# *Interprocedural Analysis & Optimization*

*COMP 512*
*Rice University*
*Houston, Texas*

*Spring 2009*





## *Motivation*

**Why consider analysis & optimization of whole programs?**

1. To produce better code around call sites
   - Avoid saves & restores
   - Understand cross-call data flow
2. To produce tailored copies of procedures
   - Often, full generality is unneeded
   - Constant-valued parameters & globals, aliases
3. To provide sharper global analysis
   - Improve on conservative assumptions
   - Particularly true for global variables
4. To present the optimizer with more context
   - Languages (& programs) with short procedures
   - Assumes that context improves code quality

Perception is that calls are expensive (& disruptive)

## *Interprocedural Analysis & Optimization*

Whole program compilation

- Old ideas
- Trendy subject

{ Automatic detection of parallelism
{ Link-time analysis & optimization

Component technologies

- Interprocedural analysis
  - Control-flow analysis (derive a call graph)
  - Data-flow analysis
  - Estimating execution frequencies
- Interprocedural optimization
  - Inline substitution & procedure cloning
  - Activation record merging, cross-jumping, …
- Recompilation analysis



## *Fact versus Folklore*

Folklore: Create a single large graph & optimize it all

- Get all the benefits of analyzing the whole program
- See all the opportunities
- This should show the upper bound on improvement

Fortran H limited analysis to 996 names, with 2 bits for the rest

Fact: The details get in the way

- Single procedure methods get overwhelmed with details
  - Imagine a data-flow analysis with all the names
  - Two choices–summarization or slow compilation
  - And most of the detail is local
- Not clear that this approach leads to better code
- It does lead (*inexorably*) to slower compilation

See, for example, "Interprocedural optimization: Experimental Results," S. Richardson & M. Ganapathi, Software--Practice and Experience, 19, 149-169, 1989.

## *Fact versus Folklore*

Folklore: Overhead of calls is significant

- For small procedures, linkage can dominate useful work
- Should eliminate calls whenever possible (*avoid the cost*)
- Lead scientific programmers to non-modular style

Fact: Calls have costs, but they also have benefits

- Actual costs are a function of procedure size and call frequency
- Calls provide a much needed separation of concerns
  - Imagine nesting all those register lifetimes
  - Imagine using a spill-everywhere allocator!
  - Few allocators have the courage to spill everything
- Eliminating calls can slow the code down



## *Fact versus Folklore*

Folklore: Modular codes need interprocedural optimization

- Higher ratio of calls to real work
- Less straight-line code for optimizer

Fact: Opportunities are limited

- Smaller procedures have smaller name spaces
  - Fewer interactions to improve
  - Fewer common overheads, like address expressions
- Procedure call optimization is more important
  - Calls take a larger fraction of the time
  - Inline calls to eliminate linkages & create context
    - → The classic strategy for OO languages
  - Avoid copying & de-referencing parameters

## *What are the problems?*

Scoping effects limit opportunities for improvement

- Intraprocedural methods assume a common name space
  - Redundancy elimination must "see" definitions
  - Constant propagation must "know" values
- Entry & calls tap into an unknown environment
- Formal→actual mapping is onto, not one-to-one

*At source level, only formals & globals interact across procedures!*

- Modularity limits use of globals
- Call-by-value forces values through memory
- Modular programs may be inherently less efficient!

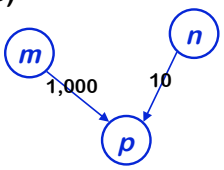
*Fortran may be the best case scenario !*



## *What are the problems?*

Different calls to *p* have different properties

- Frequency of the call
- Environment that *p* inherits
  - Mapping from parameters to value (& locations)
  - Constant values & known values
  - Size of task
- Surrounding execution context
  - Is call in a parallel loop?
  - Which registers are unused?
- Procedure-valued parameters (OOP)



