# Demand-Driven Interprocedural Constant Propagation: Implementation and Evaluation

**Tito Autrey**

Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
P. O. Box 91000, Portland, OR 97291-1000 USA

*tito@cse.ogi.edu*

June 1, 1995

## Abstract

We have developed a hybrid algorithm for interprocedural constant propagation combining two prior methods with a new demand-driven approach. We modified a prior intraprocedural constant propagator to use incrementally in a demand-driven interprocedural framework. We compare our algorithm to three prior interprocedural methods.

Burke and Cytron solve the interprocedural constant propagation problem with an algorithm that uses a pessimistic incremental intraprocedural constant propagator to iterate forward and backward over the call graph until no new information is discovered [BC86].

Wegman and Zadeck solve the intraprocedural constant propagation problem with an optimistic algorithm [WZ91]. Their algorithm solves the sparse conditional constant problem. The interprocedural version of their algorithm links the Static Single Assignment graphs of all procedures together and runs their intraprocedural algorithm over the single SSA graph.

Grove and Torczon performed experiments that show Jump Functions combined with procedure summary blocks as described by Callahan *et. al* are effective at finding interprocedural constants [CCKT86, GT93].

We show that our interprocedural algorithm, Demand-driven With Incremental Modification (DWIM), is fast and finds the same number of constants as the best interprocedural constant propagator in use today. We know of no implementations of either interprocedural Wegman-Zadeck or Burke-Cytron algorithms. Over a set of standard benchmarks we find 46.4% more interprocedural constants compared to intraprocedural constants.

## 1 Introduction

*Constant propagation* is a compiler optimization that identifies which uses of variables in a procedure have a constant value and what that value is. This is done by taking initial values which are literal constants and propagating them through the computation of the procedure. *Constant folding* is the optimization of computing the results of builtin operations (those without side-effects) on constants at compile-time. Folding uses propagation results as input and propagation uses folding results as input, so they can be performed concurrently.

Intraprocedural constant propagation has two limitations. First, at call sites, due to the possibility of side-effects, the values of global variables as well as reference and pointer parameters may change. Second, on entry into a procedure the values of formal parameters and global variables are also unknown. Interprocedural analysis and constant propagation seek to solve these problems. Computation of interprocedural side-effect information of a procedure, called *MOD* sets, tells which actual arguments and which globals may be modified; the rest remain unchanged across the call site. Interprocedural constant propagation identifies formals and globals with constant values on entry to a procedure. It may also find that the side effect on them is a function of formal parameters, global variables and constants that can be evaluated at compile-time.

We develop a new algorithm, Demand-driven With Incremental Modification (DWIM) for performing interprocedural constant propagation, describe its implementation and evaluate it over a set of benchmark programs. We will proceed to describe: constant propagation - Section 2; interprocedural analysis in support of constant propagation - Section 3; interprocedural constant propagation in general and DWIM in particular - Section 4; our evaluation experiments and results - Section 5; related work - Section 6; future work - Section 7, and we summarize in Section 8.

## 2 Constant Propagation

It is well-known that the general constant propagation problem is undecidable [KU77]. Solutions to the restricted decidable sub-problem still yield useful results. Constant propagation has several direct effects. It may mark edges as non-executable in the control flow graph (CFG) by compile-time (static binding time) predicate evaluation. Dead code elimination removes non-executable edges and code from the CFG of a procedure[1]. Dead code elimination uses propagation results as input and propagation uses the fact that nodes in the CFG have been eliminated as input, so they can be performed concurrently. Induction variable analysis uses propagation results as input, and folding and propagation use induction expressions as input, so they can all be performed concurrently. Most constants are small integers [HP90]. This allows for a more compact encoding of a program by placing constants in an instruction immediate field rather than loading them explicitly from memory. Constant propagation makes other analyses simpler and more exact which leads to useful indirect effects. For example, constant

[1]There are two forms of dead code. This form of dead code is called unreachable code. The other form is code whose results are not used, called unused code [All72].

loop limits enhance dependence analysis, enhance parallelization heuristics [EB91], and may remove run-time range checks [Kol94]. It may also make subscript expressions linear which speeds analysis and can enable loop transformations [SLY90]. It may make array section analysis simpler [MS93].

Constant propagation is generally only performed on integer and logical values. This is because they produce the biggest payoffs in terms of loop bounds, array subscripts and predicate values. Floating point (FP) constants are generally ignored because representations may be different between the architecture the compiler is running on and the one it is targeted to, and the IEEE rounding mode may be dynamic. The Convex Application Compiler does perform FP constant propagation by assuming the compiler and target architectures are the same and tracking when the rounding mode is not known to be important [MS93]. Tracking constant array element values is a fruitful area of current research. Other datatypes, such as structures, bit and string constants could all be done, but folklore holds that they do not occur frequently enough to be worth the effort.

Constant propagation is computed on a data-flow graph representation of a procedure, frequently classic use-def and def-use chains [ASU86]. A lattice is used to cast constant propagation as a global data-flow problem. Each variable use or definition is represented by a value from Kildall's three-level constant propagation lattice [Kil73]. Top, $\top$, represents an as yet unknown constant value. Bottom, $\bot$, represents a non-constant value. The constant values constitute the middle layer of the lattice. The lattice *meet* operator, $\sqcap$, is defined in the usual way.

An optimistic propagator initially assumes all uses are an as yet undiscovered constant, represented by $\top$. It then iterates over the data-flow graph to determine LHS lattice values by evaluating RHS expressions and lattice values which are propagated along the graph edges. Each use can be lowered at most twice, once to a constant, and once more to $\bot$.

A pessimistic propagator initially assumes all uses are non-constant, represented by $\bot$. It then iterates over the data-flow graph the same as for an optimistic propagator. Each use can be raised at most once, to a constant value.

The advantage of an optimistic algorithm is that it can find constants in the presence of loops in the data-flow graph. A pessimistic algorithm cannot because the initial $\bot$ will flow around the loop, consequently the meet is always $\bot$. The disadvantage of an optimistic algorithm is that it must always be run to completion; otherwise a $\bot$ lattice value may fail to be propagated, leading to an incorrect result. A pessimistic algorithm can terminate at any point leaving a correct, but conservative result, as it will only mark uses as constant that have been proved as such.

