Inlining and Devirtualization

Lecture 8

CS 501
April 24, 2013
Preliminaries

- Papers
  - *Adaptive Online Context-Sensitive Inlining*
    Hazelwood and Grove
  - *A Study of Devirtualization Techniques for a Java JIT Compiler*
    Ishizaki, et al
Inlining

long res;

void foo(long x)
{
    res = 2*x;
}

void bar()
{
    foo(5);
}

void foo(long x)
{
    res = 2*x;
}

void bar()
{
    res = 2*5;
}

void foo(long x)
{
    res = 2*x;
}

void bar()
{
    res = 10;
}
Inlining

2: enter 0
3: mul 2 x#16
4: add res_base#32760 GP
5: store (3) (4)
6: ret 8

7: enter 0
8: param 5
9: call [2]
10: ret 0

2: enter 0
3: mul 2 x#16
4: add res_base#32760 GP
5: store (3) (4)
6: ret 8

7: enter 0
8: add res_base#32760 GP
9: store (8) (9)
10: ret 0
Benefits

• Reduction on function invocation overhead

• No marshalling / unmarshalling parameters and return values

• Better instruction cache locality

• Expanded optimization opportunities

• CSE, constant propagation, unreachable code elimination, ...

• Poor man’s interprocedural optimization
Costs

- Code size
  - Typically expands overall program size
  - Can hurt icache
- Compilation time
  - Larger methods can lead to more expensive compilation, more complex control flow
Language / runtime aspects

- What is the cost of a function call?
  - C: cheap, Java: moderate, Python: expensive

- Are targets resolved at compile time or run time?
  - C: compile time, Java, Python: run time

- Is the whole program available for analysis?

- Is profile information available?
When to inline?

- Jikes RVM (with Hazelwood/Grove adaptations):
  - Call Inst. Sequence (CIS) = # of Inst to make call
  - Tiny (< 2x) : Always inline
  - Small (2-5x) : Inline subject to space constr.
  - Medium (5-25x) : Inline if hot (sub. to space)
  - Large : Never inline
Gathering profile info

- Counter-based: Instrument edges in CFG
  - Entry + loop back edges
  - Enough edges (e.g., Ball / Larus)
  - Expensive - typically removed in opt. code
- Call stack sampling
  - Periodically walk stack
  - Interrupt-based or instrumentation-based
Object-oriented languages

• OO encourages lots of small methods
  • getters, setters, ...
  • Inlining a requirement for performance
    • High call overhead wrt total execution
    • Limited scope for compiler opts
  • For Java, if you’re going to anything, do this!
  • But ... virtual methods a challenge
Virtual methods

- In general, we cannot determine the target until runtime
- Some languages (e.g., Java) allow dynamic class loading: all subclasses of A may not be visible until runtime

class A {
    int foo() { return 0; }
    int bar() { return 1; }
}

class B extends A {
    int foo() { return 2; }
}

void baz(A x) {
    y = x.foo();
    z = x.bar();
}
Virtual tables

- Object layout in a JVM:

```
virtual table
  lock word
  field1
  ...
  fieldN

foo entry
bar entry
```

A::bar()
B::foo()
Virtual method dispatch

\[ t_1 = \text{ldvtable } x \]
\[ t_2 = \text{ldvirtfunaddr } t_1, A::\text{foo} \]
\[ t_3 = \text{call } [t_2] (x) \]
\[ t_4 = \text{ldvtable } x \]
\[ t_5 = \text{ldvirtfunaddr } t_4, A::\text{bar} \]
\[ t_6 = \text{call } [t_4] (x) \]

- \( x \) is receiver object
- For a receiver object with a runtime type of \( B \), \( t_2 \) will refer to \( B::\text{foo} \).
Devirtualization