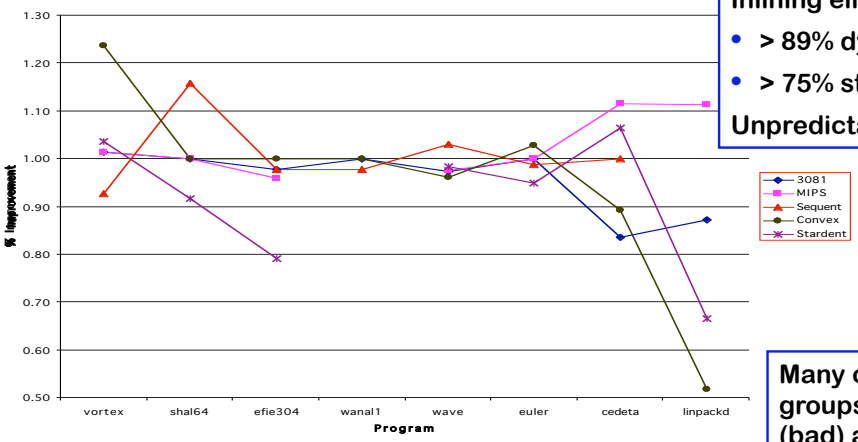

*p* must function correctly in *both* contexts

May want to optimize distinct calls differently

- Consider simple issue, such as caller-saves vs. callee-saves

## *What are the problems?*

**Profitability**




**Inlining eliminated**

- **> 89% dynamic calls**
- **> 75% static calls**

**Unpredictable results**

**Many commercial groups have secret (bad) anecdotes about inlining!**

**In real life, its hard to pick the right strategy**

- **Combination of program, compiler, input data**
- **No single strategy fits all cases**



## *What are the problems?*

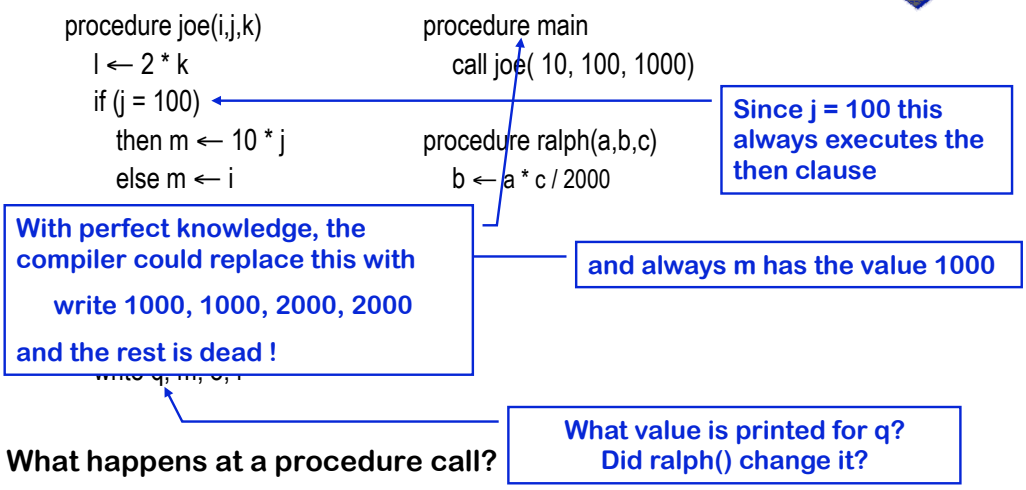
**Compilers are engineered objects**

- **Implementations contain unwritten assumptions**
  - **Code shape**
  - **Finite data structures**
- **Call sites provide some great big hints**
  - **Limit the global impact of local effects**
  - **Break lifetimes & reset analysis**
- **Smaller procedures map more easily onto small resources**
  - **32 (or 32 + 32) registers**
  - **Less control-flow ⇒ better knowledge**

**Separation of concerns**

- **Global optimization work well with plentiful resources**
- **In larger scopes, resources are taxed & separation breaks down**

## *What are the problems?*




**What happens at a procedure call?**

- **Use worst case assumptions about side effects**
- **Leads to imprecise intraprocedural information**
- **Leads to explosion in intraprocedural def-use chains**



## *Interprocedural Analysis*

**The compiler needs to understand call sites**

- **Limit loss of information at calls**
- **Shrink intraprocedural data structures**
  - **Def-use chains in PFC**
- **Solve simple intraprocedural problems**
  - **Shadow loop indices in Fortran**