Modern approaches to constant propagation are performed on sparse graphs. One of the most common sparse graphs is the Static Single Assignment (SSA) form [CFR$^+$89]. The Wegman-Zadeck Sparse Conditional Constant propagator (WZ-SCC) is a worklist algorithm [WZ91]. It has two lists, one of variable uses and the other of edges in the CFG. All edges from executable basic blocks are added to the edge list unless they come from a predicate with a known constant value, in which case only the possible flow edge is added. Uses are added when their reaching def is lowered.

A demand-driven[2] constant propagator is a backward algorithm. Starting anywhere in the data-flow graph, each variable definition demands the lattice values used to compute it and takes the meet of their values. Each use obtains its value from the meet of the reaching defs. In SSA graphs there is only one reaching def, except at $\phi$ nodes. Stoltz has shown that demand-driven constant propagation solvers with *Gated Single Assignment* are faster than WZ-SCC [SW94].

## 3 Interprocedural Analysis

Various interprocedural analyses are needed to enable interprocedural constant propagation. We implemented the following four analyses to support our study of interprocedural constant propagation: Call Graph Construction, Alias Analysis, MOD and REF Analysis, and Jump Function identification. The analyses store some of their results in a procedure summary block which is referenced by later analysis and optimization stages [CCKT86]. A brief description of the algorithms we use and comments on their implementations are given below.

```
 1:    ∀ Procedures
 2:        Parse
 3:        Collect call site information

 4:    Call graph construction
 5:    Alias analysis
 6:    MOD and REF analysis
 7:    Apply MOD and REF information to IL

 8:    ∀ Procedures
 9:        Add SSA links
10:        Add def-use links
11:        Induction variable analysis
12:        Sparse conditional constant propagation (WZ-SCC)

13:    Demand-driven walk of call-graph for interprocedural
14:    constant propagation using incremental propagator.
```

Figure 1: Compiler Phase Order

The original design of Nascent[3] was strictly as an intraprocedural compiler; it dis-

[2]Demand-driven in this case has nothing to do with lazy evaluation. Constant propagation is inherently eager. It refers to demanding the solution to a given data-flow problem at predecessor nodes before computing the current node.

[3]Nascent is the name of the Fortran compiler being developed by the Sparse group at OGI under the supervision of Michael Wolfe.

carded the body of each procedure before it started the next one. A number of enhancements were made to support interprocedural analysis and optimization. First we changed it to preserve the body of all procedures. This involved adapting the memory management layer to generate a new private heap space for each procedure. Now by reloading the heap context for a procedure the compiler can revisit it as many times as desired. Nascent has been restructured so that lists of intraprocedural phases are alternated with lists of interprocedural phases. Any particular list may be empty. For interprocedural constant propagation the phase order goes as in Figure 1. Because Nascent did not use any interprocedural information, new types of intermediate language constructs were added to convey MOD and REF information about actual arguments and global variables. Each actual argument at each call site is now identified as an in, out, or in/out parameter depending on whether it is used or/and modified directly or indirectly by the called procedure. Global variables are also attached to each call site now because they are implicit arguments. They are identified similarly to actual arguments. This enhances the completeness of the information available at the intraprocedural level.

## 3.1 Call Graph Construction

First we must construct the call multi-graph. It is a true multi-graph because a procedure may call another one multiple times. For simplicity we will refer to it as the call graph. In Fortran, the construction would be straight-forward except for the presence of procedure formals. Procedure formals are formal parameters that are bound to actual arguments which are procedure constants or procedure formals, as opposed to procedure variables which can be assigned procedure values at any point in the program. We implemented the algorithm described by Hall and Kennedy [HK92] which is complete and fast for languages with procedure formals. It finds the maximal $Boundto(pf) = \{c : c \text{ is procedure constant bound to } pf \text{ along some possible execution path}\}$ set for all procedure formals. The algorithm initializes a worklist with bindings from procedure constants, $pc$, to formal parameters, $fp$. When an element $pc \rightarrow fp$ is removed from the list, if $pc \notin Boundto(fp)$ then $pc$ is added to $Boundto(fp)$. For each call site $cs$, three cases are considered:

1) $fp$ is not an actual parameter:
   do nothing
2) $cs$ does not invoke $fp$ but $fp$ is an actual parameter:
   $\forall x \in Boundto(fp)$ add elements to the worklist binding $x$ to the formal(s) corresponding to $fp$
3) $cs$ does invoke $fp$ and $fp$ is an actual parameter:
   $\forall x \in Boundto(fp)$ add elements to the worklist binding $x$ to the formal in $x$ corresponding to $fp$

The last case is an optimization that prevents spurious bindings from being generated. This algorithm extends readily to handle procedure variables which are found in imperative languages such as C. The psuedo-language used in the examples uses the call-by-reference parameter passing mechanism.

```
proc A                        proc C (P3)
   call B (C, 2, D)        @     call P3 (P3)
   call B (C, 3, E)           end C
end A                         proc D (P4)
proc B (P1, L, P2)            end D
   if L = 42 then             proc E (P5)
*    P1 = F                   end E
   endif                      proc F (P6)
   call P1 (P2)               end F
end B
```

Figure 2: Call Graph Construction Example

In Figure 2, the line marked with a '*' is not valid in Fortran, but it is valid in C. P1 is a procedure variable, and has a new value assigned to it directly rather than as a consequence of a procedure call. Allowing arbitrary altering of the *BoundTo* sets means that more complicated analysis is required. Discounting the '*"ed line, the algorithm will identify that A calls B, B calls C, and C calls D and E. Without call graph analysis the optimizer would have to assume that B and C could call any other procedure. The statement marked with an '@' triggers case 3 in the algorithm. P3 is bound to D and E, but when D is called, D is the only value that can be passed on to P4, and the same for E with respect to P5.

