Optimizing Procedure Calls

Procedure calls can be costly

- **direct** costs of call, return, argument & result passing, stack frame maintainance
- **indirect** cost of damage to intraprocedural analysis of caller and callee

Optimization techniques:

- hardware support
- inlining
- tail call optimization
- interprocedural analysis
- procedure specialization

Inlining

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

Replace call with body of callee

- insert assignments for actual/formal mapping, return/result mapping
  - do copy propagation to eliminate copies
- manage variable scoping correctly
  - e.g. α-rename local variables, or tag names with scopes, ...

Pros & Cons:

+ 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 slow down compilation

In what part of compiler to implement inlining?
front-end? back-end? linker?

What/where to inline?

Inline where highest benefit for the cost

E.g.:

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

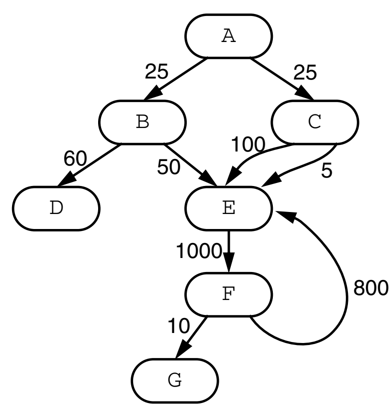
Can be chosen by:

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

Program representation for inlining

Weighted call graph: directed multigraph

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



Hard cases for building call graph:

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

Inlining using a weighted call graph

What order to do inlining?

- top-down: local decision during compilation of caller ⇒ easy
- bottom-up: avoids repeated work
- highest-weight first: exploits profile data
  - but highest-benefit first would be better...

Avoid infinite inlining of recursive calls

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

Tail call optimization

Tail call: last thing before return is a call

- callee returns, then caller immediate returns

```
int f(...) {
  ...
  if (...) return g(...);
  ...
  return h(i(...), j(...));
}
```

Can splice out one stack frame creation and tear-down, by **jumping** to callee rather than calling

- \+ callee reuses caller's stack frame & return address
- – effect on debugging?

Tail recursion elimination

If last operation is self-recursive call, turns recursion into loop ⇒ tail recursion elimination

- common optimization in compilers for functional languages
- required in Scheme language specification

```
bool isMember(List lst, Elem x) {
  if (lst == null) return false;
  if (lst.elem == x) return true;
  return isMember(lst.next, x);
}
```

Works for mutually recursive tail calls, too; e.g. FSM's:

```
void state0(...) {
  if (...) state1(...)
  else state2(...);
}
void state1(...) {
  if (...) state0(...)
  else state2(...);
}
void state2(...) {
  if (...) state1(...)
  else state2(...);
}
```

Interprocedural Analysis

Extend intraprocedural analyses to work across calls

+ avoid making conservative assumptions about:
  - effect of callee on caller
  - context of caller (e.g. inputs) on callee
+ no (direct) code increase

– doesn't eliminate direct costs of call
– may not be as effective as inlining at cutting indirect costs

Interprocedural analysis algorithm #1: supergraph

Given call graph and CFG's of procedures,
create single CFG ("control flow supergraph") by

- connecting call sites to entry nodes of callees
- connecting return nodes of callees back to calls

+ simple
+ intraprocedural analysis algorithms work on larger graph
+ decent effectiveness
  (but not as good as inlining)

– speed?
– separate compilation?
– imprecision due to "unrealizable paths"

Interprocedural analysis algorithm #2: summaries

Compute summary info for each procedure

- callee summary:
  summarizes effect/result of callee procedure for callers
- caller summaries:
  summarizes context of all callers for callee procedure

Store summaries in database
Use summaries when compiling & optimizing procedures later

For simple summaries:

+ compact
+ compute & use summaries quickly
+ separate compilation practical (once summaries computed)
– less precise analysis

A continuum in the design of summaries:

- as small as a single bit
- as large as the full source code of the callee

Examples of callee summaries

MOD

- the set of variables possibly modified by a call to a proc

USE

- the set of variables possibly read by a call to a proc

MOD-BEFORE-USE

- the set of variables definitely modified before use

CONST-RESULT

- the constant result of a procedure, if it's a constant

PURE

- a pure, terminating function, without side-effects

Computing callee summaries within a procedure

Flow-insensitive summaries can be computed without regard to control flow

+ often can be calculated in linear time
– limited kinds of information
  • cannot compute anything that depends on the relative order of execution of statements

