Demand-Driven Constant Propagation*

Eric Stoltz, Michael Wolfe, and Michael P. Gerlek

Department of Computer Science and Engineering

Oregon Graduate Institute of Science & Technology

P.O. Box 91000

Portland, OR 97291-1000

(503) 690-1121 ext. 7404

FAX: (503) 690-1553

stoltz@cse.ogi.edu

Technical Report 93-023

Abstract

Constant propagation is a well-known static compiler technique in which values of variables that are determined to be constants can be passed to expressions that use these constants. Code size reduction, bounds propagation, and dead-code elimination are some of the optimizations which benefit from this analysis.

In this paper, we present a new method for detecting constants, based upon an optimistic *demand-driven* recursive solver, as opposed to more traditional iterative solvers. The problem with iterative solvers is that they may evaluate an expression many times, while our technique evaluates each expression only once. To consider conditional code, we augment the standard Static Single Assignment (SSA) form with merge operators called γ-functions, adapted from the interpretable Gated Single Assignment (GSA) model. We present preliminary experimental results which show the number of intra-procedural constants found in common high-performance Fortran programs.

*This research supported by NSF grant CCR9113885, ARPA grant F3062-92-C-135, and grants from Intel Corporation and Matsushita Electric Industrial.

1 Introduction

Constant propagation is a static technique employed by the compiler to determine values which do not change regardless of the program path taken. In fact, it is a generalization of *constant folding* [1], the deduction at compile time that the value of an expression is constant, and is frequently used as a preliminary to other optimizations. The results can often be propagated to other expressions, enabling further applications of the technique. It is this recursive nature of the data-flow problem which suggests using a *demand-driven* method instead of the more usual iterative techniques.

In the following example, the compiler substitutes the value of 5 in S1 for `x`, which is a canonical instance of constant folding. Since the value of `x` is now constant, the compiler can propagate this value into S2, which, after applying constant folding once again, results in the determination that `y` is the constant 20. It should be noted that constant propagation for this work was applied only to scalar integer values. Propagation of real-valued expressions can be performed, but special care is required since operations on real-valued expressions are often architecturally dependent. The method outlined in this work also allows for arbitrary symbolic expression propagation [2].

```
S1:    x = 2 + 3
S2:    y = 4 * x
```

Although in general constant propagation is an undecidable problem [3], it is nonetheless extremely useful and profitable for a number of optimizations. These include dead code elimination [4], array- and loop-bound propagation, and procedure integration and inlining, which we believe to be a major source of detectable constants [5]. Due to these benefits, constant propagation is an integral component of modern optimizing commercial compilers [6, 7, 8].

The paper is organized as follows. In Section 2 we examine the standard framework employed to perform constant propagation and relate it to previous methods and algorithms. In

Section 3 we define the structure used for this work, with particular attention given to the necessary intermediate form (based upon Static Single Assignment) required to implement the algorithms we present. The basic propagation method used in our restructuring, parallelizing compiler is also given in this section. Section 4 describes the extension of the method to conditional code, and a discussion of variables within loops, which we have found is closely tied to induction variable recognition. In Section 5 we present the experimental results obtained thus far, and we close with future directions and conclusions in Section 6.

2 Background and Other Work

2.1 Framework

Constant propagation operates on a standard 3-level lattice, as shown in Figure 1. Top ($\top$) is the initial state for all symbols. When comparing two lattice element values, the meet operator ($\sqcap$) is applied, as given in Table 1. These foundations are standard for many constant propagation methods [4, 9, 5], originally introduced by Kildall [10]. Each symbol has its lattice value initialized to $\top$, which indicates that it has an as yet undetermined value. After analysis is complete, all symbols will have lattice value equal to $\perp$ (it cannot be determined to be constant), a constant value, or $\top$ (unexecutable code). We note that values can only move down in the lattice, due to the meet operator. By initializing lattice values to $\top$, an optimistic approach is taken, which assumes all symbols can be determined to be constant until proven otherwise.

Previous methods perform the analysis as an iterative data-flow problem [11], in which iterations continue until a fixed point is reached [4, 5]. We will see in the next section that an alternative demand-driven recursive algorithm offers advantages over the traditional approach.

$$
\begin{array}{cccl}
\top & \sqcap & \text{any} & = \text{any} \\
\bot & \sqcap & \text{any} & = \bot \\
\text{constant}_i & \sqcap & \text{constant}_j & = \begin{cases} \text{constant}_i & \text{if } i = j \\ \bot & \text{otherwise} \end{cases}
\end{array}
$$

Table 1: Rules for meet ($\sqcap$) operator.

