Interprocedural Analysis Useless for Code Optimization

S. Richardson, M. Ganapathi

Technical Report No. CSL-TR-87-342

November 1987

This technical report was supported by Defense Advanced Research Projects Agency under Contract No. MDA903-83-C-0335.

Interprocedural Analysis Useless for Code Optimization

S. Richardson, M. Ganapathi

Technical Report No. CSL-TR-87-342

November 1987

Computer Systems Laboratory
Department of Electrical Engineering
Stanford University
Stanford, CA 94305

Abstract

The problem of tracking data flow across procedure boundaries has a long history of theoretical study by people who believed that such information would be useful for code optimization. Building upon previous work, we have implemented an algorithm for interprocedural data flow analysis. The algorithm produces three flow-insensitive summary sets: MOD, USE, and ALIASES. The utility of the resulting information was investigated using an optimizing Pascal compiler. Over a sampling of 27 benchmarks, we found that additional optimizations performed as a result of interprocedural summary information contributed almost nothing to program execution speed.

Key Words and Phrases: compiler, optimization, data flow analysis, interprocedural analysis, aliasing.

Table of Contents

1. Introduction

The presence of procedure calls forces most compilers to make conservative assumptions for code optimization. Usually, calls within blocks are treated as complex instructions that may affect the values of many variables. Global code optimizers perform data-flow analysis by handling procedure calls conservatively, since their effect is unknown. Consider the following example:

```
a := b + c
call proc(d)
x := b + c
y := a + e
```

If proc(d) kills the definition of a, then a := b + c does not reach y := a + e. In the absence of information about proc, it must be assumed conservatively that a procedure call is the definition of every variable visible at the call-site and all variable parameters.

Careful analysis of procedures can yield additional optimization opportunities that would otherwise have been wasted. Consider the following program skeleton.

```
program Ipatest(input, output);
var
    gv: integer;
    ba: array [1..1000] of integer;
    i: integer;

    procedure Sub;
    begin
        (* body of subroutine *)
    end;

begin
    gv := 2;
    for i := 1 to 1000 do begin
        Sub;                    (* Does call modify gv?         *)
        ba[gv*i] := 0;          (* Clear ba if gv not set.      *)
    end;
end.
```

If interprocedural analysis (ipa) can determine that the call to sub does not alter gv, the address calculation inside the loop can be significantly optimized. Such an interprocedural optimization can make the program run three times faster than when optimized without interprocedural analysis. Similarly, the presence of a call within a loop can prevent invariant code from being hoisted out of the loop.

Another example:

```
program test(input, output);
var
    i, j, k: integer;
    a: array [0..2000] of integer;

        procedure check(var x: integer);
        begin
            j := 2;
            for i := 0 to 1000 do begin
                x := 3;         (* Does assignment alter j? *)
                a[i*j] := 0;
            end;
        end;

begin
    check(k);
    (* body of program *)
end.
```

Inter-procedural analysis is necessary to determine that the parameter x is not aliased to i or j inside the subroutine "check". Once this has been determined, the constant expression "x := 3" can be hoisted out of the loop and the index expression [i*j] can be optimized. The optimized program becomes:

```
program test(input, output);
var
    i, j, k: integer;
    a: array [0..2000] of integer;

        procedure check(var x: integer);
        begin
            j := 2;
            x := 3;
            for tmp := 0 to 2000 step 2 do begin
                a[tmp] := 0;
            end;
            i := 1000;
        end;

begin
    check(k);
    (* body of program *)
end.
```

A comprehensive treatment of interprocedural techniques including all facets of programming-language constructs and consequent quantitative analysis is still the subject of on-going research. Spillman analyzed the modification of variables considering almost all PL1 features, including recursion, reference parameters, procedure variables, label variables, pointers and exception handling [Spillman 71]. Allen used a very simple programming model, precluding procedure parameters and disallowing recursion but considering computation of modified-variables (MOD), used-variables (USE) and preserved-variables (PRESERVED) information [Allen 74]. Rosen considered a complicated but precise method to compute composite summary information [Rosen 76], yielding sharper information than those of earlier or later techniques. Lomet presented a simpler but less precise analogue of Rosen's method [Lomet 77]. Barth extended Spillman's technique, requiring multiple passes over the program, and computing algebraic relations across procedures and variables [Barth 77]. Banning used the call graph as a control flow graph; thus, standard global-data-flow analysis techniques were used in a one-pass algorithm to compute flow-insensitive summary information [Banning 79]. Weihl computed ranges of potential values for procedure variables, label variables and pointers independent of the program's call graph or control-flow-graph [Weihl 80] and used Barth's method to generate summary information. Myers showed that in the presence of aliasing, flow-sensitive summary problems are either IV-complete or co-NP complete [Myers 81]. Sharir and Pnueli treated the entire program as a single data-flow graph and solved problems on it [Sharir & Pnueli 81]. Jones and Muchnik considered a flexible approach to interprocedural flow analysis and programs with recursive data structures [Jones & Muchnik 82]. Cooper and Kennedy reformulated the algorithms of Banning and Myers resulting in a fast algorithm for interprocedural data flow analysis [Cooper & Kennedy 84]. They have concentrated on isolating and improving parts that arc non-linear. Specifically, they isolated the tracking and closure effect of formal-parameter bindings, thus creating a more practical basis for the application of elimination techniques to Banning's algorithm. Problems with this reformulation and corrections to fur these problems have been noted in the literature [Burke 84] [Carroll 87] [Cooper & Kennedy 87a]. Similarly, Burke and Cytron [Burke & Cytron 86] engineer the algorithm one step further to partition global-modification computation into local modification information and interprocedural binding patterns.

Research continues to focus on improving the efficiency of interprocedural techniques and analyzing the complexity of interprocedural side effect analysis [Cooper & Kennedy 87b] [Cooper & Kennedy 87c]. As we shall see later, our observation has been that efficiency, as defined by these techniques, is not the major problem. Rather, the potential benefit of the information must be questioned. In this paper we concentrate on the latter part of the problem. We take an existing algorithm and evaluate its utility in an optimizing compiler. The actual cost of implementation is contrasted with the potential gain obtained through program optimization. Thus, precision of interprocedural summary information and consequent improvement to the code can be determined.

The algorithm used in our own experiments computes flow-insensitive summary information similar to that computed by Banning panning 79] and Cooper and Kennedy [Cooper & Kennedy 84]. This information is

incorporated in an optimizing Pascal compiler [Chow 83] to attempt improvements in the execution efficiency of 27 benchmarks. These results are summarized in the concluding section.

2. Interprocedural Summary Information

The interprocedural analyzer scans through a source program just once and creates the program's call-graph along with three flow-insensitive summary sets:

- MOD(u), the set of variables global to procedure u that may be modified by a call to u.
- USE(u), the set of variables global to procedure u that may be used during a call to u.
- ALIASES(x), the set of variables which may be aliased to the variable x.