- Compiler converts virtual calls to static calls
- Benefits: enables inlining, lowers call overhead, better branch prediction on calls
- Often optimistic:
  - Make guess at compile time
  - Test guess at run time
  - Fall back to virtual call if necessary
Guarded devirtualization

\[ t_1 = \text{ldvtable } x \]
\[ t_7 = \text{getvtable } B \]
\[ \text{if } t_1 == t_7 \]
\[ \quad t_3 = \text{call } B::\text{foo}(x) \]
\[ \text{else} \]
\[ \quad t_2 = \text{ldvirtfunaddr } t_1, A::\text{foo} \]
\[ \quad t_3 = \text{call } [t_2](x) \]
\[ \ldots \]

- Guess receiver type is B (e.g., based on profile).
- Call to B::foo is statically known - can be inlined.
- Guard inhibits optimization
Guarded by method

test

```
t1 = ldvtable x

t2 = ldvirtfunaddr t1

t7 = getfunaddr B::foo

if t2 == t7
  t3 = call B::foo(x)
else
  t2 = ldvirtfunaddr t1, A::foo
  t3 = call [t2] (x)

...```

• Guess that method is B::foo

• More robust, but more overhead

• Harder to optimize redundant guards
How to guess receiver?

- Profile information
  - Record call site targets and/or frequently executed methods at run time
- Class hierarchy analysis
  - Walk class hierarchy at compile time
- Type analysis
  - Intra/interprocedural data flow analysis
Profiling

Context-insensitive

runTest

\texttt{cs1} \texttt{cs2}

\texttt{HashMap.get} 50\% 50\%

\texttt{Object.hashCode} \texttt{MyKey.hashCode}

Context-sensitive

runTest

\texttt{cs1}

\texttt{HashMap.get} 100\%

\texttt{Object.hashCode} \texttt{MyKey.hashCode}

runTest

\texttt{cs2}

\texttt{HashMap.get} 100\%

\texttt{Object.hashCode} \texttt{MyKey.hashCode}
Class hierarchy analysis

- Walk class hierarchy at compilation time
  - If only one implementation of a method (i.e., in the base class), devirtualize to that target
- Not guaranteed in the presence of class loading
  - Still need runtime test / fallback
Flow sensitive type analysis

- Perform a forward dataflow analysis propagating type information.
- At each use site, compute the possible set of types.
- At call sites, use type information of receiver to narrow targets.

```java
A a1 = new B();
a1.foo();
if (a2 instanceof C)
a2.bar();
```
Alternatives to guarding

- Guarding impose overheads
  - run-time test on every call, merge points impede optimization
- Often “know” only one target is invoked (CHA)
  - call site is monomorphic
- Alternative: compile without guards
  - recover as assumption is violated (e.g, class load)
  - cheaper runtime test vs more costly recovery
Recompilation approach

- Optimistically assume current class hierarchy will never change with respect to a call
- Devirtualize/inlinen call sites without guard
- On violating class load, recompile caller method
  - Recompiled code installed before new class
  - New invocations will call de-optimized code
  - What about current invocations?
Preexistence analysis

- Idea: if the receiver object pre-existed the caller method invocation, then the call site is only affected by a class load in future invocations.

- If new class C is loaded during execution of baz, x cannot have type C:

```java
void baz(A x) {
    ...
    // C loaded here
    x.bar();
}
```
Code-patching

- Pre-generate fallback virtual call out of line
- On invalidating class load, overwrite direct call / inlined code with a jump to the fallback code
  - Must do thread-safe!
- On x86, single write within a cache line is atomic
- No recompilation necessary
Patching

\[ t_3 = 2 // B::foo \]
\[
\text{next:}
\]
\[
\ldots
\]
\[
\text{fallback:}
\]
\[
t_2 = \text{ldvirtfunaddr } t_1, A::foo
\]
\[
t_3 = \text{call } [t_2](x)
\]
\[
goto \text{next}
\]
\[
goto \text{fallback}
\]
Performance

- Method Test, Class Test
- Codepatch
- Type Analysis
- Preexistence

Programs: compress, jess, db, javac, mpegaudio, mtrt, jack, jbb

Speed up:
- 0.9
- 1.0
- 1.1
- 1.2
- 1.3
- 1.4

Values:
- 2.25
- 2.7
- 2.33

Wednesday, April 24, 13