RC 20290 (11/29/95)
Computer Science

IBM Research Report

Flow-Sensitive Interprocedural Constant Propagation

Paul R. Carini

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

Michael Hind

Department of Mathematics and Computer Science
State University of New York at New Paltz
New Paltz, NY 12561
and IBM T.J. Watson Research Center

IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich

Flow-Sensitive Interprocedural Constant Propagation*

Paul R. Carini† Michael Hind‡

November 14, 1995

Abstract

We present a flow-sensitive interprocedural constant propagation algorithm that supports recursion while only performing one flow-sensitive analysis of each procedure. We present experimental results which show that this method finds more constants than previous methods and is efficient in practice. The results are further categorized into integer and floating point constants. We show how the flow-sensitive algorithm can be extended to propagate returned constants. We introduce new metrics for evaluating interprocedural constant propagation algorithms which measure the number of interprocedural constant values that are propagated. We use these metrics to provide further experimental results for our algorithm.

1 Introduction

Program analysis has struggled with the tradeoff between modularization and optimization in many forms. Interprocedural analysis (IPA) is a well-known technique that can be used to bridge the gap across procedure boundaries. The popularity of object-oriented languages, such as C++, increases the opportunity for, and the value of, interprocedural analysis and transformations. Constant propagation identifies constant definitions of variables and propagates the constant values to the uses of those objects. Interprocedural constant propagation (*ICP*) propagates the constant values of variables across procedure boundaries [9, 10, 22, 18, 26]. This may lead to more precise data-flow information, which can help improve the precision of other analyses, as well as enable other program transformations to generate more efficient code.

Flow-sensitive interprocedural algorithms consider intraprocedural control flow and are, in general, more precise, but less efficient, than flow-insensitive algorithms [4, 21], which do not consider intraprocedural control flow information during the analysis. Flow-sensitive interprocedural constant propagation relies upon a flow-sensitive intraprocedural constant propagation method to identify constants, and then propagates these constants interprocedurally. In this paper we introduce a flow-sensitive interprocedural constant propagation method. In addition to the flow-sensitive method, this paper also describes a flow-insensitive algorithm, which our flow-sensitive method utilizes when call graph cycles exist. The combination of these two methods enables us to approach the precision of an iterative flow-sensitive interprocedural analysis, without requiring more than one flow-sensitive analysis of each procedure. Our algorithms propagate both formal parameters and global variables, and were designed to support multiple languages, including C and C++. These algorithms have been implemented in a Fortran 90 research prototype compiler. While not as precise as a full iterative solution, the use of flow-insensitive results for back edges is a reasonable tradeoff between efficiency and precision.

---

*A preliminary version of this paper appeared in ACM SIGPLAN'95: *Conference on Programming Language Design and Implementation*, (San Diego, CA, June 18-21, 1995) [11]

†IBM T.J. Watson Research Center, P.O. Box 704 Yorktown Heights, NY 10598, carini@watson.ibm.com

‡Department of Mathematics and Computer Science, State University of New York at New Paltz, New Paltz, NY 12561, and IBM T.J. Watson Research Center, hind@mcs.newpaltz.edu

```
void main(void) {
    sub1(0);
}

void sub1(int f1) {
    int x, y;
    x = 9;
    if (f1 == 0)
        y = 1;
    else
        y = 0;
    sub2(y, 4, f1, x);
}

void sub2(int f2, int f3, int f4, int f5) {
    ... = f2 + f3 + f4 + f5;
}
```

Method	Formal Parameter Constants
Flow-Sensitive	f1, f2, f3, f4, f5
Flow-Insensitive	f1, f3, f4
Literal	f1, f3
Intra	f1, f3, f5
Pass-through	f1, f3, f4, f5
Polynomial	f1, f3, f4, f5

Figure 1: Example Program and Precision Comparison

We present measurements of the effectiveness of each technique and contrast these with results previously described in the literature. Our results are further categorized into integer and floating point constants. We also suggest metrics that characterize the effectiveness of interprocedural constant propagation. These metrics are a true reflection of the number of interprocedural constant values that are propagated. Our results show that the flow-sensitive method discovers and propagates more constants than the flow-insensitive method alone, with little extra compile-time cost. We also extend the flow-sensitive algorithm to propagate returned constants and illustrate its effectiveness with an example.

Consider Figure 1, which illustrates the difference between flow-sensitive and flow-insensitive constant propagation for formal parameters. By inspecting the call site argument information, noting that the formal parameters are not modified, and propagating this information to the called routine, a flow-insensitive algorithm can detect that the values of parameters `f1`, `f3`, and `f4` are constant. Determining that `f2` and `f5` are also constant requires an analysis of the intraprocedural control flow of `sub1`. In particular, `x` and `y` must be the same constant on all paths from the entry of `sub1` to the call of `sub2`. Since `f1` has the constant value 0, the path containing "`y = 0`" is not executed. Figure 1 also specifies the constants that would be found by the various "*jump functions*" used in [18]. These techniques are discussed in Section 5.

2 Compilation Model

In addition to intraprocedural analyses, our compiler performs a number of interprocedural analyses. Our compilation model provides an *IPA* collection phase, during which each procedure in the program is visited and the *IPA* inputs are collected and saved for later use. This information includes such items as procedure names and formal parameters, procedure call sites and arguments, global variables which are immediately modified or referenced, and an intermediate representation of the procedure. Figure 2 illustrates the steps of the compilation model. Our model is compatible with the traditional separate compilation model and with automatic program generators and utilities such as *make* and *sccs*.

