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
  - `o : ColoredRect E[o.c][] = MEM(E[o] + 12)`
  - `E[o.UR][] = MEM(E[o] + 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, and an address of a 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

```
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
  - class A { int m(); }
  - class B { int m(); }
  - class C extends A, B {} // which m?
- All methods/fields must be uniquely defined
- Problem 2: field replication
  - class A { int x; }
  - class B1 extends A { ... }
  - class B2 extends A { ... }
  - class C extends B1, B2 { ... }

Dispatch tables break

```
interface Shape {
    void setCorner(int w, Point p); 0
}

interface Color {
    float get(int rgb); 0
    void set(int rgb, float value); 1
}
```

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

  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 at 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 Mass {
    void setWeight(int kg); 1
    
} interface Color {
    float get(int rgb); 2
    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 {...}
  ```

**Dispatch with Hash Tables**

- 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
```

Option 6: Interface Table

V-Table links to array of interfaces
- scan through array for target interface
- retrieve offset and add to v-table address
Embed interface method tables inside v-table
- all methods occur in v-table anyways
- may force method to occur twice in v-table

```
V-Table (Blob fields)
I-Table get setCorner (Blob methods)
Shape 3 Color 1
```

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

```
Color  RGBColor  Blob  Rectangle  Circle  Egg
3   0     1   2   4
```

Interface Table

- Disadvantages
  - Linear-time lookup
- Advantages
  - Linear in # of interfaces implemented by an object (usually very small)
  - Good cache behavior
  - Only needs to be done once per interface
    - then constant per method!
  - Interface table can be updated dynamically