Lecture 4 - Outline

• Empirical results from dynamic sampling analysis
• Optimizations for OO programs
  • Method inlining w & w/o guards
    - Pre-existence
  • Control-flow path splitting
  • Method specialization
• Object layout for better cache performance
• JIKES RVM online FDO experiments
Dynamic Analysis - Experiences

- Empirical experience in IBM's Jalapeno JVM
- 10 benchmarks
  - SPECjvm98 (input size 10), Volano, pBob, opt-compiler
  - Running times from 1.1-4.8 seconds
  - Class file sizes from 10K-1,517K bytes
  - Machine 333Mz IBM RS/6000 powerPC 604e with 2096Mb RAM running AIX 4.3
- Instrumented all methods in applications and libraries

Instrumentation

- **Call-edge**
  - Collect caller, callee, call-site within caller at method entry
  - One counter per call edge
- **Field-access**
  - One counter per field of each class
  - Each putfield, getfield access instrumented
Exhaustive Instrumentation Overhead

On average, 88% call-edge and 60% field-access

Checking Code

Duplicated (Instrumented) Code

Low Overhead

High Overhead

Time Overhead (Full-Dup)

Percent

compress  jess  db  jace  mpegaudio  mrt  jack  opt-compiler  pBob  Volano  average

Arnold & Ryder, PLDI'01
Arnold, Hind, Ryder, OOPSLA'02
### Cost + Accuracy

<table>
<thead>
<tr>
<th>Sample Interval</th>
<th>Overhead (Full-Dup)</th>
<th>Call-edge Accuracy</th>
<th>Field-access Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>182%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>10</td>
<td>29%</td>
<td>99%</td>
<td>100%</td>
</tr>
<tr>
<td>100</td>
<td>10%</td>
<td>98%</td>
<td>99%</td>
</tr>
<tr>
<td>1,000</td>
<td>6%</td>
<td>94%</td>
<td>97%</td>
</tr>
<tr>
<td>10,000</td>
<td>5%</td>
<td>82%</td>
<td>94%</td>
</tr>
<tr>
<td>100,000</td>
<td>5%</td>
<td>71%</td>
<td>83%</td>
</tr>
</tbody>
</table>

OO Opt Using Dynamic Analysis

- Guarded inlining with dynamic dispatch
- Path splitting
- Method specialization
- Object layout for locality
- Adaptive compilation with FDO
Optimizing Dynamic Dispatch

• Early optimizations—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 for C++

• C++ source-to-source compiler changed virtual method calls to guarded inlined method calls
  • Most frequent class represents 40% of receivers at call site
  • Optimize call sites that account for over 0.1% of calls in run

• Results: profiling feedback more successful than CHA-based analysis, but gains varied over benchmarks
Tradeoffs

- Need to balance costs 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

  - Example, in Vortex research compiler
    - Do guarded inlining if there are a small (<=3) number of candidate classes and all methods can be inlined
    - Devised a quick run-time type test for objects


Speculative Guarded Inlining

- In JikesRVM
  - Can be in response to CHA or profiling
  - Guard with class/method test
  - May avoid test with pre-existence

```java
void f(A a){ ... a.m();...}
```

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

- Eliminates the need for a run-time guard on inlined method code
- Applies to call sites that are currently monomorphic, but might become polymorphic due to future class loading
  - Want to inline target method safely wrt classes being loaded
  - Find call sites where receiver is guaranteed to have been created prior to invocation of f(); can apply inlining safely for them

Pre-existence

At runtime the order of events goes like this:
1) f() is invoked and is on the stack
2) Class loading occurs and extends some classes that affect f()'s call sites
3) f() is recompiled so that new invocations of f() are guaranteed to execute correctly.
4) Existing invocations of f() contain code that is technically incorrect, but will be ok due to pre-existence. If a call site was directly inlined based on pre-existence, it is known that receiver objects at those sites *preexist* the invocation of f(), and thus preexist the class loading, so they cannot be of an unsafe type.
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 JikesRVM

**Example**

\[
\begin{align*}
x1.\text{class} &= \text{Rectangle?} \\
\text{t1} &= x1.\text{len}; \\
\text{t2} &= x1.\text{wid}; \\
x1.\text{class} &= \text{Circle?} \\
\text{t3} &= x1.\text{radius}; \\
\text{t4} &= t3 \times t3 \times \pi; \\
x2 &= \text{area}(x1); \\
x1.\text{class} &= \text{Rectangle?} \\
\text{t5} &= x1.\text{center}; \\
\text{x3} &= \text{bb}(x1);
\end{align*}
\]
Example

```
x1.class == Rectangle?
```

```
t1:=x1.len;
t2:=x1.wid;
x5:=x1;
```

```
x1.class==Circle?
```

```
x2:=area(x1);
x3:=bb(x1);
t3:=x1.radius
t4:=t3*t3*pi;
t5:=x1.center;
```

know type of x1 at compile-time here

Method Specialization

- Factoring shared code into base classes which contain virtual calls to specialized behavior in subclasses hurts run-time performance (SELF)
  - Compiler must undo effects of factorization
- At compile-time translating a customized version of code, assuming known type information (e.g., receiver type)
- Drawbacks
  - Overspecialization - multiple specialized versions may be too much alike; can lead to code bloat
  - Under-specialization - methods may only be specialized on receiver type, when could use other parameters
Vortex Profile-guided Specialization

- **Idea:** Given weighted call graph derived from profile data, eliminate heavily traveled, dynamically dispatched calls by specializing to particular patterns in their parameters.

