Compiling OO Languages

OO languages create impediments to analysis and optimization

- Dynamism
- Java semantics
- . . .

How might they facilitate optimizations?

- Hint: What are the key ideas behind the OO model?

Code and Data Reorganization

Last time

- Introduction to compiling OO languages

Today

- Specialization
- Exploit encapsulation to improve memory performance
    - Data reorganization

Specialization

Idea

- Create multiple versions of methods, one for each potential receiver
- Now each method knows the type of the receiver
- Can optimize each specialized method

Problems

- Overspecialization
  - Code explosion
  - Code bloat with little benefit because some specialized versions are almost identical
- Underspecialization
  - Some methods that are commonly invoked could be much faster if they were specialized

Specialization Example

```
class rectangle:shape {
      int length() { ... }
      int width() { ... }
      int area() { return (length() * width()); }
}

class square:rectangle {
      int size;
      int length() { return(size); }
      int width() { return(size); }
}
```

Specialize `area` for `rectangle` and `square`

– Can then inline **`length`** and **`width`**

A Brief History of Specialization

Trellis [1988], Sather [1991]

– Specialize all inherited methods for each receiver class

Self [1989]

– Only compiles (dynamically) code that actually executes
– Only dynamically compiled systems can do this

Cecil [1995]

– **Selective specialization**: only specialize when benefit is significant
– Use profile-derived weighted call graph to guide specialization
– Specialize for sets of classes with same behavior
  – *e.g.* Create one instance of `isConvex()` for `rectangle` and `square`
  – *e.g.* Create separate instances of `area()` for `rectangle` and `square`
– Specialize on arguments, too

Inlining

Idea

- Replace call site with method body
- Requires class analysis, *etc.*

Advantages?

- Eliminates method call overhead
- Specializes methods to calling context
- Specializes caller to the callee's context

Disadvantages?

- Not always possible
- Increases code size

Key to success

- Use profile information to discover where it is beneficial

Benefits of Inlining [Arnold,et al 2000]

Static call graph heuristic (SCG)

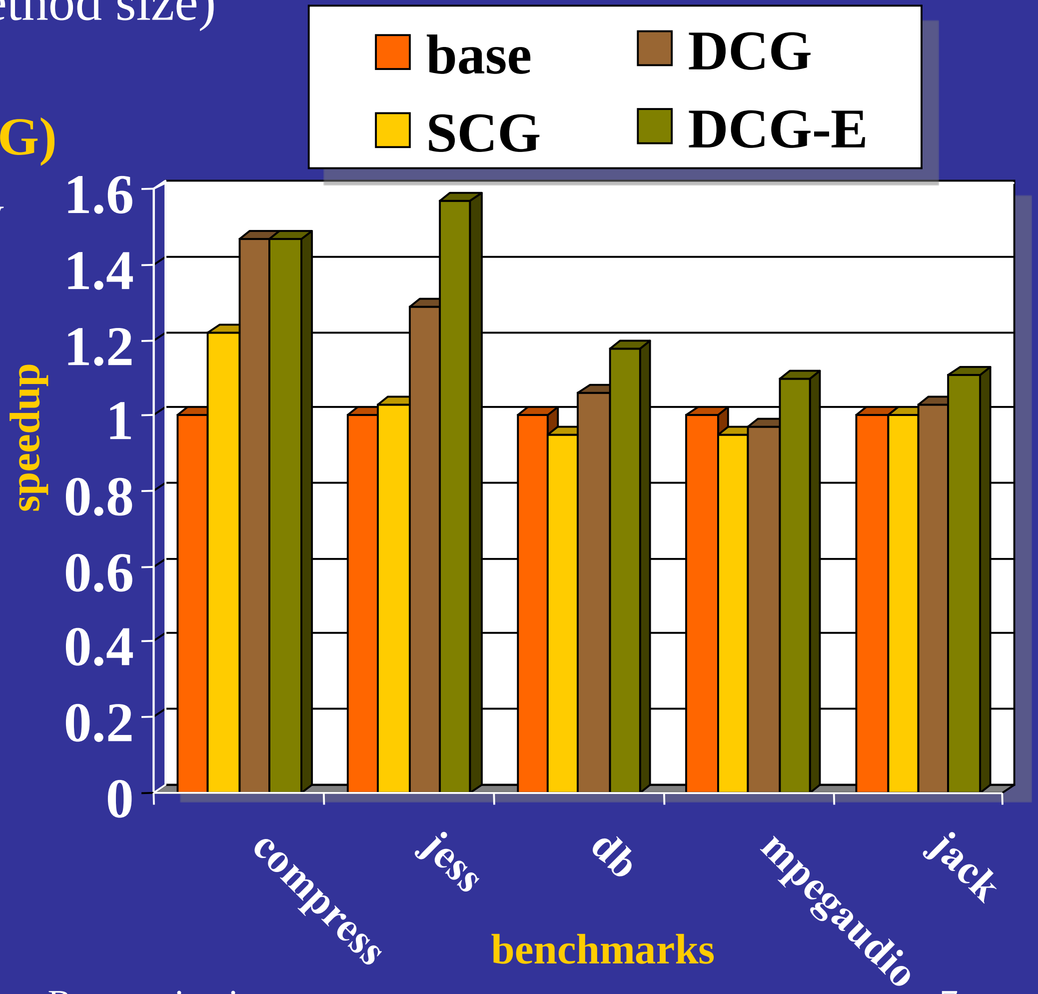
– Minimize (# of call sites × method size)

