OOPL Optimizations

- Optimizations specific to OOPLs
  - Optimizing dynamic dispatch
    - Run-time method selection
  - Method inlining
  - Control flow path splitting
  - Method specialization
  - Object layout for cache performance
  - Synchronization removal

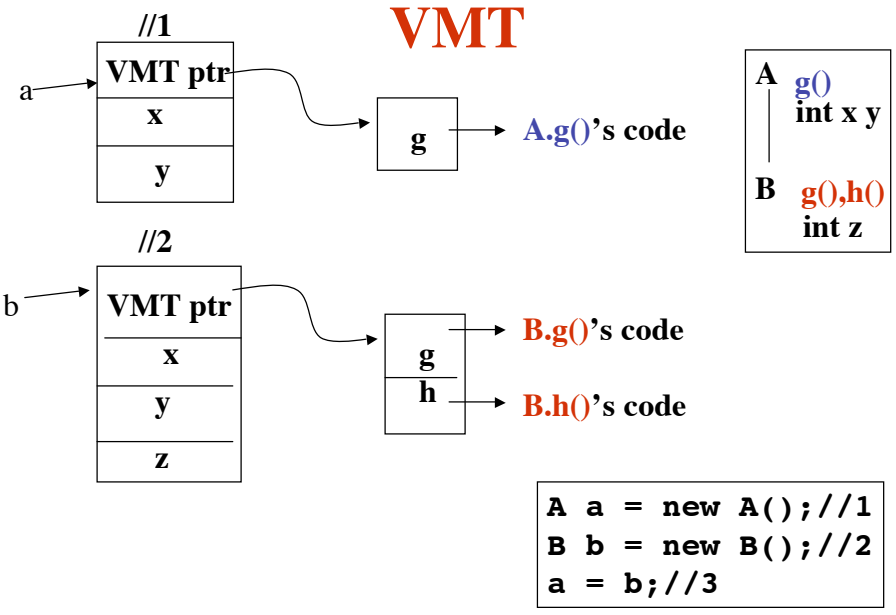
Dynamic Dispatch

- Introduced in C++ *virtuals*
- Virtual method tables (1 per class, every instance points to its class's VMT)
  - Keeps pointers to code for appropriate method of each name, for receivers whose runtime type is a specific class
  - Avoids need to store class hierarchy explicitly for run-time use
    - Put pointer to its class's VMT entry in object when it is created
  - Inherited methods are explicitly represented

Dynamic Dispatch

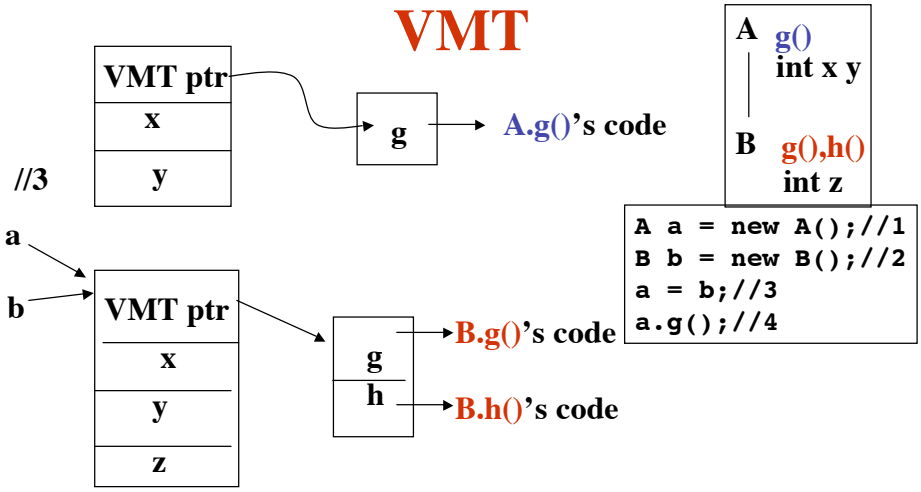
- Method call resolution:
  - Load receiver's VMT entry
  - Load method address by indexing into that table by selector number
  - Jump to that function
- Need to build additional tables because of multiple inheritance (more than one method at same index)
- Cost of resolution measured by Driesen and Holzle on early C++ programs at median of 5.2% total execution time (OOPSLA'96)