The initiation of the linking phase of an executable is used as the signal to begin the interprocedural analysis phase. After the complete program has been visited (the model includes a provision for

1. Collect IPA inputs
2. Construct the Program Call Graph
3. Perform Interprocedural Aliasing (Pointer and Reference Parameter)
4. Compute Interprocedural Mod and Ref
5. Perform Interprocedural Constant Propagation
6. Perform Reverse Topological Traversal

Figure 2: Compilation Model Overview

handling missing procedures), the interprocedural analysis phase begins by constructing the *program call graph* (PCG). An interprocedural alias analysis is then performed, which includes both reference parameter [14, 4] and pointer-induced [12, 8] alias analyses. This is followed by a phase that includes the computation of interprocedural MOD and REF information [4, 13, 6], which is implemented as a flow-insensitive traversal of the PCG. This computation relies upon the IPA inputs that are initially collected, as well as the previously computed alias information.

Next, interprocedural constant propagation is performed, which uses the interprocedural MOD and REF information along with the IPA inputs. The interprocedural constant propagation consists of two separate steps, the computation of interprocedural constants, and the transformation of a program representation to reflect these constants.

The final phase of interprocedural analysis is a reverse topological traversal (*backward walk*) of the PCG. During this traversal each procedure (node) is visited once. The intermediate representation of each procedure is first restored to memory, and optional procedure inlining and cloning may be performed. These transformations have the output of interprocedural constant propagation available to them. At this stage, the intermediate representation is transformed to reflect the results of interprocedural constant propagation, other traditional analyses and transformations are performed, upward exposed use is computed and saved, and final code is generated.

Since interprocedural analysis and transformations can introduce additional source file dependencies, all object code generation is deferred until an executable is created. As a consequence, object code for the entire program is generated whenever any source file is updated and recompiled. This mechanism insures that no object code that is based on outdated interprocedural information is used, and eliminates the possibility of error due to additional cross file dependencies.

3 Method

In this section we provide a description of our analysis methods. We describe the intraprocedural constant propagation algorithm we employ, and then describe the two variations of interprocedural constant propagation: flow-insensitive and flow-sensitive. We describe a flow-insensitive and flow-sensitive algorithm to propagate constants in the forward direction. We then show how the flow-sensitive algorithm can be extended to propagate returned constants.

Our prototype performs intraprocedural constant propagation by default. The routine is an implementation of the "*Sparse Conditional Constant*" (SCC) algorithm of Wegman and Zadeck [26], and is built upon an implementation of SSA [15] data-flow analysis. This is an optimistic algorithm that discards unreachable code during the propagation, which may permit the identification of additional constants.

An interprocedural constant propagation method (flow-insensitive or flow-sensitive) may optionally be performed. If interprocedural constants are identified, these constants are propagated to the entry point of the procedure where they are constant during the backward walk of the compiler. This propagation is equivalent to adding an assignment statement for each constant variable at the beginning of the procedure where it is constant. Assignment statements are created only for those variables that are referenced in that procedure.

3.1 Flow-Insensitive Interprocedural Constant Propagation

The flow-insensitive method is performed in a forward traversal of the PCG, after interprocedural MOD and REF information have been computed. For ease of implementation, globals are treated separately from parameters.

The flow-insensitive method for formal parameters, shown in Figure 3, is an optimistic data-flow algorithm in that all formal parameters are initialized to $\top$. It collects and propagates immediate call site constants to their called procedures. If a formal is found to be constant, and is subsequently passed to another formal at a call site without being modified in its procedure (directly or indirectly via any call site), this constant is also propagated, and the relationship between the formals is recorded in a set called *fp_bind*. To handle cycles in the PCG, a work list is used to keep track of formals that are constant, but are lowered (in the lattice algebra sense) at a later time. When processing a constant argument, if the corresponding formal is $\top$, the formal is marked as constant.

The flow-insensitive algorithm for global variables, given in Figure 3, identifies global constants by visiting any global initializations (i.e. Fortran block data) and recording those variables that are initialized to a constant on a list. Global variables that are modified anywhere in the program (i.e. those on the procedure MOD list for the main procedure) are removed from this list. The remaining variables on the list are constant for the entire program. These global constants are propagated to every procedure, but constant definitions are only created for a variable that is immediately referenced in a procedure.

Our flow-insensitive method is similar to the "*Pass-through*" parameter forward jump function method [10, 18]. They differ in that the pass-through method applies a flow-sensitive intraprocedural constant jump function before the interprocedural solution is computed. Thus the pass-through method may identify more constants to interprocedurally propagate. Our flow-insensitive method defers the application of the intraprocedural constant propagation routine until the backward traversal of the PCG, just prior to code generation.

3.2 Flow-Sensitive Interprocedural Constant Propagation

The flow-sensitive constant propagation algorithm makes use of a flow-sensitive intraprocedural method. Although any intraprocedural method can be employed, our implementation uses the SCC algorithm of Wegman and Zadeck [26]. The flow-sensitive method is performed in one forward traversal of the PCG, after procedure MOD and REF information have been computed. This processing of the PCG includes the performance of intraprocedural constant propagation as each procedure is visited. A major characteristic of our algorithm is the *interleaving* of the intraprocedural and interprocedural phases. This algorithm is similar to ones previously described for interprocedural data-flow analysis [2] and interprocedural alias analysis [12, 8].

As a flow-sensitive intraprocedural analysis can be expensive, it is only performed once per procedure, i.e. there is no iteration during the flow-sensitive analysis. If a PCG has cycles (back edges), then an optimistic flow-sensitive interprocedural algorithm with one iteration of the PCG could give an incorrect solution [9]. We address this issue by performing a flow-insensitive analysis prior to the flow-sensitive analysis, only if there are cycles in the PCG. During the flow-sensitive analysis, if a call site edge has not been processed before the called procedure (i.e. it is a back edge), we use the solution obtained by the flow-insensitive interprocedural constant propagation method for this edge. We use this same method to compute procedure USE information in one reverse topological traversal of the PCG, where REF information is used for back edges. Figure 4 presents our flow-sensitive algorithm for formal parameters and global variables.

