The Program Summary Graph and Flow-sensitive Interprocedural Data Flow Analysis

David Callahan

Department of Computer Science
P.O. Box 1892
Rice University
Houston, Texas 77251

1 Introduction

This paper discusses a method for interprocedural data flow analysis which is powerful enough to express flow-sensitive problems but fast enough to apply to very large programs. While such information could be applied toward standard program optimizations, the research described here is directed toward software tools for parallel programming, in which it is crucial.

Many of the recent "supercomputers" can be roughly characterized as shared memory multi-processors. These include top-of-the-line systems from Cray Research and IBM, as well as multi-processor computers developed and successfully marketed by many younger companies. Development of efficient, correct programs on these machines presents new challenges to the designers of compilers, debuggers, and programming environments.

Powerful analysis mechanisms have been developed for understanding the structure of programs. One such mechanism, data dependence analysis, has been evolving for many years. The product of data dependence analysis is a *data dependence graph*, a directed multi-graph that describes the interactions of program components through shared memory. Such a graph has been shown useful for a variety of applications from vectorization and parallelization to compiler management of locality.

Another application of the data dependence graph is as an aid to static debugging of parallel programs. PTOOL [4] is a software system developed at Rice University to help programmers understand parallel programs. It is within this context that we at Rice have learned of the importance of interprocedural data flow analysis. I will briefly describe the PTOOL system and explain the kind of interprocedural information valuable in such an environment.

PTOOL is designed to help locate interactions between parallel threads of control, because such interactions could lead to non-deterministic behavior in the parallel program. The primary means of expressing parallelism that PTOOL understands is the parallel DO loop. A parallel DO loop is essentially the same as a standard Fortran DO loop except that the separate iterations of the loop may execute concurrently. PTOOL attempts to prove that a parallel DO loop is operationally equivalent to executing the same loop as a standard sequential loop. This equivalence holds if there are no interactions between the separate loop iterations at the memory level, in particular, if there is no memory location M such that M is accessed by more than one iteration and M is modified by at least one iteration. Such an interaction is called a *loop-carried data dependence* [20, 18]. One of the contributions of data dependence analysis is the ability to accurately test for and compactly represent the data dependences of a program as a graph.

Parallel DO loops also differ from their sequential counterparts in that variables can be declared to be *private* to each loop iteration. Variables which are private to a particular loop represent different memory locations on each iteration and therefore cannot carry any data dependences. Only variables which hold data that is computed and used within a single loop iteration can be private to that loop iteration. Any variable that is not private to a loop is said to be *shared* with respect to that loop.

PTOOL attempts to establish which variables are shared by determining which variables, under sequential execution, would carry values into the loop, out of the loop, or across loop iterations. This determination is straightforward after definition-use [19] chains have been constructed. Accurate detection of shared variables is critical for PTOOL for two reasons. First, if many variables are incorrectly thought to be shared, then many spurious interactions will *appear* to exist, and controlling false positives is an important concern for any static debugging system. Second, most of the errors PTOOL has uncovered have involved variables which were inadvertently made private when they needed to be shared.

The efficacy of PTOOL (and any dependence-based parallelizing compiler) is limited by the accuracy of the data dependence analysis and the shared variable analysis. Our experience with PTOOL has identified the lack

Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation
Atlanta, Georgia, June 22–24, 1988

of information about the effects of external subroutines as the major source of imprecision. Consider the following program fragment:

```
REAL A(100)
COMMON // Y
DO I = 1,N
    CALL SUB(A,I,X)
        :
    A(I) = A(I) + X + Y
        :
```

If PTOOL were completely ignorant about SUB, then it would have to assume that SUB could use and modify any of the parameters, A,I, or X, and any variable global to the loop, such as Y. Such an assumption can have a catastrophic impact on the accuracy of the data dependence graph.

PTOOL was significantly enhanced by adding machinery to compute interprocedural *flow-insensitive summary* information. This information describes for each external entry, the set of formal parameter variables and global variables that *may* be used (data read from) and the set of formal parameter variables and global variables that *may* be modified by an invocation of that entry. The terms "flow-insensitive" and "may" distinguish the information from "flow-sensitive" and "must" information. Intuitively, "must" facts hold on all execution paths through a subroutine (e.g., "variable X *must* be modified") while "may" facts hold on at least one execution path. A problem is "flow-sensitive" if information about control internal to subroutines is used to compute the final set of data flow facts and " flow-insensitive" if this information is ignored. The term "summary" reflects the use of the information to summarize the behavior of the external routine.

In the previous example, assume flow-insensitive may information indicates that only X is modified and that A,I, and X may be used. We can determine that there are no interactions between invocations of the subroutine generated by variables A, I and Y . Similarly, there are no interactions between the displayed assignment statement on one iteration and the subroutine invocation on another which involve I or Y.

Flow-insensitive may summary information can dramatically reduce the number of data dependences[1]., but unfortunately there is still a significant amount of missing information which is critical to PTOOL. This information falls into two basic categories: array access information and accurate must summary information.