Banning used standard flow analysis techniques on the call-graph. Based on his work, we have derived the following equation for collecting inter-procedural MOD information:

$$MOD(u) = MOD_0(u) \cup [\cup_{e=(u,v)} map(MOD(v), PTRS(e))],$$

where

$MOD_0(u)$ is the set of non-local variables modified by procedure u, without considering variables modified by procedure calls within u.

PTRS(e) is a list of ordered pairs representing the var parameters and their bindings at each call site e. If procedure u makes a call to procedure v and passes the actual variables x, y, and z to v as formal parameters fp1, fp2, and fp3 at call site e_1, then

$$PTRS(e_1) = \{ (fp1, x), (fp2, y), (fp3, z) \}.$$

map(MOD(v), PTRS(e)) searches MOD(v) for any elements which are also forma1 reference parameters of procedure v. It then replaces these elements with the corresponding actual parameter as given by PTRS(e) for the call site e. As an example, if

$$MOD(v) = \{ a, b, c, fp1 \} \text{ and } PTRS(e) = \{ (fp1, x), (fp2, y), (fp3, z) \},$$

then

$$map(MOD(v), PTRS(e)) = \{ (a, b, c, x) \}$$

We implement this equation using the following algorithm:

```
(* Initialize *)

foreach procedure u do begin
    MOD(u) := MOD0(u);
    CALLS(u) := CALLS0(u);
end;
newsets := true;

(* Calculate MOD set *)

while (newsets) do begin
    newsets := false;
    foreach procedure u do begin
        foreach callsite e=(u, v) ∈ CALLS(u) do begin
            MOD'(v) = map (MOD(v), PTRS(e));
            if (MOD(u) ∩ MOD'(v) ≠ ∅) then begin
                MOD(u) := MOD(u) ∪ MOD'(v);
                CALLS(u) := CALLS(u) ∪ CALLS(v);
                newsets := true;
            end;
        end;
    end;
end;
```

The algorithm for collecting USE information is strictly analogous to the algorithm for collecting MOD information.

In program optimization, it is helpful to know variables that may potentially be aliased to other variables. Thus,

for each variable x in a program, we calculate ALIASES(x), the set of possible aliases for variable x. ALIASES(x) is formed by taking the transitive closure of a simpler set $\mathbf{ALIASES_0(x)}$, the set of actual parameters that are bound to formal parameter x:

```
(* Initialize *)

foreach procedure u do begin
    foreach varparameter x_u do begin
        ALIASES(x_u) := ALIASES_0(x_u);
    end;
end;

(* Calculate ALIAS set *)

change := true;
while (change) do begin
    change := false;
    foreach procedure u do begin
        foreach varparameter x_u do begin
            foreach actualparameter ap=ALIASES(x_u) do begin
                if ALIASES(x_u) n ALIASES(ap) ≠ Ø then begin
                    change := true;
                    ALIASES(x_u) := ALIASES(x_u) U ALIASES(ap);
                end; (* if *)
            end (* foreach *)
        end (* foreach *)
    end (* foreach *)
end (* while *)
```

3. Influencing Global Optimization

An existing global optimizer (UOPT) [Chow 83] was used to explore potential benefits from the computed interprocedural summary information. UOPT has been used in the Stanford U-Code compiler system since 1983 and has proven effective over a wide range of programs. It operates as an optional extra pass performed on the intermediate code output from the compiler front end, and its output in turn is used by the compiler back-end to generate machine code for a given target machine. To improve execution efficiency, the following global optimizations are performed:

1. Stack height reduction in expression evaluation.
2. Constant propagation.
3. Constant expression evaluation at compile time.
4. Address collapsing in array expressions.
5. Dead code elimination.
6. Copy propagation.
7. Common subexpression elimination.
8. Loop-invariant expression optimization.
9. Partial redundancy suppression by backward code motion.
10. Loop induction-expression optimization (strength reduction).
11. Linear function test replacement and induction variable elimination.
12. Redundant store elimination.
13. Dead variable elimination.
14. Partial redundancy suppression by forward code motion.
15. Global register allocation and assignment.

To a large extent, these optimizations depend upon the use of data flow information; the accuracy of which can theoretically be improved with the use of interprocedural summary information.

The optimizer makes the following assumptions concerning data flow:

1. Direct access of a memory location affects only the memory location accessed
2. Indirect access through a pointer to a known memory location affects only the known location.
3. Indirect access through a pointer to an unknown memory location affects all non-local variables.
4. Access of a var parameter affects access to all non-local variables. For example, a store to a var parameter kills all globals and all var parameters.
5. Access of a var parameter is affected by access to all non-local variables. For example, a store to a global kills all var parameters.
6. A procedure call affects all variables in the scope of the called procedure.

Assumptions (1) and (2) are precise and therefore cannot be improved.

Assumption (3) is a worst-case assumption that can be improved by tracking pointer dereferences. The algorithm presented here does not address the problem of tracking pointers, although in the future, pointers can be handled in a way similar to ALIASES computation. for reference parameters. The interprocedural analyzer can collect all the address ranges throughout the program whose addresses have been taken via a load-address instruction and stored somewhere or passed as a parameter. When UOPT sees an indirect store, it need kill only these addresses instead of

everything.

Assumptions (4) and (5) are worst-case assumptions that improve with the addition of the ALIASES summary set. Access of var parameter v_p affects only the aliases of v_p in the set $ALIASES(v_p)$.

Most importantly, the use of MOD and USE sets allow us to remove the worst-case restriction in assumption (6). Rather than assuming that a procedure call affects all variables in unknown ways, we can precisely convey that a call to procedure P modifies only those variables in the set MOD(P) and uses only those variables in USE(P).

3.1. Optimization with MOD Information

MOD sets allow the global optimizer to determine whether or not a given variable is killed (modified) by a given procedure call. This information may be used in turn to calculate flow information and liveness information for various types of code optimization, including:

- constant folding/constant propagation
- copy propagation
- strength reduction
- induction variable elimination
- common subexpression elimination
- redundant store elimination
- code motion

As an example, consider the following code from "pwhets", a Pascal version of the Whetstone benchmark (the various benchmarks used in this paper are described in Appendix I):

```
for j = 1 to n do
begin
    n1 = 0;
    n2 = L2 * i;
    n3 = L4 * i;
    Sub(n1, n2, n3);
end;
```

a. UNOPTIMIZED

```
for j = 1 to n do
begin
    n1 = 0;
    n2 = L2 * i;
    n3 = L4 * i;
    Sub(n1, n2, n3);
end;
```

b. **UOPT, no IPA (no change)**

```
n1 = 0;
n2 = L2 * i;
n3 = L4 * i;
for j= 1 to n do
begin
    Sub(n1, n2, n3);
end;
```

c. UOPT + IPA

Figure 3-l: Effect of MOD Sets on Optimization

The variables i, nl, n2, and n3 are global to procedure Sub. Thus, without the help of MOD sets, the global optimizer can do nothing to improve the running speed of the loop (Figure 3-lb). Because the variables are not in fact modified by the call to subroutine "Sub" $(MOD(Sub) = \emptyset)$, the use of MOD information allows us to move all three assignments out of the loop (Figure 3-lc).