VMT



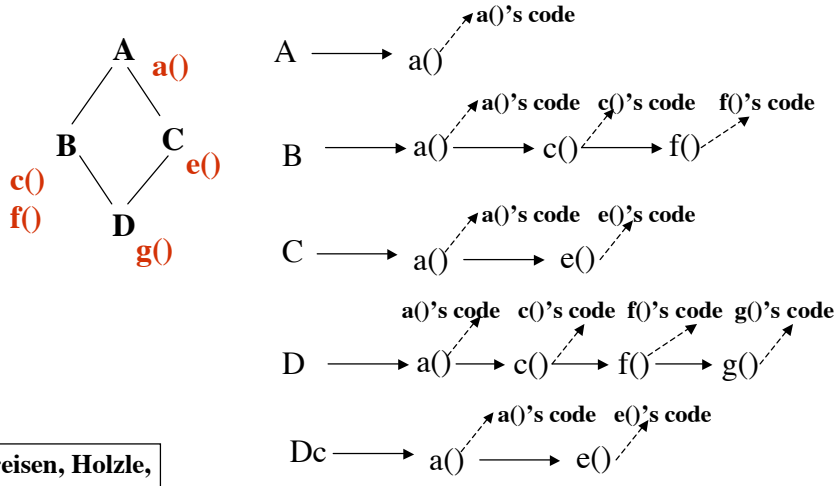

```
A a = new A();//1
B b = new B();//2
a = b;//3
```

VMT



Will get method resolution for a.g() call at 4 through B's VMT entry, using runtime type of object to which a refers.

VMT - Multiple Inheritance



Dreisen, Holzle, OOPSLA'96

Method Inlining

- **Idea: avoid overhead of context switching and expose more code to possible optimization**
  - **Can inline call sites statically shown by analysis to be monomorphic**
  - **Using guards, can inline call sites with small number of frequent callees and large frequency of occurrence**
  - **Usually inline small methods to avoid too much code size growth**
    - **Avoid in presence of recursion**

Method Inlining

- **Strategies developed are heuristic**
  - **If inline, often expose opportunities for optimizations in code**
  - **If inline, may increase register pressure and cause too much code growth**
- **Need to balance estimated gain in performance versus code growth and potential slowdown due to runtime checks**

Method Selection

- Early optimizations of VFT schemes - how to do lookup quickly?
  - Use observed profiling information about calls to predict most likely methods called
  - Encode method call as guarded inlining

```
if (shape instanceof Circle)//inline code for Circle.area()
else if (shape instanceof Square) //inline code for Square.area()
else shape.area();//regular dynamic dispatch
```

- Also can encode as runtime method selection

```
if (shape instanceof Circle)//call Circle.draw()
else if (shape instanceof Square)//call Square.draw()
else shape.draw();//regular dynamic dispatch
```


Guarded Inlining

Aigner and Holzle, "Eliminating Virtual Function Calls in C++ Programs", ECOOP96

- Aigner and Holzle: C++ source-to-source compiler changed virtual method calls to guarded inlined method calls (asked the C++ compiler to inline methods)
  - Decisions:
    - Most frequent class represents 40% of receivers at call site
    - Optimize call sites that do over 0.1% of calls in run;
  - Results: profiling feedback more successful than CHA-based analysis, but gains varied over benchmarks

Guarded Inlining

- Possible problems
  - Must worry about cost versus benefits
    - Can increase cost of dynamic dispatch for classes not in the tests
    - Can decrease overall cost of dynamic dispatch if the tested classes occur frequently enough and if further optimizations are possible in the inlined code
  - Vortex choices
    - Do guarded inlining if there are a small (<=3) number of candidate classes and all methods can be inlined

Craig Chambers, Jeffrey Dean, David Grove, "Whole-program Optimization of Object-oriented Languages, TR-96-06-02, DCSE, Univ. Washington, June 1996 ;
J. Dean, G. DeFouw, D. Groave, V. Litvinov, C. Chambers, "Vortex: An Optimizing Compiler for OO Languages", OOPSLA'96.

