Whole Program Optimization of OOPLs
C. Chambers, J. Dean, D. Grove

University of Washington
Technical Report 96-06-02

see also OOPSLA'96 paper on Vortex

Vortex

- **Cecil: OOPL with multimethods using dynamic dispatch, classes, inheritance, closures**
- **Vortex: optimizing compiler written in Cecil, emphasizes optimizations for OOPLs**
  - **Performs whole program analysis and transformation**
- **Goal: to reduce runtime performance costs of OO style through static analysis based and dynamic profiling guided techniques**

Vortex

- Static analyses
  - Intraprocedural class analysis
    - for compile-time method resolution and elimination of runtime class checks
  - Class hierarchy analysis
    - to use with exhaustive class testing and cloned codes
- Execution time profiling
  - Frequency counts
    - for prediction with cloning or splitting code
  - Selective method specialization for argument subsets

Intraprocedural Class Analysis

- A flow-sensitive, forward intraproc data-flow analysis
  - Map M: Variables $\rightarrow$ power set of Classes
  - Safety requires if M(x) = S, then S contains all the possible runtime classes of x
- Details
  - Need to define transfer functions at control flow graph nodes
  - Use fixed point iteration to solve

Interprocedural Class Analysis

- **result(x) =**
  - **x an operator, predefined class**
  - **x a message, declared return type T and all of its subclasses (cone(T))**
  - **x an instance variable of type Q, cone(Q)**
- **Assume Unknown is set of all classes**
- **Let Class be the set of incoming (reference, set of types) pairs**
- **Then**
  - **x := const** **class[x → class of const]**
  - **x:= new C** **class[x → {C}]**
  - **x:= y** **class[x → class(y)]**
  - **x:= y.foo(…)** **class[x → result(y.foo(…))]**
  - **x:= obj.var** **class[x → result(obj.var)]**
- **At control merges, union classes associated with same variable; at runtime type tests, propagate narrowed type information forward**

Comments

- **Existence of cone(C) as a type impedes optimization**
- **Can improve analysis by interleaving it with inlining**
- **Automatic inlining performed with heuristics about size of callee, expected call frequncy and whether or not function is recursive**
  - **User directives allowed**

Experiments in Inlining

- **Performed inlining speculatively**
  - **Measured optimization benefits**
  - **Stored persistently static information used in opts**
  - **Reused this cost/benefit info at different call sites with same characteristics**
  - **Results showed inlining decisions less sensitive to superficial changes in source code's structure and to thresholds set to choose inlining candidates**
- ***Research Issue: when and how to inline?***

Class Hierarchy Analysis

- **Presumes you have whole program**
- **Complication of interprocedural analysis of OOPLs: cyclic dependency between structure of call graph and static class info inferred for receivers**
- **Augments intraprocedural class analysis with knowledge of inheritance structure**
  - **Can bound Cone(Q)**

Class Hierarchy Analysis

- **Goal: program only pays for dynamic dispatch insofar as it uses it**
  - **E.g., Extensible libraries incur a compilation cost only when extended**
  - **Problems**
    - **Union and Cone(Q) may involve a bounded enumerable set of classes**
    - **Even if object type is a set of classes there may be only one possible method called**
    - **Using Vortex as data, over 50% messages that can be statically bound with CHA have more than 1 class in the receiver's type**

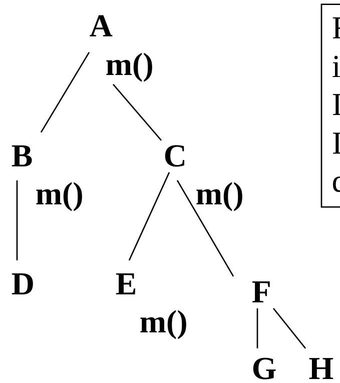
Class Hierarchy Analysis

- **Build inheritance tree for whole program and decorate with methods and their signature**
- **For each class C in tree and each relevant method f, can compute at compile time, which f is called by a C receiver**
- **Now, *compile-time method resolution* involves finding the possible type(s) of the receiver and unioning all possible methods for those types, hoping for a singleton method**

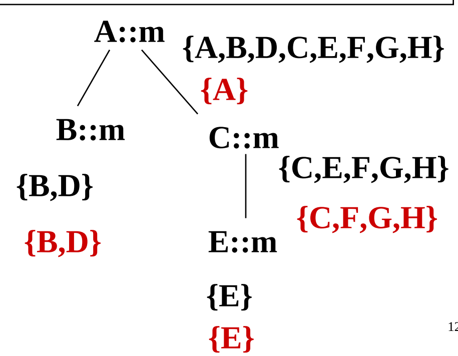
Class Hierarchy Analysis

- **Problem: need quick lookup of appropriate method for non-singleton receiver classes**
  - **Precompute *applies-to set* of method f: set of classes that resolve to this method**
  - **At method call, test receiver's type (a set of classes) for overlap with applies-to set for each potential method**
    - **for x.f(), applies-to(f_i) ∩ type(x) == ?**
    - **if there's only 1 applies-to set with an overlap, have uniquely resolved the call**
    - **memoize the result of the test for future lookup**