Call graph w/node weights (DCG)

– Same goal but uses frequency information

Dynamic call w/edge weights (DCG-E)

– Considers individual call site frequencies
– Can inline some instances of a method rather than all or nothing



Inlining Trials [Dean and Chambers'94]

Many indirect benefits of inlining

- Constant propagation, dead code elimination, loop invariant code motion

Indirect benefits of inlining

- Can't be measured by looking at the call graph, node frequencies, or link frequencies
- Often depends on information at the call site, such as specific parameters

Idea

- Perform **inlining trials** to measure cost and benefit of inlining
- Use **type group analysis** to describe info available at each call site
- Keep database of **inlining trials** indexed by the type group
- Inline a method if its call site matches a profitable inlining trial

Inlining Trials (cont)

Experimental results

– Primary benefit is reduction in compilation time (20% faster)

– Program execution time essentially the same (1% slower)

– Difficult to compare Self with other systems

  – Self uses incremental, dynamic compilation

  – Self is a pure object-oriented language

The big picture

– Preserve rich information in a database

– Perform optimization in the large, *i.e.*, across programs

Data Reorganization: Motivation

Memory speeds increasing slower than processor speeds

– Improve cache behavior to improve program performance

Clustering [Chilimbi and Larus 98]

– For small objects, place objects that tend to be accessed together on the same cache line

– The garbage collector can improve locality

  – Use a copying collector

  – Cluster while copying

  – Transparent to programmer and compiler

Limitations of Clustering

Clustering works for small objects

- In Cecil, most objects are < 16 bytes, so multiple objects fit in a cache line
- In Java, most objects are larger
  - Average of 24 bytes [Chilimbi, Davidson & Larus 99]
  - Clustering is less useful for large objects
    - *e.g.* Can't cluster 24 byte objects into 32 byte cache lines

What do we do about large objects?

- Reorganize the layout of individual objects

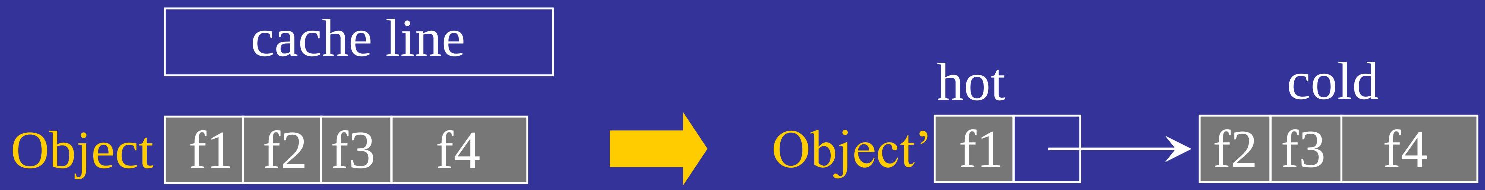
Reorganization of Large Objects [Chilimbi, Davidson, Larus 99]

Encapsulation hides implementation details

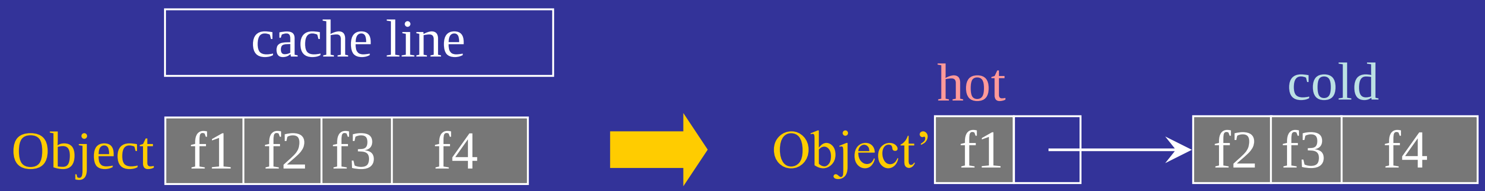
- The compiler can change the layout of an object and the programmer can't notice
- This is not true in C or C++ where the programmer can access arbitrary memory locations through pointers and pointer arithmetic
- Exploit encapsulation to improve data cache behavior