## 3.2 Alias Analysis

Aliases occur when one storage location can be accessed by more than one name. They make it hard for compilers to detect when a particular storage location is modified. There are two types of aliases for formal arguments; Type I are formal-global aliases which are created by passing globals and formals as actuals, and Type II are formal-formal aliases which are created by passing the same local, formal or global in more than one position at a call site. Much of the original work we draw upon was performed by Keith Cooper at Rice [Coo85, CK88, CK86]. The analysis uses two data structures which are built on top of the call graph. The algorithm for computing the formal-global binding graph, $\beta$, is initialized by examining all call sites in all procedures. Where a formal is bound to a global it adds a node to $\beta$, and places the global in the *Alias* set of the formal. Where a formal is bound to a formal, it adds one node for each formal and places an edge between them. Procedure-valued formals are ignored because they were processed first in precisely the same fashion in order to construct the complete call graph which is needed this analysis. Type I alias analysis is solved as a data-flow problem on $\beta$. The algorithm for computing the pair binding graph, $\pi$, is initialized by examining all call sites in all procedures. Where two actuals are aliases of one another it places nodes in the graph for the pair of actuals and the pair of formals with an edge between them. Type II alias analysis is solved as a data-flow problem on $\pi$.

Our implementation of *Alias* set computation is demand-driven, using $\beta$ as the data-flow graph. One new optimization we use is to never place self-edges in $\beta$. These occur when a procedure calls itself recursively with the formals used as actuals in the same position. These edges add no information so they may be safely ignored. The $\pi$ graph turns out to be unnecessarily large and complex to compute directly [MW93]. Instead we use a worklist algorithm on elements containing the pair of formals which are aliased as described by Mayer and Wolfe.

```
global G1, G2                 proc J (F3, F4)
proc H                           F3 = 2
   call I (G1, G2)                  Y = G1 + 3
end H                         end J
proc I (F1, F2)
   call J (F1, G1)
   call J (F2, G1)
end I
```

Figure 3: Alias Analysis Example

In Figure 3, Type-I aliases are created at the call to I in procedure H. They are passed on at the calls to J in procedure I. The first call to J in I also creates a Type-II alias because F1 has G1 as an alias. All this information is needed so that in procedure J the compiler does not try to load G1 into a register before doing the store to F3.

## 3.3 MOD and REF Analysis

MOD and REF analysis computes *MOD(p)* and *REF(p)* sets for all procedures $p$ [CK84, CK87, CK88]. *MOD* represents the side effects or externally visible write set of a procedure. *REF* represents the set of referenced formal parameters and global variables. During parsing we set *DMOD (p)* to be all formal or global parameters that are directly modified by the body of procedure *p*. *DREF* is is constructed similarly but with referenced formals and globals. The *IMOD (p)* set is *DMOD (p)* augmented with the *MOD (c)* sets of all procedures *c* called directly or indirectly by *p*. The *IREF (p)* set is constructed similarly. *MOD (p)* is the *IMOD (p)* set augmented by the *Alias (v)* sets of all variables *v* in *IMOD (p)*. The *REF (p)* set is constructed similarly. Our modification of Cooper and Kennedy's algorithm for computing *IMOD, IREF, MOD, REF* sets is to construct them concurrently by solving the data-flow problem on the call graph using the *DMOD, DREF* and *Alias* sets.

In Figure 4, MOD and REF analysis will determine that $MOD(M) = \{G5\}$, and $MOD(L) = \{F6, G5\}$ and $REF(L) = \{F7\}$. The consequences of this information in procedure K with respect to the call to L are: A is not referenced so for live-range

```
global G4, G5                proc L (F5, F6, F7)
proc K                          F6 = F7 + 7
   B = C + G4 + G5              call M ()
   call L (A, G4, C)         end L
   D = C + G4 + G5
end K
proc M
   G5 = 8
end M
```

Figure 4: MOD and REF Analysis Example

analysis it can be ignored at the call to L; G4 is modified so its value must be reloaded *after* the call; C is referenced and not modified so its value can be retained in a register across the call; G5 is also modified so its value must also be reloaded after the call.

### 3.4 Jump Functions

Jump functions[4] are a concise way to express the value of arguments in terms of literals and constant flow through a procedure [CCKT86]. The flow of constants in a procedure $p$ to a call site $s$ for each actual argument and global variable (called global argument for this discussion) $x$, is expressed by a *forward jump function* (FJF) $F_s^x$. The formal and global arguments that are used as input to $F$ are represented by the set $support(F_s^x)$. The FJF summarizes the effect on the actual $x$ of the execution of the procedure up to the point of call site $s$. The flow of constants out of a procedure $p$ for each formal and global argument $y$, is represented by a *return jump function* (RJF) $R_p^y$. The formal and global arguments that are used as input to $R$ are represented by the set $support(R_p^y)$. RJFs may be viewed as a generalization of constant folding applied to the side effects of a procedure. They summarize the effect on the formal and global parameters of executing the whole procedure. The type of a jump function is an element of the constant propagation lattice.

According to Grove and Torczon there are four FJFs of interest [GT93]. The *literal constant FJF* (LCFJF) identifies actual arguments that are literal constants. In Figure 5, formal A in procedure R has value 1 and no other constants are found. Because global variables are lexically variables and not literals, this FJF is unable to identify globals as constant. The *intraprocedural constant FJF* (IPFJF) identifies actual arguments and global variables that are intraprocedural constants. In Figure 5, formals B and C in procedure R are determined by intraprocedural constant propagation to have the values of 5 and 6 respectively, and A is still has the value 1. The *passthru parameter FJF* (PTFJF) identifies actuals that are non-modified formals or are found by IPFJF. Because it uses formal and global parameters as input it can cross multiple edges in the

[4]The term is historical and was originated by John Cocke.

```
proc Q                          proc S (Z)
   N = 5                           call T (Z+Z)
   call R (1, N, N+1)           end S
end Q                           proc T (Y)
proc R (A, B, C)                end T
   call S (A)
   B = C + A + 42
end R
```

Figure 5: Jump Function Example

call graph. In Figure 5, formal Z in procedure S has value 1 because A is unmodified in R and is bound to an intraprocedural constant value in procedure Q. The more general *polynomial parameter FJF* (PNFJF) identifies actuals that are any polynomial function of the procedure's formal and global parameters. In Figure 5, formal Y has value 2 because it is bound to a polynomial function of Z in S which was found to have the value 1.