Global variables are handled in a similar manner. The method first examines block data and identifies the variables that are assigned constant values by entering them on a list. This list is then

```
procedure meet(formal, new_value)
    orig_value = formal
    formal = formal ∧ new_value
    if orig_value ≠ ⊥ and formal = ⊥ then
            add all fp_k to the worklist, such that (formal, fp_k) ∈ fp_bind
    end if
end meet

procedure insens()
    /* First process globals */
    Collect initial constants from block data, removing those that are modified in the program
    for each procedure, p, in the program
        for each constant global variable, c
            if p references c, (i.e., c ∈ Ref(p)) then
                propagate c into p
            end if
        end loop
    end loop
    /* Then process formal parameters */
    Initialize formal parameters to ⊤; worklist, and fp_bind to empty
    for each procedure, p, in the PCG in a forward topological traversal
        for each call site, cs, in p
            for each argument, arg, at cs
                Let fp1 be the corresponding formal parameter in the called procedure.
                if arg is an immediate constant or a global constant
                    meet(fp1, arg)
                else if arg is a formal parameter of p, fp0, that is currently marked as constant
                        and is not modified (directly or indirectly) by p
                    add (fp0, fp1) to fp_bind
                    meet(fp1, fp0)
                else
                    meet(fp1, ⊥)
                end if
            end loop
        end loop
    end loop
    /* Items on the worklist represent a pass-through formal parameter that was
        constant and then was lowered to ⊥. */
    while worklist is not empty
        remove an item, fp_j, from the worklist
        if fp_j is not ⊥ then
            set fp_j to ⊥
            add all fp_k to the worklist, such that (fp_j, fp_k) ∈ fp_bind
        end if
    end loop
end insens
```

Figure 3: Flow-Insensitive ICP for Formal Parameters and Global Variables

```
Perform flow_insens for formal parameters, recording the constant status of each argument
Initialize all formal parameters to ⊤
Collect initial constants from block data as an imaginary call to main
For each procedure, p, in a forward topological traversal of the PCG
    For each call site of p
        If all arguments corresponding to a particular formal parameter of p are the same constant
            propagate the constant to the formal parameter of p
        end if
        If a global is the same constant on each call to p
            propagate the global constant to p
        end if
        /* Use flow-insensitive information if a calling procedure has not been processed. */
    end loop
    Perform a flow-sensitive intraprocedural constant propagation on p
    If a constant is used as an argument
        Record the argument as constant
        /* This will discard the flow-insensitive solution for this argument. */
    end if
    if a global constant at a call site is in the Ref set for the called procedure then
        Record the global as constant at this call site
    end if
end loop
```

Figure 4: Flow-Sensitive ICP for Formal Parameters and Global Variables

provided as input to the intraprocedural constant propagation algorithm for the entry point of the main procedure. The intraprocedural algorithm propagates global constants to call sites. If the global variable is referenced in the called procedure, or in any procedure which is called by that procedure, it is propagated interprocedurally. This is determined from the procedure REF information. (The flow-sensitive procedure USE information has not yet been computed.) The referenced variable is then added to the list of constants for that call site. When a procedure is processed, the list of constants for each call site that invokes the procedure is inspected, and a meet is performed. If a back edge exists, the flow-insensitive global solution for the back edge is used.

The ratio of the number of back edges to the total number of edges can be used as a measure of the "flow-insensitiveness" of our solution. When this ratio is zero, i.e. no back edges exist, the same results as a flow-sensitive iterative solution (that does not propagate returned constants) are achieved, without requiring iteration. As the ratio of back edges to forward edges approaches one, more flow-insensitive information is utilized. In the limit that all edges are back edges and the ratio is one, the flow-sensitive method achieves the same results as the flow-insensitive solution. For intermediate ratio values, a result is achieved which is between the two limiting solutions. Even though we adopt a more conservative flow-insensitive solution for back edges, our method still benefits from a flow-sensitive intraprocedural analysis of each procedure.

3.2.1 Returned Constants

Returned constants can be accommodated by extending our flow-sensitive method to include one additional topological traversal of the PCG which is performed in the reverse direction. During this traversal, a second flow-sensitive intraprocedural analysis of each procedure is performed to identify the procedure's set of returned constant formal parameters[1] and global variables that are propagated

[1] This includes returned function values and does not include "pass-by-value" formal parameters.

```
For each procedure, p, in the PCG
    Perform a flow-insensitive intraprocedural constant propagation on p,
        initializing the returned constant set of p to the formal parameters
            and global variables that are constant at every exit point of p.
end loop
For each procedure, p, in a reverse topological traversal of the PCG
    Perform a flow-sensitive intraprocedural constant propagation on p:
        For each call site, use the returned constant set computed for the called procedure
        /* Use flow-insensitive information if the called procedure's returned constant set
            has not been computed. */
    At the exit of p, add formal parameters and global variables that are constant
        at every exit point of p to the returned constant set of p
end loop
```

Figure 5: Flow-Sensitive ICP for Returned Constants

to the invoking call site. A flow-insensitive solution is also precomputed and used for back edges in this traversal. This enhancement is described in Figure 5, and can be performed immediately after the algorithm given in Figure 4. If performed in this order, the computation of returned constants can use the constants discovered by the forward traversal. The backward traversal can be obtained for free by performing it during the reversal topological traversal in Step 6 of Figure 2 in a similar manner to the computation of USE.