One of the most powerful features of PTOOL is its ability to analyze subscripted variable references and, for large classes of references, prove that no loop carried dependence exists. Knowing that A is used is insufficient to prove that there is no interaction with A(I) in the assignment statement. The subroutine could access just A(I), in which case no loop-carried dependence would exist, or it could access A(I+1), in which case a loop-carried dependence would exist. Callahan and Kennedy [8] describe this problem in more detail and propose a solution method called *regular section analysis*. This problem has also been examined by Triolet [27] and Burke and Cytron [6].

The second problem, which this paper addresses, involves determining whether X must be shared. Since we only know that X may be used and may be modified by SUB, it is possible that either X is used before it is modified or that X is not modified along some execution path so that the old value is used by the occurrence of X in the assignment statement. In either case, it is possible for the variable X to transmit a value into the loop or across loop iterations, forcing PTOOL to assume that X must be shared.

Our experience with PTOOL has indicated that inaccurate shared variable analysis significantly reduces PTOOL's usefulness and that almost all inaccuracies can be attributed to lack of flow-sensitive summary information. Since most of the parallelization-introduced program errors found using PTOOL have involved variables made private incorrectly, accurate shared variable analysis is critical.

In particular, this paper examines the flow-sensitive must-modify problem and the flow-sensitive may-use problem. Flow-sensitive problems are more difficult than flow-insensitive problems for two reasons. The first is that exact solutions are intractable in the presence of aliasing [22] and the second is that the control flow graph of an entire program is likely to be huge. I avoid the first by formulating the basic problems in a context-independent way first described by Lomet [21]. This point will be discussed in greater detail in section 6. The second difficulty is addressed by introducing a new graph, the *program summary graph*, which exploits the hierarchical nature of a program to allow flow-sensitive problems to be solved on a compact structure.

In the following sections, I describe the program summary graph and its construction. Also, I formulate flow-sensitive summary problems and show that they can be solved in time linear in the size of the graph. Next, I examine a reformulation to exploit properties peculiar to global variables and then discuss how the global variable problem interacts with the reference formal parameter problem. Generalizations of the program summary graph to handle constant propagation and the relationship of the program summary graph to previous work are described in section 9.

2 The Program Summary Graph

The program summary graph is an abstraction of a complete program. It summarizes the interprocedural control flow in a way that generalizes the more traditional call graph, but is more compact than the program supergraph described by Myers [22]. The program summary graph has four types of nodes: entry nodes, call nodes, exit nodes, and return nodes. There are entry and exit

[1] At Rice, we have observed up to 90% reductions in the size of data dependence graphs for loops with calls to external routines.

```
PROGRAM MAIN
CALL SUBA(A,B)
PRINT *,A
STOP
END

SUBROUTINE SUBA(X,Y)
EXTERNAL SUBB
X = 4.0
IF (X.EQ.5.0) K = SUBB(X,Y)
K = Y +K +X
RETURN
END

INTEGER FUNCTION SUBB(U,V)
SUBB = U+V
IF (U.GT.V) U = V
RETURN
END
```

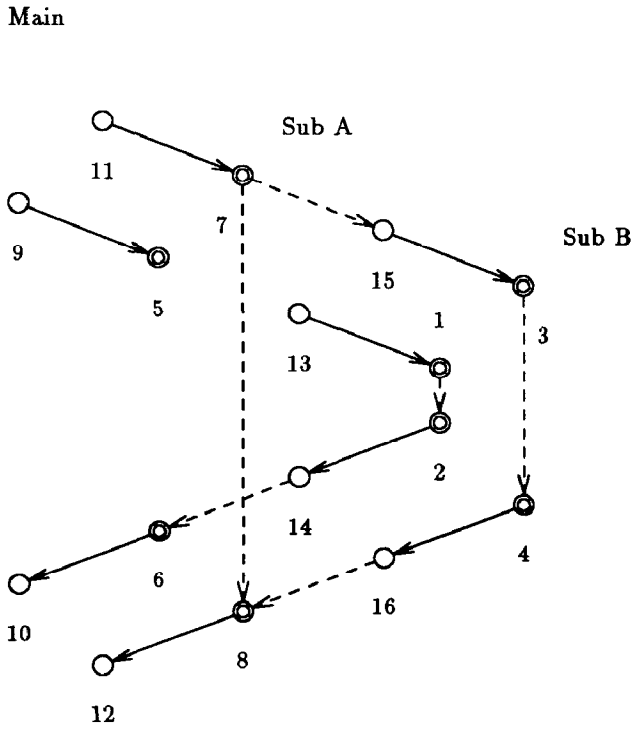

Figure 1: Program Summary Example

nodes for every formal parameter of every routine. There are call and return nodes for every actual parameter of every call site. These nodes represent the four interprocedural events: procedure entry, procedure invocation, procedure exit, and procedure return. For now, assume that global variables are treated explicitly as formal parameters and actual parameters.

There are edges from call nodes to entry nodes that correspond to the binding of formal parameters to actual parameters. There are also edges from exit nodes to return nodes that correspond to this same binding. These edges depend only on the call structure of the program, not the internals of the subroutine.