Another bit of pwhets code is shown in Figure 3-2.

```
t = 0.499975;
t2 = 0.500025;
Sub();
e1 = expr1 * t;
e4 = expr4 / t2;
```

a. **UOPT, no IPA**

```
Sub();
e1 = expr1 * 0.499975;
e4 = expr4 / 0.500025;
```

b. UOPT + IPA

Figure 3-2: Effect of MOD Sets on Optimization

The global optimizer was unable to do anything to the original code because of uncertainty as to the fates of global variables t and t2 during the call to Sub. After using interprocedural MOD information to determine that procedure Sub does not modify either of the global variables t or t2 (MOD(Sub) = Ø), the code was optimized as shown in Figure 3-2b.

Finally, consider the program given below as Figure 3-3a. Because variables gv and i are global to subroutine "Subl," the global optimizer UOPT was initially unable to optimize the loop in any way. Once the MOD sets were used to determine that neither gv nor i are modified by "Subl" (MOD(Subl) = Ø), the optimizer was able to eliminate the variable "gv" and the repetitive array index calculation "ba[gv*i]", as shown in Figure 3-3b.

```
gv = 2;
for i = 1 to 1000 do
begin
    Subl();
    ba[gv*i] = 13;
end;
```

a. UOPT, no IPA

```
for tmp = &ba[2] to &ba[2000] step 2 do
begin
     Subl();
     *tmp = 13;
end;
```

b. UOPT + IPA

Figure 3-3: Effect of MOD Sets on Optimization

3.2. Optimization with USE Information

The interprocedural summary set USE may be used by UOPT to determine whether or not a given variable is *used* by a given procedure call. As seen previously with MOD information, USE information can be used to calculate flow information and liveness for use with various types of code optimization.

The example given below illustrates how USE information can be used to perform register allocation in the presence of procedure calls.

The cyk benchmark contains the following code before optimization (Figure 3-4).

```
for i := 1 to n do begin
    for j := 1 to n do begin
        for k := 1 to n do begin
            P2[i, j, k] := true;
        end;
    end;
end;
for i := 1 to L do x[i] := 1;
cyksub(x, P2);
```

Figure 3-4: Unoptimized cyk Code

The optimizer allocates global variables i, j, and k to the registers R_0, R_1, and R_2 respectively. Without interprocedural analysis to tell it about use patterns within the procedure cyksub, it produces the code in Figure 3-5.

```
for R0 := 1 to n do begin
    for R1 := 1 to n do begin
        for R2 := 1 to n do begin
            P2[R0, R1, R2] := true;
        end;
        k := R2;                          (**)
    end;
    j := R1;                              (**)
end;
for R0 := 1 to L do x[R0] := 1;
i := R0;                                  (**)
cyksub(x, P2);
```

Figure 3-5: cyk Code Optimized without the Use of Interprocedural Information

By referring to the USE sets, the optimizer can determine that i, j, and k are not used inside procedure cyksub (USE(cyksub) = Ø), obviating the need for the three updates marked by "(**)". The resulting code is as shown in Figure 3-6 below.

```
for R0 := 1 to n do begin
    for R1 := 1 to n do begin
        for R2 := 1 to n do begin
            P2[R0, R1, R2] := true;
        end;
    end;
end;
for R0 := 1 to L do x[R0] := 1;
cyksub(x, P2);
```

Figure 3-6: cyk Code Optimized Using Interprocedural USE Sets

The savings to the algorithm from this optimization is easily calculated. The expression inside the loop is calculated exactly n^3 times. The stores occur once for each of the outer loops, for a total of n^2+1 stores. For large n, the savings translates to a speedup of

$$n^3 / (n^3 - n^2 - n)$$

USE information can also be used to perform linear test replacement, as shown by Figure 3-7. Two other procedures are within the scope of

```
lo = 1;
hi = MAXCELL;
for c = lo to hi do begin
    cell[c] = nil;
end;
```

a. UOPT, no IPA

```
lo = &cell[1];
hi = &cell[MAXCELL];
for tmp = lo to hi do begin
    *tmp = nil;
end;
```

b. UOPT + IPA

Figure 3-7: The Effect of USE on Code Optimization

variable c. Without interprocedural analysis, the global optimizer UOPT was unable to optimize the given code. With the addition of USE sets, it became known that neither of the other two procedures used variable c, and the result was code optimized as shown by Figure 3-7b.

Figure 3-8a shows more unoptimized code from the pwhets benchmark.

```
diff = 17;
for i = 1 to 1000
do begin
    diff = P(x, i) / i;
    this = diff;
end;
```

a. UNOPTIMIZED

```
diff = 17;
R0 = diff;
for i = 1 to 1000
do begin
    diff = R0;
    R0 = P(x, i) / i;
    this = R0;
end;
```

b. UOPT, no IPA

```
R0 = 17;
    for i = 1 to 1000
    do begin
    R0 = P (x, i) / i;
    this = R0;
end;
```

c. UOPT + IPA

Figure 3-8: The Effect of USE Information on Program Optimizaton

When optimized without the use of interprocedural summary information, the global optimizer allocates the variable "diff" to register R_0, as shown in Figure 3-8b. When the global optimizer is allowed to use interprocedural summary information, it finds that "diff" does not appear in the set USE(P), i.e. "diff" is not *used* by procedure P. Thus, the optimizer was able to remove the update of diff at the beginning of the loop, producing the code shown in Figure 3-8c.

3.3. Optimization with ALIAS Information

ALIAS sets are used to determine whether a given variable has potential aliases, such that unknown side effects may occur as a result of storing to the variable in question.

Note the following example of how ALIAS sets were used to improve register allocation. In the benchmark "bigfm" we find the code shown in Figure 3-9.

```
procedure Onestep(var bestgain: pinrange);

    procedure Findbest(...);

        procedure initchain(cell: cellrange; gain: pinrange);
            begin
                new(bestchain);
                bestchain^.next = nil;
                bestchain^.number = cell;
                lastlink = bestchain;
                bestgain = gain;
                bestlink = bestchain;
            end;
        end;

    begin . . . end;
begin . . . end;
```

Figure 3-9: Unoptimized Source Code from the "bigfm" Benchmark

Uncertainty as to whether var parameter "bestgain" and global variable "bestchain" are aliased initially prevented the optimizer from allocating "bestchain" to a register. Once ALIAS information was referenced, the optimizer yielded the code shown in Figure 3-10.

```
procedure Onestep(var bestgain: pinrange);

    procedure Findbest(...);

        procedure initchain(cell: cellrange; gain: pinrange);
            begin
                new(R0);
                R0^.next = nil;
                R0^.number = cell;
                lastlink = R0;
                bestgain = gain;
                bestlink = R0;
            end;
        end;

    begin . . . end;
begin . . . end;
```

Figure 3-10: Optimized "bigfm" Code, using Interprocedural Information

UOPT was unable to optimize the code given in Figure 3-11 without the help of interprocedural ALIAS sets. However, once the ALIAS sets were calculated and utilized, optimized code was produced as shown in Figure 3-12. The alias set for x was found to be: ALIASES(x) = {j}.

