Inlining

• Replace call with body of callee
• Turn parameter- and result-passing into assignments
  – do copy prop to eliminate copies
• Manage variable scoping correctly
  – rename variables where appropriate

Program representation for inlining

• Call graph
  – nodes are procedures
  – edges are calls, labelled by invocation counts/frequency
• Hard cases for building call graph
  – calls to/from external routines
  – calls through pointers, function values, messages
• Where in the compiler should inlining be performed?

Inlining pros and cons (discussion)

Inlining pros and cons

• Pros
  – eliminate overhead of call/return sequence
  – eliminate overhead of passing args & returning results
  – can optimize callee in context of caller and vice versa
• Cons
  – can increase compiled code space requirements
  – can slow down compilation
  – recursion?
• Virtual inlining: simulate inlining during analysis of caller, but don’t actually perform the inlining

Which calls to inline (discussion)

Which calls to inline

• What affects the decision as to which calls to inline?
  – size of caller and callee (easy to compute size before inlining, but what about size after inlining?)
  – frequency of call (static estimates or dynamic profiles)
  – call sites where callee benefits most from optimization (not clear how to quantify)
  – programmer annotations (if so, annotate procedure or call site? Also, should the compiler really listen to the programmer?)
Inlining heuristics

• Strategy 1: superficial analysis
  – examine source code of callee to estimate space costs, use this to determine when to inline
  – doesn’t account for post-inlining optimizations

• How can we do better?

Inlining heuristics

• Strategy 2: deep analysis
  – perform inlining
  – perform post-inlining analysis/optimizations
  – estimate benefits from opts, and measure code space after opts
  – undo inlining if costs exceed benefits
  – better accounts for post-inlining effects
  – much more expensive in compile-time

• How can we do better?

Inlining heuristics

• Strategy 3: amortized version of 2
  – perform strategy 2: an inlining “trial”
  – record cost/benefit trade-offs in persistent database
  – reuse previous cost/benefit results for “similar” call sites

Inlining heuristics

• Strategy 4: use machine learning techniques
  • For example, use genetic algorithms to evolve heuristics for inlining
    – fitness is evaluated on how well the heuristics do on a set of benchmarks
    – cross-populate and mutate heuristics

• Can work surprisingly well to derive various heuristics for compilers

Another way to remove procedure calls

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

Tail call elimination

• Tail call: last thing before return is a call
  – callee returns, then caller immediately returns

• Can splice out one stack frame creation and destruction 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, what does tail call elimination do?

Transforms recursion into loop: tail recursion elimination
- common optimization in compilers for functional languages
- required by some language specifications, eg Scheme
- turns stack space usage from $O(n)$ to $O(1)$

Addressing costs of procedure calls

- Technique 1: try to get rid of calls, using inlining and other techniques

- Technique 2: interprocedural analysis, for calls that are left

Interprocedural analysis

- Extend intraprocedural analyses to work across calls
- Doesn’t increase code size
- But, doesn’t eliminate direct runtime costs of call
- And it may not be as effective as inlining at cutting the “precision cost” of procedure calls

A simple approach (discussion)

- Given call graph and CFGs of procedures, create a single CFG (control flow super-graph) by:
  - connecting call sites to entry nodes of callees (entries become merges)
  - connecting return nodes of callees back to calls (returns become splits)

- Cons:
  - speed?
  - separate compilation?
  - imprecision due to “unrealizable paths”
Another approach: summaries (discussion)

Code examples for discussion

```c
global a;
global b;
f(p) {
    *p := 0;
global a;
    a := 5;
f(...);
}
g() {
    a := 5;
f(&a);
b := a + 10;
}
h() {
    a := 5;
f(&b);
b := a + 10;
}
```

Another approach: summaries

- Compute summary info for each procedure
- Callee summary: summarizes effect/results of callee procedures for callers
  - used to implement the flow function for a call node
- Caller summaries: summarizes context of all callers for callee procedure
  - used to start analysis of a procedure

Examples of summaries

- MOD vars possibly modified by call (callee)
- USE: vars possibly read by call (callee)
- MOD-BEF-USE: vars defined before use (callee)
- LIVE-RES: result may be live in caller (caller)
- CONST-ARGS: values of const params (caller)
- CONST-RES: values of const results (callee)
- ARG-S-MAY-PT: may point to for args (caller)
- RESULT-MAY-PT: may point-to for result (callee)
- PURE: function has no side-effects (callee)

Issues with summaries

- Level of “context” sensitivity:
  - For example, one summary that summarizes the entire procedure for all call sites
  - Or, one summary for each call site (getting close to the precision of inlining)
  - Or ...
- Various levels of captured information
  - as small as a single bit
  - as large as the whole source code for callee/callers
- How does separate compilation work?

How to compute summaries

- Using iterative analysis
- Keep the current solution in a map from procs to summaries
- Keep a worklist of procedures to process
- Pick a proc from the worklist, compute its summary using intraprocedural analysis and the current summaries for all other nodes
- If summary has changed, add callers/callees to the worklist for callee/caller summaries
How to compute callee summaries

let m: map from proc to computed summary
let worklist: work list of procs

for each proc p in call graph do
  m(p) := ?

for each proc p do
  worklist.add(p)

while (!worklist.empty()) do
  let p := worklist.remove_any();
  // compute summary using intraproc analysis
  // and current summaries m
  let summary := compute_summary(p,m);
  if (m(p) ≠ summary)
    m(p) := summary;
  for each caller c of p
    worklist.add(c)

Examples

• Let’s see how this works on some examples

• We’ll use an analysis for program verification as a running example

Protocol checking

Interface usage rules in documentation
– Order of operations, data access
– Resource management
– Incomplete, wordy, not checked

Violated rules ⇒ crashes
– Failed runtime checks
– Unreliable software

FSM protocols

• These protocols can often be expressed as FSMs
• For example: lock protocol

FSM protocols

• Alphabet of FSM are actions that affect the state of the FSM
• Often leave error state implicit
• These FSMs can get pretty big for realistic kernel protocols

FSM protocol checking

• Goal: make sure that FSM does not enter error state
• Lattice:
FSM protocol checking

- Goal: make sure that FSM does not enter error state
- Lattice: \((L, \perp, T, E, P, U)\)
  
  \(2^{\perp, T, E, P, U}\), \(\emptyset, \{U, E\}, \top, \perp\)
What went wrong?

- We merged info from two call sites of g()
- Solution: summaries that keep different contexts separate
- What is a context?

Approach #1 to context-sensitivity

- Keep information for different call sites separate
- In this case: context is the call site from which the procedure is called

Example again

```
main() {
  f() {
    g();
    l1: if(isLocked()) {
      if (...) { main(); }
      else { lock; }
      unlock;
    }
  }
}
```