CS 526

Advanced
Compiler
Construction

http://misailo.cs.Illinois.edu/courses/cs526

INTERPROCEDURAL ANALYSIS

The slides adapted from Vikram Adve

So Far…

Control Flow Analysis

Data Flow Analysis

Dependence Analysis

Points-to Analysis

Abstract Interpretation

All within a single procedure

(intraprocedural)

Today

Control Flow Analysis

Data Flow Analysis

Dependence Analysis

Points-to Analysis

Abstract Interpretation

Across multiple procedures

(interprocedural)

Today

Control Flow Analysis

Key question to answer:

How to deal with function call y = f(x)?

(we will describe this for a subset of techniques)

Abstract Interpretation

Why interprocedural analysis and optimization?

- **Produce better code around call sites**

  avoid saves, restores; understand cross-call site data flow

- **Produce tailored copies of procedures**

  often, full generality is not necessary;
  constant valued parameters, aliases

- **Provide sharper global (*intraprocedural*) analysis**

  improve on conservative assumptions

  especially true for global variables

- **Present the optimizer with more context**

  languages with short procedures; assumes context improves code

Key Challenges

Compilation Time, Memory

Key problem: scalability to large programs

- Dominated by analysis time/memory
- Flow-sensitive analyses: bottleneck often memory (!time)
- ⇒ Often limited to fast but imprecise analyses

Multiple calling environments

Different calls to P() have different properties:

- known constants, aliases, surrounding execution context (e.g., enclosing loops), function-pointer arguments, …
- frequency of the call

Key Challenges

Recursion

Recursive codes are typically like most difficult types of loops

- No induction variables, complex data structures, complex termination

Estimating profitability

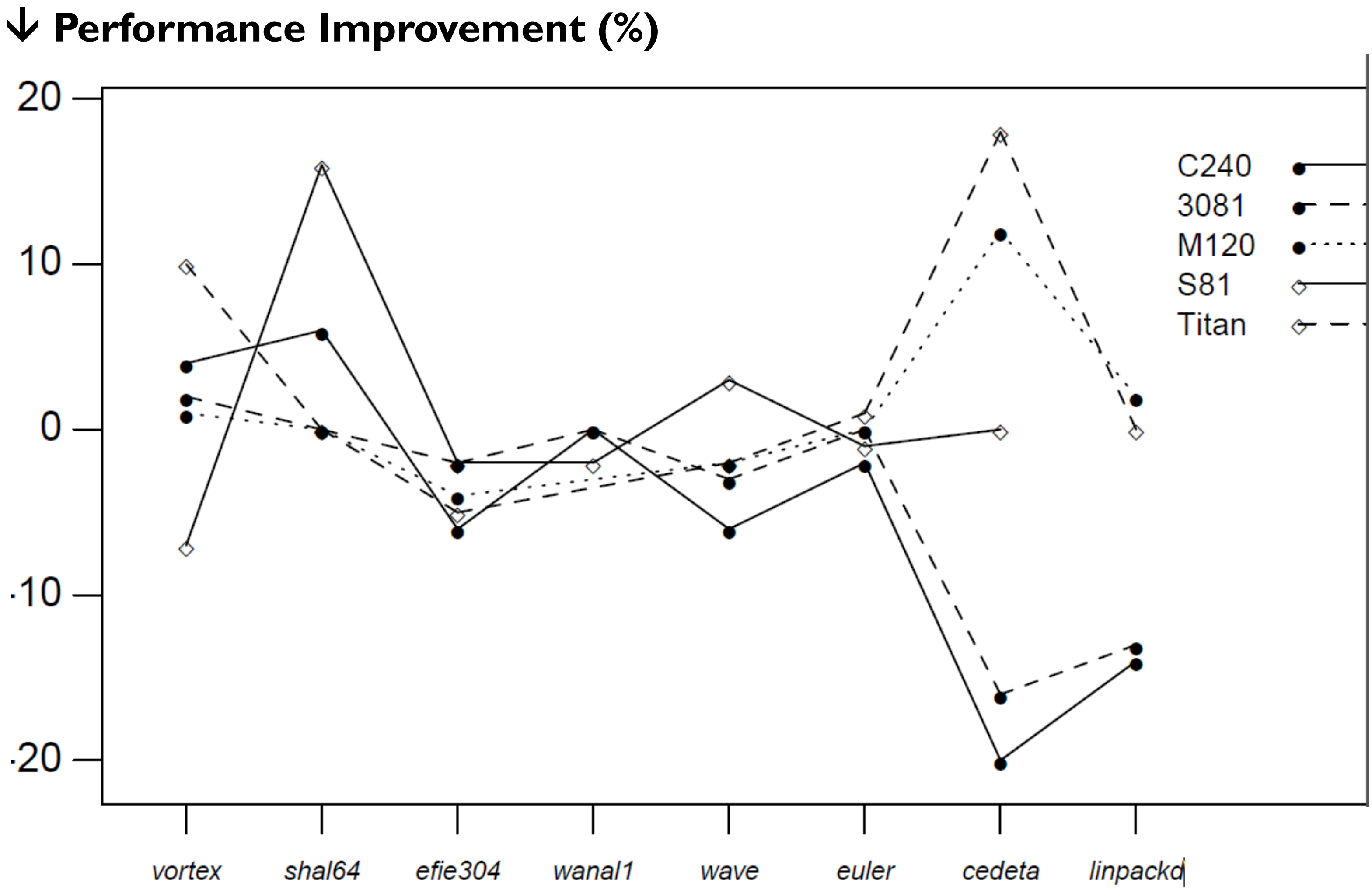
- even inlining is not clear win
- separation of concerns:
  - ignores resource constraints
  - works best with smaller procedures