Because x is not aliased to i or a, the computation x=3 was moved out of the loop.

Because x is not aliased to i, i was replaced by a register.

Because x is aliased to j through at least one path, no assumptions were made as to the value of j inside the loop.

ALIAS information allowed the global optimizer to perform linear test replacement on the program shown in Figure 3-13. Optimization without interprocedural analysis is unable to do anything to the program because of

```
program test(input, output);
var
    i, j, k: integer;
    a: array [0..2000] of integer;
        procedure init(var x: integer);
        j = 2;
        begin
            for i = 0 to 1000 do begin
                x = 3;
                a[i*j] = 0;
            end;
        end;
    begin
        init(j);
    end.
```

Figure 3-11: Benchmark,Unoptimizable without Interprocedural Information

```
program test(input, output);
var
    i, j, k: integer;
    a: array [0..2000] of integer;
        procedure init(var x: integer);
        j = 2;
        x = 3;
        begin
            for R0 = 0 to 1000 do begin
                a[R0*j] = 0;
            end;
        end;
    begin
        init(j);
    end.
```

Figure 3-12: Optimized Benchmark, Using ALIAS Sets

```
program test(input, output);
var
    i, j, k: integer;
    a: array [0..2000] of integer;
        procedure init(var x: integer);
        j = 2;
        begin
            for i := 0 to 1000 do begin
                a[i*j] = 0;
            end;
        end;
    begin
        init(k);
    end.
```

Figure 3-13: Benchmark, Unoptimizable without Interprocedural Information

possible aliasing patterns between the var parameter x and the global variables i, j, k, and the global array a. Optimization with the benefit of inter-procedural analysis thrown in yields the code shown in Figure 3-14. Because j is not aliased to var parameter x (ALIASES(x) = {k}), it was recognized as a loop-invariant constant, and the array reference inside the loop was linearized.

As a final example of where ALIAS information can help optimize programs, Figure 3-15 presents procedure "P3" from the pwhets benchmark. It is obvious that the local variables x, y, and tmp may be allocated to registers. Interprocedural analysis, specifically the use of ALIAS sets, is required to show that the var parameter z is not aliased to the non-local t before t can be allocated to a register. Using interprocedural analysis, the optimizer was able to produce the version of P3 shown in Figure 3-16.

```
program test(input, output);
var
    i, j, k: integer;
    a: array [0..2000] of integer;
        procedure init(var x: integer);
        begin
            R0 := &a[0];
            for i := 0 to 1000 do begin
                *R0 := 0;
                R0 := R0 + 2 * INT_SIZE;
            end;
        end;
    begin
        init(k);
    end.
```

Figure 3-14: Use of ALIAS Information Allows Linear Test Replacement

```
procedure P3(x, y: real; var z: real);
var tmp: teal;
begin
    tmp = (x + y) * t;
    z = ((tmp + y) * t + tmp) / t2;
end;
```

Figure 3-15: Procedure P3, Unoptimized

```
procedure P3(x, y: real; var z: real);
var tmp: real;
begin
    R0 = x;
    R1 = y;
    R2 = t;
    R3 = (R0 + R1) * R2;
    R4 = &z;
    R5 = t2;
    *R4 = [ (R3 + R1) * R2 + R3] / R5;
end;
```

Figure 3-16: Procedure P3, Optimized with Use of ALIAS Sets

4. Experimental Observations

The interprocedural analyzer was implemented using 9300 lines of Pascal source code. Interprocedural summary information was processed at a rate of approximately one second of VAX780 cpu time per 10 lines of Pascal source, adding about 15% to the total compile time. The global optimizer ran 10% to 20% slower when utilizing interprocedural information; most of this time was spent reading a given benchmark's associated interprocedural summary file for a given benchmark. This time could be substantially improved by storing the summary information in a more optimal format.

The benchmarks

27 Pascal benchmarks were used in evaluating the overall effect of interprocedural analysis on program speedup. Figure 4-l lists the names of these benchmarks, along with some statistics:

lines The number of lines of source code in the benchmark.

procs The number of procedures defined statically in each program.

lftime The percentage of loads and stores that occur within the leaf procedures of each program. Procedures that make only system calls are counted as leaf procedures.

int_{call} Within a given procedure P, MOD and USE information affects only those operations involving variables that are within the scope of one or more procedures called by P. By counting the number of accesses to such variables, we get an idea of the possible impact of MOD and USE information on the optimization of the program. This number, represented as a percentage of all memory accesses, is int_{call}. The calculation of int_{call} is discussed further below.

int_{alias} ALIAS information is important only within procedures that access var parameters. Interference only occurs between either a var parameter and a global or one var parameter and another var parameter. The percentage of memory accesses that are susceptible to such interference is calculated as int_{alias}.

The figure of merit for procedure call interference int_{call} is defined as the ratio of memory accesses susceptible to interference by procedure calls to all memory accesses. If we believe that the number of cycles saved by global optimization is proportional to the number of loads and stores eliminated, then int_{call} gives us an upper bound for the amount of speedup to be obtained through the use of MOD and USE information*.

Within a given procedure P, the number of memory accesses susceptible to interference by procedure calls is $int_{call}(P)$. For a leaf' procedure LF, $int_{call}(LF)$ is zero, **i.e. no memory** references interfere with procedure calls. For a procedure GE that calls only itself and/or up-level procedures, $int_{call}(GE)$ is equal to the number of direct accesses to global variables from within GE[2]. For a procedure LT making at least one call to a procedure within its static scope, $int_{call}(LT)$ is equal to the number of **all direct memory accesses, local or global, made by** LT.

As shown in the table, the average benchmark had 1190 lines of source code and 30 procedures. Nearly 50% of all memory accesses occurred in leaf procedures. 15% of all memory accesses had potential interference with a procedure call. 1% of all memory accesses could potentially be aliased to other variables via the mechanism of pass-by-reference parameter binding.

Note that the metrics int_{call} and int_{alias} are based on worst-case estimates of data flow patterns within a procedure. Were actual data flow taken into account, these numbers would become strictly smaller. As an example of where our worst-case estimate differs from actual numbers, consider the bubble-sort benchmark *bubble.p*. This program begins with a call to an initialization procedure *Initarr*. In the next basic block, the global variable *i* is defined, after which it is accessed 750,000 times within a loop. In this case, int_{call} counts 750,000 possible interferences whereas,

---

'Similarly, int_{alias} may give us an upper bound for the amount of speedup to be obtained thru the use of ALIAS information.

