Field Offsets

```java
class Shape {
    Point LL /* 4 */ , UR /* 8 */;
    void setCorner(int which, Point p);
}
class ColoredRect extends Shape {
    Color c /* 12 */;
    void setColor(Color c_);
}  
• Offsets of fields from beginning are same for all subclasses
• Accesses to fields are indexed loads
  ColoredRect x;
  \$[x.c] = MEM(\$[x] + 12)
  \$[x.UR] = MEM(\$[x] + 8)
• Need to know size of superclasses – can be a problem
  • e.g., Java – field offsets resolved at dynamic link/load time
```

Field Alignment

• In many processors, a 32-bit load must be to an address divisible by 4, address of 64-bit load must be divisible by 8
• In rest (e.g. Pentium), loads are 10× faster if aligned -- avoids extra load

⇒ Fields should be aligned

```c
struct {
    int x; char c; int y; char d;
    int z; double e;
}
```

Multiple Inheritance

• Mechanism: a class may declare multiple superclasses (C++)
• Java: may implement multiple interfaces, may inherit code from only one superclass
• Two problems: multiple supertypes, multiple superclasses
• What are implications of multiple supertypes in compiler?
### Semantic problems

- **Problem 1: ambiguity**
  
  ```java
  class A { int m(); }
  class B { int m(); }
  class C extends A, B {} // which m?
  ```

- **Problem 2: field replication**

```
  class A { int x; }
  class B1 extends A { ... }
  class B2 extends A { ... }
  class C extends B1, B2 { ... }
```

- All methods, fields must be uniquely defined

### Dispatch vectors break

```java
interface Shape {
    void setCorner(int w, Point p); // 0
}
interface Color {
    float get(int rgb); // 0
    void set(int rgb, float value); // 1
}
```

```
class Blob implements Shape, Color { ... }
```}

### DV alternatives

- **Option 1: search with inline cache**  
  (Smalltalk, Java)

  - For each class, interface, have table mapping method names to method code. Recursively walk upward in hierarchy looking for method name

  - **Optimization**: at call site, store class and code pointer in call site code (**inline caching**). On call, check whether class matches cache.

### Inline cache code

- Let \( t_o \) be the receiver object:

  ```asm
  mov t1, [t_o]
  cmp t1, [cacheClass434]
  jnz miss
  call [cacheCode434]
  miss: call slowDispatch
  ```

  ![Inline cache code diagram](cache data (in data segment) object object class information)

  90% of calls from a site go to same code as last call from same site
Option 2: Sparse dispatch vectors

- Make sure that two methods never allocated same offset: give Shape offset 0, Color offsets 1 and 2. Allow holes in DV!
- Some methods can be given same offset since they never occur in the same DV
- Graph coloring techniques can be used to compute method indices in reasonably optimal way (finding optimum is NP-complete!)

Sparse Dispatch Vectors

```java
interface Shape {
    void setCorner(int w, Point p); 0
}
interface Color {
    float get(int rgb); 1
    void set(int rgb, float value); 3
}
class Blob implements Shape, Color { … }
```

- Advantage: same fast dispatch code as SI case
- Disadvantage: requires knowledge of entire type hierarchy (makes separate compilation, dynamic loading difficult)

Option 3: Hash tables

- Idea: don’t try to give all method unique indices; resolve conflicts by checking that entry is correct at dispatch
- Use hashing to generate method indices
  - Precompute hash values!
  - Some Java implementations

```java
interface Shape {
    void setCorner(int w, Point p); 11
}
interface Color {
    float get(int rgb); 4
    void set(int rgb, float value); 7
}
class Blob implements Shape, Color { … }
```

- What if there’s a conflict? Entries containing several methods point to resolution code
- Basic dispatch code is (almost) identical!
- Advantage: simple, reasonably fast
- Disadvantage: some wasted space in DV, extra argument for resolution, slower dispatch if conflict
Option 5: Binary decision trees

- Idea: use conditional branches, not indirect jumps
- Unique class index stored in first object word
- Range tests used to select among $n$ possible classes at call site in $\log n$ time – direct branches to code

Shape x;
x.SetCorner(...)
mov ebx, [eax]
cmp ebx, 1
jle L1
cmp ebx, 2
je Circle$setCorner
jmp Egg$setCorner

L1: cmp ebx, 0
je Blob$setCorner
jmp Rect$setCorner

Binary decision tree

- Works well if distribution of classes is highly skewed: branch prediction hardware eliminates branch stall of ~10 cycles
  - Can use profiling to identify common paths for each call site individually
  - 90%/10% : usually a common path to put at top of decision tree
- Like sparse DVs: need whole-program analysis
- Indirect jump can have better expected execution time for >2 classes: at most one mispredict