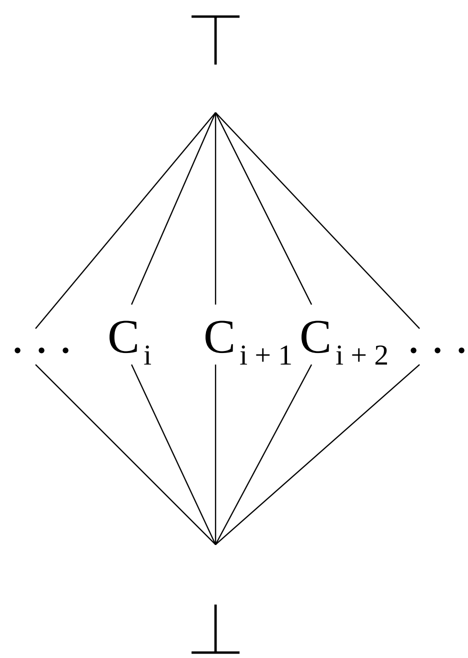

Figure 1: Standard Constant Propagation Lattice

```
S1:   z = 3              S7:    z =  3
S2:   if ( P ) then      S8:    if ( z < 5 ) then
S3:      y = 5           S9:       y = 5
S4:   else              S10:    else
S5:      y = z + 2      S11:       y = 2
S6:   endif             S12:    endif
```

(a) (b)

Figure 2: Constant propagation with (a) simple, and (b) conditional, constants

2.2 Previous Methods

2.2.1 Classification

As explained by Wegman and Zadeck [4], constant propagation algorithms can be classified in two ways: (*i*) using the entire graph or a sparse graph representation, and (*ii*) detecting simple or conditional constants. This naturally creates four classes of algorithms. It is clear that propagating information about each symbol to every node in a graph is inefficient, since not all nodes contain references or definitions of the symbol under consideration. Sparse representations, on the other hand, such as def-use or use-def chains [11], Static Single Assignment (SSA) [12], Dependence Flow Graphs (DFG) [13], or Program Dependence Graphs (PDG) [14], have all shown the virtue of operating on a sparse graph for analysis.

The distinction between simple (all paths) constants and conditional constants can be seen in Figure 2. The simple value of `y` is determined to be constant only if both branches which merge at S6 are constant with identical value, as is the case in (a). However, if the predicate which controls branching can be determined to be constant, then only one of the branches will be executed, allowing not only `y` to be recognized as constant in (b), but also identifying the other path to be dead code.

The distinction between the four types of algorithms is explained well by Wegman and

Zadeck [4], and the reader is referred to their paper for more detail. We will look at the algorithm that they present, since it incorporates both sparse graph representation and conditional code. The sparse graph employed is the SSA form, described in the next subsection.

2.2.2 Graph Preliminaries

The algorithms to convert a program into SSA form are based upon the *Control Flow Graph* (CFG), which is a graph G = $<V,E,Entry,Exit>$, where V is a set of nodes representing basic blocks in the program, E is a set of edges representing sequential control flow in the program, and *Entry* and *Exit* are nodes representing the unique entry point into the program and the unique exit point from the program. *Switch* nodes have their outgoing edges determined by a *predicate*. After a program has been converted into SSA form, it has two key properties:

1. Every use of a variable in the program has exactly one reaching definition, and

2. At confluence points in the CFG, merge functions called *ϕ-functions* are introduced. A ϕ-function for a variable merges the values of the variable from distinct incoming control flow paths (in which a definition occurs along at least one of these paths), and has one argument for each control flow predecessor. The ϕ-function is itself considered a new definition of the variable.

For details on SSA graph construction the reader is referred to the paper by Cytron et al. [12]. A sample program converted into SSA form is shown in Figure 3.

2.2.3 A Closer Look at One Algorithm

The algorithm used by Wegman and Zadeck operates on CFG edges. SSA *def-use* edges are added to the graph once the program has been transformed into SSA form.

Their algorithm works by keeping two worklists, a FlowWorkList and an SSAWorkList. Flow edges are initially marked *unexecutable*. Edges are examined from either worklist until empty,

(a)	(b)	(c)
$x = 0$	$x_0 = 0$	$x_0 = 0$
$y = 0$	$y_0 = 0$	$y_0 = 0$
$z = 0$	$z_0 = 0$	$z_0 = 0$
if (P) then	if (P) then	if (P) then
$y = y + 1$	$y_1 = y_0 + 1$	$y_1 = y_0 + 1$
endif	endif	endif
	$y_2 = \phi\ (\ y_0,\ y_1\ )$	$y_2 = \gamma\ (\ P,\ \mathit{true}\!\rightarrow y_1,\ \mathit{false}\!\rightarrow y_0\ )$
$x = y$	$x_1 = y_2$	$x_1 = y_2$
$z = 2 * y - 1$	$z_1 = 2 * y_2 - 1$	$z_1 = 2 * y_2 - 1$

Figure 3: Program in (a) normal form, (b) SSA form, and (c) GSA form

with those examined from the FlowWorkList being marked *executable*. The destination node for these edges also have their ϕ-functions evaluated by taking the meet of all the arguments whose corresponding CFG predecessors are marked *executable*. Expressions are evaluated the first time a node is the destination of a flow edge, and also when the expression is the target of an SSA edge and at least one incoming flow edge is executable. More detail can be found in the original paper [4].