Consider Figure 6 which extends Figure 1 by including returned constants. After the forward topological traversal, formal parameters f1 – f5 are found to be constant. The reverse topological traversal determines that the returned value from sub2 is constant. This result is used during the flow-sensitive intraprocedural analysis of sub1, which discovers that the returned value from sub1 is constant. This information can then be used in main, to determine if sub4 is invoked.

Note that this enhancement does not provide the full precision of a iterative solution over the PCG. This is demonstrated by the call to sub3 in sub1. Although the reverse topological traversal determines that r2 is constant, since this occurs after the only forward traversal, sub3 will not benefit from this information. To do so, would require an additional forward traversal for this example, and full iteration, in general. The table in Figure 6 summarizes the precision of our method as well as the return jump function implementation described in [18].

4 Results

Since this paper describes the behavior of interprocedural constant propagation, we have tried to distinguish the interprocedural effects from the intraprocedural ones. We measure the number of constant candidates that are interprocedurally propagated at each call site, and the number of interprocedural constants that are found to hold at the entry of a procedure. While the second metric is the main focus of this paper, the first metric is also important. The identification of constant candidates can be useful for other optimizations, such as procedure inlining and cloning. These results include both constant formal parameters and global variables. The metrics eliminate the duplicate counting that can occur if a constant interprocedural value is propagated to a variable that is referenced multiple times within a procedure. A global variable that is propagated to multiple procedures will be counted once for each procedure that it is propagated to.

We only propagate scalar variables, although we have observed that at least one benchmark would benefit from the propagation of constant array values. These results do not include the propagation of returned constants, since the implementation of this feature has not yet been completed. Our mea-

```
void main(void) {
    int r1;
    r1 = sub1(0);
    if (r1 == 0)
        sub4();
}

int sub1(int f1) {
    int x, y, r2;
    x = 9;
    if (f1 == 0)
        y = 1;
    else
        y = 0;
    r2 = sub2(y, 4, f1, x);
    sub3(r2);
    if (r2 > 0)
        return 1;
    else
        return 0;
}
```

```
int sub2(int f2, int f3, int f4, int f5) {
    ... = f2 + f3 + f4 + f5;
    return f2 + f3 + f4 + f5;
}

void sub3(int f6) {
    ...  = f6
    ⋮
}

void sub4(void) {
    ⋮
}
```

METHOD	FORMAL PARAMETER CONSTANTS	RETURNED CONSTANTS
FLOW-SENSITIVE	f1, f2, f3, f4, f5	–
FLOW-SENSITIVE WITH RETURNED CONSTANTS	f1, f2, f3, f4, f5	r1, r2
FLOW-SENSITIVE (*PCG* ITERATION)	f1, f2, f3, f4, f5, f6	r1, r2
POLYNOMIAL WITH RETURNED CONSTANTS	f1, f3, f4, f5	–

Figure 6: Example Program and Precision Comparison

Call Site Constant Candidates										
	Arguments							Globals		
program	Tot	Imm	Pct	FI	Pct	FS	Pct	FI	FS	Vis
013.spice2g6	2983	384	13%	384	13%	430	14%	0	533	302
015.doduc	483	39	8%	39	8%	43	9%	0	1	1
030.matrix300	178	25	14%	25	14%	110	62%	0	0	0
034.mdljdp2	195	11	6%	11	6%	11	6%	16	69	38
039.wave5	676	30	4%	32	5%	49	7%	74	249	231
048.ora	0	-		-		-		16	7	7
077.mdljsp2	195	11	6%	11	6%	11	6%	0	0	0
078.swm256	0	-		-		-		0	0	0
089.su2cor	644	110	17%	110	17%	110	17%	0	0	0
090.hydro2d	197	28	14%	28	14%	28	14%	0	1	1
093.nasa7	104	33	32%	33	32%	45	43%	0	3	3
094.fpppp	103	17	17%	17	17%	21	20%	0	8	4
Total	5758	688	11.9%	690	12.0%	858	14.9%	106	871	587

Table 1: Interprocedural Call Site Constant Candidates

surements were made on the Fortran subset of the SPECfp92 benchmarks [3] (except 047.tomcatv, which has no call sites), and the 030.matrix300 program from the first SPEC89 suite [25]. See [7, 19] for an analysis of interprocedural aspects of the Fortran subset of the first SPEC89 suite. The Fortran SPEC benchmarks are all written in Fortran77, which does not support recursion as a language feature. None of these benchmarks have back edges in the PCG.

Table 1 presents one metric, the number of call site constant candidates that are interprocedurally propagated. We report the number of constant actual arguments and the number of constant global candidates for each method. The first column reports the total number of arguments in each program. The column labeled Imm reports the number of immediate or literal constant arguments. The first column that is labeled FI reports the number of arguments found to be constant by the flow-insensitive interprocedural method. The first column that is labeled FS reports the number of arguments found to be constant by the flow-sensitive interprocedural method. The adjacent columns report the results as a percentage of the number of arguments. The last row records the totals for all of the benchmarks.

The flow-insensitive method only finds 2 additional constant arguments in 1 benchmark. Any increase from the flow-insensitive method results from the pass-through of constant immediate values, few of which are expected. The flow-sensitive method finds 168 additional constant arguments in 6 benchmarks, which are 2.9% of the total number of arguments, and 24% more than the flow-insensitive method.

In Table 1, the second column that is labeled FI lists the number of global constants that are initialized in a block data subprogram, and represents the number of candidates considered for propagation by the flow-insensitive algorithm. The second column that is labeled FS reports the number of global constants that reach the call site and are referenced in the called procedure (directly or indirectly), using the flow-sensitive method. The column labeled Vis reports the number of global constants that are visible in the calling procedure, using the flow-sensitive method.