Similarly there are three RJFs of interest. The *literal constant RJF* (LCRJF) identifies formal and global parameters whose side effect is assignment of a literal constant. The *intraprocedural constant RJF* (IPRJF) identifies formal and global parameters whose side effect is assignment of an intraprocedural constant, similar to the IPFJF. The *polynomial parameter RJF* (PNRJF) identifies formal and global parameters whose side effect is a polynomial function of a subset of the formal and global parameters. In Figure 5, the RJF for formal B in procedure R is a polynomial function of the formal arguments A and C. The notion of a passthru parameter RJF is captured by the argument not appearing in the *MOD* set of the procedure.

In the analysis used by Callahan *et al.*; the jump functions are extracted from the body of the procedure and stored in the procedure summary block so that the text of the program is not required. This is to support separate compilation. Since Nascent keeps the program text around, we do not need to extract the actual Jump Function description. Instead, actual parameters are inspected to determine if they are literal constants, to give the effect of LCFJF. After intraprocedural constant propagation is run, we can tell which actual parameters have constant lattice values. This gives the effect of IPFJF. We implemented a demand-driven solver for the interprocedural constant propagation. The interprocedural aspect gives the capability of the PTFJF, and the demand-driven aspect gives the capability of the PNFJF.

None of the RJFs were implemented for this experiment. The PNRJF is very complex to compute. Because the lattice values are not available until after intraprocedural constant propagation, the LCRJF would require the same complexity as the IPRJF. We discuss this under future work.

## 4 Interprocedural Constant Propagation

Interprocedural constant propagation increases all of the benefits of intraprocedural constant propagation in three ways. First, MOD analysis makes it possible to change the non-killing defs at call sites to simple uses for global variables and actual arguments which are not modified. This allows their values to be propagated past the call site. Second, formal parameters and global variables may be identified as constants at procedure entry time. Third, RJFs may identify modification by a constant value for global variables and actual parameters.

Interprocedural constant propagation also allows a completely new optimization, procedure cloning, where a procedure is specialized for specific constant values of one or more of its formal parameters. If interprocedural analysis finds that there are only a few constant values for a formal parameter then the procedure is a candidate for cloning. A clone could have a simpler CFG, and use of a clone can simplify the call graph. The drawback is that cloning can increase code size, so the compiler must make decisions on the tradeoff between space and time.

### 4.1 Demand-driven With Incremental Modification Algorithm

There are three leading theoretical descriptions of interprocedural constant propagation aside from the obvious brute force approach. Wegman and Zadeck's interprocedural algorithm on the complete program SSA graph solves the interprocedural constant propagation problem in time linear in the sum of the size of the component procedure SSA graphs [WZ91]. However, the use of global worklists is likely to lead to poor locality of reference and consequently poor compile-time performance. Burke and Cytron's interprocedural algorithm is tailored to minimize the working set of memory needed. They describe the use of an incremental intraprocedural propagator, iterating both backward and forward over the call graph [BC86]. On the practical front, Grove and Torczon have demonstrated that procedure summary blocks with jump functions find a very high percentage of the available constants [GT93]. RJFs also capture modification by a constant as a result of a procedure call. This information can be applied on a call site by call site basis which is more powerful than simply merging the results of all call sites together.

We chose to couple the last two observations along with some significant improvements. Burke and Cytron suggested using a pessimistic propagator because it is able to function incrementally. Recall that an optimistic algorithm would have to reinitialize the lattice values to $\top$ before beginning and a pessimistic algorithm can not deal with loops in the CFG. We chose to use an optimistic algorithm, WZ-SCC, for the initial setting of the lattice values and to use a pessimistic algorithm for the incremental propagator. An added feature of WZ-SCC is that it marks unreachable code, so we simplify the CFG during the initial pass. RJFs can easily be incorporated into DWIM.

The order of the compiler phases in Figure 1 is straightforward. The Alias and MOD/REF analyses need the complete call graph. The SSA algorithm needs to know whether a given IL node is a use, def, or non-killing def so the *MOD* sets need to be reflected into the IL. Induction variable analysis is done before constant propagation because constant propagation can look at induction expressions to find additional constants. The demand-driven character of the interprocedural driver is an important aspect of the algorithm. It provides the power of PNFJFs with no more complexity than the IPFJF.

Our incremental algorithm described in Figure 4.1 is a modification of WZ-SCC. We want to retain the full power of an SCC propagator and combine it with the incremental capability of a pessimistic algorithm. We use two worklists, the *SSAlist* for SSA edges, the *AntiFlowlist* for nonexecutable flow edges. Our incremental algorithm takes as parameters a procedure (implicit) and one formal argument. The *SSAlist* is initialized with all uses of this parameter. The *AntiFlowlist* is initially empty. We assume an initial intraprocedural constant propagation has been performed on the procedure. In the incremental analysis, we can only raise non-constant values to constants and mark executable edges as non-executable. New constant defs may only alter expressions whose value is currently $\perp$, so the check at line 22 is for consistency only. The check for an executable incoming edge is to avoid useless computation. Visit-$\Phi$ checks for lattice value changing (lowering) in line 18 to avoid useless computation. In Visit-Expression, for any predicate which may change, the current SSA and CFG information is based on all paths from a predicate being executable, so if a predicate becomes constant then the new information is which edges are **NOT** executable. For nonexecutable edges we must check the destination block to see if any information crossing that edge is merged at a $\phi$-node. If the block has become dead code, i.e., it has no executable incoming edges, then we add all outgoing edges to the *AntiFlowlist*.