This algorithm finds all simple constants, plus additional constants that can be discovered when the predicate controlling a switch node is determined to be constant. The time complexity is proportional to the size of the SSA graph, and each SSA edge can be processed at most twice.

Since ϕ-functions are re-evaluated each time an edge with that node as a destination is examined, Wegman and Zadeck note that expressions which depend on the value of a ϕ-function may be re-evaluated twice for each of its operands. For example, in this program fragment:

```
        if ( P )

        then
10        y1 = 1
          z1 = 2

        else
20        y2 = 1
          z2 = 3

        endif
30      y3 = φ(y1, y2)
        z3 = φ(z1, z2)
        x1 = y3 + z3
```

if `P` is not constant, the expression for $\mathtt{x}_1$ may be evaluated many times. If the flow edge from `10` is processed first, then $\mathtt{x}_1$ equals 3, and it may stay at 3 if the SSA edges for `y` are examined next. Eventually, $\mathtt{x}_1$ will evaluate to $\perp$, as the merge for `z` becomes non-constant. It is this multiple expression evaluation which we seek to avoid.

3 SSA using FUD Chains for Simple Constants

3.1 FUD Chains

In our restructuring compiler, Nascent [15], we also convert the intermediate representation into SSA form. In order to achieve the single-assignment property each new definition of a variable receives a new name. Practically, however, this is undesirable (managing the symbol table explosion alone precludes this option), so the SSA properties are maintained by providing links between each use and its one reaching definition. Instead of providing def-use links, as is the common implementation [4, 13], we provide use-def links, giving rise to an SSA graph comprising factored use-def chains (FUD chains). This approach yields several advantages, such as constant space per node and an ideal form with which to perform demand-driven analysis[16].

Our analysis of programs begins within a framework consisting of the CFG and an SSA data-flow graph. Each basic block contains a list of intermediate code tuples, which themselves are linked together as part of the data-flow graph. Tuples are of the form $<$*op,left,right,ssalink,lattice*$>$, where *op* is the operation code and *left* and *right* are the two operands (both are not always required, e.g. a unary minus). The *ssalink* is used for fetches and arguments of ϕ-functions, as well as indexed stores (which are not discussed further in this paper). The *ssalink*, if applicable, represents the one reaching definition for the variable in question at that point in the program. The *left*, *right*, and *ssalink* fields are pointers: they are all essentially use-def links. Thus, to perform an operation associated with any tuple, a request (or *demand*) is made for the information at the target of these links. Each tuple also has a lattice element assigned to it, *lattice*, initialized to $\top$.

We first show how to implement simple constant propagation within our framework. Our algorithm efficiently propagates simple constants in the SSA data-flow graph by demanding the lattice value from the unique definition point of each use. We visit all CFG nodes, examining each of its tuples, calling *propagate*() recursively on any unvisited left or right tuples. Assignment statements are evaluated, calling *propagate*() on all references with an *ssalink*. When a ϕ-function is encountered, recursive calls to the arguments are made, followed by taking the meet of those arguments. In the case of data-flow cycles, characterized by ϕ-functions at loop-header nodes, $\bot$ is returned. The algorithm is given in Figure 4.

Several points are noted regarding this algorithm:

- This is not an iterative solver. It is a recursive demand-driven technique which will completely solve the graph in the absence of cycles. The order in which basic blocks are visited is not important.

- This is an optimistic solver, since all symbols are initialized to $\top$. We find the same class of simple constants as other non-conditional solvers, such as Kildall [10] and Reif and

```
∀ t ∈ tuples,
    lattice( t ) = ⊤
    unvisited( t ) = true

Visit all basic blocks B in the program
    Visit all tuples t within B
        if unvisited ( t ) then propagate( t )

propagate ( tuple t )
    unvisited ( t ) = false
    if ssa_link( t ) ≠ ∅ then
        if unvisited( ssa_link( t ) ) then propagate( ssa_link( t ) )
        lattice( t ) =  lattice( t ) ⊓ lattice( ssa_link( t ) )
    endif
    if unvisited( left ( t ) ) then propagate( left( t ) )
    if unvisited ( right ( t ) ) then propagate( right( t ) )
    case on type (t )
        constant C: lattice( t ) = C
        arithmetic operation:
            if all operands have constant lattice value
            then lattice( t ) = arithmetic result of
                lattice values of operands
            else lattice( t ) = ⊥
            endif
        store: lattice( t ) = lattice( RHS )
        φ-function:
            if loop-header φ then lattice( t ) =  ⊥
            else  lattice( t ) = ⊓ of φ-arguments of t
            endif
        default: lattice( t ) = ⊥
    end case
end propagate
```

Figure 4: Demand-driven propagation of simple constants.