[2]Indirect accesses through pointers are not affected by interprocedural analysis.

filename	lines	procs	ltime	int_{call}	int_{alias}
ack	15	2	0%	0%	0%
uniforum	18	1	100%	0%	0%
fib	19	2	0%	0%	0%
loop1 la	22	1	100%	0%	0%
sieve	33	1	100%	0%	0%
cyk	44	2	100%	0%	0%
intmm	76	6	99%	0%	0%
queen	64	2	0%	1%	0%
bubble	84	5	0%	68%	0%
perm	86	7	41%	12%	0%
strings	107	6	56%	35%	0%
place	119	9	97%	2%	0%
tree	122	8	16%	52%	0%
simple	172	5	63%	22%	0%
puzzle	155	5	95%	0%	0%
dhrystone	251	12	46%	32%	4%
pwhets	350	10	64%	19%	3%
tlb	354	14	23%	59%	0%
newtlb	355	14	14%	27%	0%
bigfm	541	18	33%	11%	0%
ccal	799	27	8%	18%	3%
bench	941	48	75%	3%	0%
dnf	1585	37	44%	1%	8%
hopt	2197	92	66%	12%	11%
simu	3458	101	11%	13%	5%
macro	7484	179	33%	26%	10%
upasmips	13768	235	5%	39%	16%
(average)	1230	31	47%	15%	1%

Figure 4-1: Benchmark Statistics

once we actually do the data flow analysis, we find that there **is no** interference between the procedure call and ***i.*** Simple inspection of the statistical data **for *bubble.p*** shows that the actual **number of interferences is** almost nil, even though our algorithm reports a possible 68% interferences between variables and procedure calls,

Each benchmark was run four times, with varying degrees of interprocedural analysis taken into account. The results are reported in figure 4-2 under four colums labeled as shown below:

uopt, no ipa The program was compiled and optimized, but **no** inter-procedural summary information was used. Worst-case assumptions were made concerning procedure calls and aliasing. The program speedup is reported as compared to the running time of an unoptimized program. For example, 1.09 means 9% faster.

uopt + alias The program was compiled and optimized using ALIAS summary information. Program speedup is reported versus the running time of an unoptimized program. In parentheses is shown the incremental difference between the running time of the program as compiled and optimized without the use of any interprocedural summary information (plain vanilla optimization) versus the running time of the program as compiled and optimized with ALIAS information added (i.e. **uopt, no ipa** versus **uopt + alias).** A difference of 0% means that the addition of ALIAS information caused the running time of the program to improve by less than 1%.

uopt + **mod/use** **The** program was compiled and optimized using MOD and USE summary information. Program speedup is reported versus the running time of an unoptimized program. In parentheses is shown the difference between the plain vanilla optimization for this program versus the running time of the program as compiled with MOD and USE information (i.e. **uopt, no ipa** versus **uopt + mod/use).**

uopt + both **The** program was compiled and optimized using all available interprocedural summary

information (MOD, USE, and ALIAS). Program speedup is reported versus the running time of an unoptimized program. In parentheses is shown the difference between plain vanilla optimization for this program versus the running time of the program as compiled with all available summary information (i.e. uopt, **no ipa** versus **uopt + both).**

filename	uopt, no ipa	uopt + alias	uopt + mod/use	uopt + both
ack	1.11	1.11 (0%)	1.11 (0%)	1.11 (0%)
bench	2.07	2.07 (0%)	2.07 (0%)	2.07 (0%)
bigfm	1.53	**1.54 (0%)**	1.53 (0%)	1.53 (0%)
bubble	3.00	3.00 (0%)	3.00 (0%)	3.00 (0%)
ccal	1.16	1.16 (0%)	1.17 (0%)	1.17 (0%)
cyk	**1.60**	**1.60 (0%)**	**1.60 (0%)**	1.60 (0%)
dhrystone	**1.17**	**1.17 (0%)**	**1.18 (0%)**	1.18 (0%)
dnf	**1.28**	**1.28 (0%)**	**1.28 (0%)**	1.28 (0%)
fib	**1.09**	**1.09 (0%)**	**1.09 (0%)**	1.09 (0%)
hopt	**1.06**	**1.06 (0%)**	**1.06 (0%)**	1.06 (0%)
intmm	2.07	2.07 (0%)	2.07 (0%)	2.07 (0%)
looplla	**2.91**	**2.91 (0%)**	**2.91 (0%)**	**2.91 (0%)**
macro	**1.15**	**1.15 (0%)**	**1.15 (0%)**	**1.15 (0%)**
newtlb	**1.19**	**1.19 (0%)**	**1.27 (7%)**	**1.27 (7%)**
perm	**1.27**	**1.27 (0%)**	**127 (0%)**	1.27 (0%)
place	2.74	2.74 (0%)	2.74 (0%)	2.74 (0%)
puzzle	2.45	2.45 (0%)	2.45 (0%)	2.45 (0%)
pwhets	**1.09**	**1.09 (0%)**	**1.09 (0%)**	1.09 (0%)
queen	**1.55**	**1.55 (0%)**	**1.55 (0%)**	1.55 (0%)
sieve	**2.19**	**2.19 (0%)**	**2.19 (0%)**	2.19 (0%)
simple	**1.48**	**1.48 (0%)**	**1.48 (0%)**	1.48 (0%)
simu	**1.22**	1.22 (0%)	**1.22 (0%)**	1.22 (0%)
strings	**1.53**	1.53 (0%)	**1.55 (1%)**	**1.55 (1%)**
tlb	**1.07**	**1.07 (0%)**	**1.19 (12%)**	**1.19 (12%)**
tree	**1.20**	**1.20 (0%)**	**1.25 (4%)**	1.25 (4%)
uniforum	**1.31**	**1.31 (0%)**	**1.31 (0%)**	1.31 (0%)
upasmips	**1.04**	**1.04 (0%)**	**1.04 (0%)**	**1.04 (0%)**

Figure 4-2: Benchmark Improvement

Of a suite of 27 **benchmarks, only two showed an improvement greater than** 2%. **The failure of these** benchmarks to produce satisfactory results leads to the following conclusions.

- The call graphs for these benchmarks are not complex. Much of a program's running time is spent either in leaf procedures or in areas where procedure calls do not interfere significantly with data flow patterns.
- Aliasing patterns in these benchmarks are not complex.

Our work has been limited to the domain of Pascal programs running on a sequential machine. However, our **results seem to concur with those of Allen, Callahan, and Kennedy [Allen et al 86 86], who report a similar lack of** improvement for FORTRAN programs running in the context of a vectorizing compiler.