Guarded Inlining

- How to accomplish?
  - Single class membership tests
  - VMT (virtual method table) pointer in C++

cost: 5 instructs

x = obj.m( ); becomes id_x = obj.class_id;
if (id == D) … elseif (id == E)…else ….;

  - Test types in bottom-up order wrt class hierarchy
  - On each branch of the if-then-else, inline a method, 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)

Subclass 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
    - Want to know `inheritsFrom(obj,B)` function
    - N classes numbered 0 to (N-1)
    - Build NxN bit matrix X, X[k,j] = true if class k is subclass of class j

      cost: 5-6 instructs
      NxN bits of space

Inlining in JikesVM

- Speculative inlining with guards
  - Done in response to CHA or profiling
  - Guard with class/method test
  - May avoid test with pre-existence

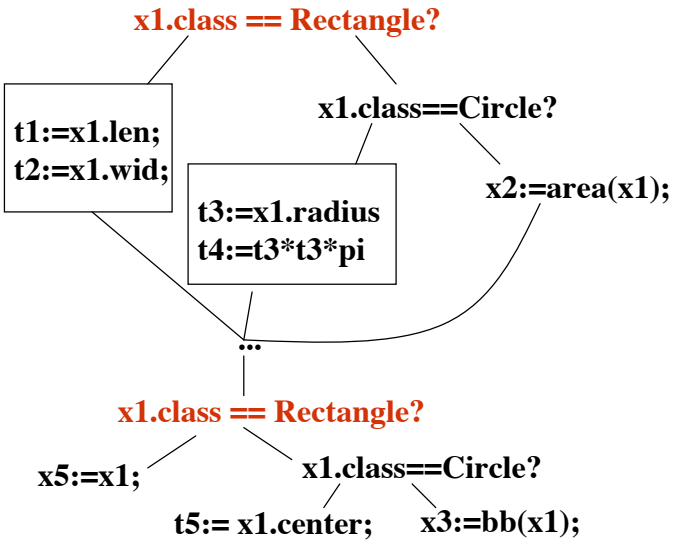
    `void foo(A a){ … a.m();…)`

    if object referred to by `a` can be shown to have been created prior to when `foo` is invoked, then it is valid when executing the inlined code.

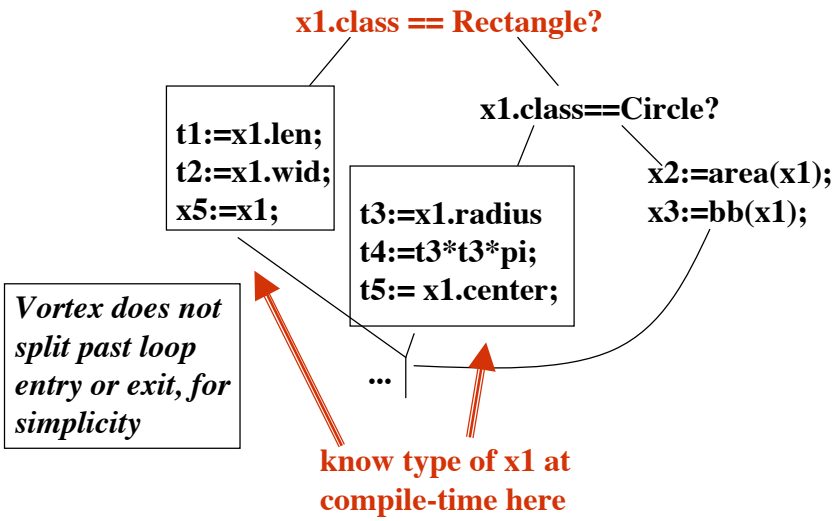
Path Splitting

- **Idea: to avoid redundant tests and increase extent of code for which types of some objects are known**
- **To avoid redundant type tests, split control flow path between merge following one occurrence of a class test and the next occurrence of same class test**
  - **Duplicates code**
- **Vortex does this lazily**
- **Feedback-directed splitting in adaptive Jikes VM**