There are also edges from entry and return nodes to call and exit nodes. These edges summarize the control flow structure of the subroutine and their construction begins with solving the standard data flow problem of reaching definitions: a definition of variable x *reaches* a point p in the subroutine if there is an execution path from the definition to p along which x is not killed. Local to each routine, we can construct reaching information treating each actual parameter at a call site to be a use followed by a definition that kills. Each entry statement defines all parameters and each **RETURN** statement uses all parameters.

An example is shown in Figure 1. Doubled circles represent entry and exit pairs while single circles represent call and return pairs. Each entry is directly above the corresponding exit and each call is directly above the corresponding return. Dashed lines represent intra-procedural edges based on local reaching information while the solid lines represent inter-procedural edges. The following table describes the relationship between node numbers in the figure and the actual and formal parameters in the program:

		A	B			X	Y
Call	SUBA	9	1	Call	SUBB	13	15
Return	SUBA	10	12	Return	SUBB	14	16
		X	Y			U	V
Entry	SUBA	5	7	Entry	SUBB	1	3
Exit	SUBA	6	8	Exit	SUBB	2	4

An edge is added to the program summary graph whenever a definition from an entry reaches a call site. The sink of the edge is the call node associated with the actual parameter. The source of the edge is the entry node associated with the formal parameter. An edge is added to the program summary graph whenever a definition from an entry reaches a **RETURN** statement. The sink of the edge is the exit node associated with the parameter and the source is the entry node associated with the parameter. An edge is added to the program summary graph whenever a definition reaches from a call site to either another call site or to a **RETURN** statement. The source of the edge is the return node associated with the parameter and the sink is either the call node associated with

the actual parameter of the other call site or the exit node associated with the parameter. The local reaches information summarizes "definition-free" execution paths inside each subroutine for a particular variable and it is clear that all "definition-free" execution paths for each variable through the entire program are summarized in the program summary graph.

An important question for later complexity analysis is: "how large does the program summary graph get?". The interprocedural edges in the program summary graph corresponding to the call graph are clearly proportional to program length and the number of variables. Unfortunately, for programs using unrestricted GOTO statements, the number of intra-procedural edges could grow quadractically with program length if control flow is sufficiently complex. With the following definitions:

l	=	program length
c_m	=	maximum number of call sites in any procedure
v_p	=	average number of actual parameters at call sites
v_g	=	total number of global variables

We see that the size of the program summary graph is $O((l + c_m^2)(v_p + v_g))$, worst case.

I have implemented a prototype within the PFC program transformation system [5] which builds a program summary graph for reference formal parameters. The table in Figure 2 shows statistics collected with PFC about a few numerical Fortran programs. The numeric columns give size in lines, number of call sites, total number of reference formal parameters, total number of actual parameters, and total size of program summary graph. While I make no argument that these are necessarily representative, the data supports the assumption that program summary graph size grows linearly with program length. The data also supports the assumption made by Cooper and Kennedy [12] that the average number of actual parameters is a small constant independent of program length. I conclude that we can expect the program summary graph restricted to formal parameters to be proportional to program length, $O(l)$, and the total program summary graph to be proportional to program length times the number of global variables, $O(l \cdot v_g)$.

3 Flow-sensitive KILL and USE

First I examine KILL and USE. I define these sets as, respectively, the variables that *must* be modified and the variables that may be used *before* being overwritten. In more traditional data flow terms, these are, respectively, the set of variables *killed* by the subroutine invocation and the set of variables that are *live on entry* to the subroutine. I will discuss *live on exit* in section 7. I use the name USE for the flow-sensitive problem even though it is already used in the literature for the flow-insensitive problem, because the flow-sensitive solution is the better

Program	Size	Calls	F's	A's	PSG
MFLOPS	526	51	3	35	153
SCALGAM	835	22	76	111	842
BARO	984	8	15	17	131
EULER	1200	32	17	55	254
SIMPLE	1925	58	34	182	882
VA3D4	3738	69	51	122	224
MDSJE23	4619	42	20	38	250
LSODES	5632	134	344	1132	5217
MCNPC	40959	941	245	3246	13474

Figure 2: Size of Program Summary Graphs

for most applications. This is not the case for modifications: you cannot ignore variables in MOD, the set of variables which *may* be modified, that are not in KILL.

These problems are formulated as simple distributive data flow problems [17]. For KILL, the lattice of solution values has two values: "must be modified" ($\top$) and "may not be modified" ($\bot$) with $\bot \sqsubset \top$. A variable must be modified if and only if there is no path from the entry node for that variable to the exit node for that variable. This fact is captured as the greatest fixed point of the data flow equations:

$$Kill(x) = \begin{cases} false & \text{if } x \text{ is an exit node;} \\ \bigwedge_{\langle x,y\rangle \in E} Kill(y) & \text{if } x \text{ is either an entry or a return node;} \\ Kill(y) \vee Kill(z) & \text{if } x \text{ is a call node, } y \text{ is the corresponding return node and } z \text{ is the corresponding entry node.} \end{cases}$$

where $\top$ is encoded as boolean *true* and $\bot$ as boolean *false*, the meet operator, $\wedge$, is "and" and $\vee$ is "or". More precisely, the set $KILL(p)$ of variables killed by invocation of p is equal to the set of all v such that $Kill(e_v^p)$ where e_v^p is the entry node associated with variable v and procedure p. The above equations are solved using a standard iterative algorithm shown in Figure 3. This algorithm propagates facts backward through the graph. The above equations could be reversed to propagate information forward.

