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
- turn parameter- and result-passing into assignments
  - 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 args & returning results

\+ can optimize callee in context of caller, and vice versa

– can increase compiled code space requirements

– can slow down compilation

A question: where in compiler should inlining be implemented?
front-end? back-end? linker?

Which calls to inline?

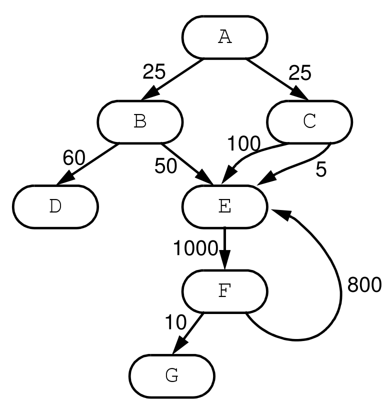
Inline calls of known functions with highest benefit for the cost
E.g.:
- most-frequently executed call sites
- call sites with small callees
- call sites with callees that benefit most from optimization
- sole call site of callee, if remove callee function after inlining

Must avoid infinite inlining of recursive calls

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

A program representation for inlining

Weighted call graph: a labeled, directed multigraph
- nodes are procedures
- edges are calls, labeled 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 consider inlining calls?

Top-down
- can be done locally, on demand, during compilation of caller
  - \+ easy to implement
  - – cannot tell if all calls of function are inlined away

Bottom-up
- – requires a global pre-pass
- \+ can identify if all calls of function are inlined away
- \+ avoids repeated transitive inlining work

Highest-weight first
- – requires a global pre-pass
- \+ can identify if all calls of function are inlined away
- \+ can exploit high-weight interior call edges
  - – still doesn't account for varying *benefits* of inlining

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
  - callee will return directly to caller's caller
- – 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 e.g. Scheme language specification
  - \+ turns stack space usage from O(N) to O(1)

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

Tail mutual-recursion elimination

Works for mutually recursive tail calls, too

E.g., FSM's written as mutually recursive functions:

```
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
  - entries become merges
- connecting return nodes of callees back to calls
  - returns become splits

+ 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

Use summaries when compiling & optimizing procedures later

Can store summaries in persistent database

Properties of typical summaries:

+ are compact
+ quick to compute & use
+ allow separate compilation (once summaries computed)
– sacrifice some analysis precision

In general, any amount of info can be captured by a summary

- as small as a single bit
- as large as the whole source code of the callee/callers

Examples of callee and caller 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

LIVE-RESULT
- whether result may be live in caller

CONST-ARGS
- the constant values of those formals that are constant

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

ARGS-MAY-POINT-TO
- may-point-to info for formal parameters

RESULT-MAY-POINT-TO
- may-point-to info for the result

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' callee summaries, transitively

Therefore, compute callee summaries bottom-up in call graph
- when encounter call site, consult previously computed summary

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

Computing caller summaries across procedures

A procedure's caller summary depends on *all* its callers
- requires complete knowledge of all call sites of a procedure, i.e. whole-program info

Therefore, compute caller summaries top-down in call graph, starting from `main`
- when encounter call site, merge call site info into callee's caller summary

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

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)

General case: **context-sensitive** interprocedural analysis

- function returns different results for different calling contexts

Simpler case: **context-insensitive** interprocedural analysis

- first merge all calling contexts into a combined context
- then function returns a single result to all calling contexts

[Supergraph yields (most precise) context-insensitive analyses]

Simplest case: **context-oblivious** interprocedural analysis

- function’s result doesn’t depend on calling context

[Simple bottom-up callee summaries are context-oblivious]

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

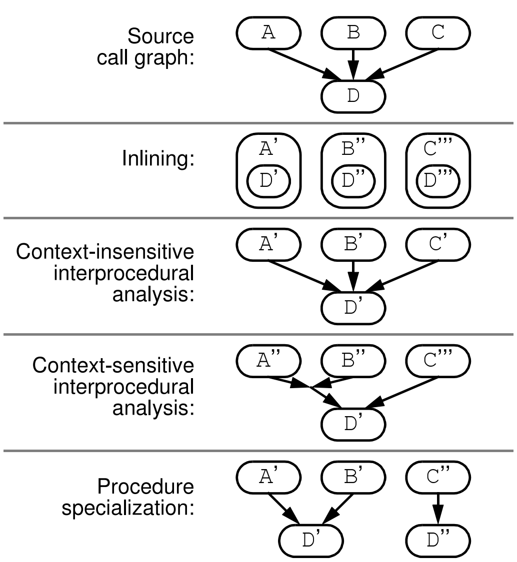
Partial function: handles only subset of possible call site infos, e.g. those actually occurring in a program

- a table mapping calling context to corresponding result info, grown as the program is analyzed

+ 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

Compile multiple versions of a procedure, each for a different calling context



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 context-insensitive interprocedural analysis:

+ better optimization of caller and callee

Versus context-sensitive interprocedural analysis:

+ better optimization of callee

Interprocedural Constant Propagation

[Callahan, Cooper, Kennedy, & Torczon, PLDI 86]

Goal: for each procedure, for each formal, identify whether all calls of procedure pass a particular constant to the formal

- e.g. stride argument passed to LINPACK library routines

Sets up lattice-theoretic framework for solving problem

- store const-prop domain element for each formal
  - initialize all formals to T
- worklist-based algorithm to find interprocedural fixed-point:

