Interprocedural Analysis and Optimization

Mod/Ref Analysis
Alias Analysis
Constant Propagation
Procedure Inlining and Cloning

Introduction

- Interprocedural Analysis
  - Gathering information about the whole program instead of a single procedure
    - Examples: side-effect analysis, alias analysis
- Interprocedural Optimization
  - Modifying more than one procedure, or
  - Using interprocedural analysis

Interprocedural Side-effect Analysis

- Modification and Reference Side-effect
  - MOD(s): set of variables that may be modified as a side effect of call at s
  - REF(s): set of variables that may be referenced as a side effect of call at s

```
COMMON X,Y
...
DO I = 1, N
S0:    CALL P
S1:    X(I) = X(I) + Y(I)
ENDDO
```

- Can vectorize if
  - P neither modifies nor uses X
  - P does not modify Y

Interprocedural Alias Analysis

```
SUBROUTINE S(A,X,N)
   COMMON Y
   DO I = 1, N
S0:    X = X + Y*A(I)
   ENDDO
END
```

- Could we keep X and Y in different registers?
  - What happens if S is called with parameters S(A,Y,N)?
    - Y is aliased to X on entry to S (Fortran uses call-by-ref)
    - Can't put X and Y in different registers
- For each parameter x, compute ALIAS(p,x)
  - The set of variables that may refer to the same location as formal parameter x on entry to p

Call Graph construction

- Interprocedural analysis must model how procedures call each other
  - Two approaches: call graph and interprocedural control flow graph
- Call Graph: G=(N,E) model call relations between procedures
  - N: one vertex for each procedure
  - E: p->q: if procedure p calls q; one edge for each possible call
- Construction must handle function pointers (procedure parameters)

```
    SUBROUTINE S(X,P)
S0:     CALL P(X)
        RETURN
    END
```

- P is a procedure parameter to S
  - What values can P have on entry to S?
  - CALL(s): set of all procedures that may be invoked at s (alias analysis)

Flow Insensitive Side-effect Analysis

- Goal: compute what variables may be modified by each procedure
  - Interprocedural analysis
- Assumptions
  - Procedure definitions are not nested inside one another
  - All parameters passed by reference
  - Each procedure has a constant number of parameters
  - Procedures may recursively invoke each other
- We will formulate and solve the MOD(s) problem

Solving MOD

- MOD(s): variables modified by the call of procedure p at call site s

$$MOD(s) = DMOD(s) \cup \bigcup_{x \in DMOD(s)} ALIAS(p,x)$$

  - DMOD(s): set of variables directly modified as side-effect of call at s

$$DMOD(s) = \{v \mid s \Rightarrow p, v \xrightarrow{s} w, w \in GMOD(p)\}$$

  - GMOD(p): set of global variables and formal parameters of p that are modified, either directly or indirectly as a result of calling p

```
S0: CALL P(A,B,C)
SUBROUTINE P(X,Y,Z)
        INTEGER X,Y,Z
        X = X*Z
        Y = Y*Z
END
```

GMOD(P)={X,Y}

DMOD(S0)={A,B}

Solving GMOD

- GMOD(p) contains two types of variables
  - IMOD(p): variables explicitly modified in body of P
  - Variables modified as a side-effect of some procedure invoked in p
    - Global variables are viewed as parameters to a called procedure

$$GMOD(p) = IMOD(p) \cup \bigcup_{s=(p,q)} \{z \mid z \xrightarrow{s} w, w \in GMOD(q)\}$$

  - May take a long time to converge due to recursive procedure calls

Solving GMOD

- Decompose GMOD(p) differently to get an efficient solution
- Key: Treat side-effects to global variables and reference formal parameters separately

$$GMOD(p) = \boxed{IMOD^{+}(p)} \cup \boxed{\bigcup_{s=(p,q)} GMOD(q) \cap \neg LOCAL}$$

  - where

$$IMOD^{+}(p) = IMOD(p) \cup \bigcup_{s=(p,q)} \{z \mid z \xrightarrow{s} w, w \in RMOD(q)\}$$

  - ***RMOD(p):*** set of formal parameters that may be modified in p, either directly or by used as actual parameter to call another procedure q