Lewis [17].

- When at a merge node, we take the meet of the demanded classification of the ϕ-arguments. By Table 1, this will result in a constant iff all ϕ-arguments are constant and identical.
- Each expression is evaluated once, since the node containing the expression will only be evaluated after all referenced definitions are classified.
- The asymptotic complexity is proportional to the size of the SSA data-flow graph, since it requires each SSA edge to be examined once.
- In the presence of data-flow cycles (due to loops in the CFG), the solver will fail to classify constant valued tuples, either as a function of the loop's trip count or even if it remains constant throughout the loop. A more complex solver, such as the one described in the next section, is needed to account for cycles.

4 Constants within Conditionals and Loops

4.1 Extending SSA to GSA

When demanding the classification of a variable at a merge node, we take the meet of the demanded classification of its ϕ-arguments, as noted in the last section. However, if only one of the branches will, in fact, be taken, we would like to only propagate the value along that path. In previous methods, as illustrated in Section 2, the predicate at a branch (or split) node is first evaluated, and if found constant, the executable edge is added to a worklist. Thus, when the corresponding merge node is processed, expressions at that node will be evaluated in terms of the incoming executable edges. As we have seen, this may result in expressions being evaluated more than once.

In our method, if a symbol demands the value from a merge node, we want to process the predicate that determines the path to follow. Examine Figure 3(b). When attempting to classify $\mathtt{x}_1$, the value is demanded from the use-def SSA link of $\mathtt{y}_2$, which points to the ϕ-function. However, a ϕ-function is not interpretable [18]. Thus, we have no information about which path may or may not be taken. Since the predicate P in our example determines the path taken, if P is constant, we can determine which argument of the ϕ-function to evaluate. If P is not constant, the best we can do is to take the meet of the ϕ-arguments.

Augmentation of the ϕ-function is needed to include this additional information. We extend the SSA form to a *gated single assignment* form (GSA), introduced by Ballance et al. [18], which allows us to evaluate conditionals based upon their predicates. Figure 3 shows a simple program converted to GSA form. Briefly, ϕ-functions are reclassified into μ- and γ-functions. Most ϕ-functions contained within loop-header nodes are renamed μ-functions, while most other ϕ-functions are converted to γ-functions*. The γ-function, $\mathtt{v}{=}\gamma(\mathtt{P}, \textit{true} \rightarrow \mathtt{v}_1, \textit{false} \rightarrow \mathtt{v}_2)$, means *if* P *then* $\mathtt{v}{=}\mathtt{v}_1$ *else* $\mathtt{v}{=}\mathtt{v}_2$. In this form, the γ-function represents an `if-then-else` construct, but it is also extended to include more complex branch conditions, such as `case` statements. Several important notes are necessary:

- We provide the complete algorithm to convert ϕ-functions to γ- and μ-functions in Appendix A. A similar method employed by Havlak [19], aimed at value-numbering, *thins* the γ-function to eliminate paths that cannot reach a merge point. Essentially, if all arguments save one are $\top$, then the entire argument structure is reduced to the one non-$\top$ argument. Thinning misses identifying constants in some situations, such as shown in Figure 5.

- It is convenient to insert nodes into the CFG such that the header node of a loop has

*A ϕ-function cannot be converted to a μ- or γ-function in the presence of irreducible loops.

exactly two predecessors, one from within the loop and one from without. A *preheader* node is inserted to accomplish this task, and we also insert a *postbody* node that is the target for all loop back edges.

- Multiple levels of conditionals result in nested γ-functions. Examine Figure 5, which is an unstructured code fragment (although structured code with nested `if-then` constructs also result in nested γ-functions). Figure 5(b) shows the program translated into GSA form. It is quite an interesting example for constant propagation, since if we know the value of predicate `P` we always know what possible value of `x` can reach the merge at `40`. However, if we don't know `P`, then the value of predicate `Q` becomes crucial:

  - If `Q` is true, only $\mathtt{x}_1$ can reach `40`.
  - If `Q` is false, we have no clear information on what value of `x` to propagate.