**Interprocedural effects limit intraprocedural analysis**

- **Grove & Torczon showed major impact of call sites on SCCP**
  - **Each call site killed many potential constants**
- **Knowledge about modifications eliminated most of it**

## *Interprocedural Analysis*

Definitions

- May problems describe events that might happen in a call
  - *May Modify* sets include any name the call might define
  - *May Reference* sets include any name the call might use
- Must problems describe events that always happen in a call
  - *Must Modify* set describes KILLs

Computation can consider or ignore control-flow in procedure

- Flow-insensitive analysis ignores intraprocedural control-flow
- Flow-sensitive analysis tracks intraprocedural control-flow

Some combinations don't make sense
  - Flow-insensitive must modify sets?

In the presence of aliasing and recursion, flow-sensitive analysis problems are either NP complete or Co-NP complete
(Eugene Myers, 8th POPL)



## *Interprocedural Analysis*

Classical Problems have appeared in the literature

- Constructing the call graph
- May summary problems – MAYMOD & MAYREF
- Alias tracking – MAYALIAS
- Constant propagation
- Kill information – MUSTMOD

New-fangled problems appear all the time

- Alignment propagation for caches
- Propagating data distributions for FORTRAN D
- Placing instrumentation & picking checkpoints

## *Constructing the Call Graph*

**Solution: Ryder, 1979 (non-recursive Fortran)**

- **Build subgraph described by literal constants**
- **Propagate sets of values for procedure variables**
- **Complexity is linear in size of call graph**

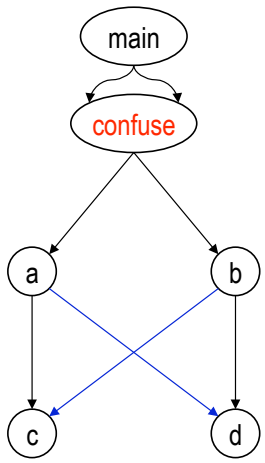
**Procedure-valued variables complicate the process**

- **Must track values of variables (constant propagation)**
  - **Typically (& fortunately) no arithmetic**
- **Results can be approximate (overestimate) or precise**



## *Constructing the Call Graph*

```
procedure main
    call confuse(a,c)
    call confuse(b,d)
procedure confuse(x,y)
    call x(y)
procedure a(z)
    call z()
procedure b(z)
    call z()
procedure c()
    …
procedure d()
    …
```




**Imprecise call graph**

## *Constructing the Call Graph*

Algorithms that handle recursion

- Callahan, Carle, Hall, Kennedy (87 & 90)
  - Iterative algorithm
  - Build a precise graph
  - O(N + $PE^{CP}$) logical steps
- Burke (87)
  - Interval analysis
  - Conservative, approximate graph
  - O($NE_p^2P$ + *d* $E_pEP$) bit-vector steps
- Hall, Kennedy (90)
  - Iterative algorithm
  - Conservative, approximate graph
  - O(N + PE) logical steps

CP is maximum number of formal parameters to any procedure in the program



## *Constructing the Call Graph*

All of these algorithms assume Fortran

- Newer ones include recursion (C)
- No procedure-valued functions (no return)
- Calls to variables are textually obvious

More complex situations

- Assignment to procedure-valued variables
- Procedure-valued functions (static or dynamic)
- Arithmetic on procedure-valued functions
- Object-oriented programs
  - Value-based bindings

## *Alias Analysis*

The Approach

- Factor base problem into alias-free solution + ALIAS sets
- Combine two to solve the base problem
- ALIAS set tracks effects of parameter binding

Definition

*When a memory location can be accessed by > 1 name, those names are said to be aliases of each other*

Strategy

- Identify sites that introduce aliases
- Propagate pairs of aliases around the call graph

{
1. a "global" variable is passed as an actual parameter
2. a single name is passed in > 1 parameter positions p
}



## *Summary Problems*

MOD: $v \in \text{MOD}(s) \Leftrightarrow$ executing s might change v's value

REF: $v \in \text{REF}(s) \Leftrightarrow$ executing s might change v's value

These formulations can be solved with a flow-insensitive method

The equations

$$\text{GMOD}(p) = \text{IMOD}(p) \cup \cup_{e=<p,q>} b_e(\text{GMOD}(q))$$

$$\text{DMOD}(e) = b_e(\text{GMOD}(q))$$

Where

GMOD(p): variables that may be modified by an execution of p
IMOD(p): variables that may be modified locally in p
DMOD(s): variables that may be modified by call site s, ignoring aliasing

