Optimizing Compilers

Optimizations for Object-Oriented Languages

Markus Schordan

Institut für Computersprachen
Technische Universität Wien

Overview

- Object layout and method invocation
  - Single inheritance
  - Multiple Inheritance
- Devirtualization
  - Class hierarchy analysis
  - Rapid type analysis
  - Inlining
- Escape Analysis
  - Connection graphs
  - Intra-procedural
  - Inter-procedural

Object Layout and Method Invocation

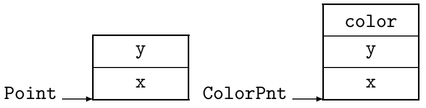
The memory layout of an object and how the layout supports dynamic dispatch are crucial factors for performance.

- Single Inheritance
  - with and without virtual dispatch table
- Multiple Inheritance
  - embedding superclasses
  - trampolines
  - table compression

Single Inheritance Layout

```
class Point {
    int x, y;
}
```

```
class ColorPnt extends Point {
    int color;
}
```

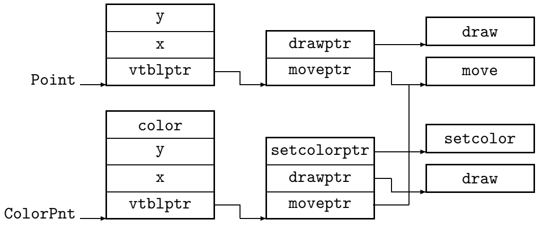


- Memory layout of an object of a superclass is a prefix of the memory layout of an object of the subclass
- Instance variables access requires just one load or store instruction

Single Inheritance Layout with vtbl

```
class Point {
  int x, y;
  void move(int x, int y) {...}
  void draw() {...}
}
```

```
class ColorPnt extends Point {
  int color;
  void draw() {...}
  void setcolor(int c) {...}
}
```



Invocation of Virtual Methods with vtbl

- Dynamic dispatching using a vtbl has the advantage of being fast and executing in constant time.
- It is possible to add new methods and to override methods
- Each method is assigned a fixed offset in the virtual method table (vtbl)
- Method invocation is just three machine code instructions

```
LDQ vtblptr,(obj)           ; load vtbl pointer
LDQ mptr,method(vtblptr)    ; load method pointer
JSR (mptr)                  ; call method
```

- One extra word of memory is needed in each object for the pointer to the virtual method table (vtbl)

Dispatch Without Virtual Method Tables

Despite the use of branch target caches, indirect branches are expensive on modern architectures.

The pointer to the class information and virtual method table is replaced by a type identifier:

- A type identifier is an integer representing the type of the object
- It is used in a dispatch function which searches for the type of the receiver
- Example: SmallEiffel (binary search)
- Dispatch functions are shared between calls with the same statically determined set of concrete types
- In the dispatch function a direct branch to the dispatched method is used (or it is inlined)

Example

Let type identifiers T_A, T_B, T_C, and T_D be sorted by increasing number. The dispatch code for calling $x.f$ is:

if $id_x \leq T_B$ **then**
  if $id_x \leq T_A$ **then** $f_A(x)$
  else $f_B(x)$
else if $id_x \leq T_C$ **then** $f_C(x)$
  else $f_D(x)$

Comparison with dispatching using a virtual method table

- Empirical study showed that for a method invocation with three concrete types, dispatching with binary search is between 10% and 48% faster
- For a megamorphic call with 50 concrete types, the performance is about the same

Multiple Inheritance

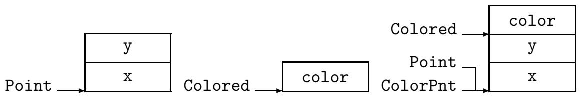
- Extending the superclasses as in single inheritance does not work anymore
- Fields of superclass are embedded as contiguous block
- Embedding allows fast access to instance variables exactly as in single inheritance
- Garbage collcection becomes more complex because pointers also point into the middle of objects