An important difference from the optimistic intraprocedural algorithm are: a $\phi$-node that has value $\perp$ must be reexamined as these are the interesting nodes for a pessimistic algorithm; they need not be examined by an optimistic algorithm as the algorithm only lowers nodes in the lattice. However, SSA edges from formal arguments are not special-cased to $\perp$ because the algorithm has precise information about their value and need make no assumptions.

Our algorithm is a Demand-driven With Incremental Modification interprocedural constant propagator (DWIM). This algorithm combines the demand-driven approach which enhances the power of simple Jump Functions with an initial optimistic intraprocedural constant propagator to handle loops in the CFG and a pessimistic incremental propagator. We are very precise about only examining information that may have changed and only propagating information that has definitely changed. Careful modification of the WZ-SCC algorithm, allows us to obtain the benefit of incremental dead code elimination.

## 5 Experiments

We evaluate the effectiveness of demand-driven interprocedural constant propagation by running it on several sets of standard benchmark programs, the RICEPS benchmark

```
Incremental (n)
Initialize SSAlist with all edges from n
While NotEmpty(SSAlist) and NotEmpty(AntiFlowlist)
        Take an item from a list
        If item is from SSAlist and destination is a $\phi$-node then
            call Visit-$\Phi$
        If item is from SSAlist and destination is an expression then
            call Visit-Expression
        If item is from AntiFlowlist then
            mark edge as nonexecutable
            $\forall\phi$-nodes in destination block call Visit-$\Phi$
            If all incoming edges to this block are unexecutable then
                add all outgoing edges to the AntiFlowlist
End Incremental

Visit-$\Phi$ (n)
If at least one incoming edge is executable then
        evaluate $\phi$-node
        If lattice value of $\phi$-node changes then
            add all uses of the $\phi$-node to the SSAlist
End Visit-$\Phi$

Visit-Expression (n)
If at least one incoming edge is executable and the value is $\perp$ then
        evaluate the full expression
        If the lattice value of the expression changes then
            If target is an assignment then
                add all SSA edges to the SSAlist
            If target is a predicate then
                add all outgoing edges NOT determined by the -
                predicate to the AntiFlowlist
End Visit-Expression
```

Figure 6: Incremental Pessimistic Propagator

suite[5], the PERFECT Club suite[6] and the Mendez suite[7]. They are all used for evaluating optimizing Fortran compilers. Tables 1, 2 and 3 give an idea of the size and complexity of the various programs. The *'s indicate programs that have comments counted in their number of lines.

Table 1: PERFECT Club static characterization

| PERFECT Club | lines | procs | blocks | insts | fetches | stores | preds | calls |
|---|---|---|---|---|---|---|---|---|
| adm | 4165 | 97 | 2694 | 70523 | 4503 | 511 | 272 | 388 |
| arc2d | 2747 | 39 | 1638 | 40884 | 4517 | 186 | 51 | 101 |
| bdna | 3793 | 43 | 2093 | 62054 | 3528 | 400 | 171 | 285 |
| dyfesm | 4401 | 78 | 2093 | 26824 | 1708 | 193 | 130 | 168 |
| flo52 | 1850 | 28 | 1468 | 43906 | 2566 | 234 | 108 | 226 |
| main | 2812* | 28 | 1350 | 45839 | 4655 | 720 | 118 | 140 |
| mdg | 1028 | 16 | 458 | 20493 | 703 | 116 | 39 | 90 |
| ocean | 2577 | 36 | 1277 | 87568 | 1508 | 443 | 153 | 380 |
| qcd | 1780 | 35 | 1294 | 23458 | 1307 | 439 | 89 | 150 |
| spec77 | 3399 | 65 | 2890 | 73188 | 3331 | 360 | 121 | 605 |
| spice | 18521* | 128 | 7428 | 338814 | 14372 | 3396 | 1956 | 1889 |
| track | 2192 | 32 | 970 | 24313 | 940 | 227 | 150 | 113 |
| trfd | 418 | 7 | 537 | 7112 | 702 | 141 | 22 | 16 |

Table 2: RICEPS static characterization

| RICEPS | lines | procs | blocks | insts | fetches | stores | preds | calls |
|---|---|---|---|---|---|---|---|---|
| boast | 7212 | 58 | 5810 | 123968 | 9568 | 1317 | 698 | 198 |
| ccm | 18709 | 145 | 5432 | 188100 | 13575 | 2648 | 540 | 648 |
| hydro | 13049 | 36 | 1250 | 65798 | 2465 | 364 | 208 | 307 |
| qcd | 2353* | 34 | 1274 | 24221 | 1319 | 347 | 100 | 256 |
| simple | 1239 | 8 | 521 | 29358 | 1741 | 114 | 26 | 60 |
| sphot | 876 | 7 | 431 | 12192 | 588 | 104 | 91 | 55 |
| track | 3735* | 34 | 959 | 23944 | 931 | 218 | 141 | 108 |
| wanal1 | 1718 | 11 | 2161 | 21958 | 1886 | 118 | 31 | 104 |
| wave | 7520* | 91 | 3371 | 96628 | 6495 | 734 | 452 | 378 |

Recall that constant propagation performs two primary functions for us, it finds constant uses and defs of variables, and it finds predicates which can be evaluated at compile-time. These two functions are only loosely interdependent so we chose to measure both of them. We could count programmer variables which have constant value, but this is not meaningful. For example, a particular variable might have several different constant values within a procedure or it might be constant only over part of a procedure. Specifically, we count all uses and all defs which have a constant lattice value and we count all predicates which can be evaluated at compile-time. The compile-time predicates indicate simplification of the CFG, and we want to know how much.

[5]ccm, linpackd, qcd and track have procedure calls with an incorrect number of arguments. This breaks interprocedural analysis so they are discarded.

[6]bdna, flo52 and track have the same problem.

[7]euler, mhd2d and shear have the same problem.

Table 3: MENDEZ static characterization

| MENDEZ | lines | procs | blocks | insts | fetches | stores | preds | calls |
|---|---|---|---|---|---|---|---|---|
| baro | 930* | 7 | 644 | 9872 | 1030 | 57 | 46 | 32 |
| euler | 1200* | 14 | 738 | 19382 | 965 | 125 | 120 | 91 |
| mhd2d | 927* | 14 | 428 | 11038* | 653 | 164 | 21 | 108 |
| vortex | 710* | 20 | 302 | 7654 | 365 | 27 | 17 | 50 |

We also count the number of call sites in each procedure so we can measure the compile-time simplification of the call graph.