  If *thinning* were used, the γ-function at `40` would reduce to: $\mathtt{x}_2 = \gamma($ P, $t \rightarrow \mathtt{x}_1$, $f \rightarrow \mathtt{x}_0)$. If `P` is not constant, the meet of its arguments is $\perp$. However, if `Q` is known to be true, the constant value $\mathtt{x}_1$ will be missed using thinning, since the *false* side of predicate `P` is prematurely reduced to $\mathtt{x}_1$, instead of $\top$.

- As described by Ballance et al., μ-functions also contain a predicate – it determines whether another execution of the loop will take place. We don't require an executable intermediate form, and, as we shall see, other techniques efficiently handle loops.

- Only reducible flow graphs can be converted into GSA form. An irreducible graph contains loops with multiple entries – this leads to problems both in loop *detection* (we classify loops according to the *natural loop* [11] definition), and working with control dependence (in an irreducible graph, the transitive control dependence of a node can skip over the immediate dominator). The Appendix provides more detail.

```
   x = 2                       x0=2
   if ( P ) goto 30            if ( P ) goto 30
   if ( Q ) goto 50            if ( Q ) goto 50
   else goto 40                else goto 40
30 x = 3                    30 x1=3
40 y = x                    40 x2 = γ( P, t→x1, f→γ( Q, t→ ⊤, f→ x0))
                               y1=x2
50 continue                 50 continue
   (a)                         (b)
```

Figure 5: Conditional code which results in nested γ-functions

4.2 Conditional Constant Propagation

Once converted into GSA form, we can improve upon the *propagate*() routine to take advantage of predicates that can be determined to be constant. When encountering a γ-function, we first attempt to evaluate the predicate. If constant, we follow the indicated branch, propagating constant values as found. If not constant, we take the meet of its arguments. The revised algorithm is given in Figure 6.

We may encounter ϕ-functions in a program with irreducible loops [11]. In this case, ϕ-functions cannot be converted to GSA form, but we can still detect simple constants.

Several comments need to be made regarding this algorithm:

- Due to lack of space, we have only dealt with integer constants, not logical or enumerated types.
- We have not covered arithmetic simplifications, including special cases such as zero times anything (including $\perp$) equals zero.
- Reaching a μ-function returns $\perp$. This is due to the separate solver used for loops, discussed next.

```
∀ t ∈ tuples,
      lattice( t ) = ⊤
      unvisited( t ) = true

Visit all basic blocks B in the program
      Visit all tuples t within B
            if unvisited ( t ) then propagate( t )

propagate ( tuple t )
      unvisited ( t ) = false
      if ssa_link( t ) ≠ ∅ then
            if unvisited( ssa_link( t ) ) then propagate( ssa_link( t ) )
            lattice( t ) = lattice( t ) ⊓ lattice( ssa_link( t ) )
      endif
      if unvisited ( left ( t ) ) then propagate( left( t ) )
      if unvisited ( right ( t ) ) then propagate( right( t ) )
      case on type (t )
            constant C: lattice( t ) = C
            arithmetic operation:
                  if all operands have constant lattice value
                  then lattice( t ) = arithmetic result of
                        lattice values of operands
                  else lattice( t ) = ⊥
                  endif
            store: lattice( t ) = lattice( RHS )
            φ-function: lattice( t ) = ⊓ of φ-arguments of t
            γ-function:
                  if lattice( predicate ) = C then
                        lattice( t ) = lattice value of
                        γ-argument corresponding to C
                  else lattice( t ) = ⊓ of all γ-arguments of t
                  endif
            μ-function: lattice( t ) = ⊥
            η-function: lattice( t ) = lattice ( η-argument )
            default: lattice( t ) = ⊥
      end case
end propagate
```

Figure 6: Demand-driven propagation with conditional constants.

4.3 Loops

Cycles in the GSA data-flow graph are the result of loops within the original program. The variables defined within these cycles are detected with induction variable analysis. Induction variables are traditionally detected as a precursor to strength reduction, and more recently for dependence analysis with regard to subscript expressions. We have developed methods for detecting and classifying induction variables (including non-linear induction variables [20, 21, 22]) based on strongly-connected regions in the SSA data-flow graph [2]. These techniques make use of an exit function, the η-function, which holds the exit value of a variable assigned within the loop. The exit value may be a function of the loop tripcount (which may itself be an expression determined to be constant), or may be invariant with respect to the loop. An exit expression is held by the η-argument, which if constant can be used to propagate values outside of the loop. Ballance et al. introduced η-functions in their GSA form. We place η-functions in *postexit* nodes as part of our SSA translation phase. For each edge exiting the loop, a postexit node is inserted outside the loop, between the source of the exit edge (within the loop body) and the target (outside the loop).