Object Memory Layout (without vtbl)

```
class Point {
    int x, y;
}
```

```
class Colored {
    int color;
}
```

```
class ColorPnt extends Point, Colored {
}
```



Dynamic Dispatching for Embedding

- Allows fast access to instance variables exactly as with single inheritance
- For every superclass
  - virtual method tables have to be created
  - multiple *vtbl* pointers are included in the object
- The object pointer is adjusted to the embedded object whenever explicit or implicit pointer casting occurs (assignments, type casts, parameter and result passing)

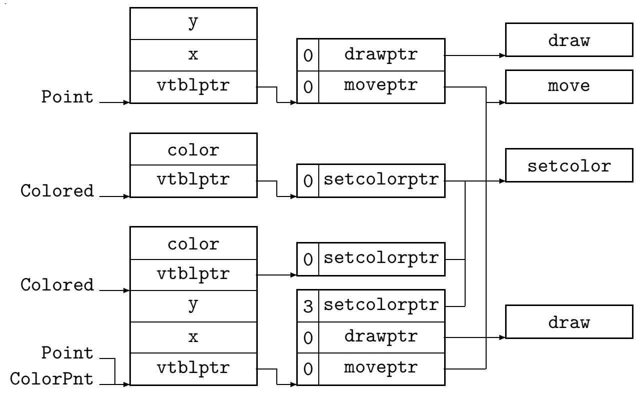
Multiple Inheritance with vtbl

```
class Point {
    int x, y;
    void move(int x, int y) {...}
    void draw() {...}
    }

class Colored {
    int color;
    void setcolor(int c) {...}
    }

class ColorPnt extends Point, Colored {
    void draw() {...}
    }
```

Multiple Inheritance with vtbl



Pointer Adjustment and Adjustment Offset

Pointer adjustment has to be suppressed for casts of null pointers:

```
Colored col; ColorPnt cp; ...;
col = cp;  // if (cp!=null)col=(Colored)((int*)cp+3)
```

Problem with implicit casts from actual receiver to formal receiver

- Caller has no type info of formal receiver in the callee
- Callee has no type info of actual receiver of the caller
- Therefore this type info has to be stored as an adjustment offset in the vtbl

Method Invocation with vtbl

Method invocation now takes 4 to 5 machine instructions (depending on the architecture).

```
LD  vtblptr,(obj)            ; load vtbl pointer
LD  mptr,method_ptr(vtblptr) ; load method pointer
LD  off,method_off(vtblptr)  ; load adjustment offset
ADD obj,off,obj              ; adjust receiver
JSR (mptr)                   ; call method
```

This overhead in table space and program code is even necessary when multiple inheritance is not used (in the code).

Furthermore, adjustments to the remaining parameters and the result are not possible.

Trampoline

To eliminate much of the overhead a small piece of code, called trampolin is inserted that performs the pointer adjustments and the jumps to the original code.

The advantages are

- smaller table size (no storing of an offset)
- fast method invocation when multiple inheritance is not used
  - the same dispatch code as in single inheritance

The method pointer `setcolorptr` in the virtual method table of `Colorpoint` would (instead) point to code which adds 3 to the receiver before jumping to the code of method `setcolor`:

```
ADD obj,3,obj                ; adjust receiver
BR  setcolor                 ; call method
```


Lookup at Compile-Time

Invoking a method requires looking up the address of the method and passing control to it.
In some cases, the lookup may be performed at compile-time:

- There is only one implementation of the method in the class and its subclasses
- The language provides a declaration that forces the call to be non-virtual
- The compiler has performed static analysis that can determine that a unique implementation is *always* called at a particular call site.

In other cases, a runtime lookup is required.

Dispatch Table

In principle the lookup can be implemented as indexing a two-dimensional table. A number is given to

- each method in the program
- each class in the program

The method call

```
result = obj.m(a1,a2);
```

can be implemented by following three actions:

1. Fetch a pointer to the appropriate row of the dispatch table from the object `obj`.
2. Index the dispatch table row with the method number.
3. Transfer control to the address obtained.