Table 2 presents a second metric, the number of interprocedurally propagated constants that are found to hold at the entry of a procedure.[2] The column labeled Tot reports the number of formal parameters in each benchmark. The first column that is labeled FI reports the number of formal parameters found to be constant by the flow-insensitive interprocedural method. The first column that is labeled FS reports the number of formal parameters found to be constant by the flow-sensitive

[2] This table corrects an earlier version of this paper [11].

Interprocedural Propagated Constants								
	Formal Parameters					Globals		
program	Tot	FI	Pct	FS	Pct	Procs	FI	FS
013.spice2g6	307	4	1%	4	1%	120	0	45
015.doduc	133	2	2%	2	2%	41	0	1
030.matrix300	32	2	6%	15	47%	5	0	0
034.mdljdp2	40	3	8%	3	8%	36	38	40
039.wave5	258	5	2%	9	3%	79	0	61
048.ora	0	-		-		3	18	23
077.mdljsp2	40	3	8%	3	8%	35	0	0
078.swm256	0	-		-		8	0	0
089.su2cor	57	4	7%	4	7%	25	0	0
090.hydro2d	42	7	17%	7	17%	40	0	0
093.nasa7	64	15	23%	22	34%	23	0	0
094.fpppp	70	4	6%	7	10%	13	0	2
Total	1043	49	4.7%	76	7.3%	428	56	172

Table 2: Interprocedural Propagated Constants

interprocedural method. The adjacent columns report the results as a percentage of the number of formal parameters. The last row records the totals for all of the benchmarks.

In order for a formal to be detected as constant, all corresponding arguments must have the same constant value. The flow-insensitive method finds 49 constant formal parameters in 10 benchmarks, which are 4.7% of the total. The flow-sensitive method finds 76 constant formal parameters, which are 7.3% of the total. The flow-sensitive method finds 27 additional constant formal parameters, in 4 benchmarks, which are 2.6% of the total number of formal parameters, and 55% more than the flow-insensitive method.

Table 2 also reports the number of global constants interprocedurally propagated by the flow-insensitive and flow-sensitive methods. The column labeled Procs lists the number of procedures in each program that are reachable from the main procedure, including the main procedure. The second column that is labeled FI reports the total number of global constants, found with the flow-insensitive analysis, that occur at the beginning of a procedure and are referenced directly in that procedure. This amounts to a count of the number of constants that are initialized in a block data subprogram and not defined elsewhere in the program. The second column that is labeled FS reports the total number of global constants, found with the flow-sensitive analysis, that occur at the beginning of a procedure and are referenced directly in that procedure.

The flow-insensitive method finds 56 global constants in these benchmarks, which is greater than the number of constant formal parameters that it discovers (49). The flow-sensitive method finds 172 global constants in the same benchmarks, which is more than twice the number of constant formal parameters that it discovers. The flow-sensitive method finds more than three times the number of global constants than the flow-insensitive method.

Our metrics are based on the count of the number of interprocedural constants that are propagated to and used by each procedure, rather than counting the total number of constant substitutions. These metrics eliminate possible distortion by counting each interprocedural constant that is propagated to a procedure and referenced, only once, regardless of the number of references within a procedure. Substitutions that arise purely from intraprocedural propagation are not counted. An advantage of this metric is that it is independent of any intraprocedural constant propagation method. We wish to emphasize that our flow-sensitive interprocedural constant propagation method can use any flow-sensitive intraprocedural constant propagation method. The number of constants that are propagated by our flow-sensitive method is dependent upon the intraprocedural method used.

Call Site Integer Constant Candidates										
	Arguments							Globals		
program	Tot	Imm	Pct	FI	Pct	FS	Pct	FI	FS	Vis
013.spice2g6	2069	384	19%	384	19%	426	21%	0	452	251
015.doduc	93	39	42%	39	42%	39	42%	0	0	0
030.matrix300	122	25	20%	25	20%	110	90%	0	0	0
034.mdljdp2	172	11	6%	11	6%	11	6%	5	0	0
039.wave5	228	30	13%	32	14%	45	20%	29	78	78
048.ora	0	-		-		-		0	4	4
077.mdljsp2	172	11	6%	11	6%	11	6%	0	0	0
078.swm256	0	-		-		-		0	0	0
089.su2cor	291	110	38%	110	38%	110	38%	0	0	0
090.hydro2d	62	28	45%	28	45%	28	45%	0	1	1
093.nasa7	61	33	54%	33	54%	45	74%	0	3	3
094.fpppp	70	17	24%	17	24%	21	30%	0	8	4
Total	3340	688	20.6%	690	20.7%	846	25.3%	34	546	341

Table 3: Interprocedural Call Site Integer Constant Candidates

Our implementation optionally propagates floating point constants. We have found that the propagation of floating point constants does provide a measurable improvement in the performance of certain programs, including some of the SPEC benchmarks. The results previously reported in this section include the propagation of floating point as well as integer constants.

Tables 3 and 4 report the integer only results from Tables 1 and 2 respectively. ¿From Table 3, it is apparent that almost all of the constant arguments are integers. The flow-sensitive method finds that more than 25% of the integer arguments are constants. For the flow-insensitive method, the number of integer global candidates is 32% of the total, while for the flow-sensitive method, the number of integer global candidates is 63% of the total. In Table 4, the number of constant formal parameters found by each method is the same as in Table 2. No integer global constants are found by the flow-insensitive method. The flow-sensitive method discovers 67 integer global constants. The propagation of only integer constants mainly causes a reduction in the number of global constants that are propagated. Even without the propagation of floating point constants, the flow-sensitive method still discovers approximately the same number of global constants as formal parameter constants.