Solve on either the call graph or the binding graph $\beta$

## *Constant Propagation*

The Problem

Annotate each procedure with a set CONSTANTS(p) that contains < *name,value* > pairs

Approximate the answers &

**A completely different approach:**

Wegman & Zadeck, in their work on Sparse Conditional Constant Propagation, suggest performing all interprocedural constant propagation by inlining during intraprocedural constant propagation.

This will work, but involves all the problems of inlining and compiling very large procedures. The idea is not completely practical.

We could consider, however, doing this for the purposes of analysis. It would yield a large set of constants. The problem, however, is that mapping those constants back onto the non-inlined program may require some deep reasoning and/or additional transformations.



## *Constant Propagation*

Three important effects to model

1. Constant values available at call sites
   - Perform local constant propagation to find values
   - Use results as initial information
2. Transmission of values across call sites
   - Model this with an iterative data-flow framework
   - It will behave, but must use lattice depth to prove time bound
3. Transmission of values through procedure bodies
   - Can imagine many schemes for handling this issue
   - Values transmitted forward & backward (*returns*)
   - Approximate procedure's behavior with *jump functions*

## *Constant Propagation*

Jump functions

- Varying levels of complexity
- Use return jump functions for parameters & global variables
- Use summary information (MOD) in jump function bodies

Given a set of jump functions

- Can use a simple iterative data-flow solver
- If *support* of jump function is bounded, have a linear time bound
- As in SCCP, values are either TOP, BOT, or $c_i$


Callahan, Cooper, Kennedy, & Torczon, SIGPLAN 86, June 1986
Grove & Torczon, PLDI 93, June 1993


## *Pointer Disambiguation*

Problem is quite hard

- Intraprocedural analog is harder than SCCP
- Problem is inherently interprocedural

Two competing formulations

- ALIAS formulation (*tuples of aliased pointers*)
- POINTS-TO formulation
- Propagation involves modeling effects of assignments

Other complications

- Many values have no explicit names (*malloc()*)
- Syntax obscures their types, sizes, and other attributes

## *Enough Analysis, What Can We Do?*

There are interprocedural optimizations

- Choosing custom procedure linkages
- Interprocedural common subexpression elimination
- Interprocedural code motion
- Interprocedural register allocation
- Memo-function implementation
- Cross-jumping
- Procedure recognition & abstraction



## *Linkage Tailoring*

Should choose the best linkage for circumstances

- Inline, clone, semi-open, semi-closed, closed
- Estimate execution frequencies & improvements
- Assign styles to call sites

} The choices interact

Practical approach

- Limit choices (*standard, cloned, inlined*)
- Clone for better information & to specialize (*based on idfa*)
- Inline for high-payoff optimizations
- Adopt an aggressive standard linkage
  - Move parameter addressing code out of callee (& out of loop)
  - Parts of prologue & epilogue that are predictable in caller

## Improving Linkages

Attack the actual inefficiencies

- Save & restore code
  - Live across a call ⇒ target callee-saves register
  - Not live across a call ⇒ target caller-saves register
  - Leaf procedure ⇒ target caller-saves register
- Optimize actual save & restore code
  - Code space issue ⇒ use a common library routine
  - Avoid saving anything that is not needed
- Generate parameter bindings in a way that will allocate well
  - Copy actual into formal & count on coalescing (when possible)
- Return address in a register?
  - Make it the first spill (store at entry, load at exit)
- Expose pre-call & post-return sequences to LICM & GCSE
  - Expand operations before optimization



## *Interprocedural Common Subexpression Elimination*

Consider the domain carefully

- Procedures only share parameters, globals, & constants
- No local variables in an ICSE
- Not a very large set of expressions
  - Includes, however, parameter & global variable addresses

Possible schemes

- Create a global data area to hold ICSEs
  - Sidestep issue of register pressure
- Ellide unnecessary parameters
  - Speeding up linkages

