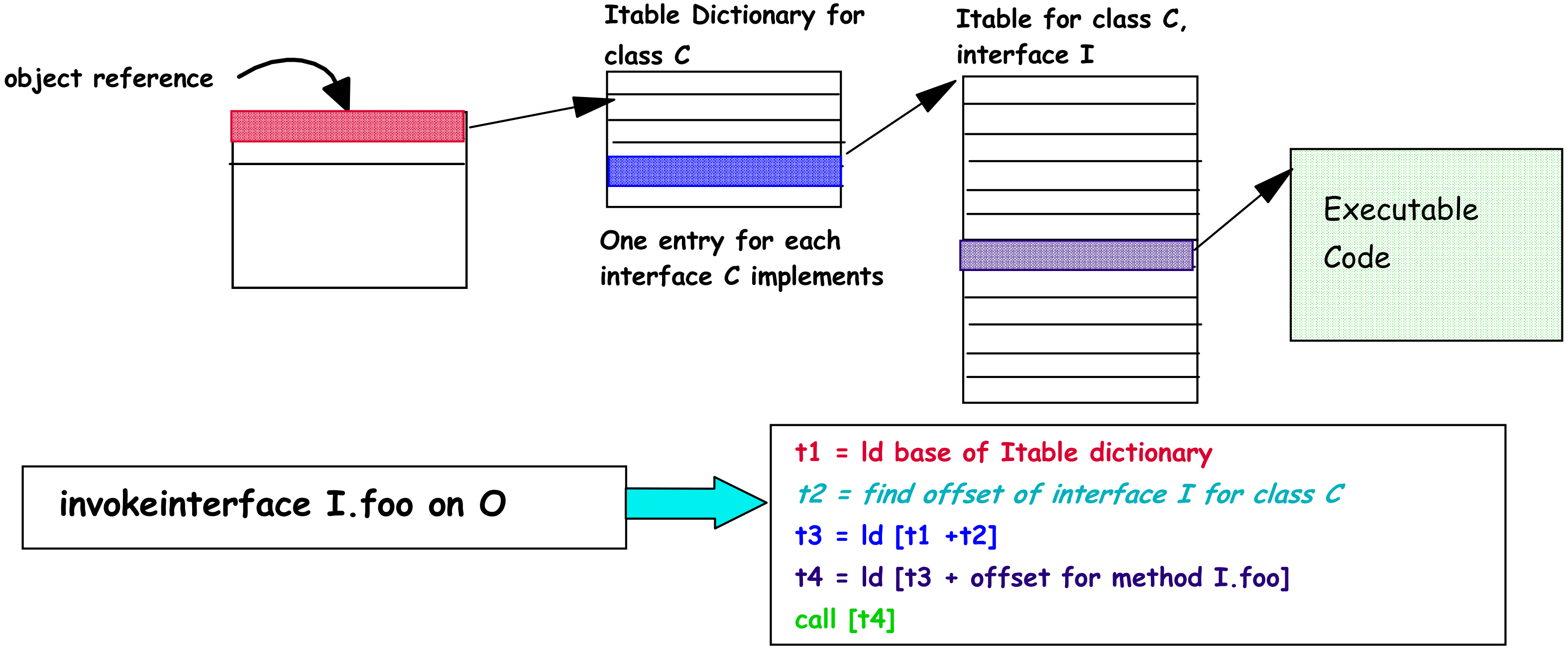


invokevirtual A.foo on O

```
t1 = ld base of VMT
t2 = ld [t1 + offset for A.foo]
call [t2]
```

Interface Table Dispatch: *Search* Variant

- Variant 2: Packed dictionary of itables [eg. Fitzgerald et al. 99]