The baseline for comparison is standard intraprocedural constant propagation using WZ-SCC and induction variable analysis. One experiment measures just the use of *MOD* sets to show how many constants are preserved across call sites. Another experiment measures the effectiveness of Jump Functions in addition to *MOD* sets. The JFs measured are the LCFJF and the PNFJF.

Table 4: Totals From Constant Propagation Over the Whole Benchmark Set

| Fetches | | | | Predicates | | | |
|---|---|---|---|---|---|---|---|
| Intra | + *MOD* | + LCFJF | + PNFJF | Intra | +*MOD* | + LCFJF | + PNFJF |
| 817 | 177 | 41 | 161 | 42 | 11 | 9 | 29 |

The results in Table 4 show that we find 21.7% more constant fetches with *MOD* sets over vanilla intraprocedural WZ-SCC. We find an additional 26.7% more constant fetches over WZ-SCC when we add the LCFJF and 46.4% more when we use the PNFJF instead of the LCFJF. We find 26.2% more constant predicates with *MOD* sets over vanilla intraprocedural WZ-SCC. We find an additional 47.6% more constant predicates over WZ-SCC when we add the LCFJF and 116.7% more when we use the PNFJF instead of the LCFJF. Fetches represent eliminated *load* instructions and predicates represent eliminated *branch* instructions plus dead code. See Appendix A for a program by program breakdown of the statistics.

The current implementation of Nascent has some limitations. It does not parse EQUIVALENCE statements, so specific programmer directed aliases are not available to the alias analysis. Nascent does not use *Alias* information intraprocedurally, so defs are not correctly handled in the presence of aliases. Happily, the Fortran standard declares that modifying an aliased variable is not required to be supported. Nascent does not make globals that are implicitly passed through a procedure visible, so useful constant propagation would be very difficult and was not implemented. Also Nascent does not parse DATA statements or BLOCKDATA procedures. These are an important source for constants, especially for global variables, so the numbers presented above are strictly conservative.

The results show only first order effects. We do not perform incremental update of the call graph and hence no incremental update of *Alias, MOD* and *REF* sets, and no iteration over the improvement to interprocedural information aside from the constants themselves is done. If this were done in an environment that supported separate compilation it would increase the amount of interfile dependences. These results are also based on static measurements. Nascent does not have a code generator so it is not yet possible to determine the actual affect on the run-time performance seen by a user.

The results in Table 5 show that we find 21.3% more dead call sites with *MOD* sets over vanilla intraprocedural WZ-SCC. We find an additional 21.7% more dead call sites over WZ-SCC when we add the LCFJF and 31.2% more when we use the PNFJF instead of the LCFJF.

Table 5: Total Dead Call Sites and Basic Blocks for the Whole Benchmark Set

| Dead Call Sites | | | | Dead Basic Blocks | | | |
|---|---|---|---|---|---|---|---|
| Intra | + *MOD* | + LCFJF | + PNFJF | Intra | +*MOD* | + LCFJF | + PNFJF |
| 221 | 47 | 1 | 21 | 756 | 259 | 17 | 30 |

We find 34.3% more dead basic blocks with *MOD* sets over vanilla intraprocedural WZ-SCC. We find an additional 36.5% more dead basic blocks over WZ-SCC when we add the LCFJF and 40.5% more when we use the PNFJF instead of the LCFJF. See Appendix A for a program by program breakdown of the statistics.

## 6 Related Work

We compare our work to some of the original theoretical work in the area. There is a distinct shortage of papers evaluating the effectiveness or cost of performing interprocedural analysis, optimization and in particular constant propagation. Grove and Torczon have evaluated the ParaScope[8] research compiler, and Metzger and Stroud have evaluated the Convex Application Compiler, a commercial compiler [MS93]. We will make comparisons to the Rice group under Ken Kennedy, to Wegman and Zadeck's extension of their intraprocedural SCC algorithm, to Burke and Cytron's work and to partial evaluation techniques.

Callahan *et al.* describe a method for efficient interprocedural constant propagation that easily supports separate compilation, but provide no evaluation of its effectiveness [CCKT86]. They use procedure summary blocks so that they do not need the procedure text available when they perform interprocedural analysis. They expand the idea of forward and return Jump Functions. The FJFs summarize the partial computations from procedure invocation to a given call site and the RJFs summarize the side effects (except I/O) of a procedure. By providing summary information their method minimizes the effects of interprocedural constant propagation on recompilation effort. They build the Jump Function expression trees after parsing but before interprocedural analysis. They use a WZ-SCC constant propagator with a symbolic expression evaluator for the Jump Functions. The expression evaluator handles aliasing. They trade completeness against support for separate compilation and they choose data structures that give them time linear in the size of the call graph with some reasonable assumptions.

Grove and Torczon evaluate the effectiveness of the various types of Jump Functions [GT93]. Their algorithm only handles integer constants and it does not track them in and out of arrays. First, in a bottom-up walk of the call graph, it builds RJFs from an SSA graph, then it destroys the SSA graph. Next, in a top-down walk of the call graph, it builds the FJFs from a new SSA graph. Then the algorithm uses ParaScope's standard interprocedural data-flow solver to perform the actual interprocedural constant propagation. If a formal parameter becomes constant, then the FJFs are evaluated for all call sites. If an actual parameter becomes constant, then the data-flow solver makes another pass over the entire call graph. They use the programs in the SPEC and PERFECT benchmark suites that ParaScope could compile. They noted that PNFJFs found no more constants than the PTFJFs, and complete[9] constant propagation found 0.8% more constants compared to PNFJF, both using *MOD* sets.

Grove and Torczon found 72.8% more constants with full interprocedural constant propagation over vanilla intraprocedural constant propagation. There are several reasons why their

[8]ParaScope is a parallelizing compiler developed by Ken Kennedy's group at Rice.