We are able to propagate constants through loops (single and nested) by taking advantage of specialized solvers which detect and classify a large assortment of linear and non-linear induction variables. Interested readers may obtain a description of this work via *anonymous ftp* to cse.ogi.edu [23].

5 Experimental Results

To gauge the effectiveness of our routines, we measured the number of constants (both simple and conditional) on Fortran scientific codes found in the PERFECT, RICEPS, and MENDEZ benchmark suites, and several miscellaneous but important routines. A constant is considered propagated if there was a fetch of a constant. Folded constants are counted separately.

Results are shown in Table 2. The vast majority of constants (95%) are simple constants. Most conditional constants were as a result of loop analysis. Although a few predicates controlling switch nodes are determined constant, we believe these are mainly due to guards; not until interprocedural analysis and inlining are implemented do we expect to see many conditional constants propagated. Results are also shown for the number of folded constants.

With a total of 88013 lines of code analyzed, we found 1.5 simple constants per procedure, on average, with one constant every 55 lines. For conditional constants, we found 1.6 per procedure, and one constant every 52 lines.

To obtain valid comparisons with other algorithms, notably Wegman and Zadeck's, we are currently implementing a version of our compiler that transforms intermediate forms into a version of SSA which supports their data structures. We plan timing tests for several constant propagation algorithms on the same test suite of Fortran programs as was used in this section.

6 Future Work and Conclusions

We plan many extensions to this work. One important topic is interprocedural analysis and procedure integration, an area where we believe many constants will be found. Although some work has already been done in this area [5, 24, 25], we would like to apply our demand-driven style to the problem.

Dead-code can currently be identified with our technique, but we have not yet developed the algorithm fully. It may well be that dead code is best identified using edges instead of nodes, as pointed out by Wegman and Zadeck.

Traditional SSA form has been criticized for lacking a method to propagate constants determined by predicate analysis [13]. In the following fragment

routine	*lines*	*procs*	*FC*	*SC*	*CP*	*NCP*	*CC*
PERFECT club							
adm	4165	97	3	102	0	271	102
arc2d	2747	39	9	98	0	51	98
bdna	3793	43	1	42	0	171	56
dyfesm	4401	78	1	4	0	130	5
flo52	1850	28	15	77	0	108	78
mdg	1028	16	1	0	0	39	0
mg3d	2537	28	9	249	0	118	255
ocean	2577	36	20	1	0	153	1
qcd	1780	35	5	15	2	87	17
spec77	3399	65	33	28	2	119	38
track	2192	32	19	1	0	150	1
trfd	418	7	4	8	2	20	8
RICEPS							
boast	7212	58	38	21	2	696	24
ccm	18709	145	91	507	3	537	529
linpackd	468	11	0	14	0	32	21
simple	1239	8	52	172	1	25	172
sphot	876	7	1	2	0	32	2
wanal1	1718	11	52	43	3	28	43
MENDEZ							
euler	1183	14	6	17	4	116	17
mhd2d	827	14	19	56	0	21	56
shear	848	16	56	34	3	39	34
vortex	564	20	2	13	0	17	13
MISC							
comp3	1477	1	0	0	0	188	0
comps	7707	24	4	26	11	765	32
eispack	7587	68	27	3	0	711	3
livermore	5003	38	28	54	2	142	62
vector	1708	101	2	10	0	39	10
Total	88013	1040	498	1597	35	4864	1677

Table 2: Experimental runs to detect propagated constants. FC = folded constant, SC = simple constant, CP = constant predicate, NCP = non-constant predicate, CC = conditional constant.

```
if (x₁ = 1) then
  i₀ = x₁
else
  j₀ = x₁
endif
```

it is desirable to be able to assign i_0 constant value. A sophisticated compiler may analyze the guard and determine that under the range of the true side of the conditional, x_1 will always be 1. This notion of a *derived assertion* is not new [8], but to our knowledge has not yet been integrated into the SSA form. Using demand-driven SSA form, derived assertions can easily be captured by inserting dummy assignments. We propose a new SSA operator, the ρ-function, which serves as the new definition of its variable. By examining the right-hand side of the predicate, the above fragment becomes:

```
if (x₁ = 1) then
  x₂ = ρ(1)
  i₀ = x₂
else
  j₀ = x₁
endif
```

Now constant propagation may easily be performed via the argument of the ρ-function, which may be constructed of actual operations in the intermediate form.