Tables 5 and 6 report the floating point only results from Tables 1 and 2 respectively. ¿From Table 5, it is apparent that there are almost no constant arguments that are floating point variables. The flow-insensitive method finds none, and the flow-sensitive method finds just 12 floating point constant arguments, which is .5% of the total number of floating point arguments. Both methods find a significant number of global constant floating point candidates. The flow-insensitive method finds 72 global constant floating point candidates, which is more than twice the number of global constant integer candidates reported in Table 3. Table 6 shows that no floating point constant formal parameters are found by either method. The flow-insensitive method finds 56 global floating point constants. All of the global constants found by the flow-insensitive method are floating point constants. 105 of the 172 global constants discovered by the flow-sensitive method are floating point constants.

We have also measured the compilation time for both of our interprocedural constant propagation methods. The flow-sensitive method increases the analysis phase of the compilation by 50% over the flow-insensitive method. This result is consistent over all of the benchmarks. Since the analysis phase contributes only a small fraction of the overall compilation time, the increase in the overall compilation time is typically small ($< 1\%$).

Interprocedural Propagated Integer Constants								
	Formal Parameters					Globals		
program	Tot	FI	Pct	FS	Pct	Procs	FI	FS
013.spice2g6	117	4	3%	4	3%	120	0	31
015.doduc	30	2	7%	2	7%	41	0	0
030.matrix300	21	2	10%	15	71%	5	0	0
034.mdljdp2	37	3	8%	3	8%	36	0	2
039.wave5	103	5	5%	9	9%	79	0	28
048.ora	0	-		-		3	0	4
077.mdljsp2	37	3	8%	3	8%	35	0	0
078.swm256	0	-		-		8	0	0
089.su2cor	28	4	14%	4	14%	25	0	0
090.hydro2d	16	7	44%	7	44%	40	0	0
093.nasa7	35	15	43%	22	63%	23	0	0
094.fpppp	46	4	9%	7	15%	13	0	2
Total	470	49	10.4%	76	16.2	428	0	67

Table 4: Interprocedural Propagated Integer Constants

Call Site Floating Point Constant Candidates										
	Arguments							Globals		
program	Tot	Imm	Pct	FI	Pct	FS	Pct	FI	FS	Vis
013.spice2g6	914	0	0%	0	0%	4	.4%	0	81	51
015.doduc	390	0	0%	0	0%	4	1%	0	1	1
030.matrix300	56	0	0%	0	0%	0	0%	0	0	0
034.mdljdp2	23	0	0%	0	0%	0	0%	11	69	38
039.wave5	448	0	0%	0	0%	4	.9%	45	171	153
048.ora	0	-		-		-		16	3	3
077.mdljsp2	23	0	0%	0	0%	0	0%	0	0	0
078.swm256	0	-		-		-		0	0	0
089.su2cor	353	0	0%	0	0%	0	0%	0	0	0
090.hydro2d	135	0	0%	0	0%	0	0%	0	0	0
093.nasa7	43	0	0%	0	0%	0	0%	0	0	0
094.fpppp	33	0	0%	0	0%	0	0%	0	0	0
Total	2418	0	0%	0	0%	12	.5%	72	325	246

Table 5: Interprocedural Call Site Floating Point Constant Candidates

Interprocedural Propagated Floating Point Constants								
	Formal Parameters					Globals		
program	Tot	FI	Pct	FS	Pct	Procs	FI	FS
013.spice2g6	190	0	0%	0	0%	120	0	14
015.doduc	103	0	0%	0	0%	41	0	1
030.matrix300	11	0	0%	0	0%	5	0	0
034.mdljdp2	3	0	0%	0	0%	36	38	38
039.wave5	155	0	0%	0	0%	79	0	33
048.ora	0	-		-		3	18	19
077.mdljsp2	3	0	0%	0	0%	35	0	0
078.swm256	0	-		-		8	0	0
089.su2cor	29	0	0%	0	0%	25	0	0
090.hydro2d	26	0	0%	0	0%	40	0	0
093.nasa7	29	0	0%	0	0%	23	0	0
094.fpppp	24	0	0%	0	0%	13	0	0
Total	573	0	0%	0	0%	428	56	105

Table 6: Interprocedural Propagated Floating Point Constants

Call Site Integer Constant Candidates										
	Arguments							Globals		
program	Tot	Imm	Pct	FI	Pct	FS	Pct	FI	FS	Vis
015.doduc	93	39	42%	39	42%	39	42%	0	0	0
020.nasa7	55	33	60%	33	60%	42	76%	0	0	0
030.matrix300	122	25	20%	25	20%	110	90%	0	0	0
042.fpppp	70	17	24%	17	24%	21	30%	0	8	4
Total	340	114	33.5%	114	33.5%	212	62.4%	0	8	4

Table 7: Interprocedural Call Site Integer Constant Candidates (SPEC89)

5 Discussion

The most closely related work is that of Grove and Torczon [18]. They use the number of intraprocedural substitutions as the metric for measuring the effectiveness of the interprocedural constant propagation methods. This method was originally suggested by Metzger and Stroud [22] and is useful for assessing the overall effect of constant propagation. However, this metric makes it difficult to distinguish interprocedural and intraprocedural effects. Since we are more interested in the interprocedural effects, we suggest that a more appropriate measure of the effectiveness of interprocedural constant propagation can be obtained by measuring the number of constant formal parameters and the number of constant globals that are propagated to each procedure. These are the constant values that are propagated interprocedurally.