## *Interprocedural Loop-Invariant Code Motion*

**Generalities**

- **Finding interprocedural loop nests**
- **Computing interprocedural AVAIL** **(imagine ILCM)**
- **Moving code across procedure boundaries**
  - **And tracking it for later compile-link cycles …**

**All are difficult**

**What about invocation invariant expressions?**

- **Expression whose value is determined at point of call**
- **Find a subset of these *iie*'s**
- **Hoist them to the prolog, or hoist them across the call**



## *Invocation Invariant Expressions*

**Do iie's exist? (*yes*)**

**Is moving them profitable? (*it should be*)**

**Can we engineer this into a compiler? (*this is tougher*)**

| Distinct *iie* s found in FMM | | | | | | |
|---|---|---|---|---|---|---|
| Routine | size | add | sub | mult | conv | total |
| decomp | 933 | 3 | 1 | 0 | 1 | 5 |
| fehl | 621 | 33 | 0 | 0 | 1 | 34 |
| fib | 57 | 2 | 2 | 0 | 0 | 4 |
| fmin | 601 | 11 | 0 | 0 | 1 | 12 |
| rfk45 | 191 | 6 | 5 | 6 | 0 | 17 |
| rkfs | 1502 | 23 | 1 | 0 | 1 | 25 |
| seval | 219 | 1 | 0 | 0 | 0 | 1 |
| solve | 287 | 1 | 1 | 1 | 1 | 4 |
| spline | 1105 | 16 | 4 | 4 | 1 | 25 |
| svd | 2372 | 3 | 0 | 0 | 1 | 4 |
| zeroin | 462 | 8 | 0 | 0 | 1 | 9 |

**8.9% to 0.2% (static)**

**1.66% on average**

## *Interprocedural Register Allocation*

Some work has been done

- Chow's compiler for the MIPS machine
  - Often slowed down the code
- Wall's compiler at DEC SRC did link-time allocation
  - Showed consistent improvement
  - Target machine had scads of registers (IA 64?)

Critiques

- Arithmetic of costs is pretty complex
- Requires good profile or frequency information
- Need a fair basis for comparing different uses for $r_i$



## *Memo-function implementation*

Idea

- Find pure functions & turn them into hashed lookups

Implementation

- Use interprocedural analysis to identify pure functions
- Insert stub with lookup between call & evaluation

Benefits

- Replace evaluations with table lookup
- Potential for substantial run-time savings
- Should share table implementation with other functions

## *Cross-jumping*

**Idea**

- Procedure epilogs come in two flavors
  - Returned value & no returned value
- Eliminate duplicates & save space

**Implementation**

- At start of each block, compare ops before predecessor branch
- If identical, move it across the branch
- Repeat until code stops changing

**Presents new challenges to the debugger**



## *Procedure Abstraction*

**Idea**

- Recognize common instruction sequences
- Replace them with (very) cheap calls

**Experience**

- Need to abstract register names & local labels
- Use suffix trees ~ 1% slower for each 2% smaller
- Causes havoc for debugger

## *Where Can IDFAO Have An Impact?*

There are few killer interprocedural optimizations

- Inlining and cloning, particularly for OO programs
- Analysis to avoid method lookup in class hierarchy
- Trace caches in dynamic systems

What about traditional Algol-like languages?

- MayMod helps single-procedure constant propagation (CP) — Grove & Torczon 93
  - Lets SCCP pass constants across a call
- Procedure cloning on forward constants helps ICP — Metzger & Stroud 92
- Pointer analysis enables register promotion (Lecture 22) — Lu & Cooper 97
- Inlining reduces call overhead — Scheiffler 77
  - Particularly with small procedures, data abstraction languages (CLU), & OOLs (C++) — OOPLSA



## *What Mechanisms are Needed?*

Analyzer needs access to entire program's text

Three alternatives

- Programming environment, like $R^n$
  - Compiler can access text as needed
  - Compiler can analyze compilation dependences
- Classic compiler and a program repository (*Convex*)
  - Simplified version of $R^n$
- Link time analysis & optimization
  - Redo entire analysis & optimization on every compile
  - No change to programming model