Field Splitting

- For objects that are about the size of a cache line
  - Divide the fields into **hot fields** and **cold fields**



Field Splitting



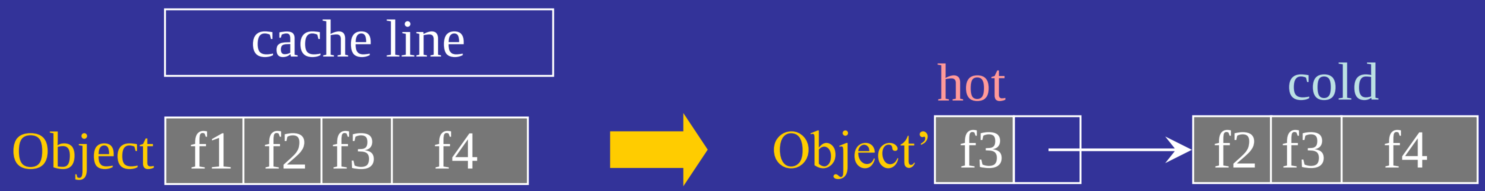
Hot fields vs. cold fields

- Hot fields are those that are accessed more frequently
- Hot fields can now be clustered for improved cache behavior
- Access to cold fields is slower: requires an extra level of indirection

Two Computer Science Principles

- Optimize the common case
- You can solve any problem with an extra level of indirection

Field Splitting (cont)



Identifying hot fields

– Use profiling to gather information on field usage
– Results will suffer if they are input-dependent

Identify potential classes to split

– Only consider classes that are commonly accessed
– Define **Live Classes** as those whose total field accesses exceed some threshold:

$A_i > LS/(100 \times C)$, where LS = total field accesses in program
C = total number of classes
A_i = total number of accesses to fields in class i

Identifying Fields to Split

Additional restrictions on Live Classes

- Must have at least two fields
- Must be larger than 8 bytes

Splitting Heuristic

- Our goal is to identify classes with a large **temperature** difference between hot and cold fields
  - Why?
- Start by identifying **cold fields**
  - An average field would be accessed A_i/F_i times, where F_i is the number of fields in class i
  - Cold fields are those not accessed at least $A_i/(2 \times F_i)$ times
- All other fields are **hot fields**

Identifying Fields to Split (cont)

Temperature Difference

- Define **temperature difference** as follows

  $TD(class_i) = (max(hot(class_i)) - 2 \times \sum cold(class_i)) / max(hot(class_i))$

  where $hot(class_i)$ and $cold(class_i)$ are the number of references to the hot and cold fields of $class_i$, respectively

  - The temperature difference identifies at least one really hot field
- Split those classes whose TD > 0.5
  - *i.e.*, Split if $max(hot(class_i)) > 2 \times \sum cold(class_i)$
- Can split an object into multiple cold portions if necessary

Lots of magic numbers in these heuristics

Field Splitting Transformation

Cold fields are placed in a new object

- Cold members are `public` to allow access by the hot portion of the object
- Translate references to fields in the cold portion

Example

```
class A {
  protected long a1;
  public int a2;
  public float a3;
  A() {
    . . .
    a3 = . . .;
  }
}
```

⇒

```
class A {
  public int a2;
  public coldA coldAref;
  A() {
    coldAref = new coldA();
    coldAref.a3 = . . .;
  }
}
class coldA {
  public long a1;   ⇐
  public float a3;
  coldA() { . . .}
}
```

Note: Java now supports nested classes
Does this change the implementation?

Field Splitting Transformation (cont)

Example with Inheritance

```
class B extends A {
  public long b1;
  public int b2;
  B() {
    . . .
    b2 = a1 + 7;
  }
}
```

```
class b extends A {
  public int b2;
  public coldB coldBref;
  B() {
    coldBref = new coldB();
    b2 = coldAref.a1 + 7;
  }
}
class coldB {
  public long b1;
  coldB() { . . .}
}
```

Treat class b independently

- The fields of class b can also be split
- If class a has been split, class b has to have access to class a's cold fields

Field Splitting Issues

Persistence

- Objects that are copied to or from external devices cannot be transformed transparently (*e.g.* RMI)

Splitting into multiple versions

- Can create multiple versions if program exhibits phase behavior with different hot and cold access patterns
- Is this beneficial?

Stability of heuristics

- How much do the heuristics change from program to program and from machine to machine?