Alias Analysis

- Recall definition of MOD(s)

$$MOD(s) = DMOD(s) \cup \bigcup_{x \in DMOD(s)} ALIAS(p, x)$$

- Need to
  - Compute ALIAS(p,x)
  - Update DMOD to MOD using ALIAS(p,x)
- Key Observations
  - Two global variables can never be aliased of each other.
  - Global variables can only be aliased to formal parameters
  - The number of aliases for each variable is bounded by the number of formal parameters and global variables
    - Not true in C/C++ code when data can be dynamically allocated

Update DMOD to MOD

```
SUBROUTINE P
   INTEGER A
S0:    CALL S(A,A)
END

SUBROUTINE S(X,Y)
   INTEGER X,Y
S1:    CALL Q(X)
END

SUBROUTINE Q(Z)
   INTEGER Z
   Z = 0
END
```

GMOD(Q)={Z}

DMOD(S1)={X}

MOD(S1)={X,Y}

Interprocedural Optimizations

- The goal of interprocedural analysis is to enable whole program optimizations
- Can we understand procedural calls just like regular statements?
  - MOD/REF -- set of variables modified/referenced in procedure
  - ALIAS -- set of aliased variables in a procedure.
- Eliminating the boundary of procedures
  - Procedure inlining and cloning(specialization)
- Can enhance the scope of many optimizations
  - Constant propagation
  - Redundancy elimination
  - Loop optimizations

Procedure Inlining

- Replace a procedure invocation with the body of the procedure being called
  - Advantages:
    - Eliminates procedure call overhead.
    - Allows more optimizations to take place
  - However, overuse can cause slowdowns
    - Breaks compiler procedure assumptions.
    - Function calls add needed register spills.
    - Changing function forces global recompilation.

Procedure Cloning

- Often specific values of function parameters result in better optimizations.

```
PROCEDURE UPDATE(A,N,IS)
  REAL A(N)
  INTEGER I = 1,N
     A(I*IS-IS+1)=A(I*IS-IS+1)+PI
  ENDDO
END
```

If we knew that IS != 0 at a call, then loop can be vectorized.

If we know that IS != 0 at specific call sites, clone a vectorized version of the procedure and use it at those sites.

Hybrid optimizations

- Combinations of procedures can have benefit.
- One example is loop embedding:

```
DO I = 1,N
   CALL FOO()
ENDDO

PROCEDURE FOO()
   …
END
```

⇒

```
CALL FOO()

PROCEDURE FOO()
   DO I = 1,N
      …
   ENDDO
END
```

Constant Propagation

- Propagating constants between procedures can significantly improve performance
- Dependence testing can be made more precise

```
SUBROUTINE FOO(N)
   INTEGER N,M
   CALL INIT(M,N)
   DO I = 1,P
      B(M*I + 1) = 2*B(1)
   ENDDO
END
```

```
SUBROUTINE INIT(M,N)
   M = N
END
```

Enable more accurate dependence analysis if N is a constant

- Challenge:need to model data-flow across procedural boundaries

Constant Propagation

- Definition: Let s = (p,q) be a call site, and let x be a parameter of q. The jump function J_s^x
  - Gives the value of formal parameter x used to invoke q in terms of incoming parameter values of procedure p
  - Models a transfer function for each call site
    - caller parameters ==> callee parameters
- We construct an interprocedural value graph:
  - Add a node to the graph for each jump function J_s^x
  - If x is used to compute to J_t^y, where t is a call site in procedure q, then add an edge between J_s^x and J_t^y for every call site s = (p,q) in some procedure p
  - Model control flow (call relations) between jump functions
- Apply the constant propagation algorithm to this graph.
  - Might want to iterate with global propagation

Jump Functions

```
PROGRAM MAIN
   INTEGER A
α  CALL PROCESS(15,A)
   PRINT A
END
SUBROUTINE PROCESS(N,B)
   INTEGER N,B,I
β  CALL INIT(I,N)
γ  CALL SOLVE(B,I)
END
SUBROUTINE INIT(X,Y)
   INTEGER X,Y
   X = 2*Y
END
SUBROUTINE SOLVE(C,T)
   INTEGER C,T
   C = T*10
END
```