[9]Applies dead code elimination after each pass over the call graph. For their benchmark set two passes were sufficient.

results differ from ours. Grove and Torczon use as a metric "the number of constants substituted into the program text". This is equivalent to our fetches metric. However, they used a different set of benchmarks, including the SPEC89 FP codes, `doduc, fpppp, linpackd, matrix300, snasa7, simple`. If we count just the benchmarks in common between the two experiments, Grove and Torczon find only 46.6% more constants and we find 2.9% more. Allowing only a single pass over the call graph, Grove and Torczon find only 43.7% more constants. Only one of these programs, `ocean`, has constants in a BLOCKDATA procedure. So it is the only significant benchmark where RJFs affect the number of constants found. If this benchmark is removed from consideration then Grove and Torczon find only 5.1% more constants and we find 1.5% more. The difference can be accounted for by our lack of support for global variables and RJFs.

Burke and Cytron note that interprocedural constant propagation can be modeled as incremental intraprocedural constant propagation [BC86]. They point out that optimistic algorithms, which assume a value is constant until proven otherwise, such as WZ-SCC, are ill-suited to incremental propagation. They suggest using a pessimistic algorithm combined with iteration forward and backward over the call graph, updating the procedure summary information, to achieve the best possible result. To keep the memory requirements down they use only the current node's intraprocedural data-flow graph along with all the procedure summary information. They are willing to trade away efficiency for effectiveness.

Wegman and Zadeck develop the powerful Sparse Conditional Constant algorithm [WZ91]. This algorithm relies on properties of an SSA graph, the lack of which in def-use chains prevented an earlier discovery of the algorithm. The algorithm is linear in the size of the SSA graph. They describe an interprocedural version that connects up the SSA graphs of the procedures by adding edges from call sites to procedure entry points, and edges from the exit points back to the call sites. This gives a complete solution, linear in the sum of the sizes of the SSA graphs of the component procedures. However, this approach fails to capture the results of RJFs. Since RJFs are functions of formal parameters rather than actual parameters they represent effects with respect to one call site rather than across all call sites. The interprocedural algorithm uses one def and one edge worklist for the whole program which is likely to lead to poor locality of reference. Wegman and Zadeck also suggest aggressive procedure inlining to allow for call site specific optimizations. They do note that this can lead to large programs and that it fails in the case of cycles in the call graph if a naive inlining algorithm is used. The envision this approach being more effective in a functional language setting.

Constant propagation is one of the primary tools used in partial evaluation [CD93]. In partial evaluation a program as specialized with respect to some of its inputs and a new one is generate by a partial evaluation function. Given a partition of the binding time of a program's inputs, partial evaluators perform binding time analysis on a program to determine for each expression when it can be evaluated. Partial evaluators are divided into two classes, offline for compile time evaluation and online for run time evaluation. An imperative language compiler is an offline partial evaluator that uses constant propagation and constant folding to partially evaluate a program. Constant propagation algorithms identify statically bound inputs to expressions and fold the results to make statically bound inputs for other expressions. Polyvariant partial evaluators may generate multiple specialized instances of an input function where some of its inputs are constants. In imperative language compilers this is called procedure cloning. A partial evaluator must make the same sort of space versus time tradeoff decisions as a compiler.

## 7 Future Work

There are several tasks to complete to demonstrate the full capability of this algorithm: make all implicit globals explicit in the intermediate representation; implement the polynomial RJF; and, process COMMON blocks, EQUIVALANCE statements, BLOCKDATA procedures, and DATA statements. This will ensure that all sources of constants are utilized and that the capability of propagating them is in place.

This work can be extended in several directions. Among the most interesting are to build a unified solver for constant propagation, induction variable detection and dead code removal. To carry it even further, an incremental version suitable for assisting interprocedural constant propagation can be investigated. In the same vein, a set of incremental algorithms for interprocedural information would enable higher order effects. All of these should be integrated with a procedure cloning and inlining capability.

A completely new area would be to extend constant analysis to the discovery of write-once variables, especially those that are defined by READ statements. A number of programs have constants represented as DATA statements, whereas others get them from a file or the command line at run time. Compile time analysis can support dynamic specialization using run time code generation and run time linking. However, a number of parallel optimizations can benefit from constant information to assist with partitioning data and specializing code with respect to the processor number it is executing on. Run time resolution of this information when it cannot be discovered at compile time is expensive. By delaying some of the code generation until early run time, a lot of interpretation may be avoided, or performed once as a compilation. Programs that get their input data from files can receive the same benefits of constant propagation as those which have their constants inline.

## 8 Summary

We have described the implementation of several interprocedural analyses in Nascent: complete call graph, Alias, and MOD/REF. We presented DWIM, a new hybrid algorithm for solving the interprocedural constant propagation problem, which uses a demand-driven approach with two intraprocedural propagators, an optimistic initial one and pessimistic incremental one. The pessimistic algorithm was developed from an existing powerful optimistic one. We have performed preliminary evaluation of DWIM and we have shown it to be as effective as the best approach based on procedure summary information.

## 9 Bibliography

## References

[All72] F. E. Allen. *A Catalogue of Optimizing Transformations*, pages 1–30. Prentice Hall, Englewood Cliffs, NH, 1972.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[BC86] Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 162–175, July 1986.

[CCKT86] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 152–161, July 1986.

[CD93] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, 1993.

[CFR$^+$89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, February 1989.

[CK84] Keith D. Cooper and Ken Kennedy. Efficient computation of flow-insensitive interprocedural summary information. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 247–258, June 1984.

[CK86] Keith D. Cooper and Ken Kennedy. Fast interprocedural alias analysis. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 29–41, February 1986.

[CK87] Keith D. Cooper and Ken Kennedy. Efficient computation of flow-insensitive interprocedural summary infomation (a correction). Technical Report TR87-60, Rice University, 1987.

[CK88] Keith D. Cooper and Ken Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 57–66, August 1988.

