Interprocedural Constant Propagation

David Callahan
Keith D. Cooper
Ken Kennedy
Linda Torczon

Department of Computer Science[†]
Rice University
Houston, Texas

Abstract

In a compiling system that attempts to improve code for a whole program by optimizing across procedures, the compiler can generate better code for a specific procedure if it knows which variables will have constant values, and what those values will be, when the procedure is invoked. This paper presents a general algorithm for determining for each procedure in a given program the set of inputs that will have known constant values at run time. The precision of the answers provided by this method are dependent on the precision of the local analysis of individual procedures in the program. Since the algorithm is intended for use in a sophisticated software development environment in which local analysis would be provided by the source editor, the quality of the answers will depend on the amount of work the editor performs. Several reasonable strategies for local analysis with different levels of complexity and precision are suggested and the results of a prototype implementation in a vectorizing Fortran compiler are presented.

1. Introduction

Fortran programmers have learned to expect optimizing compilers that generate excellent code for a single procedure. Twenty-five years of development has led to a well-understood collection of principles for building optimizing compilers and almost every commercially available computer system now offers one. One problem remains, however. The quality of the code produced by a good Fortran compiler declines considerably in the presence of calls to independently compiled procedures. The main reason for this is that the compiler must make worst case assumptions about what happens on the side of the interface that it cannot see. For example, when compiling the called procedure, the standard linkage convention requires that all the registers be saved and restored, even though many of them may not be in use at the point of call.

Interprocedural data flow analysis attempts to overcome this problem by propagating information about data usage and creation among the procedures that form a single program, so that the compiler can take advantage of contextual information to generate better code for any single procedure. An important interprocedural data flow analysis problem is the determination of which parameters[1] to a given procedure will be constant at run time. Based on this information, a compiler could perform a number of useful optimizations that are unavailable to it in current compilation schemes. For example, many subroutines in the LINPACK library [DBMS 79] have a parameter to indicate the stride of indexing for some array. In the typical program constructed from LINPACK, this stride is passed the integer constant ‘‘1’’. In the absence of better information, the compiler must assume that the parameter could take on any value, precluding the application of many

[†] This research has been supported by the National Science Foundation through grants MCS 81-21844 and MCS 83-03638 and by IBM Corporation.

[1] Here we extend the term *parameter* to cover global or imported variables.

optimizations. In particular, the value ‘‘0’’ would preclude vectorization of array operations within the procedure and a non-constant value would preclude loop unrolling, both of which have proven effective in improving performance of programs constructed from LINPACK [Dong 80].

Constant propagation is a code improvement technique in which the compiler reasons about the values that variables may assume at run time. If the compiler is able to establish that a variable will always have a known constant value at a particular use point it may replace an occurrence of that variable by an occurrence of the constant itself. Whenever all the inputs to an operation are replaced by known constant values, the operation may be performed at compile time and the constant result further propagated to other uses. When performed in a systematic way, constant propagation can lead to the evaluation at compile time of many operations that might be repeatedly performed at run time. This can significantly improve the performance of the compiled program.

Techniques for constant propagation within a single procedure have been widely discussed in the literature [Kild 73, Kenn 78, Kenn 81, WeZa 85]. However, these techniques have not been extended to propagate constants across procedure calls[2] for reasons of practicality. In order to compute which variables are constant at every invocation of a given procedure, the system must have knowledge of the behavior of every procedure in the program. In a traditional separate compilation system, the compiler doesn’t even know which procedures the program includes until the link editing step immediately prior to execution. Even if the compiler had this knowledge, without a database similar to those found in a programming environment, it would need to examine every procedure in the program in order to compile just one of them. This appears to be prohibitively expensive.

[2] A notable exception is the paper by Wegman and Zadeck [WeZa 85], which describes a single-procedure algorithm that can be used to evaluate the effectiveness of inline expansion of a called procedure. The technique propagates the known constants at a single call site into the called procedure to determine the size of the code that would result after inline expansion and useless code elimination. This approach does not extend naturally to handle procedures that are not expanded inline and may be called from more than one site.

In an attempt to address these problems, the project at Rice University has been developing a programming environment for Fortran that supports the development, compilation and optimization of *whole programs* [HoKe 85, CoKT 85, CoKT 86]. A central goal of this project is to experiment with the systematic development of interprocedural data flow information. In , the *source editor* provides local information about the behavior of individual subroutines and a *program composition editor* records which subroutines comprise the program. Once local information is available and the program composition is known, the computation of interprocedural data flow information is the responsibility of the *program compiler*. Most of the required interprocedural information is developed by solving data flow analysis problems on the program’s call graph. Since the call graph of a program is likely to be large, it is important for the program compiler to employ efficient algorithms for solving these data flow problems. Previously, Cooper and Kennedy have reported nearly-linear algorithms for the determination of side effects [CoKe 84] and potential aliases [Coop 85].