Dispatch Table Compression (1)

- Virtual Tables
  - effective method for statically typed languages
  - methods can be numbered compactly for each class hierarchy to leave no unused entries in each vtbl
- Row Displacement Compression
  - idea: combine all rows into a single very large vector
  - it is possible to have rows overlapping as long as an entry in one row corresponds to empty entries in the other rows
  - greedy algorithm: place first row; for all subsequent rows: place on top and shift right if conflicts exist.
  - unchanged: implementation of method invocation
  - penalty: verify class of current object at the beginning of any method that can be accessed via more than one row

Dispatch Table Compression (2)

- Selector Coloring Compression
  - graph coloring: two rows can be merged if no column contains different method addresses for the two classes
  - graph: one node per class; an edge connects two nodes if the corresponding classes provide different implementations for the same method name
  - coloring: each color corresponds to the index for a row in the compressed table
  - each object contains a reference to a possibly shared row
  - unchanged: implementation of method invocation code
  - penalty: if classes C1 and C2 share the same row and C1 implements method m whereas C2 does not, then the code for m should begin with a check that control was reached via dispatching on an object of type C1.

Devirtualization

Devirtualization is a technique to reduce the overhead of virtual method invocation.

The aim of this techique is to statically determine which methods can be invoked by virtual method calls.

- If exactly one method is resolved for a method call, the method can be inlined or the virtual method call can be replaced by a static method call.

The analyses necessary for devirtualization also improve the accuracy of the call graph and the accuracy of subsequent interprocedural analyses.

Class Hierarchy Analysis

The simplest devirtualization technique is class hierarchy analysis (CHA), which determines the class hierarchy *used* in a program.

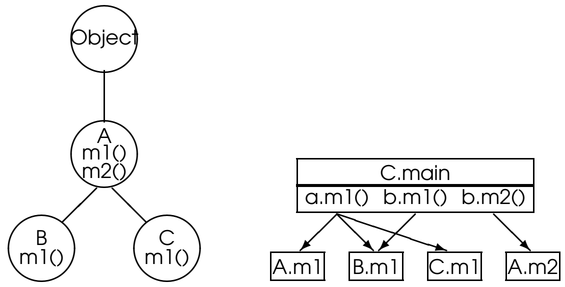
The information about all referenced classes is used to create a conservative approximation of the class hierarchy.

- The transitive closure of all classes referenced by the class containing the main method is computed.
- The declared types of the receiver of a virtual method call are used for determining all possible receivers

Example: Class Hierarchy Analysis

```
class A extends Object {
    void m1() {...}
    void m2() {...}
    }
class B extends A {
    void m1() {...}
    }
class C extends A {
    void m1() {...}
    public static void main(...) {
        A a = new A();
        B b = new B();
        ...
        a.m1(); b.m1(); b.m2();
        }
    }
```


Example: Class Hierarchy and Call Graph



CHA Algorithm

$main$	// the main method in a program
$x()$	// call of static method x
$type(x)$	// the declared type of the expression x
$x.y()$	// call of virtual method y in expression x
$subtype(x)$	// x and all classes which are a subtype of class x
$method(x, y)$	// the method y which is defined for class x

$callgraph := main$
$hierarchy := \{\}$
for each $m \in callgraph$ **do**
    for each $m_{stat}()$ occuring in m **do**
        if $m_{stat} \notin callgraph$ **then**
            add m_{stat} to $callgraph$
    for each $e.m_{vir}()$ occuring in m **do**
        for each $c \in subtype(type(e))$ **do**
        $m_{def} := method(c, m_{vir})$
        if $m_{def} \notin callgraph$ **then**
            add m_{def} to $callgraph$
            add c to $hierarchy$

Rapid Type Analysis (1)

Rapid type analysis uses the fact that a method m of a class c can be invoked only if an object of type c is created during the execution of the program.