object reference

Itable Dictionary for class C

One entry for each interface C implements

Itable for class C, interface I

Executable Code

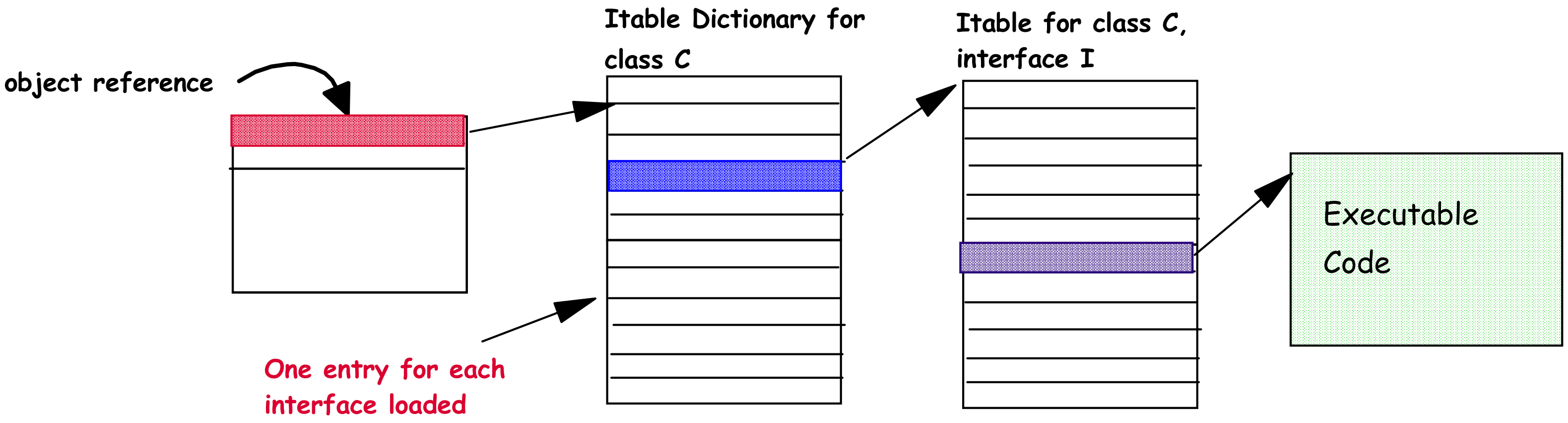
invokeinterface I.foo on O

```
t1 = ld base of Itable dictionary
t2 = find offset of interface I for class C
t3 = ld [t1 +t2]
t4 = ld [t3 + offset for method I.foo]
call [t4]
```

- Drawback: *need to find offset t2 of interface I for class C*
  - Sometimes, compiler can determine t2 statically (should virtualize call)
  - If not, need dispatch time search (cache, binary or linear) to find t2

Interface Table Dispatch: *Direct* Variant

- Variant 1: Array of itables [Krall & Grafi 97]



object reference

Itable Dictionary for class *C*

Itable for class *C*, interface I

Executable Code

One entry for each interface loaded

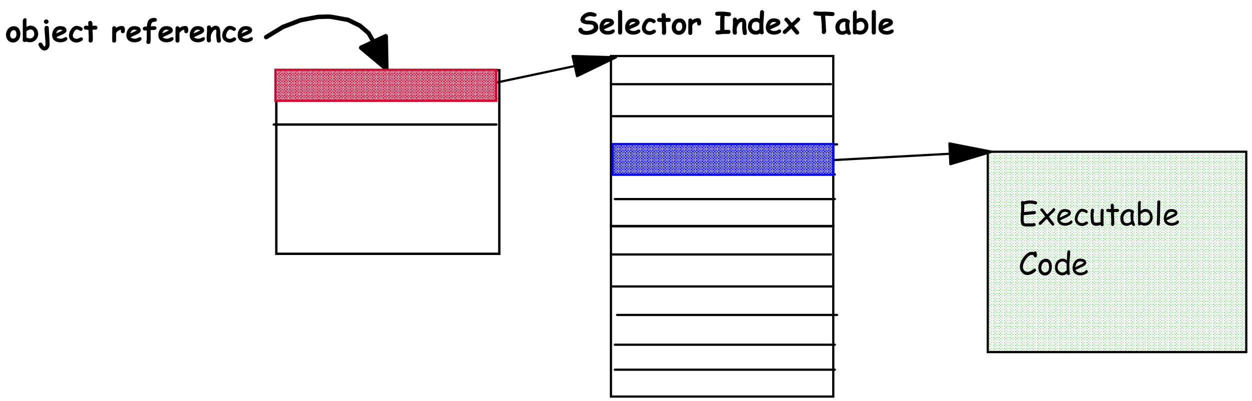
invokeinterface I.foo on O

```
t1 = ld base of Itable dictionary
t2 = ld [t1 + offset for interface I]
t3 = ld [t2 + offset for method I.foo]
call [t3]
```