5. Conclusions

An interprocedural analyzer has been implemented and integrated into an existing Pascal compiler [Chow 83]. The efficiency of the algorithm used to perform inter-procedural analysis was not considered an issue, as our concern was to explore the result of using such analysis in an optimizing compiler. Our observations reveal that in an optimizing Pascal compiler, interprocedural summary information can produce interesting and tangible optimizations. Unfortunately, these optimizations do not happen in the frequently executed parts of a program. Thus, due to the characteristics of Pascal programs, inter-procedural information is unnecessary for code optimization.

ALIAS information, which is the most difficult to compute algorithmically, turns out to be the least useful. Our data show that only about 1% of the loads and stores in a given program are susceptible to being potentially aliased to a var parameter, and that the actual speedup gained by making such relationships explicit is almost nil.

MOD and USE information fares little better. On average, we find that 15% of loads and stores potentially interfere with procedure calls within the program. The actual speedup gained after using MOD and USE information is about 1%. The difference comes from the fact that worst-case assumptions are either correct or superfluous most of the time. For instance, if a given variable is defined immediately after a procedure call, the procedure call's possible use of the variable is superfluous. Similarly, if inter-procedural information tells us that a given variable is used by a given procedure call, this merely affirms the fact that our worst case assumption (all procedure calls affect all variables) is correct in this particular case.

Altogether, these statistics effectively defuse potential optimization opportunities that would otherwise have arisen due to lack of interprocedural summary information. Thus, the effective improvement in the results of optimizing transformations does not justify the expense of determining side-effects and aliases. It seems that a simple static scan could prove a reasonably practical alternative for inter-procedural analysis. For example, since 'Must' items are not needed in global code-optimization, other value-items can be easily derived by a global optimizer.

Although our observations have been limited to the domain of Pascal programs, it is interesting to note that others [Allen et al 86 86] have reached similar conclusions in the domain of vectorizing FORTRAN programs. They claim that USE and MOD sets increase the accuracy of information, but in the final analysis they do not help programs run faster. They also claim that alias analysis does not help much because aliasing is usually very rare. In addition, they note that interprocedural constant propagation does not produce significant run-time improvements. Similarly, Burke [Burke 84] notes that the potential benefits of interprocedural summary information are not significant when compiling for sequential machines. Our results concur with these observations and they tend to verify that potential benefits of inter-procedural optimization are indeed small for Pascal programs.

Appendix I
Benchmark Descriptions

ack	Ackerman's function.
bench	Combination of the 10 Stanford benchmarks: bubble, dnf, perm, queen, tower, quick, tree, mm, puzzle, and fft.
bigfm	An implementation of the Fiduccia-Mattheyses algorithm as described in 19th DA, pp 175-181. This algorithm reads a netlist of a hyper-graph from standard input and produces a partition of that graph into two (roughly) equal sized sections, written to standard output. The algorithm adapts the Kemighan-Lin algorithm for graph partition by adding data structures to speed the calculation of the biggest gain, and the updating of gains of neighbors for each move.
bubble	Bubble sort.
ccal	Interactive desk calculator, written by Warren Cory of Stanford University.
cyk	$O(n^3)$ Cocke-Younger-Kasami parsing algorithm based on dynamic programming. See Intro. to Automata Theory, Languages, and Computation, by J. E. Hopcroft and J. D. Ullman.
dhrystone	An integer version of whetstone.
dnf	A normalizer from arbitrary boolean expressions to Disjunctive Normal Form.
fib	A heavily recursive algorithm forprinting out the Fibonacci sequence.
hopt	Optimizer for intermediate code, written by Edwin Hou of Stanford University.
intmm	Multiply two integer matrices.
looplla	Sums the elements of two large arrays in a unique way.
macro	Macro expansion preprocessor for the SCALD computer-aided &sign system developed by T. M. McWilliams and L. C. Widdoes of Stanford University. It is described in Stanford University Technical Report 152, Digital Systems Laboratory, March 1978.
newtlb	TLB and cache simulator. Same as "tlb.p" except with a different set of constants (i.e. different sizes for the cache and TLB).
perm	A permutation program written by Denny Brown to illustrate recursion.
place	Evaluate and find placements assuming cheap line placement.
puzzle	An undocumented compute-bound program from Forest Baskett.
pwhets	A Pascal version of the whetstone program.
queen	A heavily recursive solution to the 8 queens problem.
sieve	Sieve of Eratosthenes: adapted by Chris Rowen from Sept 1981 Byte magazine: "A High-Level Language Benchmark" by Jim Gilbreath, pp. 180498 vol 6 no 9.
simu	Simulation of an Operating System.
strings	A hash table for character strings.
tlb	TLB and cache simulator.
tree	sorts an array.
uniforum	Multiplication of two integer arrays.

Appendix II
Bibliography Notes

[Aho et al 86] Compiler textbook.

[Allen 74] Allen presents an algorithm for finding (1) "uses of data items in a procedure which can be affected by outside definitions," and (2) "data items which may be changed when a procedure is referenced." The algorithm is restricted to non-recursive procedures connected by a well-defined control flow graph. These restrictions are removed in a later paper [Allen & Schwartz 74]. The algorithm is implemented in IBM's Experimental Compiler System (ECS).

[Allen & Cocke 76]
Data flow analysis in IBM's PL/I oriented Experimental Compiling System.

[Allen & Schwartz 74]
An extension of a previously-presented inter-procedural data flow algorithm [Allen 74] which uses "overestimates" and "worst-case assumptions" to allow for recursive procedures and a control flow graph that is not well defined.

[Allen et al 80] A comprehensive discussion of the Experimental Compiler System (ECS), representing "a new compiler construction methodology that uses a compiler base which can be augmented to create a compiler for any one of a wide class of source languages." The system calculates interprocedural summary information and incorporates procedure inlining and interprocedural constant propagation. No experimental results or numbers are reported.

[Allen et al 86 86] Kennedy and his associates report on the results of their work at Rice. Very little increase in vectorization is found after the-application of GMOD sets, GUSE sets, and constant propagation. Aliasing effects are found to be unimportant to the programs under consideration. The analysis proves useful only as part of PTOOL, an interactive tool for finding parallelism in programs.

[Ball 79] Ball presents a technique for predicting how much improvement would be gamed in a given program if constant folding and test elision were applied to the program.

[Banerjee 86] A review of Triolet's work on parallelization of call statements in a multiprocessor environment.

Banning 79] The classic algorithm for practical interprocedural analysis, later adapted by Cooper.

[Barth 771 Single-pass algorithm for finding interprocedural "may" summary information.

[Barth 78] This algorithm for finding inter-procedural summary information is said to be "precise up to symbolic execution."

[Burke 84] Burke calculates MOD information and derives USE and REF. He makes the observation that summary information is of no great help in optimizing programs on a sequential machine architecture.

[Burke & Cytron 86]
Interprocedural dependence analysis and parallelization in a multiprocessor environment.

[Callahan et al 86] An algorithm to help implement constant propagation.

[Chow & Rudmik 82]
An algorithm for determining aliases in programs.