In addition to constant propagation, the explicit representation of derived assertions may be advantageous if bounds information can be expressed. In this fragment,

```
if (n₀ > 0) then
  for i=1, n₀
    ...
  endfor
endif
```

if the compiler cannot determine any value for n_0, then it cannot be determined if the body of the loop will ever be executed within the range of the `if`. However, analysis of the guard condition assures the loop will be executed at least once. If limit information can be encoded in the argument of the ρ-function, the loop may be transformed:

```
if (n_0>0) then
  n_1 = ρ(>0)
  for i=1, n_1
    ...
  endfor
endif
```

Now it is clear from the expression describing the tripcount that the loop will be executed at least once, since the lower limit of n is known.

Other planned projects include run-time analysis and value numbering. We are interested in obtaining timing results that demonstrate how much execution time is saved for the increased analysis done at compile time. These are interesting tradeoffs, and remain an open question. Although not constant propagation *per se*, the structure of GSA lends itself particularly well to implementing value numbering, as has been shown by Havlak [19]. Finally, we want to extend our work into the area of non-integer and symbolic expression propagation.

We have presented a new demand-driven method for performing conditional constant propagation, which works on sparse data-flow graphs, finds the same class of constants as previous algorithms, but avoids evaluating expressions more than once. We have detailed specific algorithms to accomplish this task, and have presented preliminary data on the number of constants found in scientific Fortran codes (and, as noted in the last section, we are building a comparative experiment). We believe this is a promising approach with many opportunities for extensions.

References

[1] Jean-Paul Tremblay and Paul G. Sorenson. *The Theory and Practice of Compiler Writing*. McGraw-Hill, New York, NY, 1985.

[2] Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *submitted for publication*, September 1993.

[3] J. Kam and J. Ullman. Monotone data flow analysis frameworks. *Acta Informatica 7*, pages 305–317, 1977.

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

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

[6] Steve S. Muchnick. Optimizing compilers for SPARC. *Sun* Technology, pages 161–173, 1988. Summer.

[7] D. Blickstein, P. Craig, C. Davidson, R. Faiman, K. Glossop, R. Grove, S. Hobbs, and W. Noyce. The GEM optimizing compiler system. *Digital Technical Journal*, 4:121–136, 1992. Special Issue.

[8] P. Lowney, SA. Freudenberger, T. Karzes, W. Lichtenstein, R. Nix, J. O'Donnell, and J. Ruttenberg. The Multiflow trace scheduling compiler. *The Journal of Supercomputing*, 7:51–142, 1993.

[9] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. In *Proceedings of Sigplan Symposium on Compiler Construction*, volume 21, June 1986.

[10] G. A. Kildall. A unified approach to global program optimization. In *Conference Record of the First ACM Symposium on Principles of Programming Languages*, pages 194–206, October 1973.

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

[12] 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 Trans. on Programming Languages and Systems*, 13(4):451–490, October 1991.

[13] Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 78–89, June 1993.

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

[15] Michael Wolfe, Michael P. Gerlek, and Eric Stoltz. Nascent: A Next-Generation, High Performance Compiler. Oregon Graduate Institute of Science & Technology *unpublished*, 1993.

[16] Eric Stoltz, Michael P. Gerlek, and Michael Wolfe. Extended SSA with factored use-def chains to support optimization and parallelism. In *1994 ACM Conf. Proceedings Hawaii International Conference on System Sciences*, January 1994. *to appear*.

[17] John H. Reif and Harry R. Lewis. Symbolic evaluation and the global value graph. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 104–118, January 1977.

[18] Robert A. Ballance, Arthur B. Maccabe, and Karl J. Ottenstein. The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proc. ACM SIGPLAN '90 Conf. on Programming Language Design and Implementation*, pages 257–271, White Plains, NY, June 1990.

[19] Paul Havlak. Construction of thinned gated single-assignment form. In *Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, August 1993.

[20] Michael Wolfe. Beyond induction variables. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 162–174, June 1992.

[21] Mohammed R. Haghighat and Constantine D. Polychronopoulos. Symbolic program analysis and optimization for parallelizing compilers. In *Workshop on Languages and Compilers for Parallel Computing*, pages 355–369, 1992.

[22] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the automatic parallelization of four Perfect-Benchmark programs. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 65–83. Spinger-Verlag, 1992. LNCS no. 589.

[23] Michael P. Gerlek. Detecting induction variables using SSA form. Technical Report 93-014, Oregon Graduate Institute of Science & Technology, 1993.

[24] Mary Hall. *Managing Interprocedural Optimization*. PhD thesis, Department of Computer Science, Rice University, 1991.

[25] R. Metzger and S. Stroud. Interprocedural constant propagation: an empirical study. *ACM Letters on Programming Languages and Systems*, June 1992.