Example



Example



Method Specialization

- Using known type information at compile-time to translate a *customized* version of code assuming that information (e.g., receiver type)
- Factoring shared code into base classes which contain virtual calls to specialized behavior to subclasses hurts run-time performance
  - Compiler must *undo* effects of factorization
- Vortex: profile-guided selective specialization
  - **Idea:** given weighted call graph derived from profile data, eliminate heavily traveled, dynamically dispatched calls by specializing to particular patterns in their parameters

Method Specialization

- **Drawbacks**
  - **Overspecialization - multiple specialized versions may be too much alike**
  - **Under-specialization - 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`**

Vortex Specialization Algm

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

Questions asked in Vortex

- 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?

Vortex Specialization Algm

- May change a statically bound call to the unspecialized method to a dynamic test to choose between specialized versions OR can leave original (unspecialized) 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

Vortex Specialization Algm

- Chosen cost/benefit threshold: 1000 invocations for specializable call
- Drawbacks
  - Doesn't consider code growth
  - Treats all dynamic dispatches as same benefit
  - NO global view on code growth as perform the optimization

Object Layout for Locality

- **Idea:** want good cache performance so profile usage of object fields; rearrange object storage so frequently used fields occupy same cache line, where possible
  - Avoids cache misses
  - Affects data layout in storage
  - Uses profiling to measure field usage

Cache Optimization

- *Structure splitting* - viable optimization for Java
  - Improve cache performance of objects comparable to or larger than a cache block
  - Idea:
    - Profile use of fields of objects to identify some as hot (frequently used) vs cold (seldom used);
    - Automatically split class to associate cold fields of an object with another class only accessed indirectly
    - Change all existing references to the new structure
  - For larger objects that span multiple cache blocks, reorder fields by temporal affinity of use
  - Performance improvements of 18-28%, with 22-66% of improvement coming from class splitting

T. Chilimbi, B. Davidson, J.R. Larus, "Cache-conscious Structure Defn", PLDI'99

Benchmarks

- Java benchmarks used

Table 1: Java benchmark programs.

Program	Lines of Code[a]	Description
cassowary	3,400	Constraint solver
espresso	13,800	Martin Odersky's drop-in replacement for javac
javac	25,400	Sun's Java source to bytecode compiler
javadoc	28,471	Sun's documentation generator for Java source
pizza	27,500	Pizza to Java bytecode compiler

a. Plus, a 13,700 line standard library (JDK 1.0.2).

T. Chilimbi, B. Davidson, J.R. Larus, "Cache-conscious Structure Defn", PLDI'99

Experimental Procedure

- Analyzed and instrumented bytecode to collect field info (type, size) from application
- Execute instrumented code to obtain field access frequencies and numbers/kinds of objects created
- Split classes, choosing based on static + dynamic data
- Java bytecode recompiled to reflect splitting decisions (use Vortex to obtain native code)

Setup

T. Chilimbi, B. Davidson, J.R. Larus,
"Cache-conscious Structure Defn", PLDI'99

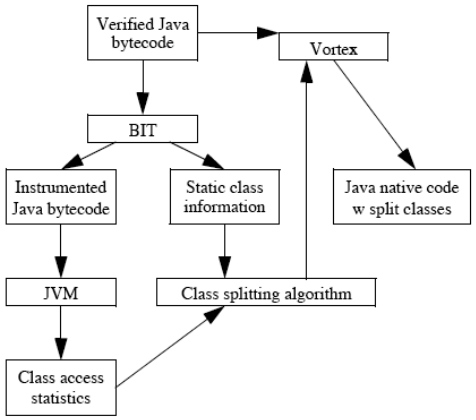


Figure 3. Class splitting overview.