Solution #1:
Reduction to Intraprocedural

1. **Conservative:**
   - Analyze each function separately
   - At every function call, invalidate all global variables
   - The result for each function is conservative, for all values of the input variables

2. **Inlining:**
   - At each call, insert the function body
   - Can optimize better, use local values of variables
   - However, the control flow graph grows exponentially
   - Also, recursion causes problems

Inlining Benefits



An Experiment with Inline Substitution, Cooper et al. 1991

Solution #2:
Analyze Global Flows

Create Whole-Program CFG

- Possible unrealizable paths
- Tradeoff between precision and space

Call String Approach

- Maintain the context of caller, each call site can have a different analysis
- Call context simulates stack
- Finite unrolling for recursion

Realizable Paths

Definition: Realizable Path

A program path is realizable iff every procedure call on the path returns control to the point where it was called (or to a legal exception handler or program exit)

Whole-program Control Flow Graph?

Conceptually extend CFG to span whole program:

- split a call node in CFG into two nodes: CALL and RETURN
- add edge from CALL to ENTRY node of each callee
- add edge from EXIT node of each callee to RETURN

Problem: This produces many unrealizable paths

Focusing only on realizable paths requires context-sensitive analysis

MOP and MVP Solutions

Previously, we learned about meet-over-paths (MOP) solutions for dataflow equations

- These were desired solutions of the analysis

For interprocedural analysis, we need to define a new **meet-over-valid-paths (MVP)** solution, which only combines dataflow facts over the realizable paths.

- Avoids the paths induced by conservative whole-program CFG.
- These would be the desired solutions of interprocedural problems

Building the Call Graph

Function pointer variables make this problem hard!

Fortran: only formal arguments (no assignment)

C, C++, Java, . . . : arbitrary function pointer variables and uses

```
void main () {
   confuse(a,c)
   confuse(b,d)
}

void confuse(x,y) { x(y) }

void a(z) { z() }
void b(z) { z() }
void c { ... }
void d { ... }
```

Languages with Function Pointer Assignment

Approach 1: Solve CALLS and ALIAS separately

- Compute whole-program call graph
- Solve ALIAS
- Refine call graph

(Iterate ALIAS and CALLS until there are no changes)

Approach 2: Solve CALLS and ALIAS simultaneously

Context-sensitive alias analysis algorithms can discover call graph as they propagate points-to sets:

- Liang and Harrold (FSE 1999)
- Fähndrich, Rehof and Das (PLDI 2000)
- Lattner and Adve (PLDI 2007)

Call Graph: Previous Results

Fortran with Recursion

Precise graph: Callahan, Carle, Hall, Kennedy (87, 90)

- $O(N^{vmax+1})$ logical steps N = #procedures
  vmax = max. #procedure-valued parameters for any procedure

Conservative, approximate graph: Hall, Kennedy (90)

- $O(N + PE)$ logical steps P = #procedures passed as parameters

Object-oriented Languages

A framework for call graph construction algorithms, David Grove, Craig Chambers. *ACM TOPLAS,* 23(6), November 2001