- **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 \( c \)

J. Dean, G. DeFouw, D. Groave, V. Litvinov, C. Chambers,
"Vortex: An Optimizing Compiler for OO Languages", OOPSLA'96.

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?

J. Dean, G. DeFouw, D. Groave, V. Litvinov, C. Chambers,
"Vortex: An Optimizing Compiler for OO Languages", OOPSLA'96.
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
- **Structure splitting** for Java objects to improve cache performance of objects comparable to or larger than a cache block

---

Structure Splitting

- **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
  - Payoff: all the hot fields are cache-resident
- **Performance improvements of 18-28%, with 22-66% of improvement coming from class splitting**

---

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

Table 1: Java benchmark programs.

<table>
<thead>
<tr>
<th>Program</th>
<th>Lines of Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cassowary</td>
<td>3,400</td>
<td>Constraint solver</td>
</tr>
<tr>
<td>espresso</td>
<td>13,800</td>
<td>Martin Odersky’s drop-in replacement for javac</td>
</tr>
<tr>
<td>javac</td>
<td>25,400</td>
<td>Sun’s Java source to bytecode compiler</td>
</tr>
<tr>
<td>javadoc</td>
<td>28,471</td>
<td>Sun’s documentation generator for Java source</td>
</tr>
<tr>
<td>pizza</td>
<td>27,500</td>
<td>Pizza to Java bytecode compiler</td>
</tr>
</tbody>
</table>

n. Plus, a 13.700 line standard library (JDK 1.0.2).

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
Sizes of Live Java Objects

Table 3: Most live Java objects are small.

<table>
<thead>
<tr>
<th>Program</th>
<th>Avg. # of live small objects</th>
<th>Bytes occupied (live small objects)</th>
<th>Avg. live small object size (bytes)</th>
<th>Avg. # of live large objects</th>
<th>Bytes occupied (live large objects)</th>
<th>% live small objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>casowary</td>
<td>25,648</td>
<td>586,204</td>
<td>22.9</td>
<td>1,669</td>
<td>816,592</td>
<td>93.8</td>
</tr>
<tr>
<td>espresso</td>
<td>72,316</td>
<td>2,263,763</td>
<td>31.3</td>
<td>563</td>
<td>722,037</td>
<td>99.2</td>
</tr>
<tr>
<td>javac</td>
<td>64,898</td>
<td>2,013,496</td>
<td>31.0</td>
<td>194</td>
<td>130,206</td>
<td>99.7</td>
</tr>
<tr>
<td>javadoc</td>
<td>92,170</td>
<td>1,894,308</td>
<td>30.5</td>
<td>210</td>
<td>148,648</td>
<td>99.6</td>
</tr>
<tr>
<td>pizza</td>
<td>51,121</td>
<td>1,657,847</td>
<td>32.4</td>
<td>287</td>
<td>569,344</td>
<td>99.9</td>
</tr>
</tbody>
</table>

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

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)
Dynamic Compilation w FDO

- **Idea**: only optimize performance-critical sections of code
  - Reduces compilation delays and time cost
- **Adaptive optimization**: discover and optimize *hot spots* in the code
  - Idea: use online profile info to optimize methods
- **IBM's Jikes RVM** for Java explored many of the ideas initially presented in SELF compilers and is refining the methodology of adaptive optimization, using online profiling and FDO
Online FDO Experiments

- Embed full duplication framework in Jikes Research VM for adaptive optimization trials
  - Insert instrumentation at highest optimization level (O2) so see optimization effects in profile
  - Instrumentation is intraprocedural edge counters
  - Optimizations used: splitting, code positioning (for code locality), loop unrolling, adaptive inlining

Online Profiling Strategy

Hot Profile(1) → Unopt → JIT Opt → Instru. Sampling
Hot Profile(2) → Re-evaluation (phase shift)
Profile-guided Opt

Arrows represent recompilation steps in the 2-phased profiling
Splitting

Splitting is tail duplication of code to eliminate merges that cause dataflow info to be lost

How to measure performance?

Factors
- Overhead of instrumentation
- Effectiveness of FDO's
- Underlying adaptive optimization system

Measure steady-state performance of SpecJvm98 codes
- Requires running benchmarks in harness multiple times (to total time of 4 minutes on size 100)
### Peak Performance Gains

- **FDO vs Adaptive VM w/o online profiling**
- **Loop Unrolling**: 7.1%
- **Inlining**: 4.8%
- **Splitting**: 5.8%
- **Code Reordering**: 8.0%

**ProLangs**

### SPECjbb2000

**Arnold, et al OOPSLA'02**

**Figure 10: Performance improvement of FDO on the SPECjbb2000 server benchmark**

ACACES-4 July 2007 © BG Ryder
Measuring Precision

• Run sampling framework to record call edges
• Run perfect profile recording every call
• Compare percentage of sample collected attributed to a particular call edge to corresponding percentage in the perfect profile.
Measuring Accuracy

- *Overlap* is minimum of these two percentages
- *Overlap percentage* is sum of overlaps for all edges (Feller 98)
  - Any sample will be less than or equal to 100%
  - A sample identical to perfect profile has 100% overlap
  - If sampling overestimates the percentage for some call site then it must underestimate the percentage for another call site

Sample & Perfect Profiles (Javac)

Javac 93.8% overlap
### Setup


![Diagram showing the setup process](image)

**Figure 3. Class splitting overview.**

### 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