To continue the example of Figure 1, exit nodes 2,4,6 and 8 are initialized to *false*, the rest to *true*. In PFC, *work_list* is implemented as a stack. The initial stack is: 8,6,4, and 2. The other nodes found to be "not killed" are, in the order visited: 16,7,14,3,15,1, and 13. This indicates that only parameter X of subroutine SUBA is killed. Note that the only values $Kill(e)$ which are relevant are those where e is an entry node. In particular, the values of nodes 9,10,11, and 12 in Figure 1 do not have meaningful information since the entire body of MAIN is not available.

An optimization is possible when MOD is available. If a parameter is not even modified, there is no need to check

```
/*  initialize all nodes to true   */
/*  except exit nodes              */
work_list ← ∅
foreach  node x do
    if is_exit?(x) then
        Kill(x) ← false
        add x to work list
    else Kill(x) ← true
while  not empty(work_list) do
    take some x off work_list
    if is_exit?(x) or  is_call?(x) then
        foreach ⟨y, x⟩ ∈ E do
            if Kill(y) then
                Kill(y) ← false
                add y to work list
    else if is_entry?(x) then
        foreach ⟨y, x⟩ ∈ E do
            z is the return node
            corresponding to y
            if not Kill(z) then
                Kill(y) ← false
                add y to work list
    else
        /*  x is a return          */
        y is the call node corresponding
        to x
        if Kill(y) then
            z is the entry node such
            that ⟨y, z⟩ ∈ E
            if  not Kill(z) then
                Kill(y) ← false
                Add y to work list
```

Figure 3: Algorithm to compute KILL

for KILL information. This observation is exploited in the equations:

$$\widehat{Kill}(x) = \begin{cases} \text{not}(v \in MOD(p)) & \text{if } x \text{ is an exit node associated with variable } v \text{ in procedure } p; \\ \bigwedge_{\langle x,y\rangle \in E} \widehat{Kill}(y) & \text{if } x \text{ is either an entry or a return node;} \\ \widehat{Kill}(y) \vee ((v \in MOD(p)) \wedge \widehat{Kill}(z)) & \text{if } x \text{ is a call node, } y \text{ is the corresponding return node, } z \text{ the corresponding entry node and } v \text{ is the variable associated with } z \text{ in procedure } p. \end{cases}$$

There is a corresponding change in the initialization shown in Figure 3. Here, $MOD(p)$ represents the set of variables which *may* be modified by an invocation of procedure p. More precisely, the modified equations directly compute $KILL(p) \cup \overline{MOD}(p)$, where $\overline{MOD}(p)$ is the set complement of $MOD(p)$. This set can then be intersected with $MOD(p)$ to get $KILL(p)$.

This change will not affect the complexity of the algorithm, but I have observed a small (15%) improvement in the solution time for the reference formal parameters subproblem. For the example of Figure 1, the initial stack is: 6 and 2. Nodes visited are: 14,1 and 13. This indicates that, of the formal parameters that are modified, only X of SUBA is killed.

A variable is "used" if there is a path from the entry node corresponding to that variable to a use of that variable such that the variable is not overwritten along the path. $USE(p)$ is the set of variables which may be used before being redefined during invocation of procedure p. For this problem, we introduce a special node "use" into the program summary graph to indicate any use in any local context. To the program summary graph described above, edges are added from entry and return nodes to "use" if the corresponding definition in the local context reaches some use. The path problem is now described by the least fixed point of the data flow equations:

$$Use(x) = \begin{cases} true & \text{if } x \text{ is the special "use" node;} \\ \bigvee_{\langle x,y\rangle \in E} Use(y) & \text{if } x \text{ is either an entry or a return node;} \\ Use(z) \vee (\text{not } Kill(x) \wedge Use(y)) & \text{if } x \text{ is a call node, } y \text{ is the corresponding return node and } z \text{ is the corresponding entry node.} \end{cases}$$

and again, more precisely: $USE(p)$ is the set of variables v such that $Use(e_v^p)$ where e_v^p is the entry node associated with variable v and procedure p.

These equations can be solved in time linear in the size of the program summary graph using the algorithm shown in Figure 4. For the example of Figure 1, the edges to "use" (not shown in figure) would be from: 3,1,16,14,7, and 10. These edges are the initial work list, again implemented as a stack. The other nodes found to be used are, in the order visited: 11,13, and 15.