Table 7 and Table 8 present our results for those first release SPEC89 benchmarks that were reported by Grove and Torczon. In order to provide a better comparison, these results present the integer only measurements and do not include the propagation of floating point constants. We only include measurements for procedures that are reachable from the main procedure, whereas the results presented in [18] include measurements for all procedures [17].

Table 8 indicates that the flow-insensitive method does not find any global constants in these benchmarks. The flow-sensitive method only finds two global constants in 1 benchmark. Most global variables are not constant at each call site, and thus, few are considered as candidates for propagation.

Interprocedural Propagated Integer Constants								
	Formal Parameters					Globals		
program	Tot	FI	Pct	FS	Pct	Procs	FI	FS
015.doduc	30	2	7%	2	7%	41	0	0
020.nasa7	29	15	52%	19	66%	17	0	0
030.matrix300	21	2	10%	15	71%	5	0	0
042.fpppp	46	4	9%	7	15%	13	0	2
Total	126	23	18.3%	43	34.1%	76	0	2

Table 8: Interprocedural Propagated Integer Constants (SPEC89)

Intraprocedural Substitutions			
program	Polynomial	FI	FS
015.doduc	287	288	288
020.nasa7	336	205	344
030.matrix300	138	14	250
042.fpppp	56	25	79
Total	817	532	961

Table 9: Comparison of Intraprocedural Substitutions

Our method for selecting and propagating global constants creates a sparse list of global variable constant candidates at each call site. The implementation of Grove and Torczon builds a jump function for every global variable at each procedure [17].

We have also measured the number of substitutions performed by the intraprocedural method for these benchmarks. Since we do not propagate returned constants, we have used the "*No Return Jump Function*" results of Grove and Torczon in the column labeled Polynomial of Figure 9. For these benchmarks, there was no difference between the *polynomial* and *pass-through* results. Both the polynomial and pass-through jump functions use the flow-sensitive "*Intraprocedural Constant Jump Function*" to identify additional constants before the interprocedural solution is computed. However, both the polynomial and pass-through jump functions may not identify some constants, as illustrated in Figure 1. The results of our flow-insensitive method are clearly inferior to the *no return polynomial jump function* results. This is not surprising since the flow-insensitive method only considers literal constants as candidates, combined with the pass-through effect.

On the other hand, our flow-sensitive method discovers 17.6% more constants than the polynomial or pass-through methods in these benchmarks. We achieve the same number of constant substitutions for both the flow-insensitive and flow-sensitive interprocedural constant propagation methods on the 015.doduc benchmark. This is also the benchmark where our results are about the same as the polynomial jump function result. The flow-sensitive method obtains significantly better results than the flow-insensitive method on the remaining three benchmarks. The flow-sensitive method also obtains better results than the polynomial jump function on these same benchmarks.

Two features distinguish our work from the work of Grove and Torczon. The first is our handling of recursive calls; we use flow-insensitive results for back edges. The second is our interleaving of the flow-sensitive intraprocedural analysis and the interprocedural propagation. Grove and Torczon also perform a flow-sensitive intraprocedural analysis of each procedure to compute a forward jump function and propagate the results of this analysis interprocedurally. However, the interprocedural propagation does not occur in their method until after all of the jump functions have been computed. Although the work performed by both methods is equivalent, the increased precision of our results arises from this interleaving. The interleaving can be considered as a method for using the updated (or most recently computed) information as soon as it is available.

6 Related Work

Wegman and Zadeck [26] present several intraprocedural constant propagation techniques, one of which (SCC) we employ intraprocedurally. They describe how to extend their algorithms interprocedurally, by using procedure integration to increase the effects of constants that are propagated. This extension would capture the effect of returned constants, but may not be efficient, in practice. No experimental results are presented.

Callahan, Cooper, Kennedy, and Torczon [10] describe a method based on *jump functions*, which summarize the value of an argument based on the formal parameters of the routine containing the call site. They describe several possible jump functions, with a precision/cost tradeoff. All but the literal jump function include an intraprocedural component that is flow-sensitive. After the jump functions are computed, an interprocedural propagation step is performed to propagate constants to each routine. Grove and Torczon [18] present experimental numbers for various jump functions based on the number of constants that are substituted intraprocedurally. It is not clear how globals can be efficiently handled in this framework. The creation of a jump function for each global variable for each call site can add substantial overhead. Their method does not handle call graph cycles [17], but does propagate returned constants.

Burke and Cytron [9] discuss interprocedural constant propagation in high-level terms in the context of data dependence analysis. They suggest a pessimistic flow-sensitive interprocedural algorithm that can be modeled as the solution of an incremental intraprocedural problem. Their algorithm does not support recursion but does support static call graph cycles, and the propagation of returned constants. No experimental results are given.

Allen, Burke, Charles, Cytron, and Ferrante [1] describe a flow-insensitive interprocedural algorithm for constant propagation of formal parameters, based on the bindings of formal parameters. It detects immediate argument constants and the transitive effect of passing their corresponding parameters as arguments. They do not handle returned constants and their program model does not include programs that are recursive. No experimental results are given.

Metzger and Stroud [22] implemented a form of the Callahan, Cooper, Kennedy, and Torczon interprocedural constant propagation algorithm to help aid procedure cloning. Their results suggest that goal-directed procedure cloning based on constant propagation can substantially increase the number of interprocedural constants.

Binkley [5] suggests an interprocedural constant propagation algorithm based on the *system dependence graph*, a variation of the *program dependence graph* [16, 20]. He utilizes symbolic execution and live-code analysis, but does not report experimental results.

Sagiv, Reps, and Horwitz [23, 24] describe a method for computing the *copy constant propagation* and *linear constant propagation* subsets of the interprocedural constant propagation problem in polynomial time. Their method handles recursion and returned constants, and is context sensitive, in that it computes a meet over all valid paths. Results are given for the C subset of the SPEC92 benchmarks.