- It refines the class hierarchy (compared to CHA) by only including classes for which objects can be created at *runtime*.

1. The pessimistic algorithm includes all classes in the class hierarchy for which instantiations occur in methods of the call graph from CHA.

Rapid Type Analysis (2)

2. The optimistic algorithm

- Initially assumes that no methods besides *main* are called and that no objects are instantiated.
- It traverses the call graph initially ignoring virtual calls (marking them in a mapping as potential calls only) following static calls only.
- When an instantiation of an object is found during analysis, all virtual methods of the corresponding objects that were left out previously are then traversed as well.
- The live part of the call graph and the set of instantiated classes grow interleaved as the algorithm proceeds.

Escape Analysis

The goal of escape analysis is to determine which objects have lifetimes which do not stretch outside the lifetime of their immediately enclosing scopes.

- The storage for such objects can be safely allocated as part of the current stack frame – that is, their storage can be allocated on the run-time stack.
- The transformation also improves the data locality of the program and, depending on the computer's cache, can significantly reduce execution time.

Objects whose lifetimes are confined to within a single scope cannot be shared between two threads.

- Synchronization actions for these objects can be eliminated.

Escape Analysis by Abstract Interpretation

A prototype implementation of escape analysis was included in the IBM High Performance Compiler for Java.

The approach of Choi et al. attempts to determine whether the object

- escapes from a method (i.e. from the scope where it is allocated)
- escapes from the thread that created it
  - the object can escape a method but does not escape from the thread
  - the converse is not possible (if it does not escape the method then it cannot escape the thread)

Escape States

The analysis uses a simple lattice to represent different escape states:

NoEscape ($\top$)
|
ArgEscape
|
GlobalEscape ($\bot$)

State	escapes the method	escapes the thread
NoEscape	no	no
ArgEscape	may (via args)	no
GlobalEscape	may	may

Connection Graphs

We are interested only in

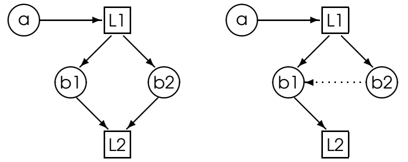
- following the object O from its point of allocation
- knowing which variables reference O
- and which other objects are referenced by O fields.

We "abstract out" the referencing information, using a graph structure where

- a circle node represents a variable
- a square node represents objects in the heap
- an edge from circle to square represents a reference
- an edge from square to circle represents ownership of fields

Example: Connection graphs

```
A a = new A();   // line L1
a.b1 = new B();  // line L2
a.b2 = a.b1;     // line L3
```



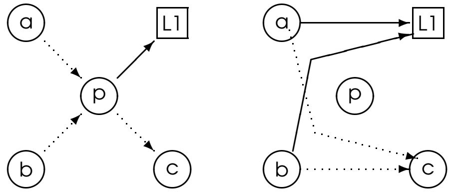
Simple Version *Using Deferred Edges*

An edge drawn as a dotted arrow is called a deferred edge and shows the effect of an assignment from one variable to another (example: created by the assignment in line 3)

Intraprocedural Abstract Interpretation

Actions for assignments involve an update of the connection graph.

- An assignment to a variable p kills any value the variable previously had. The kill function is called byPass(p):



Before *After*

Analyzing Statements (1)

p = new C(); // line L The operation $byPass(p)$ is applied. An object node labeled L is added to the graph - and nodes for the fields of C that have noninstrinsic types are also created and connected by edges pointing from the object node.

p = q; The operation $byPass(p)$ is applied. A new deferred edge from p to q is created.

p.f = q; The operation $byPass$ is *not* applied for f (no strong update!). If p does not point to any node in the graph a new (phantom) node is created. Then, for each object node connected to p by an edge, an assignment to the field f of that object is performed.

Analyzing Statements (2)

p = q.f; If q does not point at any object node then a phantom node is created and an edge from q to the new node is added. Then $byPass(p)$ is applied and deferred edges are added from p to all the f nodes that q is connected to by field edges.

