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

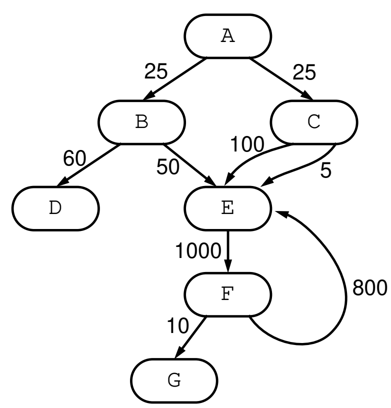
Determined by:

- explicit programmer annotations
  - annotate procedure or call site?
- automatically
  - get execution frequencies from **static estimates** or **dynamic profiles**

Program representation for profile-guided inlining

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

Some inlining design decisions

Where to do inlining?

- in compiler front-end? back-end? in linker?

What order to do inlining?

- top-down: easy
- bottom-up: avoids repeated work
- highest-weight first: exploits profile data

What hazards to avoid?

- recursive calls
- activation stack size explosion
- register pressure

Assessing costs and benefits of inlining

Strategy 1: superficial analysis

- examine source code of callee to estimate space costs
- – doesn't account for recursive inlining, post-inlining optimizations

Strategy 2: deep analysis, "optimal inlining"

- perform inlining
- perform post-inlining optimizations, estimate benefits from optimizations performed
- measure code space after optimizations
- undo inlining if costs exceed benefits
- \+ better accounts for post-inlining effects
- – much more expensive in compile-time

Strategy 3: amortized version of strategy 2 [Dean & Chambers 94]

- perform strategy 2: an "inlining trial"
- record cost/benefit trade-offs in persistent database
- reuse previous cost/benefit results for "similar" call sites
- \+ **faster** compiles than superficial approach, in Self compiler