Clearly, execution cost for KILL and USE is proportional to the size of the program summary graph. Figure 5 contains more information derived from the prototype implementation. The second column is the total number of edges in the program summary graph. The remaining columns are the times to solution in milliseconds on an IBM 3081 of the three problems: flow-insensitive MOD for parameters (FIP), flow-insensitive MOD for globals (FIG), and flow-sensitive for parameters (FSP). These times are only the time to solve the data flow problems and do not include time to set up the various graphs and other data structures. The first two problems are solved using the basic iterative algorithm described in Cooper's dissertation [10]. The low absolute solution times also

```
/*  initialize all nodes to true   */
/*  except source of edges to      */
/*  ''use''                        */
work_list ← ∅
Use(*) ← false
foreach  edge ⟨x, use⟩ ∈ E do
    Use(x) ← true
    add x to work list
while not empty(work_list) do
    take some x off work_list
    if is_entry?(x) or  is_call?(x) then
        foreach ⟨y, x⟩ ∈ E do
            if not Use(y) then
                Use(y) ← true
                add y to work list
    else
        /*  x is a return          */
        y is the call node corresponding
        to x
        if not Use(y) then
            z is the entry node such
            that ⟨y, z⟩ ∈ E
            if not Kill(z) then
                Use(y) ← true
                Add y to work list
```

Figure 4: Algorithm to compute USE

Program	Size	PSG	FIP	FIG	FSP
MFLOPS	526	153	1	3	5
SCALGAM	835	842	12	14	69
BARO	984	131	1	6	9
EULER	1200	254	8	53	23
SIMPLE	1925	882	1	30	48
VA3D4	3738	224	2	62	54
MDSJE23	4619	250	1	92	18
LSODES	5632	5217	211	27	445
MCNPC	40959	13474	48	1246	862

Figure 5: Time to solutions

establish that the flow-sensitive problems for parameters are viable for use in a compiler.

4 Handling global variables in KILL and USE

This section formulates KILL and USE for programs with only global variables. In the next section I show how separate approaches for parameters and globals will be combined into a general technique. Such a decomposition of the basic problem allows the special aspects of global variables to be isolated and exploited. The decomposition into global variable and reference formal parameter subproblems was pioneered by Cooper and Kennedy [11].

Global variables are different from reference formal parameters in two regards. The first is that the number of pairs of global variables and entry names or call sites is likely to be very large, much larger than the total number of formal parameters and certainly not a constant independent of program size, so the global variable component of the program summary graph could be very large. The second difference is that, while formal parameters have a different name than the actual they are bound to, every global variable is implicitly bound to itself at each call site.

The size problem can be alleviates by using a bit-vector implementation for globals. This mitigates the problem with the number of variables by working with different variables in parallel and also reduces the size of the program summary graph by maintaining information for only a single pseudo-variable which represents all global variables. Furthermore, the fact that there is no name change at call sites means that no complicated binding map need be maintained.

The above data flow equations hold if the variables represent sets of variables (realized as bit-vectors) rather than single variables but we need "mask" vectors to indicate which definitions do not reach due to local kills. For example, in the following subroutine, along all paths from the entry of `TEMP` to the call site, variable `A` is killed:

```
SUBROUTINE TEMP
COMMON // A,B,C
A = 1
CALL SUB
B = A + C
RETURN
END
```

The local reaching information for all global variables can be summarized by a single edge in the program summary graph annotated with the information that `A` does not reach the call site. The new data flow equations are:

$$Kill(x) = \begin{cases} \vec{false} & \text{if } x \text{ is an exit node;} \\ \bigwedge_{e=\langle x,y\rangle \in E} (Kill(y) \vee Kill(e)) & \text{if } x \text{ is either an entry or a return node;} \\ Kill(y) \vee (Kill(z) \wedge Global(z)) & \text{if } x \text{ is a call node, } y \text{ is the corresponding return node and } z \text{ is the corresponding entry node.} \end{cases}$$

where $Kill(e)$ is the vector of global variables such that there is no path from x to y which is free of definitions of that variable and $Global(z)$ is the vector of variables global to entry z. For the previous example, there is an edge from the entry of `TEMP` to `CALL SUB` and the *Kill* vector for that edge represents the set {`A`}. For the edge

from the return of CALL SUB to the exit of TEMP, the $Kill$ vector would represent the set {B}.

The "optimization" which used the flow-insensitive MOD information can be applied and may be very important since global variables should display significant locality. The "optimized" equations are:

$$Kill(x) = \begin{cases} \text{not } MOD(e) & \text{if } x \text{ is an exit node associated with entry } e \text{ and } MOD(e) \text{ is the set of global variables which may be modified by } e; \\ \bigwedge_{e=(x,y)\in E}(Kill(y) \vee Kill(e)) & \text{if } x \text{ is either an entry or a return node;} \\ Kill(y) \vee (MOD(e) \wedge Kill(z) \wedge Global(z)) & \text{if } x \text{ is a call node, } y \text{ is the corresponding return node and } z \text{ is the corresponding node associated with entry } e. \end{cases}$$

This bit-vector formulation can be solved with any of the general data flow techniques. The general iterative approach [17] gives a time bound of $O(l \cdot d)$ bit-vector operations where l is the program length factor described in section 2 and d is the maximum number of back edges along an acyclic path in a depth first search tree of the call graph. If the call graph happens to be reducible, one of the fast solvers can be applied to get $O(l \cdot \ln l)$ solutions using techniques of Graham and Wegman [14] or $O(l \cdot \alpha(l,n))$ for n nodes in the call graph using the algorithm of Tarjan [26].