```
worklist := {Main};
while worklist ≠ ∅ do
  P := remove_any(worklist);
  processProcedure(P);
end
processProcedure(P) {
  foreach call site C in P do
    compute C's actuals from P's formals;
    C's callee's formals ∩= C's actuals;
      if changed or first time,
        add c's callee to worklist;
}
```


Jump functions

How to quickly compute info at *C*'s actuals from *P*'s formals?

Define *jump functions* to relate actual parameter at a call site to formal parameters of enclosing procedure

Different degrees of sophistication:

- **all-or-nothing**:
  only if actual is an intraprocedural constant
- **pass-through**:
  also, if formal a constant, then actual a constant
- **symbolic interpretation**:
  do full intraprocedural constant propagation

Can define similar jump functions for procedure results, too

- a total summary function for callers
- push callers on worklist if procedure's result info changes

No experimental results reported :(

Interprocedural pointer analysis for C

[Wilson & Lam 95]

A may-point-to analysis

Copes with "full" C

Key problems:

- how to represent pointer info in presence of casts, pointer arithmetic, etc.?
- how to perform analysis interprocedurally, maximizing benefit at reasonable cost?

Pointer representation

Ignore static type information, since casts can violate it

Ignore subobject boundaries, since pointer arithmetic can cross them

Treat memory as composed of untyped *blocks*

- each local & global variable is a separate block
- malloc returns a block

Assume pointer arithmetic won't cross blocks, since it's not portable

Location sets

A location set represents a set of memory locations within a block

Location set = (*block*, *offset*, *stride*)

- represent all memory locations {*offset* + i * *stride* | i ∈ Ints}
- if stride = 0, then precise info
- if stride = 1, then only know block
- simple pointer arithmetic updates offset

Examples:

Expression	Location Set
`scalar`	(scalar, 0, 0)
`struct.F`	(struct, *offsetof*(F), 0)
`array[i]`	(array, 0, *sizeof*(array[i]))
`array[i].F`	(array, *offsetof*(F), *sizeof*(array[i]))
`*(&p + x)`	(p, 0, 1)

At each program point,
a pointer may point to a set of location sets

Interprocedural pointer analysis

Want to map pointer information between caller and callee

- caller → callee:
  analyze callee given pointer relationships of formals
- callee → caller:
  update pointer relationships after call returns

Option 1: supergraph-based, context-insensitive approach

+ simple
– may be too expensive
– smears effects of callers together,
  hurting results after call returns

Some context-sensitive interprocedural analyses

Option 2a: reanalyze callee for each distinct caller

+ avoids smearing among direct callers
  (but smears across indirect callers)
– may do unnecessary work

Option 2b: reanalyze callee for *k* levels of calling context

+ less smearing
– more unnecessary work

Option 2c: reanalyze callee for each distinct calling path from `main` [Emani *et al.* 94, ...]

+ avoids all smearing
– cost is exponential in call graph depth
– recursion?

Partial summary function indexed by call site or calling path (aka **call string**), not calling context

+ a bounded number (of acyclic call strings)
– variations in call string not identical to variations in calling context

Another context-sensitive interprocedural analysis

Option 3: reanalyze for each distinct calling points-to context

- i.e., a callee summary function,
  from points-to on entry to output points-to on exit

+ avoids all smearing, even in face of recursion
+ reuse results of across equivalent call sites
– worst-case cost is O(|`Proc`| * |`PtsTo`|)

A design choice: total vs. partial summary functions

[Wilson & Lam 95]: partial summary functions

- represent function as a set of ordered pairs
  (input points-to → output points-to)
- only represent those pairs that occur during analysis
- compute pairs lazily, top-down
  - requires whole-program analysis

(Other work has explored total summary functions)

Caller/callee mapping

To compute input context from a call site,
translate into terms of callee

Modeled as **extended parameters**:

- each formal and referenced global gets a node, as does each value referenced through a pointer

Goal: make input context as general as possible
(to be reusable across many call sites)

- represent abstract points-to pattern from callee's perspective, not direct copy of actual aliases in caller
  - treat extended parameter nodes as distinct iff caller nodes are distinct
- only track points-to pattern that's accessed by callee (ignore irrelevant points-to)

Tricky details:

- constructing callee model of aliases from caller aliases
- checking new caller against existing callee input patterns
- mapping back from callee output pattern to real caller aliases
- pointers to structs & struct members ("nested" pointers)

Experimental results

For C programs < 5K lines,
analysis time was < 16 seconds and
avg # of analyses per fn was < 1.4

Analysis results were used to better parallelize two C programs

Questions:

- with bigger programs, how will # analyses per fn grow? i.e. how will analysis time scale?
- what is impact of alias info on other optimizations?

[Ruf 96]: for smallish C programs (< 15K lines),
context-*insensitive* alias analyses are just as effective as context-*sensitive* ones

Cheaper interprocedural pointer analyses

(All are context-insensitive)

Andersen's algorithm [94]: **flow-insensitive** points-to

- a single points-to graph for each procedure, as a whole

Vs. the flow-sensitive points-to algorithm from class:

- the flow-sensitive algorithm has a possibly distinct points-to graph at each program point
- the flow-insensitive points-to graph will be a superset of the union of each of these graphs
- use SSA form to retain effect of flow-sensitivity for local variables

Type-based alias analysis [Diwan *et al.* 98]: just use static types

- pointers of different static types without common subtypes cannot alias

+ "trivial", yet surprisingly effective

– restricted to statically-typed, type-safe languages with restricted multiple subtyping or whole-program knowledge

– may info only

Almost-Linear-Time Pointer Analysis

[Steensgaard 96]

Goal: scale interprocedural analysis to million-line programs

- flow-sensitive, context-sensitive analysis too expensive
- aim for linear-time analysis

Approach: treat alias analysis as a **type inference** problem
(inspired by a similar analysis by Henglein [91])

- give each variable an associated "type variable"
  - each struct or array gets a single type variable
  - each alloc site gets a type variable
- make one linear pass through the entire program; whenever one pointer var assigned to/computed from another, *unify* the type variables of their targets
  - near-constant-time unification using union/find data structures
- when done, all unified variables are **may**-aliases, un-unified variables are definitely non-aliasing

Details:

- don't do unification if assigning null or non-pointers (conditional join stuff in paper)
- pending list to enable one single pass through program

Example