Appendix – The γ-Conversion Algorithm

The complete algorithm to translate a program from SSA form (already augmented with η-functions, the loop-exit place-holders) into GSA form is provided. This algorithm essentially renames loop-header ϕ-functions as μ-functions, while creating an interpretable γ-function to replace other ϕ-functions. We note that this translation is only possible with reducible flow graphs. In reducible graphs the initial switch node to determine program flow affecting a merge is always the immediate dominator.

This algorithm relies heavily on the concept of *control dependence*. Informally, X is control dependent on Y if one path from Y *must* reach X, while another path may avoid X. Cytron et al. [12] showed that control dependence is equivalent to dominance frontiers in the reverse CFG. We compute control dependence only on the *forward* CFG, eliminating back edges.

Roughly half the γ-functions can be reduced. This reduction can occur in two ways:

1. The same predicate occurs more than once in a γ-function. In this case, the value of the first occurrence of the predicate can prune the nested predicate. The *reduce*() function accomplishes this task.

2. If all γ-arguments have the same value, then the γ-function can be replaced by the value of the arguments.

As an example of *reduce*(), examine this code fragment:

```
        x0 = 0
        if( P ) goto 30
10      x2 = γa
        y1 = x2
        goto 40
30      x1 = 1
        if( Q ) goto 10
40      x3 = γc
```

Before reduce, the γ-function at `10` will be:

$$\mathtt{x}_2 = \gamma_a(\mathtt{P}, t \rightarrow \gamma_b(\mathtt{Q}, t \rightarrow \mathtt{x}_1, f \rightarrow \top), f \rightarrow \mathtt{x}_0)$$

And the γ-function at `40` will be:

$$\mathtt{x}_3 = \gamma_c(\mathtt{P}, t \rightarrow \gamma_d(\mathtt{Q}, t \rightarrow \gamma_a, f \rightarrow \mathtt{x}_1), f \rightarrow \gamma_a)$$

After applying the first reduction rule, the γ-function at `40` (γ_c) becomes:

$$\mathtt{x}_3 = \gamma_c(\mathtt{P}, t \rightarrow \gamma_d(\mathtt{Q}, t \rightarrow \mathtt{x}_1, f \rightarrow \mathtt{x}_1), f \rightarrow \mathtt{x}_0)$$

Next, the second reduction rule is applied, yielding:

$$\mathtt{x}_3 = \gamma_c(\mathtt{P}, t \rightarrow \mathtt{x}_1, f \rightarrow \mathtt{x}_0)$$

Replacing ϕ-Functions with γ- and μ-Functions

```
last_φ(∗) ⇒ previous φ-function processed at this basic block
current_γ(∗) ⇒ γ-function under consideration for this basic block
labels = branch values which correspond to outedges from a basic block
        If there is only one successor, the branch label is true
ssa_link = reaching definitions corresponding to a fetch
        or an argument from a φ, γ, or μ function

last_φ(∗) = ∅
current_γ(∗) = ∅

while  list of basic blocks not empty do
    B = next block in topological order from the CFG
    idom = immediate dominator of B
    for  each φ-function f in B do
        if f ∈ loop-header, then replace φ with μ
        else
            for each predecessor pred of B do
                lab = branch label of edge from pred to B
                ssa_link = φ-argument of f which corresponds to pred
                process( f, pred, lab, ssa_link  )
            enddo
            replace f with reduce( current_γ( idom ) )
        endif
    enddo
enddo

process( function f, basic block b, label lab, def link )
    if last_φ(b) ≠ f
        last_φ(b) = f
        if b has more than 1 successor
            send = current_γ(b) = build_gamma(b)
        else
            current_γ(b) = ∅
            send = link
        endif
        for each control predecessor  cp of b do
            if b ≠ idom then
                cp_lab = branch label from cp which executes  b
                process( f, cp, cp_lab, send )
            endif
        enddo
    endif
    if current_γ(b) ≠ ∅
        for argument a of current_γ(b) ∋ label a = lab
            set  ssa_link(a) = link
        endfor
    endif
end process
```

```
build_gamma( basic block bg )
    γ-predicate = switch function in bg
    for each successor succ of bg do
        e = label from bg to succ
        add γ-argument with label = e and ssa_link = Top
    enddo
    return γ
end build_gamma

reduce( object r )
    if r is not a γ-function return r
    predicate = switch operator of γ-function r
    if predicate already on the stack
        arg = γ-argument of r whose label matches the branch value of predicate
        return reduce( ssa_link( arg ) )
    endif
    for all γ-arguments a of r do
        push onto stack( predicate, label of a )
        ssa_link( a ) = reduce(ssa_link( a ) )
        pop off stack( predicate )
    enddo
    if all γ-arguments a of r have identical ssa_link return ssa_link( a )
    else return r
end reduce
```