[Coo85] Keith D. Cooper. Analyzing aliases of reference formal parameters. In *Proceedings of the Twelfthh Annual ACM Symposium on Principles of Programming Languages*, pages 281–290, January 1985.

[EB91] Rudolf Eigenmann and William Blume. An effectiveness study of parallelizing compiler techniques. In *Proceedings of the 1991 International Conference on Parallel Processing*, August 1991.

[GT93] Dan Grove and Linda Torczon. Interprocedural constant propagation: A study of jump function implementations. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 90–99, June 1993.

[HK92] Mary W. Hall and Ken Kennedy. Efficient call graph analysis. *ACM Letters on Programming Languages and Systems*, 1(3):227–242, Sept 1992.

[HP90] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Mateo, CA, 1990.

[Kil73] G. Kildall. A unified approach to global program optimization. In *Proceedings of the First Annual ACM Symposium on Principles of Programming Languages*, 1973.

[Kol94] Priyadarshan Kolte. Optimization of array subscript range checks. Technical report, Oregon Graduate Institute, 1994.

[KU77] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

[MS93] Robert Metzger and Sean Stroud. Interprocedural constant propagation: An empirical study. *ACM Letters on Programming Languages and Systems*, 2(4):213–232, December 1993.

[MW93] Herbert G. Mayer and Michael Wolfe. Interprocedural alias analysis: Implementation and empirical results. *Software– Practice and Experience*, 23(11):1201–1233, November 1993.

[SLY90] Zhiyu Shen, Zhiyuan Li, and Pen-Chung Yew. An empirical study of Fortran programs for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):356–364, July 1990.

[SW94] Eric Stoltz and Michael Wolfe. Constant propagation: A fresh, demand-driven look. In *Symposium on Applied Computing*. ACM SIGAPP, March 1994.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

## 10 Appendix A

In the constants found table, the first column, Fetches, is the number of constant fetches found by intraprocedural analysis. The second column is the additional ones found with *MOD* sets, the third column is number of ones over column two found with both *MOD* sets and the LCFJF, and the fourth column is the additional ones in excess of column three when the PNFJF is used instead of the LCFJF. Columns five through eight are the same, respectively, but for constant predicates. In the second table, dead edges, columns one through four are the same as in the first table, but for call sites eliminated, and columns five through eight are basic blocks eliminated. In the third table, dead code, columns one through four are the same as in the second table, but for intermediate code tuples eliminated. In cases where static predicates are found some fetches and predicates may become unexecutable, the number is indicated after a slash ("/").

There seems to be an anomaly introduced by the MOD/REF transformation that updates the call site information which causes additional fetches to appear. The additional number found is indicated in parentheses in the appropriate column.

In table three, all the ICs eliminated by MOD/REF are for simplification at call sites to account for non-referenced global parameters. A few have actual code elimination due to new constant predicates found, and these are called out in parentheses.

Table 6: Constants Found per Benchmark

| Program | Intra | + | + | + | Intra | + | + | + |
|---|---|---|---|---|---|---|---|---|
| | Fetches | *MOD* | LC | PN | Preds | *MOD* | LC | PN |
| adm | 101 | (12) | 3 | | 1 | | | |
| arc2d | 75 | (9) | 8/3 | 17 | | | 2/1 | |
| dyfesm | 6 | (-9) | 7/3 | 11 | | | 3 | |
| main | 255 | (1) | 1 | 5/8 | | | | 4 |
| ocean | 1 | (46) | 2 | 2/17 | | | | 2/1 |
| qcd | 12 | (-9) | 1 | 6 | | | | |
| spice | 162/1763 | -13/471 | 3/1 | 1/1 | 32/199 | 1/90 | 3 | 1 |
| trfd | 5 | | 1 | | | | 1 | |
| boast | 50/216 | 6/164 | 3 | 46/29 | 2/6 | 7 | | 18/1 |
| hydro | 20 | 14/15 | | | | | | |
| simple | 2 | 163/14 | | | | 2 | | |
| sphot | 4 | (-1) | | | | | | |
| wanal1 | 47/92 | (-7) | 6 | | 5 | | | |
| wave | 55/13 | (-12) | | | 2/3 | 1/-2 | | |
| baro | 12 | (-1) | 6 | 2 | | | | |
| vortex | 10 | 7(2) | | 71 | | | | 4 |

Table 7: Dead Edges and Blocks Found

| Program | Intra | + | + | + | Intra | + | + | + |
|---|---|---|---|---|---|---|---|---|
| | Calls | *MOD* | LC | PN | Blocks | *MOD* | LC | PN |
| adm | | | | | 5 | | | |
| arc2d | | | | | | | 4 | |
| dyfesm | | | | | | | 6 | |
| main | | | | | | | | 6 |
| ocean | 1 | | 1 | | 41 | | 6 | |
| qcd | | | | | | | | |
| spice | 216 | 49 | | | 482 | 288 | 1 | 2 |
| trfd | | | | | | | | |
| boast | | 2 | | 21 | 55 | -25 | | 20 |
| hydro | | | | | | | | |
| simple | | | | | | | | |
| sphot | | | | | | | | |
| wanal1 | | | | | 164 | | | |
| wave | 4 | -4 | | | 9 | -4 | | |
| baro | | | | | | | | |
| vortex | | | | | | | | 2 |

Table 8: Dead Code Found

| Program | Intra | + | + | + |
|---|---|---|---|---|
| | ICs | *MOD* | LC | PN |
| adm | 403 | 6666 | | |
| arc2d | | 838 | 34 | |
| dyfesm | | 2815 | 12 | |
| main | | 1667 | | 34 |
| ocean | 476 | 41876 | | 611 |
| qcd | | 1698 | | |
| spice | 46281 | (4854)70095 | 11 | 19 |
| trfd | | 107 | | |
| boast | 1584 | (-1050)12389 | 949 | |
| hydro | | 10211 | | |
| simple | | 2499 | | |
| sphot | | 906 | | |
| wanal1 | 1113 | 337 | | |
| wave | 480 | (39)10750 | | |
| baro | | 211 | | |
| vortex | | 785 | | 6 |