7 Conclusions

This paper describes a flow-sensitive algorithm for performing interprocedural constant propagation. The algorithm supports recursion, while only performing one flow-sensitive analysis of each routine. Experimental results are presented which show that this method finds more constants than previous methods and is efficient in practice. We show how the flow-sensitive algorithm can be extended to propagate returned constants. New metrics are introduced for evaluating interprocedural constant propagation algorithms which are based on the constant values that are interprocedurally propagated. The number of call site candidates and interprocedural constants provide better indications of the success achieved by *ICP*. We have used these metrics to provide both integer and floating point measurements of the SPEC benchmarks for our algorithms.

8 Acknowledgements

We are grateful to Fran Allen for insisting that we examine this problem, and for her support. We acknowledge useful discussions with Fran Allen, Michael Burke, Jong-Deok Choi, Rajeev Gulati and Yong-fong Lee during the initiation of this effort.

Our implementation was part of the interprocedural analysis phase of a Fortran 90 research compiler prototype. The prototype was the joint effort of colleagues at the IBM Santa Teresa Development Laboratory, the IBM Canada Toronto Programming Laboratory, and the IBM Thomas J. Watson Research Center. Many of the ideas in the interprocedural analysis phase of the prototype were based on earlier results of the PTRAN group at Watson. In particular, our algorithm was based on an algorithm for computing upward exposed USE, first conceived by Michael Burke.

We thank Yong-fong Lee for his assistance with the implementation of floating point support. We thank Dan Grove for providing additional details of his experiment. We acknowledge useful discussions of related work with Tom Marlowe. We are grateful to Michael Burke, John Field, Dan Grove, Roger Hoover, Yong-fong Lee, Ganesan Ramalingam, Barbara Simons, Harini Srinivasan, Laureen Treacy, and Ken Zadeck for suggested improvements to an earlier version of this paper.

References

[1] Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante. An overview of the ptran analysis system for multiprocessing. *Journal of Parallel and Distributed Computing*, 5(5):617–640, 1988.

[2] Frances E. Allen. Interprocedural data flow analysis. In *IFIP Congress*, pages 398–402. North Holland, 1974.

[3] Subra Balan and Walter Bays editors. Spec announces new benchmark suites cint92 and cfp92. Technical report, Systems Performance Evualation Cooperative, March 1992. *SPEC Newsletter* 4(1).

[4] John Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *6th Annual ACM Symposium on the Principles of Programming Languages*, pages 29–41, January 1979.

[5] David Binkley. Interprocedural constant propagation using dependence graphs and a data-flow model. In *International Conference on Compiler Construction*, 1994.

[6] Michael Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Transactions on Programming Languages and Systems*, 12(3):341–395, July 1990.

[7] Michael Burke and Paul Carini. Compile-time measurements of interprocedural data-sharing in fortran programs. RC 17389 76684, IBM – T.J. Watson Research Center, November 1991.

[8] Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Lecture Notes in Computer Science, 892*, pages 234–250. Springer-Verlag,

1995. Proceedings from the *7th International Workshop on Languages and Compilers for Parallel Computing.* Extended version published as Research Report RC 19546, IBM T. J. Watson Research Center, September, 1994.

[9] Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. In *SIGPLAN '86 Symposium on Compiler Construction*, pages 162–175. ACM, July 1986. *SIGPLAN Notices, 21(7).*

[10] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. In *SIGPLAN '86 Symposium on Compiler Construction*, pages 152–161, July 1986.

[11] Paul R. Carini and Michael Hind. Flow-sensitive interprocedural constant propagation. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 23–31, 1995. *SIGPLAN Notices, 30(6).*

[12] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *20th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 232–245, January 1993.

[13] Keith Cooper and Ken Kennedy. Efficient computation of flow insensitive interprocedural summary information. In *SIGPLAN '84 Symposium on Compiler Construction*, volume 19, pages 247–258, June 1984. *SIGPLAN Notices* 19(6).

[14] Keith Cooper and Ken Kennedy. Fast interprocedural alias analysis. In *16th Annual ACM Symposium on the Principles of Programming Languages*, 1989.

[15] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, pages 451–490, October 1991.

[16] Jeanne Ferrante, Karl J. Ottenstein, and Joe Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[17] Dan Grove. Private Communication, October 1994.

[18] Dan Grove and Linda Torczon. Interprocedural constant propagation: A study of jump function implementations. In *SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 90–99, June 1993. *SIGPLAN Notices* 28(6).

[19] Michael Hind, Michael Burke, Paul Carini, and Sam Midkiff. An emperical study of precise interprocedural array analysis. *Scientific Programming*, 3(3):255–271, 1994.

[20] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and Michael Wolfe. Dependence graphs and compiler optimizations. In *8th Annual ACM Symposium on the Principles of Programming Languages*, pages 207–218, 1981.

[21] Thomas J. Marlowe, Barbara G. Ryder, and Michael G. Burke. Defining flow sensitivity in data flow problems. Technical Report RC 20138 (89049), IBM Thomas J. Watson Research Center, July 1995.

[22] R. Metzger and S. Stroud. Interprocedural constant propagation: An empirical study. *ACM Letters on Programming Languages and Systems*, 2(1–4), 1993.

[23] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *TAPSOFT'95*, 1995.

[24] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. Technical Report TR-1284, Computer Sciences Department, University of Wisconsin, Madison, WI, August 1995.

[25] Joseph Uniejewski. Spec benchmark suite: Designed for today's advanced systems. Technical report, Systems Performance Evualation Cooperative, Fall 1989. *SPEC Newsletter* 1(1).

[26] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, 1991.