5 Combining Parameters and Globals

The problems for formal parameters and global variables interact in two ways. One way occurs when a global variable is passed as an actual parameter and bound to a formal parameter. The second way occurs, in languages that allow nesting of procedure declarations and lexical scoping of names, when a formal parameter of procedure p is accessed as a global variable by a procedure lexically nested inside of p.

In a language without procedure nesting, such as Fortran or C, the second interaction does not occur. The two problems can be integrated by solving for the formal parameter first, then updating the global problem. The only update needed occurs when a global variable is bound to a formal parameter of a procedure to which it is not global. If the procedure kills the parameter, then we can update the $Kill(e)$ set for the incoming edges e to that call site to reflect this fact.

If the variable is global to that procedure, then an alias exists and we insist that both symbolic names be killed independently.[2] If the variable is killed as a parameter, the variable is killed if and only if it is killed as a global as well. In this case, no change need be made. If the variable is not killed as a parameter, we must prevent it from being marked killed. Unfortunately, we have to adjust the data flow equations to retain this fact: $Kill(x)$ is equal to

$$Kill(y) \vee ((Kill(z) \wedge Global(z))) \wedge ParmKilled(x))$$

if x is a call node, y is the corresponding return node, z is the corresponding entry node, and the vector $ParmKilled(x)$ is the set of variables which are either not passed as parameters at call site x or are killed if they are passed as parameters.

Each problem is approximate in the absence of the solution of the other. By first solving the parameter problem, that solution can be used when solving the global variable problem. The resulting approximation to the global variable problem can in turn be used to improve the formal parameter problem. These problems can then be iterated to a solution.

A more straightforward solution is to alter the distinction between global variable and parameter. A variable which is a formal parameter at any level is treated explicitly, following the machinery of section 3. All other variables are handled using the machinery of the previous section. Under these conditions, the second interaction cannot occur (by definition of "global variable"). The assumption that the average number of formal parameters, v_p, is independent of program size indicates that the program summary graph restricted to parameters has $O(l)$ edges. Now we must assume that a procedure at nesting depth d has $O(d)$ "parameters", since all formal parameters of enclosing procedures will be represented explicitly. So the parameter subproblem will have $O(l \cdot d)$ expected execution time, where l is program length and d is the average nesting depth. Again it is plausible to assume that d is independent of program length, and so we again get an $O(l)$ expected execution time for the parameter problem. The execution time for the global problem remains unchanged.

6 The effects of aliasing on USE and KILL.

Two variables are said to be *aliased* if they can both refer to the same or overlapping storage locations. To see that KILL and USE (as formulated here) are imprecise in the presence of aliases, consider the fragment:

[2] See next section.

```
CALL SUB(X,X,Y)
:
SUBROUTINE SUB(A,B,C)
IF (A.LT.B) THEN
    A = C
    A = A+B
ELSE
    B = C
    B = A+B
ENDIF
RETURN
END
```

Note that neither A nor B is individually killed and both appear to be used in SUB, but there is no path along which variable X is unmodified and so X is killed. There is also no path along which X is used before being overwritten.

Lomet [21] showed that you can approximate the exact call-path specific information can be approximated by solving the "no-alias" problem and requiring each symbolic name which is part of the alias-set to be independently killed for KILL, or that USE is implied if any symbolic name in the alias-set is used. Thus, there is a clean up phase, after KILL and USE are computed and alias information is approximated [9], to get safe information for use locally in each procedure.

The effects of aliases do not arise in the formulations of KILL and USE because they are based on symbolic names rather than storage locations: the context of the subroutine is ignored for these problems. By ignoring the context, the complexity is greatly reduced. Our experience indicates that aliasing is very rare in Fortran and so very little precision should be lost.

7 The effects of aliasing on LIVE

I want to look at a closely related problem: live on exit (LIVE). A formal parameter f is in LIVE if there is some path from the exit node associated with f to a use node. For example, parameter Y of SUB2 appears to be LIVE since the value of Y may be used as B in SUB1:

```
SUBROUTINE SUB1(A,B,C)
CALL SUB2(A,B)
A = 4 + C
A = B
RETURN
END
SUBROUTINE SUB2(X,Y)
:
```

Further, parameter X appears to be dead (assuming this is the only call to SUB2) since A is killed immediately after the call to SUB2. The natural formulation of LIVE over the program summary graph used for KILL and USE is not only imprecise but is actually incorrect in the presence of aliasing. If an alias exists between A and B in the previous example, then Y is not live on exit, so LIVE is imprecise. On the other hand, if an alias exists between A and C, then X is live on exit since its value will be used on the alias of C in SUB1, hence LIVE is not even conservative.