[Cooper 83] A proposal for a project that would use inter-procedural information to aid program design and construction.

[Cooper 85] A set of techniques for analyzing aliasing patterns.

[Cooper & Kennedy 84]
A study of Banning's algorithm The original algorithm is decomposed into two subproblems, each of which has an efficient solution. Running time improves, from O(nodes x edges) to "nearly linear." A later paper corrects a couple of flaws in the logic [Cooper & Kennedy 87a].

[Cooper et al 86a] For use in an incremental compilation environment. The paper presents a test for determining which procedures in a program must be recompiled following an editing session.

[Cooper et al 85] A description of the IR^n Programming Environment. No numbers are reported

[Cooper et al 86b] Another description of the IR^n Programming Environment.. No numbers are reported.

[Cooper et al 86c] [Cooper et al 85] in journal form.

[Coutant 86] An algorithm for finding aliases, retargetable for different machines/languages. Pointers are handled as well as ref parameter aliasing. The algorithm was implemented for C, Pascal, COBOL, and FORTRAN/77. The optimizer gets impressive results; it is not clear how much program speedup is due to alias analysis. Plans are made to integrate the algorithm into an inter-procedural global optimizer, which includes "handling of pointer alias set initialization, side effects of procedure calls, and the binding of formal and actual paramaters." It will be interesting to see if they gain any speedup.

[Fosdick & Osterweil 76]
Using data flow algorithms to search for programming errors.

[Jones & Muchnik 82]
A new approach to data flow analysis of procedural programs and programs with recursive data structures is described The method depends on simluation of the interpreter for the subject programming language using a retrieval function to approximate a program's data structures.

[Lomet 773 A method for identifying and reducing the hazards caused by aliasing. Also see [Rosen 79]

[Masinter 80] A data base for program flow information, implemented in LISP.

[Morel & Renvoise 81]
Using interprocedural analysis to perform elimination of redundant expressions.

[Myers 80] Solves Banning's algorithm [Banning 79] by dividing it into two subproblems, much as Cooper and Kennedy do later [Cooper & Kennedy 84]. Myers uses transitive closure to solve the subproblem dealing with aliases, and his solution is consequently slower than that of Cooper and Kennedy.

[Myers 81] Where Banning [Banning 79] and an earlier Myers paper present algorithms for computing may information, this paper looks at the remaining problems of live, avail, and must-summary.

Richardson & Ganapathi 87]
A bibliography of work in interprocedural analysis to June 1987.

[Rosen 76] A less readable form of [Rosen 79].

[Rosen 79] The interprocedural algorithm presented here derives information specific to *each procedure call*. Similar to Lomet [Lomet 77], except better in that it obtains the "sharpest possible local information."

[Ryder 871 Incremental ipa in the style of the IR^n environment. The Rutgers version is called ISMM.

[Sharir & Pnueli 81]
The two techniques presented here *are the functional approach* and *the call-strings approach* to interprocedural data flow analysis. The functional approach makes use of summary information in the classic style. The call-strings approach turns the entire program into a single flowgraph.

[Spillman 71] A not-very-in-depth discussion of some problems in PL/I, including side effects due to aliasing, pointers, label variables, and procedure invocations.

[Torczon 85] A description of the optimizing compiler used in the IR^n programming environment,

[Triolet 85] Triolet presents a method of parallelizing CALL statements in FORTRAN.

[Triolet et al 86] Direct parallelization of CALL statements; more of Triolet's work. The algorithm uses the new objects *Region* and *Execution Context.*

[Wall 86] Wall describes his algorithm for global (interprocedural) allocation at program link time. Only local variables are allocated to registers. Profile information is used to find out the usage frequency of each local variable.

[Weihl 80] An algorithm for handling the problem of procedure variables, label variables, and general aliasing relationships.

[Weiser 84] Weiser presents the concept of program slicing. In program slicing, one concentrates on one data flow problem at a time; parts of the program not related to the given problem are "sliced" away in order to simplify the problem.

References

[Aho et *al* 86] Aho, A. V., Sethi, R. and Ullman, J. D.
Compilers, Principles, Techniques and Tools.
Addison-Wesley, Reading, Mass, 1986.

[Allen 74] Allen, F. E.
Interprocedural Data Flow Analysis.
In Proceedings of the IFIP Congress 1974, pages 398-402. ***North*** Holland, Amsterdam, 1974.

[Allen & Cocke 76]
Allen, F.E., Cocke, J.
A program data flow analysis procedure.
CACM 19(3):137-147, March, 1976.

[Allen & Schwartz 74]
Allen, F.E., Schwartz, J.
Determining the Data Relationships in a Collection of Procedures.
IBM Research Report RC 4989 (22125), I.B.M. Yorktown Heights, 1974.

[Allen et al 80] Allen, F.E., Carter, J.L., Fabri, J., Ferrante, J., Harrison, W.H., Loewner, P.G., Trevillyan, L.H.
The Experimental Compiling System
IBM 24(6), November, 1980.

[Allen et al 86 86] Allen, Randy, David Callahan and Ken Kennedy.
An Implementation of Interprocedural Data Flow Analysis in a Vectorizing Fortran Compiler.
Technical Report COMP TR86-38, Dept.. of Computer Science, Rice University, May, 1986.

[Ball 79] Ball, JE.
Predicting the effects of optimization on a procedure body.
In Proc. '79 Sym. on Compiler Construction. ACM, Colorado, August, 1979.

[Banerjee 86] Banerjee, Utpal.
A Direct Parallelization of Call Statements -- A Review.
Technical Report 576, Center for Supercomputer Research and Developement, University of Illinois, Urbana, 1986.

[Banning 79] Banning, J.P.
An Efficient Way to Find the Side Effects of Procedure Calls and the Aliases of Variables.
In ***Proc. 6th POPL Conference***, pages 724-736. ACM, 1979.

[Barth *771* Barth, Jeffrey M.
An Interprocedural Data Flow Analysis Algorithm
In Proc. 4th POPL Conference, pages 119-131. ACM, January, 1977.

[Barth 78] Barth, J.M.
A Practical Interprocedural Data Flow Analysis Algorithm.
CACM 21(9):724-736, September, 1978.

[Burke 84] Burke, M.
An Interval Analysis approach toward Interprocedural Data Flow.
Research Report, IBM Yorktown Heights RC 10640, I.B.M, 1984.

[Burke & Cytron *86*]
Burke, M. and Cytron, R.
Interprocedural Dependence Analysis and Parallelization.
In Proceedings of the SIGPLAN '86 Symposium on Compiler Construction. ACM, Palo Alto, California, June, 1986.

[Callahan et al 86] Callahan, D., Cooper, K., Kennedy, K. and Torczon, L.
Interprocedural Constant Propagation.
In ***Proceedings of the SIGPLAN '86 Symposium on Compiler Construction.*** ACM, Palo Alto, California, June, 1986.

