CSE P 501 – Compilers

Inlining and Devirtualization

Hal Perkins

Winter 2016

References

- *Adaptive Online Context-Sensitive Inlining*
  Hazelwood and Grove, ICG 2003
- *A Study of Devirtualization Techniques for a Java JIT Compiler*
  Ishizaki, et al, OOPSLA 2000

- Earlier versions of this lecture by Vijay Menon, CSE 501, Sp09 & Jim Hogg, CSE P 501 Sp14

Inlining

```
long res;

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

void bar() {
   foo(5);
}
```

→

```
long res;

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

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

→

```
long res;

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

void bar() {
   res = 10;
}
```

Benefits

- Removes overhead of function call
  - No marshalling / unmarshalling parameters and return values
  - Better instruction cache locality
- Bonus: expands optimization opportunities
  - CSE, constant propagation, unreachable code elimination, ...
  - Poor person's interprocedural optimization

Costs

- Code size
  - Typically expands overall program size
  - Can hurt instruction cache
- 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 (virtual dispatch), 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?
  - “separate compilation”
- Is profile information available?
  - If “m” is rarely called, don’t inline it

When to inline?

Jikes RVM (with Hazelwood/Grove adaptations):

- Call Instruction Sequence (CIS) = # of instructions to make call
  - Tiny (function size < 2x call size): Always inline
  - Small (2-5x): Inline subject to space constraints
  - Medium (5-25x): Inline if hot (subject to space constraints)
  - Large : Never inline

Gathering profile info

- Counter-based: Instrument edges in CFG
  - Entry + loop back edges
  - Enough edges (enough to get good results without excessive overhead)
  - Expensive - typically removed in optimized code
  - Depends critically on the "training sets"
- Call stack sampling
  - Periodically walk stack
  - Interrupt-based or instrumentation-based
  - May gather info on what calls what (callsite info)

Object-oriented languages

- OO encourages lots of small methods
  - getters, setters, ...
  - Inlining is a requirement for performance
    - High call overhead wrt total execution
    - Limited scope for compiler optimizations without it
  - For Java, C#, if you're going to anything, do this!
  - But ... virtual methods are a challenge

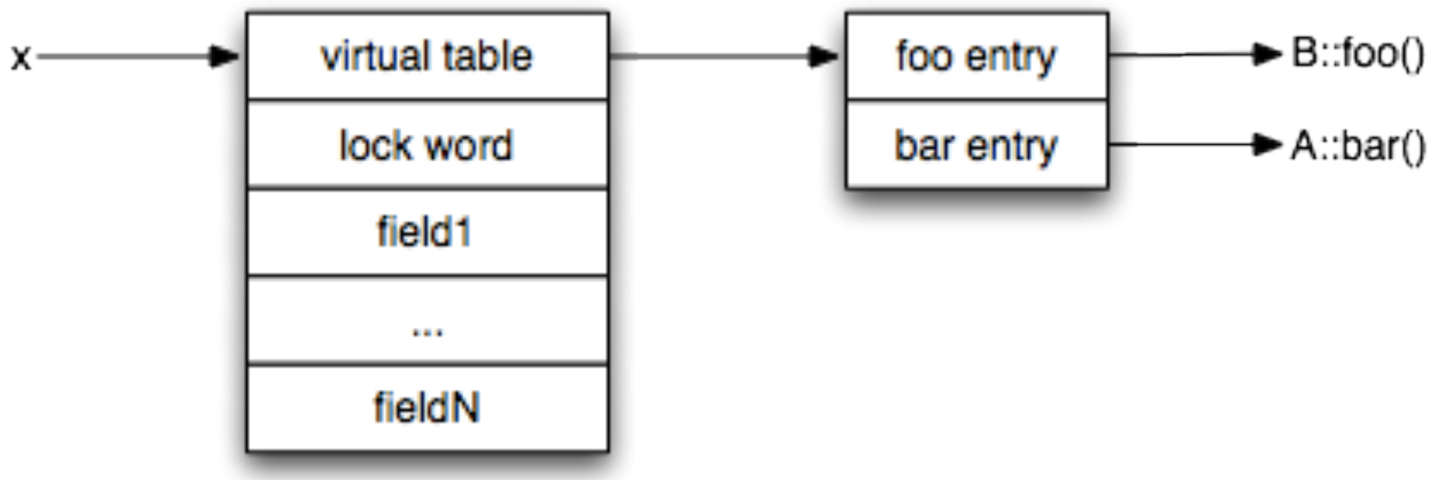
Virtual methods

```
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();
}
```

- 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

Virtual tables

- Object layout in a JVM:



Virtual method dispatch

```
Source:
y = x.foo();
z = x.bar();
```

```
t1 = ldvtable x
t2 = ldvirtfunaddr t1, A::foo
t3 = call [t2] (x)
t4 = ldvtable x
t5 = ldvirtfunaddr t4, A::bar
t6 = call [t4] (x)
```

- x is the *receiver* object
- For a receiver object with a runtime type of B, t2 will refer to B::foo.

Devirtualization

- Goal: change virtual calls to static calls in compiler
- Benefits: enables inlining, lowers call overhead, better I-cache performance, better indirect-branch prediction
- Often optimistic:
  - Make guess at compile time
  - Test guess at run time
  - Fall back to virtual call if necessary

Guarded devirtualization

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

- Guess receiver type is B (based on profile or other information)
- Call to B::foo is statically known - can be inlined
- But 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 outside guard
- 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
  - “monomorphic” vs. “polymorphic”
- Class hierarchy analysis
  - Walk class hierarchy at compile time
- Type analysis
  - Intra / interprocedural data flow analysis

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.

```
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
  - 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 wrt a call
- Devirtualize and/or inline 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?
- Nice match with JIT compiling

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:

```
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 be thread-safe!
  - On x86, single write within a cache line is atomic
- No recompilation necessary

Patching - before

```
 t3 = 2 // B::foo (inlined)
next:
  ...

fallback:
  t2 = ldvirtfunaddr t1, A::foo
  t3 = call [t2] (x)
  goto next
```

Patching - after

```
~~t3 = 2 // B::foo (inlined)~~   ←  goto fallback
next:
  ...

fallback:
  t2 = ldvirtfunaddr t1, A::foo
  t3 = call [t2] (x)
  goto next
```