The context information that can be ignored for USE and KILL cannot be ignored for LIVE. The functions that summarize the LIVE information of subroutines must depend on the context information.

The problem can still be done if the "errors" due to lack of context information can be corrected. One simple approach would be to propagate information through the program summary graph to mark every formal parameter that may be bound to a parameter that is involved in an alias. Any such variable is automatically marked LIVE, and LIVE is corrected for variables not part of an alias set. This is essentially the same solution we employ for constant propagation [7]: make the context-dependent functions context-independent by detecting dangerous aliases and then replace the context-dependent functions with "bottom" wherever appropriate. Again, if aliasing is rare, no significant loss of precision should occur.

8 Side Costs

Flow-insensitive formulations of data flow problems over the call-graph require very little information about each routine compared to the information required for construction of the program summary graph. If the initial information had to be recomputed for every compilation, the cost might become prohibitive. Of course, this is not the case. Only recently modified routines need to be re-analyzed. All of the Rice systems (PFC,PTOOL and $\mathbb{R}^n$) include database mechanisms for preserving initial information across compilations. The most sophisticated is employed in the $\mathbb{R}^n$ programming environment [13]. That system presumes an intelligent editor to provide initial information for the interprocedural analysis. The clean separation of intra-procedural information from inter-procedural information described for the program summary graph makes it straightforward to have the editor maintain the more sophisticated information. The cost of collecting initial information for the program summary graph for a particular routine is incurred at most once per editing session of that routine.

A side benefit of not using alias information in the formulation of the summary information is that no dependence is placed between these two problems. In particular, changes to the alias information do not necessarily require recomputation of summary information. Since call graphs will frequently be very tree-like, and summary problems as formulated are pure "backward" flow problems (with respect to invocation order), very fast (but not truly incremental) updates are possible by exploiting the structure of the call graph.

Finally, the local reaches information can be updated very quickly from the interprocedural information since there are no structural changes to the local flow graph [25, 23].

9 Related Work

Allen [2] solves the reaching definitions problem for programs with acyclic call graphs. This is done by visiting each procedure in "reverse invocation order" so that whenever a subroutine call is encountered, the set of variables which are upwards exposed to the entry are known as well as the must-be-modified sets. These sets are used to solve the local reaching definitions problem in the caller. Allen and Schwartz [3] extend Allen's work by "overestimating" reaching information and then refining through iteration. Here, this work has been extended with complexity analysis and re-engineering to make the time to solution acceptable for a compiler.

Lomet [21] formulates the must-not-be-modified problem, PRESERVED, in terms of local reaching definitions but uses very conservative approximations by making worst case assumptions. Lomet provides justification for solving these flow-sensitive problems under "no-alias" assumption by showing that PRESERVED can be adjusted to be correct but conservative once alias information is known. Previous work did not discuss the effects of aliasing. I have adopted the "no-alias" approach but improved on the basic reaching definitions information.

Rosen [24] formulates flow-sensitive versions of MOD, USE, and PRESERVED over the entire program control flow graph where the sets are indexed not only by variable names but also call-path specific aliasing patterns.

Myers [22] formulates the interprocedural must-summary problem over the *program supergraph* which explicitly represents all basic blocks and control flow in the program. He establishes that must-summary is co-NP in the presence of aliasing but argues that in practice, aliasing will have limited effects and provides a precise algorithm for solving the LIVE, AVAIL and must-summary problems.

The approach to interprocedural constant propagation presented by Cooper, Callahan, Kennedy, and Torczon [7] provides the seeds of the ideas leading to the program summary graph. Callahan and Kennedy [8] have shown how to extend the lattice used for constant propagation in a way which also computes KILL. The general notion of *jump functions*, which map values available upon subroutine entry to values available at call sites, and *return jump functions*, which map to values at exit points, is a direct generalization of the program summary graph: each node represents a jump function to be evaluated and the edges between nodes define the *support* of that function. This general graph defines a set of mutually recursive functions. By restricting the classes of functions to bit AND or OR, as in the program summary graph, solutions (functional fixed points) can be computed very quickly. Here I have described applications for KILL and USE but we could also, with slight modifications, compute an approximation to the set of available constants. By allowing more complex functions, the behavior and effects of subroutines can be approximated to any degree, but closed form solutions may not exist.

As part of the PTRAN project [1], flow-sensitive MOD and USE are computed for non-recursive Fortran programs using techniques essentially the same as described by Allen [2]. They do not compute *KILL* information but do have some alias information.

Hudak [16] gathers compile-time facts about a functional program via techniques called *abstract interpretation* and *collecting interpretation of expressions*. He uses program text to define functions over data domains which approximate in some way real data domains. We can contrast that with the discussion of the previous paragraph where the program summary graph is viewed as using approximate *functions* as well as approximate data domains.

Cooper and Kennedy [12] fully develop the flow-insensitive interprocedural summary problem and detail the interactions between the problems for reference formal parameters and global variables. Here I have extended their machinery to the flow-sensitive problems KILL and USE.