For each statement one graph represents the state of the program at the statement.

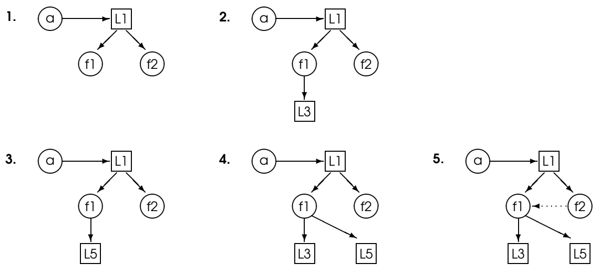
At a point where two or more control paths converge, the connection graphs from each predecessor statements are merged.

Example: Connection Graphs (1)

Suppose that the code inside some method is as follows. The declarations of classes `A`, `B1` and `B2` are omitted.

```
A a = new A();       // line L1
if (i > 0)
   a.f1 = new B1();  // line L3
else
   a.f1 = new B2();  // Line L5
a.f2 = a.f1;         // Line L6
```

Example: Connection Graphs (2)



G_1: out: A a = new A(); // line L1
G_2: out: a.f1 = new B1(); // line L3
G_3: out: a.f1 = new B2(); // Line L5
G_4: out: $G_2 \cup G_3$
G_5: out: a.f2 = a.f1; // Line L6

Interprocedural Abstract Interpretation (1)

Analyzing methods:

- It is necessary to analyze each method in the reverse order implied by the *call graph*.
- If method A may call methods B and C, then B and C should be analyzed before A.
- Recursive edges in the call graph are ignored when determining the order.
- Java has virtual method calls – at a method call site where it is not known which method implementation is being invoked, the analysis must assume that all of the possible implementations are called, combining the effects from all the possibilities.
- The interprocedural analysis iterates over all the methods in the call graph until the results converge (fixed point)

Interprocedural Abstract Interpretation (2)

- A call to a method *M* is equivalent to copying the actual parameters (i.e. the arguments being passed in the method call) to the formal parameters, then executing the body of *M*, and finally copying any value returned by *M* as its result back to the caller.
- If *M* has already been analyzed intraprocedurally following the approach described above, the effect of *M* can be summarized with a connection graph. That summary information eliminates the need to re-analyze *M* for each call site in the program.

Analysis Results (1)

After the operation $byPass$ has been used to eliminate all deferred edges, the connection graph can be partitioned into three subgraphs:

Global escape nodes: All nodes reachable from a node whose associated state is *GlobalEscape* are themselves considered to be global escape nodes (Subgraph 1)

- the nodes initially marked as *GlobalEscape* are the static fields of any classes and instances of any class that implements the `Runnable` interface.

Argument escape nodes: All nodes reachable from a node whose associated state is *ArgEscape*, but are not reachable from a *Global Escape* node. (Subgraph 2)

- the nodes initially marked as *ArgEscape* are the argument nodes $a_1, \ldots, a_n$.

Analysis Results (2)

No escape nodes: All other nodes have *NoEscape* status. (Subgraph 3).

The third subgraph represents the summary information for the method because it shows which objects can be reached via the arguments passed to the method

All objects created within a method M and that have the *NoEscape* status after the three subgraphs have been determined can be safely allocated on the stack.

References

- Material for this 6th lecture

  `www.complang.tuwien.ac.at/markus/optub.html`
- Book

  Y. N. Srikant, Priti Shankar:

  The Compiler Design Handbook: Optimizations & Machine Code Generation

  CRC Press; 1st edition, (928 pages, ISBN: 084931240X), 2002.
  - Chapter 6 (Optimizations for Object-Oriented Languages)
- J-G Choi, M. Gupta, M. Serrano, V.C Sreedar, and S. Midkiff Escape Analysis for Java, in Proceedings of OOPSLA'99, ACM Press, pp.1-19.