```
void foo(int* a, int* b) {
  ... /* are *a and *b aliases? */ ...
}
int g;
void bar() {
  ...
  int* x = &g;
  int* y = new int;  // alloc1
  foo(x, y);
  ...
}
void baz(int* e, int* f) {
  ...
  int* i = ... ? e : f;
  int* j = new int;  // alloc2
  foo(i, j);
  ...
}
void qux(int* p, int* q) {
  ... /* are *p and *q aliases? */ ...
  baz(p, q);
}
```


Results

Analyze 75K-line program in 15 seconds,
25K-line program in 5.5 seconds
(more recent versions: Word97 (2.1Mloc) in 1 minute)

+ fast!
+ linear time complexity

[Morgenthaler 95]:
do this analysis *during parsing*, for 50% extra cost

Quality of alias info?

- Steensgaard: pretty good, except for smearing struct elements together
  - another Steensgaard paper extends algorithm to avoid smearing struct elements together, but sacrifices near-linear-time bound

[Das 00]:
extension with higher precision results that analyzes Word97 in 2 minutes

[Fahndrich *et al.* 00]: a context-sensitive extension

- "polymorphic type inference"

Type inference is an intriguing framework for fast, coarse program analysis

[DeFouw, Chambers, & Grove 98]: for OO systems