Horowitz, Reps and Binkley [15] define a *linkage grammer* which is structurally very similar to the program summary graph. Their goal is determine all input parameters that might affect the value of an output parameter. Rather than using reaches information, they compute intra-procedural edges based on "slice" information: an edge exists from an entry or return node (using my terminology) to a call or exit node if the source of the edge affects the value at the sink. Determining if a particular input parameter affects the value of an output parameter can now be done by showing a path exists from the input parameter to the output parameter.

10 Summary

The program summary graph is an effective data structure for solving the flow-sensitive summary problems KILL and USE. By summarizing local control flow with reaching information and exploiting bit-vector techniques for globals, the size of the flow-sensitive problem is managed. It may be less effective for the context-dependent problems, such as LIVE, but many of these problems can be made context independent by pre-computing dangerous contexts (aliases) and "correcting" the solutions. Finally, the structures of the $\mathbb{R}^n$ programming environment and the program summary graph substantially mitigate the side costs of collecting initial information and maintaining correct local information.

References

[1] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the PTRAN analysis system for multiprocessing. In *Proceedings of the First International Conference on Supercomputing*, Springer-Verlag, Athens, Greece, 1987.

[2] F. E. Allen. Interprocedural data flow analysis. In *Proceedings IFIP Congress 74*, North-Holland Publishing Co., Amsterdam, 1974.

[3] F. E. Allen and J. T. Schwartz. *Determining the data relationships in a collection of procedures.* Research Report RC 4989, IBM T. J. Watson Research Center, August 1974.

[4] J. R. Allen, D. Baumgartner, K. Kennedy, and A. Porterfield. PTOOL: a semi-automatic parallel programming assistant. In *Proceedings of the 1986 International Conference on Parallel Processing*, IEEE Computer Society Press, August 1986.

[5] J. R. Allen and K. Kennedy. *PFC: a program to convert Fortran to parallel form.* Technical Report MASC-TR 82-6, Dept. of Mathematical Sciences, Rice University, March 1982.

[6] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 162–175, June 1986.

[7] D. Callahan, K. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, June 1986.

[8] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. In *Proceedings of the First International Conference on Supercomputing*, Springer-Verlag, Athens, Greece, 1987. Available as Rice University, Department of Computer Science Technical Report TR87-56, July 1987.

[9] K. Cooper. Analyzing aliases of reference formal parameters. In *Conference Record of the Twelfth ACM Symposium on the Principles of Programming Languages*, January 1985.

[10] K. Cooper. *Interprocedural Data Flow Analysis in a Programming Environment.* PhD thesis, Dept. of Computer Science, Rice University, April 1983.

[11] K. Cooper and K. Kennedy. Efficient computation of flow insensitive interprocedural summary information. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices Vol. 19, No. 6*, July 1985.

[12] K. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of the ACM SIGPLAN 88 Conference on Program Language Design and Implementation*, Atlanta, GA, June 1988.

[13] K. Cooper, K. Kennedy, and L. Torczan. The impact of interprocedural analysis and optimization in the $\mathbb{R}^n$ programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):419–523, October 1986.

[14] S. Graham and M. Wegman. A fast and usually linear algorithm for global data flow analysis. *Journal of the ACM*, January 1976.

[15] S. Horowitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 88 Conference on Program Language Design and Implementation*, Atlanta, GA, June 1988.

[16] P. Hudak. A semantic model of reference counting and its abstraction. In *Conference Record of the 1986 Symposium on Lisp and Functional Programming*, pages 351–363, 1986.

[17] J. B. Kam and J. D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):159–171, January 1976.

[18] K. Kennedy. *Automatic translation of Fortran programs to vector form.* Technical Report 476-029-4, Dept. of Mathematical Sciences, Rice University, October 1980.

[19] K. Kennedy. A survey of data flow analysis techniques. In S. S. Muchnick and M. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 1–54, Prentice-Hall,New Jersey, 1981.

[20] D. J. Kuck. *The Structure of Computers and Computation.* Volume 1, John Wiley & Sons, New York, 1978.

[21] D. Lomet. Data flow analysis in the presence of procedure calls. *IBM Journal of Research and Development*, 21(6):559–571, November 1977.

[22] E. Myers. A precise interprocedural data flow algorithm. In *Conference Record of the Eigth ACM Symposium on the Principles of Programming Languages*, pages 219–230, January 1981.

[23] L. L. Pollock and M. L. Soffa. *An incremental version of iterative data flow analysis.* Rice COMP TR87-58, Dept. of Computer Science, Rice University, August 1987.

[24] B. K. Rosen. Data flow analysis for procedural languages. *Journal of the ACM*, 26(2):322–344, April 1979.

[25] B. G. Ryder and M. D. Carroll. An incremental algorithm for software analysis. In *Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pages 171–179, 1986.

[26] R. E. Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, July 1981.

[27] R. Triolet, F. Irigion, and P. Feautrier. Direct parallelization of call statements. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 176–185, June 1986.