Calculating Applies-to Sets



For each function, find subtree of inheritance tree.
Initialize applies-to sets at C to Cone(C)
In depth first order, subtract descendent sets from parents



Class Hierarchy Analysis

- **Problems**
  - **If *super* is used as a receiver, then this analysis doesn't work because the invocation through super is not accounted for in the applies-to set**
    - **This violates the initialization of an applies-to set(f) in class C to Cone(C).**
  - **For multimethods, need more than receiver type in lookup**
    - **Can do lookup on k-tuple of runtime types or do some precomputation at compile-time (Java) to restrict runtime choices**

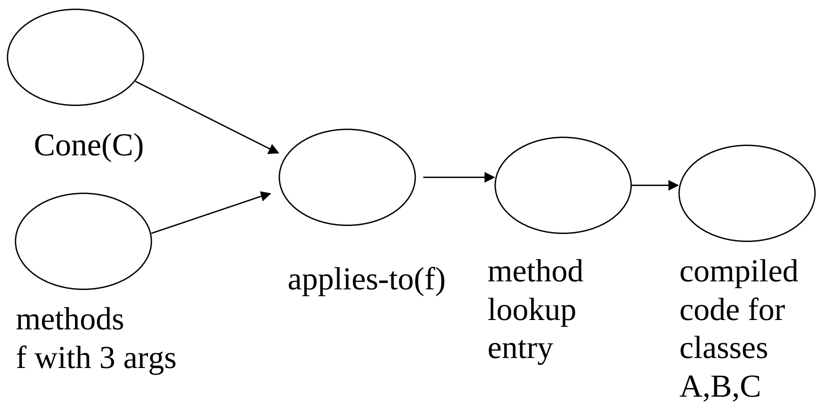
Implementation Notes

- **Vortex implementation**
  - **Overlap test efficiency using bitstring rep for sets is most difficult for set differences and cones**
    - **Note testing algorithm allows for use with extensible class sets**
  - **O(N) test where N is number of classes in program**

Analysis/Transfn Interdependences

- **Incremental compilation requires intermodule dependency information**
  - **DAG of module dependences for information used during compilation**
  - **Granularity chosen has DAG half the size of the program representation**
  - **Claims to have saved 7 times recompilation over C++ header-based scheme and factor of 2 over Scheme's finer--grained mechanism**

Example



Annotations versus Analysis

- **Requiring *virtual* keyword forces programmer to choose what client can override early**
- **Keywords make change onerous (changing design means changing old source code)**
- **CHA can often obtain an unique static binding for *virtual* functions in specific apps**
  - **About 1/2 calls in codes in paper were statically bindable through CHA, but needed to be virtual**
- **Java's *final* helps analysis**

Exhaustive Class Testing

- **Empirical studies show that often call sites actually call only a few methods**
- **Can insert explicit type tests for all potential classes at a call site (if it's a small number)**
  - **Must worry about cost/benefits**
  - **Want to improve code performance at minimal cost in time and space**
  - **Vortex choices: do if small (<=3) number of candidate classes and all methods would be inlinable; test methods in BU tree order**

Exhaustive Class Testing

- **Single class membership tests** *cost: 5 instructs*
  - **Virtual function table pointer in C++**

  x = obj.m( ); becomes id_x = obj.class_id;
  if (id == D) … elseif (id == E)…else ….;
  - **On each branch, inline and further optimize using known type of receiver**
  - **Problems:**
    - **Large hierarchies make testing impractical (especially if many classes use same function)**
    - **Sensitivity to program changes (extended classes have to be instantiated in old test code)**

Exhaustive Class Testing

- **Cone tests (subclass relation check)**
  - **Single inheritance** *cost: 6 instructs*
    - **x has y as ancestor in tree iff x.l >= y.l && x.h<=y.h where l is preorder# and h is postorder#**
  - **Multiple inheritance** *cost: 5-6 instructs NxN space*
    - **N classes numbered 0 to (N-1)**
    - **Build NxN bit matrix X, X[k,j] = true if class k is subclass of class j**