Sizes of Live Java Objects

Table 3: Most live Java objects are small.

Program	Avg. # of live small objects	Bytes occupied (live small objects)	Avg. live small object size (bytes)	Avg. # of live large objects	Bytes occupied (live large objects)	% live small objects
cassowary	25,648	586,304	22.9	1699	816,592	93.8
espresso	72,316	2,263,763	31.3	563	722,037	99.2
javac	64,898	2,013,496	31.0	194	150,206	99.7
javadoc	62,170	1,894,308	30.5	219	148,648	99.6
pizza	51,121	1,657,847	32.4	287	569,344	99.4

Observations: sizes of live objects after a GC averaged over execution; note smaller size than 64byte cache block

Chilimbi et al, PLDI'99

Details

- **Tradeoffs**
  - **Pack more hot class instances into cache block**
    - **Cost of additional reference from hot to cold portion; Code growth; More objects in memory overall; Extra indirection for each cold field access**
- **Heuristics to choose classes**
  - **Only split *live* classes with total field accesses exceeding a threshold: $A_k \geq LS / (100*C)$**
    - **A_k: #fields accesses in class k; LS: total # field accesses; C: total number of classes with at least one field access**

  Plus larger than 8 bytes with 3 or more fields.

Details

- Heuristics to choose fields
  - Cold fields accessed no more than A_k/(2*F_k) times where F_k is # fields in class k
  - To split requires at least 8 bytes cold
  - Use heuristics to avoid overly aggressive splitting
- Split class transformation
  - Hot classes and their accesses are same
    - Additional new field per object refers to new cold class
    - Need to alter constructors to create new cold class instance and assign it to the new field
  - Cold field counterpart class created with public fields, inherits from Object, only method is constructor
  - Change accesses to cold fields to indirect accesses through new field

Findings

- Measured class splitting potential with 2 inputs per benchmark
  - 17-46% of all accessed classes are candidates with 26-100% having field access profiles that justify splitting
    - Claim the splitting algorithm is insensitive to input data used to profile (measured between the 2 inputs)
  - Split classes account for 45-64% total number of program field accesses
    - Temperature differentials high (77-99%) indicating strong differences between hot and cold field accesses
    - Modest additional memory needs (13-74KB)

Optimization Results

Table 6: Impact of hot/cold object partitioning on L2 miss rate.

Program	L2 cache miss rate (base)	L2 cache miss rate (CL)	L2 cache miss rate (CL + CS)	% reduction in L2 miss rate (CL)	% reduction in L2 miss rate (CL + CS)
cassowary	8.6%	6.1%	5.2%	29.1%	39.5%
espresso	9.8%	8.2%	5.6%	16.3%	42.9%
javac	9.6%	7.7%	6.7%	19.8%	30.2%
javadoc	6.5%	5.3%	4.6%	18.5%	29.2%
pizza	9.0%	7.5%	5.4%	16.7%	40.0%

Table 7: Impact of hot/cold object partitioning on execution time.

Program	Execution time in secs (base)	Execution time in secs (CL)	Execution time in secs (CL + CS)	% reduction in execution time (CL)	% reduction in execution time (CL + CS)
cassowary	34.46	27.67	25.73	19.7	25.3
espresso	44.94	40.67	32.46	9.5	27.8
javac	59.89	53.18	49.14	11.2	17.9
javadoc	44.42	39.26	36.15	11.6	18.6
pizza	28.59	25.78	21.09	9.8	26.2

Chilimbi et al, PLDI'99

Synchronization Removal

- **Java library methods are often synchronized for use in multi-threaded applications**
- **If program is single-threaded or threads do not share data, then unnecessary**
  - **Use *escape analysis* to find objects which escape the thread that creates them**
    - **If none found, then no need for synchronization**