- Drawbacks compared to virtual call
  - One extra indirection on dispatch
  - Space overhead of Itable dictionaries and ITables

Selector Index Tables

- Interface dispatch has same runtime cost as virtual dispatch



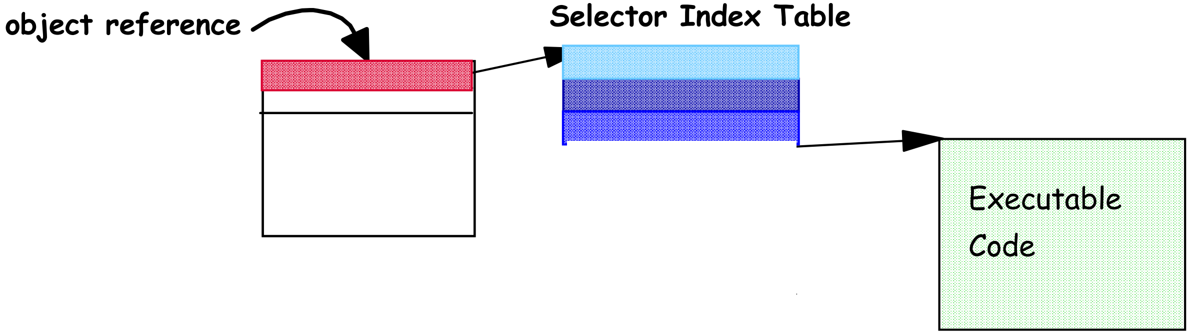
invokeinterface I.foo on O

```
t1 = ld base of SIT
t2 = ld [t1 + offset for I.foo]
call [t2]
```

- Selector Index Table
  - One entry for every interface method signature loaded
  - Very space-inefficient

Selector Index Table Coloring

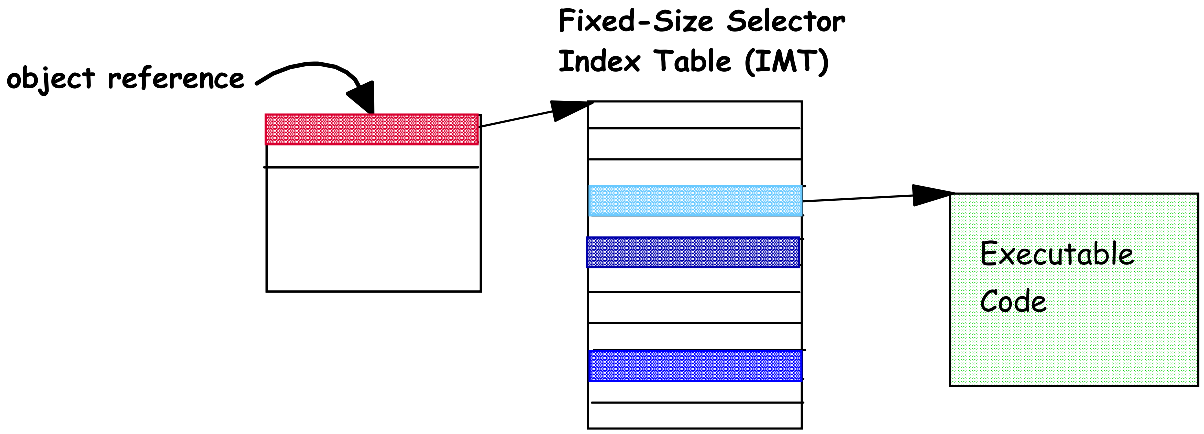
- Note that Selector Index Table for a class is sparse
  - A class only implements some interfaces
  - [Dixon et al. 89]:color interface methods for each class and pack densely