Other Static Analyses

- **Constant propagation (value flow)**
- **Instance variable optimizations**
  - **Elimination of redundant reads and writes**
  - **Use of base+offset addressing with value flow info**
- **Dead store elimination**
  - **e.g., when inlined object constructor does initializations overriden by caller code**
- **Dead object elimination**
  - **Can use *escape analysis* to see which objects exist past creator block's lifetime**

Profile-guided Optimizations

- **Execution frequency data**
  - **Guides inlining decisions and scales down optimization in infrequently used methods**
  - **Provides input to guide receiver class prediction**
  - **Guides selective specialization of code**
- **Granularity of data collection**
  - **How much calling context to save?**
  - **How many method calls to fold together?**
  - **Balance efficiency of data gathering (cost) versus utility of data gathered (profitability)**

Receiver Class Distributions

- Each distribution associated with set of method calls
  - Message summary - all messages with same name
  - Call-site-specific (1-CCP)
    - *k-Call Chain Profile* delimits k dynamically enclosing call sites (stored in factored tree form)
  - Call-chain-specific (n-CCP)
- Collect histogram for each of receiver classes
  - Shows if a few classes dominate
  - Shows which classes are most common

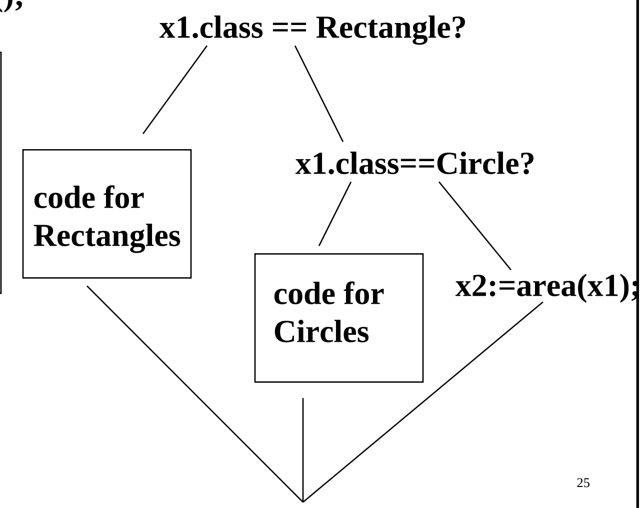
Receiver Class Distributions

- Vortex gathers info off-line in separate training runs of program and uses into in optimization
- Alternative: SELF does dynamic compilation, gathering profiling info as program runs and recompiling using this info for “hot methods”
- Code generated like exhaustive class testing except types are those seen in profiles, not statically gathered

Example

x2 := x1.area();

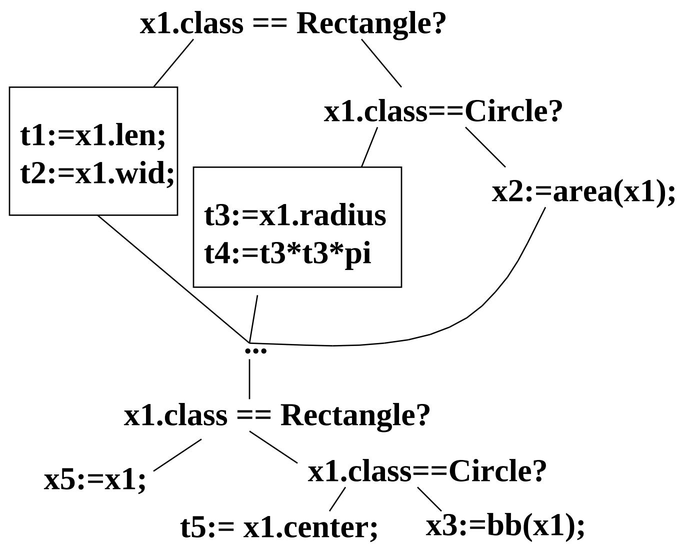
Assumption:
benefits of avoiding dynamic dispatch or inlining outweigh runtime cost of class test



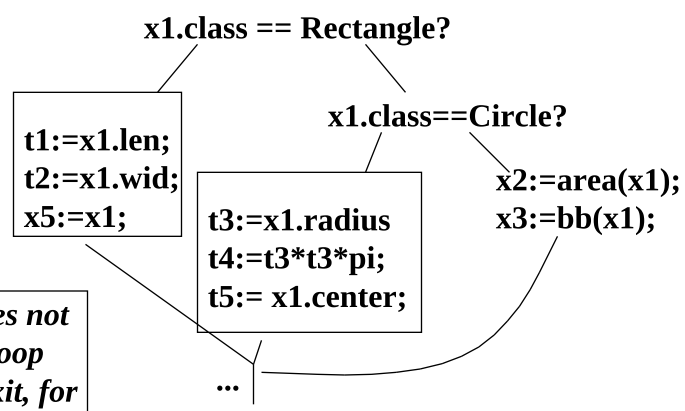
Splitting

- **To avoid redundant tests, can split control flow path between merge following one occurrence of a class test and the next occurrence of same class test**
- **Vortex does this lazily**