Flow-sensitive summaries must take control flow into account

– may require iterative dfa
+ more precise info possible

Converting to SSA form and then doing a flow-insensitive analysis is often as precise as doing a flow-sensitive analysis

Computing callee summaries across procedures

If procedure includes calls, then its callee summary depends on its callees' summaries, transitively

Therefore, compute callee summaries bottom-up in call graph

What about recursion?
What about calls *to* external, unknown library functions?
What about calls *from* external, unknown library functions?
What about program changes?

Examples of caller summaries

CONST-ARGS

- the constant values of the formal parameters of a procedure, for those that are constant

ARGS-MAY-POINT-TO

- may-point-to info for formal parameters

LIVE-RESULT

- whether result may be live in caller

Computing caller summaries across procedures

Caller summary depends on *all* callers

- requires knowledge of all call sites, e.g. whole-program info

Therefore, compute caller summaries top-down in call graph

If procedure contains a call, merge info at call site with caller summary of callee

What about recursion?
What about calls *to* external, unknown library functions?
What about calls *from* external, unknown library functions?
What about program changes?

Summary functions

Idea: generalize callee summary into a callee summary *function*

- take info at call site (**calling context**) as argument
- compute info after call site as result

Also called **context-sensitive** or **polyvariant** interprocedural analysis

Previous callee summaries are **context-insensitive**: constant summary functions which ignore their input

Example calling contexts:

- which formal parameters have what constant values
- what alias patterns are present on entry
- whether the result is live (a backwards "calling" context)

Key design point for context-sensitive interprocedural analysis: how precise is the calling context?

+ more precise contexts give more precise result info
– more precise contexts take longer to produce & use summaries

Kinds of summary functions

Total function: handles all possible call site info

+ compute once for callee, e.g. bottom-up
+ reuse for all callers
– can be expensive/difficult to compute/represent precise total function

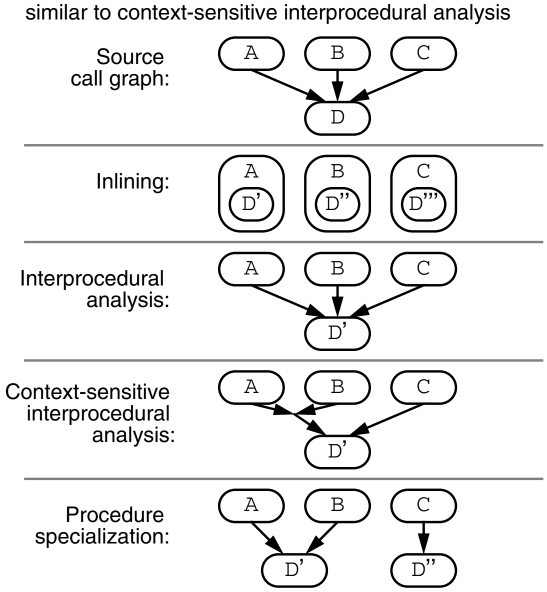
Partial function: handles only subset of possible call site infos, e.g. those actually occurring in a program

+ compute on demand when encountering new call sites, top-down
+ can be easier to represent partial functions precisely
– can analyze callee several times
– not modular

Procedure specialization

A.k.a. procedure cloning, customization

Halfway between inlining and interprocedural analysis, similar to context-sensitive interprocedural analysis



Abstract process

Given set of call sites of procedure `P`

e.g. $\{c_1, c_2, c_3, c_4, c_5\}$

Partition into equivalence classes of "similar" call sites (for instance, those with same calling context)

e.g. $\{\{c_1, c_2\}, \{c_3, c_4\}, \{c_5\}\}$

Copy `P` for each class, change calls accordingly

Do (context-insensitive) interprocedural analysis on changed call graph

Versus inlining:

+ less code explosion
+ works in presence of recursion

Versus interprocedural analysis:

+ better optimization of caller and callee

Versus context-sensitive interprocedural analysis:

+ better optimization of callee