- Need a way of building J_γ^I
- For parameter x of procedure p, define R_p^x to be the output value of x in terms of input parameters of p

$$R_{INIT}^X = \{2 * Y\} \qquad R_{SOLVE}^C = \{T * 10\}$$

$$R_{PROCESS}^B = \begin{cases} R_{SOLVE}^C(J_\gamma^T(N)) & C \in MOD(\gamma) \\ undefined & otherwise \end{cases}$$

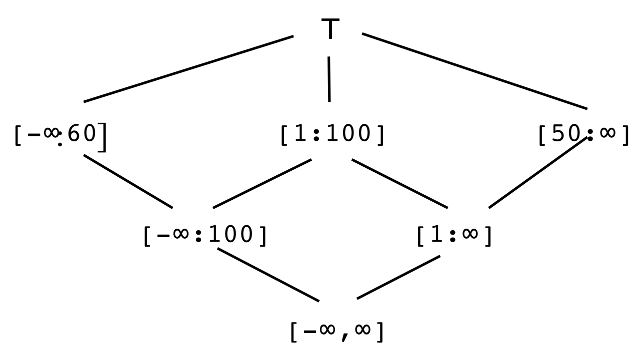
$$J_\gamma^T = \begin{cases} R_{INIT}^X(N) & I \in MOD(\beta) \\ undefined & otherwise \end{cases}$$

$$J_\alpha^N = 15 \qquad J_\beta^Y = N$$

Symbolic Analysis

- Prove facts about values of variables
  - Find a symbolic expression for a variable in terms of other variables.
  - Establish a relationship between pairs of variables at some point in program.
  - Establish a range of values for a variable at a given point.

Range Analysis:



- Jump functions and return jump functions return ranges.
- Meet operation is now more complicated.
- If we can bound number of times upper bound increases and lower bound decreases, the finite-descending-chain property is satisfied.

Array Section Analysis

- Consider the following code:

```
DO I = 1,N
   CALL SOURCE(A,I)
   CALL SINK(A,I)
ENDDO
```

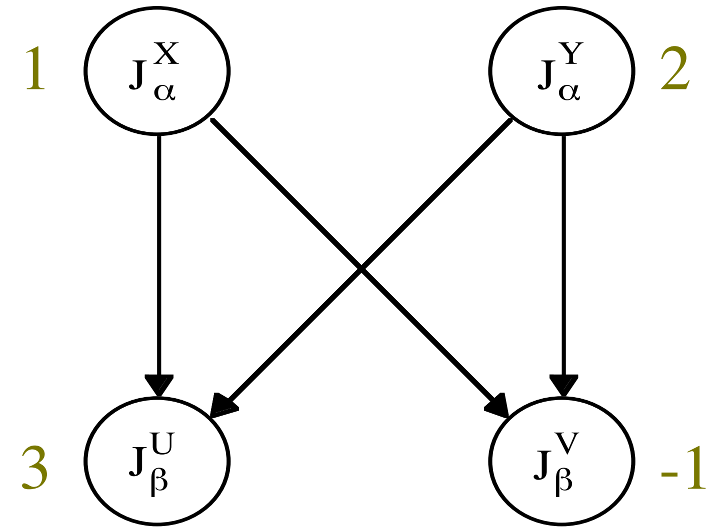
Does this loop carry dependence?

Let $M_A(I)$ be the set of locations in array modified on iteration I and $U_A(I)$ set of locations used on iteration I. Then has a carried true dependence iff

$$M_A(I_1) \cap U_A(I_2) \neq \emptyset \quad 1 \leq I_1 < I_2 \leq N$$

Example

```
PROGRAM MAIN
    INTEGER A,B
    A = 1
    B = 2
α   CALL S(A,B)
END
SUBROUTINE S(X,Y)
    INTEGER X,Y,Z,W
    Z = X + Y
    W = X - Y
β   CALL T(Z,W)
END
SUBROUTINE T(U,V)
    PRINT U,V
END
```



The constant-propagation algorithm will Eventually converge to above values.

Whole Program Optimization

- What we have covered
  - Call graph construction
  - Mod/ref analysis
  - Alias analysis
  - Constant propagation
  - Procedure inlinine and cloning
- Practical concerns
  - Requires the source code of multiple procedures (whole program)
  - Requires recompilation of interdependent procedures when program is modified