- Describes several alternative algorithms in a common framework
- Incorporates class hierarchy analysis, MOD, exception analysis, escape analysis

Solution #3:
Functional Approach

Previous: Saves space, but still iterates many times of the function

Goal: Establish the input/output relationship for the function, i.e., compute function summary

- Analyze once, compute function summary
- At call sites, specialize this summary, without looking at the body
- For recursive calls, unroll

Classification of IP* Analyses

Flow-insensitive: computes a single result for entire program/procedure

- Can be solved in time polynomial in the size of the call graph (Banning, POPL, 1979)

Flow-sensitive: computes distinct result for each program point

- NP-complete or Co-NP complete (Myers, POPL, 1981).

Context-insensitive: includes realizable and unrealizable paths

Context-sensitive: explicitly excludes unrealizable paths

May problems describe events that may happen as the result of executing a given call

Must problems describe events that always happen when a given call is executed

IP* = Interprocedural

Classification of IP Analyses

Call Graph:

- represents how the procedures (subprograms) are being called within the program code
- Nodes represent procedures, e.g., f, g…
- Edges (f, g) specify the caller and the callee,
  e.g., procedure f calls procedure g.
- A cycle in the graph indicates recursive procedure calls

Classical IP problems

Side-effect problems: "backward" IP dataflow problems

Propagation problems: "forward" IP dataflow problems

(where backward and forward refer to call-graph).

- **CALLS:** Constructing the call graph
- **ALIAS:** Alias analysis
- **MOD:** Variables possibly modified due to a call
- **REF:** Variables possibly used due to a call
- **KILL:** Variables definitely modified before use due to a call
- **USE:** Variables possibly used before being modified due to a call
- **CONST:** Constant propagation

Interprocedural Side-Effect Problems

"A Schema for Interprocedural Modification Side-Effect Analysis with Pointer Aliasing," W. Landi et al., ACM TOPLAS, March 2001.

Problems (for a call site s)

- **MOD(s):**
  v ∈ MOD(s) iff statement s might change v's value
- **MOD(P):**
  v ∈ MOD(F) iff function F might change v's value
- Similarly REF(s), REF(F):
  v ∈ REF(*) iff statement/function might reference v's value

Interprocedural Side-Effect Analysis

Compute: MOD(s), MOD(F), REF(s), REF(F)

Strategy

1. Perform interprocedural alias analysis (perhaps context-sensitive)
2. Compute direct side-effects of assignments
3. Solve dataflow equations iteratively on the Interprocedural Control Flow Graph
   - Use context (reaching aliases – **RAs**) in each dataflow equation

Interprocedural Side-Effect Analysis

Assumptions:

- Simple programs
- No global variables
- “By-reference” passing: pointers

Interprocedural Side-Effect Analysis

From Local Analysis:

- **DIRMOD(s):** variables directly modified by assignment s
- **B_C(VarSet):** Translates VarSet from names in callee (F) to names in caller at call-site C

IP dataflow problem is decomposed into several dataflow equations. They are solved by iteration on the call graph.

Interprocedural Side-Effect Analysis

CondLMOD(n, RA):

variables modified by assignment n due to aliases after any predecessor of n, under context RA

$$\text{condLMOD}(n, RA) = \bigcup_{p:p\rightarrow n} \left\{ X_1 \;\middle|\; \begin{array}{l} (X_1, X_2) \in Alias(p, RA) \\ \wedge \; X_2 = \text{DIRMOD}(n) \end{array} \right\}$$

Interprocedural Side-Effect Analysis

CondIMOD(P, RA):

variables modified by assignments in procedure P, under context RA

$$\text{condIMOD}(P, RA) = \bigcup_{\text{assignments } n \in P} \text{condLMOD}(n, RA)$$

Interprocedural Side-Effect Analysis

PMOD(P,RA):

variables modified by procedure P under RA

$$\textsf{PMOD}(P, RA) \;=\; \textsf{condIMOD}(P, RA) \;\cup \bigcup_{\substack{C_Q \in P \,:\, \text{call to } Q \\ RA' \in \textit{contexts\_of}(C_Q, RA)}} b_{C_Q}(\textsf{PMOD}(Q, RA'))$$