Performance Results

Benchmarks

Program	Lines of Code	Description
cassowary	3,400	Constraint solver
espresso	13,800	Drop-in replacement for Java
javac	25,400	Java to bytecode compiler
javadoc	28,471	Java documentation generator
pizza	27,500	Pizza to bytecode compiler

Opportunity

- Significant number of classes are large enough to split: 16%-46%
- Of these candidates, 26%-100% have profiles that justify splitting
- Cold fields
  - Variables used to handle errors
  - Fields for storing limit values
  - Auxiliary objects not on the critical path

Performance Results

Effects of Splitting

- Access to split classes: 45%-64% of accessed fields
- Reduces class sizes by 17%-23%
- High normalized temperature differences

Performance Results

Sun E5000

1MB L2 cache

64 byte L2 line size

CL: Chilimbi and Larus cache concious cache co-location by a copying garbage collector

CS: Class splitting

Miss Rates

Program	L2 miss rate	L2 miss rate (CL)	L2 miss rate (CL+CS)	Δ(CL)	Δ(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%

Performance Results

Execution Time (seconds)

Program	base	CL	CL+CS	Δ(CL)	Δ(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%

Limitations of Field Splitting

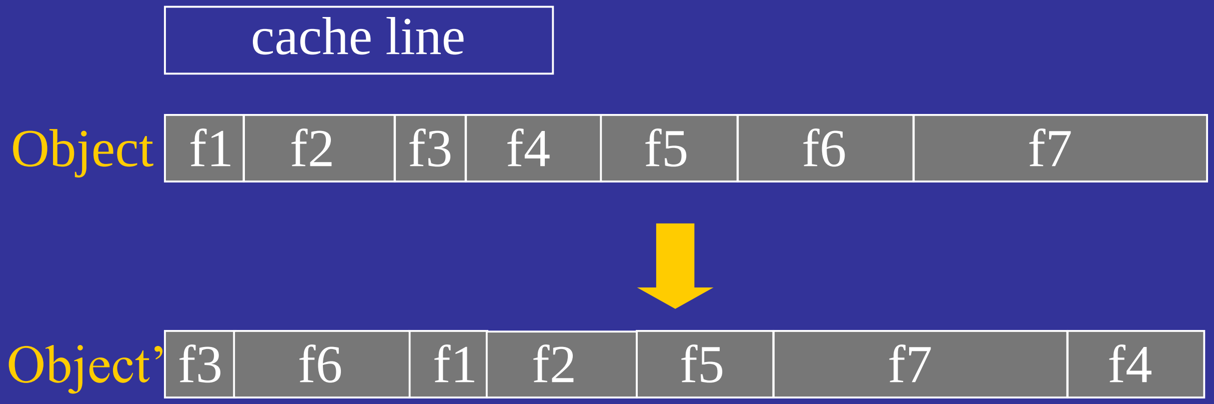
Field Splitting

- Only works for objects that are about the same size as a cache line
- What do we do about objects that are larger than a cache line?

Reorganization of Larger Objects

Field Reordering

- Order the fields within an object so that those that are accessed together are stored together
- Why might this pay off?



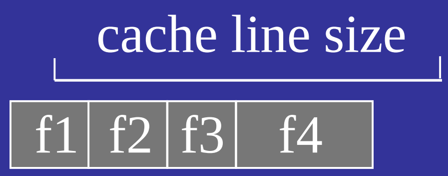
Field Reordering

Basic Idea

- Use profiling to get information about accesses to fields
- Construct **field affinity graphs** for each object **instance**
  - A **field affinity graph** is a weighted graph
    - Nodes represent fields
    - Edges connect fields that are accessed in close temporal proximity
    - Edge weights are proportional to the frequency of contemporaneous accesses
  - Temporal proximity defined to be 100ms
    - Results not sensitive to this parameter (as determined by varying this value between 50ms and 1000ms)
- Combine all instance affinity graphs for an object into a single affinity graph
- Use the object's field affinity graph to reorder fields

Greedy Field Reordering Heuristic

- Start with the two fields with the highest weighted edge in the field affinity graph
- Iteratively add to the layout the field that maximizes **configuration locality**
  - **Configuration locality** computes for each field the sum of its weighted affinities with **neighboring fields** in the layout
  - Two fields are **neighboring fields** if they lie within a cache line of each other in the layout

layout 

f1, f2, f3 and f4 are all neighboring fields

  - This notion of neighbors is approximate, since alignment may actually place two neighboring fields on different cache lines
  - To account for this uncertainty, the weights are scaled inversely with the distance between two fields

Field Reordering Performance

Summary of Performance Results

- Results for commercial C programs (Microsoft SQL)
  - Improved cache utilization 8%-25%
  - Improved execution time 2%-3%
- No experimental results for Java

Data Reorganization Summary

- Field splitting and field reordering are promising ideas
- Encapsulation provides an opportunity to change data organization

Concepts

Specialization

- Costs and benefits
- Inlining trials

Memory behavior

- Memory system performance is important to overall program performance

Exploiting OO features

- Encapsulation provides freedom to rearrange data

Next Time

Lecture

– Field analysis

Assignment 4

– Due Friday