[Carroll 87] **Carroll, M. D.**
Dataflow update bia attribute and dominator update.
PhD thesis, Rutgers University, 1987.

[Chow 83] Chow, F.
A Portable Machine-Independent Global Optimizer - Design and Measurements.
PhD thesis, Stanford University, December, 1983.

[Chow & Rudmik 82]
Chow, **AL. and Rudmik, A.**
The Design of a Data Flow Analyzer.
In *Proc. '82 Sym. on Compiler Construction,* pages **106-l** 13. **ACM, Boston, Massachusetts,** June, 1982.

[Cooper 83] **cooper, K.**
Interprocedural data flow analysis in a programming environment.
PhD thesis, Rice University, April, 1983.

[Cooper 85] Cooper, K.
Analyzing aliases of reference forma-l parameters.
In *Proc. 12th POPL Conference.* **ACM, January,** 1985.

[Cooper & Kennedy 84]
Cooper, K. and K. Kennedy.
Efficient Computation of Flow Insenstitive Interprocedural Summary Information.
In *Proc. '84 Sym. on Compiler Construction,* pages 247-258. **ACM, Montreal, Canada, June,** 1984.
Correction appears as technical report 87-60, Dept. of Computer Science, Rice University.

[Cooper & Kennedy 87a]
Cooper, K. and K. Kennedy.
Efficient Computation of Flow-Insensitive Interprocedural Summary Information -- A Correction.
Technical Report 87-60, Rice University, October, 1987.

[Cooper & Kennedy 87b]
Cooper, K. and K. Kennedy.
Complexity of Interprocedural Side-Effect **Analysis.**
Technical Report 87-61, Rice University, October, 1987.

[Cooper & Kennedy 87c]
Cooper, K. and K. Kennedy.
Interprocedural Side-Effect Analysis in Linear Time.
Technical Report 87-62, Rice University, October, 1987.

[Cooper et al 85] Cooper, K., K. Kennedy and L. Torczon.
The Impact of Interprocedural Analysis and Optimization on the Design of a Software DevelopmentEnvironment.
In *Proc. '8.5 Sigplan Symposium.* **ACM, July,** 1985.

[Cooper et al 86a] Cooper, **K.,** K. Kennedy and L. Torczon.
Inter-procedural Optimization: Eliminating unnecessary recompilation.
In Proceedings of the SIGPLAN '86 Symposium on Compiler Construction. **ACM, Palo Alto,** California, June, 1986.

[Cooper et al 86b] **Cooper, K., K. Kennedy and L. Torczon.**
Optimization of Compiled code in the R^n programming environment.
In Proc. 19th Hawaii International Conference on System Sciences. **January, 1986.**

[Cooper et al 86c] Cooper, K., K. Kennedy, and L. Torczon.
The Impact of Interprocedural Analysis and Optimization in the R^n Programming Environment.
ACM Transactions on Programming Languages and Systems 8(4):491, 1986.

[Coutant 86] Coutant, D.
Retargetable High Level Analysis,
In Proc. 13th POPL Conference, pages 110-l 18. ACM, January, 1986.

[Fosdick & Osterweil 76]
Fosdick and Osterweil.
Data Flow Analysis in Software Reliability.
ACM Computing Surveys 8(3):305-330, September, 1976.

[Jones & Muchnik 82]
Jones, Neil D. and Steven S. Muchnik.
A Flexible Approach to Interprocedural Flow Analysis and Programs with Recursive Data Structures.
In Conf. Rec. of the Ninth ACM Symp. on Princ. of Prog. Lung., pages 66-74. ACM, January, 1982.

[Lomet 77] Lomet, D.
Data Flow Analysis in the presence of procedure calls.
IBM Journal of Research and Development 21, 6, I.B.M., 1977.

[Masinter 80] Masinter, L.
Global Program Analysis in an Interactive Environment.
Technical Report SSC-80-1, Xerox PARC, 1980.

[Morel & Renvoise 81]
Morel, B. and Renvoise, A.
Interprocedural Elimination of Partial Redundancies.
In Muchnik, Steven S. and Neil D. Jones (editor), ***Program Flow Analysis: Theory and Applications.*** Prentice-Hall, Englewood Cliffs, NJ, 1981.

[Myers 80] Myers, E.
A Precise and Efficient Algorithm for Determining Existential Summary Data Flow Information.
Technical Report CU-CS- 175-80, Dept. of Computer Science, University of Colorado, Boulder, 1980.

[Myers 81] Myers, E.
A precise interprocedural data flow analysis algorithm.
In Proc. 8 POPL Conference. **ACM,** January, 1981.

Richardson & Ganapathi 87]
Richardson, S., and M. Ganapathi.
Interprocedural Analysis -- A Bibliography.
SIGPLAN Notices 22(6): 12-17, June, 1987.

[Rosen 76] Rosen, B.
Data Flow Analysis for Procedural Languages.
Research Report., IBM Yorktown Heights RC 5948, LB.M, 1976.

[Rosen 79] Rosen, B.
Data Plow Analysis for Procedural Languages,
JACM 26(2):322-344, *1979.*

[Ryder 871 Ryder, Barbara G.
An Application of Static Program Analysis to Software Maintenance.
In Proc. 20th Hawaii International Conference on System Sciences, pages 82-9 1. January, 1987.

[Sharir & Pnueli 81]
Sharir, *M.* and Pnueli, A.
Two Approaches to inter-procedural data flow analysis.
In Muchnik, ***Steven S.*** and ***Neil*** D. Jones (editor), ***Program Flow Analysis: Theory and Applications.*** Prentice-Hall, Englewood Cliffs, NJ, 198 1.

[Spillman 71] Spillman, T.C.
Exposing **Side-Effects in a PL/I Optimizing Compiler.**
In Proceedings of the IFIP Congress 1971, pages 376-38 **1. North** Holland, Amsterdam, 197 1.

[Torczon 85] Torczon, L.
Compilation dependencies in an ambitious optimizing compiler.
PhD thesis, Rice University, May, 1985.

[Triolet 85] Triolet, R.
Interprocedural Analysis for Program Restructuring with PARAFRASE.
Technical Report 538, Center for Supercomputer Research and Developement, University of Illinois, Urbana, 1985.

[Triolet et al 86] **Triolet, R., Irigoin, F. and Feautrier, P.**
Direct Parallelization of Call statements.
In *Proc. '86 Sym. on Compiler Construction.* ACM, Palo Alto, California, June, 1986.

[Wall 86] **Wall, D.**
Global Register Allocation at Link Time.
In *Proc. '86 Sym. on Compiler Construction.* ACM, Palo Alto, California, June, 1986.

[Weihl 80] **Weihl, W.E.**
Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables.
In *Seventh Sym. on Principles of Programming Languages,* pages 83-94. ACM, January, 1980.

[Weiser 84] **Weiser, M.**
Program slicing.
***IEEE Trans. on Software Engineering* 10(4):352-357, 1984.**