Interprocedural Side-Effect Analysis

CMOD(n,RA):

variables modified by statement n under RA

$$\text{CMOD}(n, RA) = \begin{cases} \text{condLMOD}(n, RA) & \text{if } n \text{ is an assignment} \\ \bigcup\limits_{RA' \in contexts\_of(n,RA)} b_n(\text{PMOD}(Q, RA')) & \text{if } n \text{ is a call to Q} \\ \phi & \text{otherwise} \end{cases}$$

Interprocedural Side-Effect Analysis

Finally:

$$\mathsf{MOD}(n) \quad = \bigcup_{all\ \ contexts\ \ RA\ \ for\ \ P} \mathsf{CMOD}(n, RA))$$

$$\mathsf{MOD}(P) \quad = \bigcup_{all\ \ contexts\ \ RA\ \ for\ \ P} \mathsf{PMOD}(P, RA))$$

IP Constant Propagation

The problem

Compute sets of pairs (*name,value)* at entry to each function and after each call site, where *value* is an element of the usual CONST lattice ($\top$,$\bot$, or constant value).

Key considerations

1. Constant values available at call sites
    - deriving initial information
2. Transmission of values across call sites and returns
    - interprocedural data-flow problem
3. Transmission of values through procedure bodies
    - single procedure data flow (*jump function*)

IP Constant Propagation

Challenges:

1. Overall problem is undecidable.
2. Constant propagation is flow-sensitive:

⇒ Must have all procedures in memory simultaneously

Solution: Capture approximate effects of function bodies with "**jump functions**."

Callahan, Cooper, Kennedy, and Torczon, "Interprocedural constant propagation", SIGPLAN 86, July 1986.

Interprocedural Constant Propagation: A Study of Jump Function Implementations, Dan Grove and Linda Torczon. PLDI 1993.

IP Constant Propagation

Build interprocedural value graph

- analogous to the SSA graph used in SCCP
- standard CONST lattice: values are either $\top$, (constant), or $\bot$

Use a standard iterative approach:

- maintain a worklist of formal parameters
- add a parameter to the worklist every time it changes value
- any parameter changes value at most twice

IP Constant Propagation

Use two types of jump functions:

- **forward jump function:** value passed to a formal parameter at a call-site (as function of formal parameters of caller)
- **return jump function:** each return value from a procedure (as a function of formal parameters of the procedure)

Example Jump Functions

Literal Constant Jump Function:

$J_s^y = c$, if y is the literal constant c at call site s (else, ⊥)

Intraprocedural Constant Jump Function:

$J_s^y = c$, if intraprocedural analysis can prove y = c at call site s (else, ⊥)

Pass-through Parameter Jump Function:

$J_s^y = c$, (as above), or

x, if y = x at s and x is a formal parameter of caller (else, ⊥)

Polynomial Parameter Jump Function:

$J_s^y = c$(as above), or

f($\vec{x}$) if y = f($\vec{x}$) at s, where $\vec{x}$ are formal parameters of caller and f is a polynomial function (else, ⊥)

INTERPROCEDURAL OPTIMIZATIONS

Inline Substitution

The code from one subroutine is substituted at the call site; formal parameters are replaced by actual parameters:

```
int f (int x) {
    int r = g(x);
    return r; }
int g(int y) {
    return 2*y}
```

```
int f (int x) {
    int r = 2*x;
    return r;
}
```

- Can always be applied
- But can be too expensive (exponential blowup)
- Recompilation of a single function will cause project recompilation

Function Cloning

Specialize function for specific values of the parameters

```
int f(int a[], int s) {
  for (i=0;i<len(a);i++)
    a[i*s-s+1]=
         a[i*s-s+1]+3;
}
```

Vectorizable when s>0,
not vectorizable when s=0

```
int f_s1(int a[], int s) {
  for (i=0;i<len(a);i++)
    a[i*s-s+1]=a[i*s-s+1]+3;
}
int f_s0(int a[], int s) {
  for (i=0;i<len(a);i++)
    a[1]=a[1]+3;
}
```

- Enhances the applicability of constant propagation

Separate Compilation

The problem

Interprocedural data flow analysis introduces subtle dependences

- optimized procedures are program-specific
- correctness of object code depends on whole program

Changing one procedure can force many compilations:

- the procedure, itself, for different programs
- other procedures within those programs

Solution: Separate Compilation

- Allows subsets of a program to be compiled separately and then linked together into a final executable.
- After a module is changed, only need to re-do selected optimizations on selected procedures
- Analysis to determine which files were changed: **dataflow!**