Interprocedural Analysis and Optimizations

Procedure calls can be costly

- **direct** costs of call, return, argument passing
- **indirect** cost of damage to intraprocedural analysis

Optimization techniques:

- hardware support
- inlining
- interprocedural analysis
- procedure specialization

Inlining

A.k.a. procedure integration, unfolding, beta-reduction

Replace call with body of callee

- need to manage variable scoping correctly

+ eliminate overhead of call/return sequence
+ eliminate overhead of passing arguments and returning results
+ can optimize callee in context of caller, and vice versa

– can increase compiled code space requirements
– can increase register pressure in caller
– can slow down compilation

What/where to inline?

Inline where highest benefit for the cost

- most frequently executed call sites
- call sites with small callees
- call sites with callees that benefit most from optimization

Determined by:

- explicit programmer annotations
- automatically
    - get execution frequencies from **static estimates** or **dynamic profiles**

Profile-guided automatic inline expansion

IMPACT optimizing C compiler
[Chang, Mahlke, Chen, & Hwu 92]

Use dynamic profile information to guide inline-expansion

Paper discusses some important issues:

- where in compilation process to do inlining
- how to represent programs
- avoiding hazards
- in what order to do inlining
- details on how to implement inlining

Results:

- performance improvement ~ 10%
- code space increase ~ 15%

Where to do inlining?

In linker?

+ can handle asm, library code, multiple languages

In compiler?

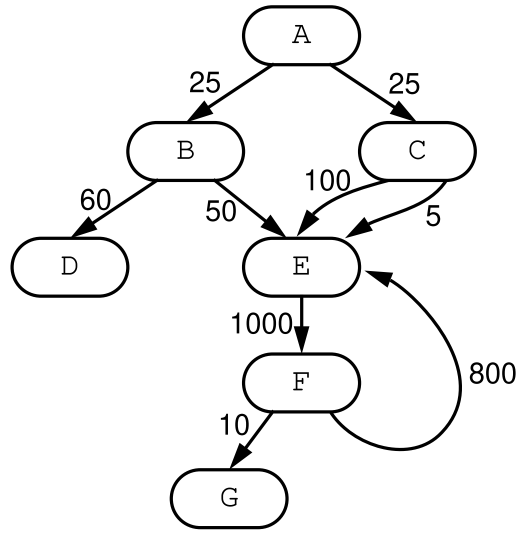
+ can apply post-inlining optimizations
– hurts separate compilation

Debugging?

How to represent programs?

Answer: **weighted call graph**

- directed multigraph
- nodes are procedures
- edges are calls, weighted by invocation counts/frequency



Hard cases:

- calls to/from external routines
- calls through pointers, function values, messages

Hazards?

Straightforward:

- callee unavailable
- multiple potential callees
- callees using `varargs`

Subtle:

- activation stack explosion

```
inline proc bar() {
  int a[1000];
  ...
}
proc foo(int i) {
  bar();
  if (i > 0) foo(i-1);
}
```

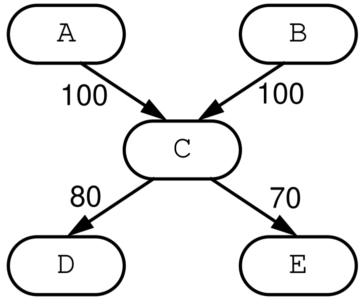
Harder:

- recursion
  - don't inline lowest-weight arc in cycle of inlinable arcs
  - inline around cycle k times

In what order to do inlining?

Strategy 1:

- inline-expand highest-weight arcs first
- stop when reach space budget or below weight threshold

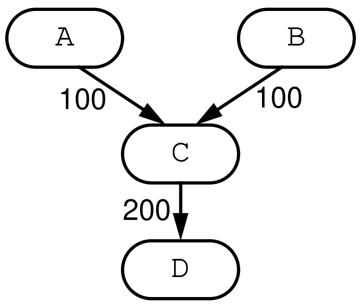


Strategy 2:

- select arcs to inline first, then do inlining bottom-up

– harder to use "space budget" cut-off method

Another example




```
proc A() {
  C(1);
}
proc B() {
  C(2);
}
proc C(int i) {
  if (i = 1) {
    D(); D();
  } else {
    ...
  }
}
```


What is profitable to inline?

How to make better automatic inlining decisions
[Dean & Chambers 94]

Benefit & cost of inlining depends on post-inlining optimizations

- can be hard to predict w/o doing them
- expensive to run optimizations to see if it's profitable

Idea: amortize cost of analysis over multiple "similar" call sites

- keep persistent database of inlining costs & benefits
- first time:
  - if plausible to inline, do it, optimize it, measure resulting costs & benefits
  - track what call site information led to important optimizations
  - back out inlining if not profitable
- later times:
  - check database for info for call site with similar info

Empirical results

- for Self system (pure OOL with heavily-used library), results of automatic inlining less sensitive to "callee size" threshold