- Coloring problems for Java:
  - Must know all the classes *a priori*

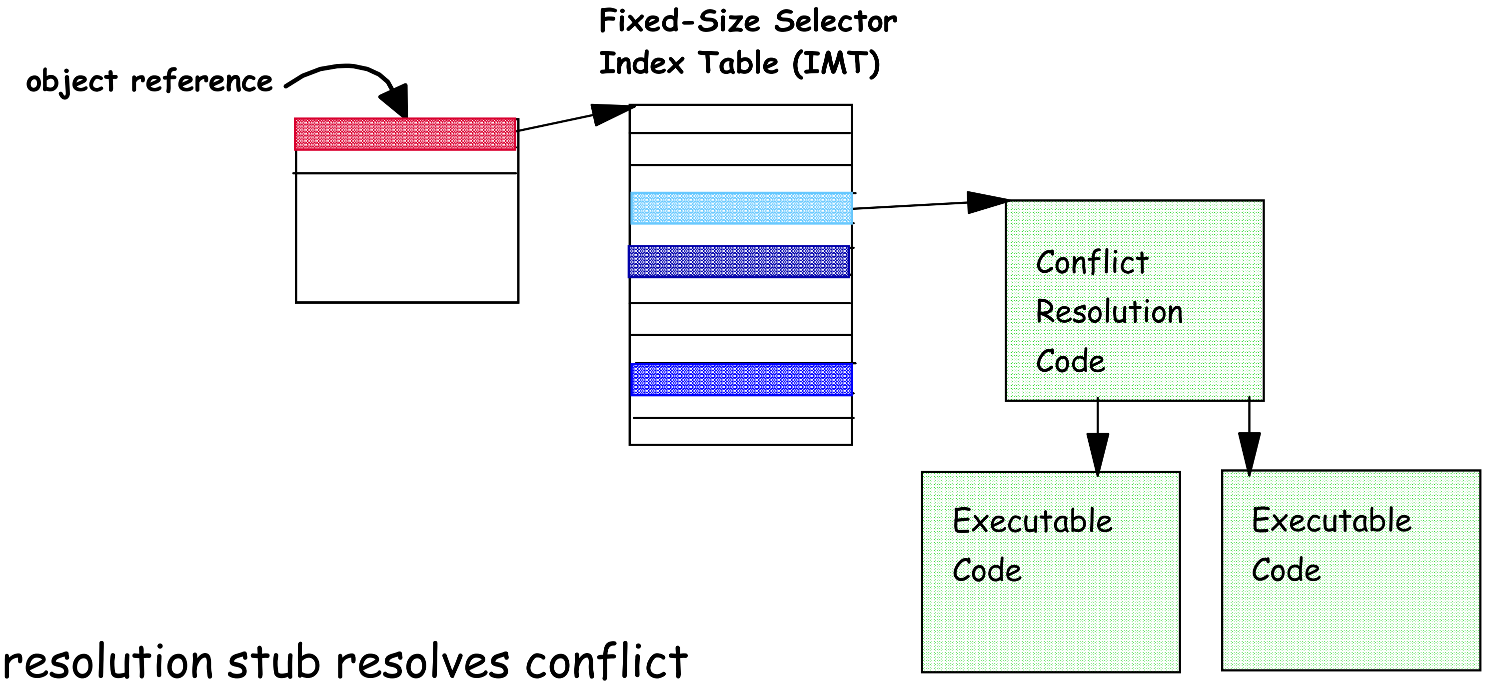
Jikes RVM IMT Solution

- Fixed-size selector-index tables (Interface Method Table (IMT))
- Map interface method to known IMT slot with hash function



- Don't need to know interfaces a priori
- Common case: interface dispatch almost same sequence as virtual dispatch
- Potential drawback: conflicts in mapping to IMT

Jikes RVM IMT Solution



- Conflict resolution stub resolves conflict
- Hidden parameter identifies proper method to dispatch
- Conflict resolution stub generated when conflict detected

  i.e. during interface loading
- Extra instruction in caller to set up hidden parameter
- Extra space for IMT and conflict resolution code (if required)

Summary of Typechecking & Dispatch

- Lots of options, prior + future research
- VM core, classloading, JIT
    - Act in concert to maintain invariants,
    - Very specific knowledge needed by (some portion of) all subsystems
- Memory model can matter (PPC vs. x86)
- Operations are so common, you have to do a very good job
- Must evaluate on very large benchmarks (jvm98, jbb too easy)

Discussion

- Object model cross-cuts, but can be encapsulated
- Object model evolution is inevitable, but often quite painful (even in well-designed systems)
- “Common” model of header + data won’t handle everything you might want to do
  - Arraylets
  - Split objects
  - Handles
  - Highly compressed pointers & values