Example



Example



Vortex does not split past loop entry or exit, for simplicity

Profiling

- Want profiles to be stable across inputs
- Want profiles to be stable across program versions
- Want peaked profiles to find some methods are more frequently called than others
- Vortex trials on C++ and Cecil programs
  - 71% of C++ messages (72% Cecil) were sent to most common receiver class
  - in C++, 36%(Cecil 50%) dynamic dispatches occurred at call sites with *single receiver class*!

Profile Stability

- Two metrics studied on profiles derived from different inputs to same programs
  - *FirstSame:* same most common receiver class
  - *OrderSame:* 2 distributions are same only if they are comprised of same classes in same frequency order
  - in C++,
    - for FirstSame, 99% match for method summary and 79% match for 1-call-site-specific
    - for OrderSame, 28% match for method summary and 45% match for 1-call-site-specific

Profile Stability

- **Gathered profiles across different versions of Vortex using 6 month version control history**
  - **OrderSame metric was not similar**
  - **FirstSame found distributions stable**
    - **Fewer than 5% method summaries changed over entire 6 month period**
- **Claim: this validates utility of profiles for optimization of future versions of a program**

Method Specialization

- **Factoring shared code into base classes which contain virtual calls to specialized behavior subclasses hurts runtime performance**
- **Compiler must *undo* effects of factorization**
- **Vortex, profile-guided selective specialization**
  - **Idea: given weighted call graph derived from profile data, eliminate heavily travelled, dynamically dispatched calls by specializing to particular patterns in their parameters**

Method Specialization

- **Drawbacks**
  - **Overspecialization - multiple specialized versions may be too much alike**
  - **Underspecialization - methods may only be specialized on receiver type**
- **Pass-through call sites use formals of caller as arguments to callee, *specializable call sites***
  - **f(A a,B b,C c){…a.s(c)….} can specialize s() for set of known static types of a and c**

Some Questions

- **How is set of classes which enable specialization of pass-through arc calculated?**
- **How should specializations for multiple call sites to same method be combined?**
- **If a method *f* is specialized, how can we avoid converting statically bound calls to *f* into dynamically bound calls?**
- **When is an arc important to specialize?**

Specialization Algorithm

- At a pass-through edge, determine most general class set tuple for pass-through formals that allows static binding of call
- Must combine class set tuples from different call sites in same method, somehow
  - Have info on specific class sets for args but not on their occurrence in specific combinations
  - Vortex: try all plausible combinations and be careful about code blowup (didn't occur in practice)

Specialization Algorithm

- May change a statically bound call to the unspecialized method to a dynamic test to choose between specialized versions OR can leave original translation as target of statically bound call
- Cascading specializations - tries to recursively specialize caller to match the specialized callee
  - Has effect of hoisting dynamic dispatch to lower frequency parts of call graph

Specialization Algorithm

- **Cost/benefit threshold: 1000 invocations**
- **Drawbacks**
  - **Doesn't consider code growth**
  - **Treats all dynamic dispatches as same benefit**
  - **NO global view on code growth as do the optimization**

Performance Studies

- **Methodology**
  - **Vortex compiles into C which is compiled using gcc2.6.3 -O2.**
  - **Reported times are average over 11 runs of SPARC 20/61 with 128 MB memory**
  - **Space measured as bytes in a stripped executable**
  - **Training data different from input measured (except for *richard*)**

Data

- **Five benchmark Cecil programs**
- **Effects (Figure 18)**
  - ***unopt* - no optimizations of sends**
  - ***intra(*or i) - intraproc class analysis, automatic inlining, hard-wired class prediction for built-ins, intraprocedural opts (e.g., splitting, dead code elim)**
  - ***i + CHA***
  - ***i + CHA + exh* - plus exhaustive class testing for call sites with small# of candidates**
  - ***i + CHA + exh + spec* - plus selective specialization**

Improvements Reported

(1) hard-wired class prediction (e.g., +)

(2) 60% performance improvement from adding *CHA*

(3) 18% performance improvement from adding exhaustive class testing

(4) 2.8 performance improvement by adding profile-guided class prediction to *intra*

Hypothesis: interprocedural class analysis techniques might further narrow performance gap between purely static and static+profile-guided system

OOPL Comparison

- C++ 3-4 times faster than Cecil 3-4 faster than Smalltalk-80
- Based on performance on 2 benchmarks

- Base is i+CHA+exh+spec in the data gathering figures