In this paper, we present a scheme for efficiently determining which variables have known constant values on invocation of the various procedures in a given program. Of necessity, our method is approximate. Kam and Ullman have shown that a precise solution to the simpler problem of intraprocedural constant propagation is not computable [KaUl 77]. Even the approximate problem commonly solved by compilers within a single procedure is intractable in an interprocedural setting because it has the property of *flow sensitivity* [Myer 81]. Therefore, we present a collection of methods that all use the same algorithm for propagating constants across the call graph but differ in the precision of the information used to model the behavior of individual procedures. This collection of algorithms clearly exhibits the tradeoff between complexity of local analysis by the source editor and the precision of the approximate sets of constants — more effort by the editor leads to finding a larger set of variables with known constant values.

The discussion begins in Section 2 with a presentation of a very precise method based on inline substitution and a discussion of the drawbacks of this approach. The algorithm we propose to use in the environment is presented in

Section 3. It is based on an efficient algorithm for constant propagation in a single procedure due to Reif and Lewis [ReLe 82] as adapted by Wegman and Zadeck [WeZa 85]. Our algorithm models the body of a procedure with functions that describe the values of variables at a call site in terms of the values of variables on entry to the procedure containing the call. Section 4 describes several methods for defining these functions and contrasts the completeness of the resulting constant sets. Section 5 describes our experience with an actual implementation of interprocedural constant propagation in a Fortran vectorizer. Finally, conclusions and future research are discussed in Section 6.

2. Inline Substitution

Perhaps the most obvious way to solve the problem is to convert the program into one large procedure using systematic inline substitution and employ an efficient single-procedure method such as the Reif-Lewis algorithm [ReLe 82] or the Wegman-Zadeck algorithm [WeZa 85]. Recursive procedures can be handled in this scheme by introducing an explicit stack, combined with unrolling. This approach is very effective at propagating constants because, in effect, each call site has its own private copy of the called procedure. Hence, constant propagation is not inhibited by having to merge constant sets from several call sites to produce information for a procedure that is invoked from more than one location. Furthermore, inline substitution makes parameter aliasing explicit in the program, permitting the analysis to be even more precise. When applied to the example in Figure 1, the method based on inline substitution will yield the program in Figure 2.

In spite of its effectiveness, we reject this technique for practical use on several grounds. First, the growth in the program's code size after substitution is potentially exponential [Sche 77]. Since the performance of the best algorithms for single-procedure constant propagation is roughly linear in the size of the procedure, the overall cost of constant propagation is potentially exponential in the size of the unexpanded program. Second, systematic use of inline expansion makes the program difficult to maintain. After each change to a single procedure, the system must either re-expand the program or attempt to recreate the change in the expanded, optimized program, a process that is likely to be very expensive.

```
procedure main;
  α: call joe(10,100,1000);
end;

procedure joe(i, j, k);
     l = 2 * k;
     if (j=100) then m = 10*j else m = i;
  β: call ralph(l, m, k);
     o = m * 2;
     q = 2;
  γ: call ralph(o, q, k);
     write q, m, o, l;
end;

procedure ralph(a, b, c);
     b = a * c/2000;
end;
```

Figure 1. An example problem

```
procedure main;
     write 1000, 1000, 2000, 2000;
end;
```

Figure 2. After inline substitution.

Therefore, we seek a method that performs constant propagation on the whole program while keeping the individual procedures separate. Furthermore, that method should have an efficiency that is proportional to the aggregate size of all the code in the program. In return for achieving these goals, we are willing to sacrifice some constant propagation effectiveness.

In spite of its drawbacks, inline substitution deserves study because it provides us with a standard by which to measure the effectiveness of other techniques. Furthermore, it can be extremely valuable as an optimization technique if used in a controlled way [Ball 79]. In fact, interprocedural constant propagation can provide the information required to efficiently determine when inline substitution can be profitably

employed [Coop 83, WeZa 85]. In the next section we present the general strategy used in the programming environment to propagate constants across procedure boundaries without using inline expansion.

3. General Approach

The goal of interprocedural constant propagation is to annotate each procedure in the program with a set CONSTANTS(p) of <*name*,*value*> pairs. A pair $\langle x,v \rangle \in$ CONSTANTS(p) indicates that variable x has value v at every call site that invokes p. This set approximates reality; every pair in CONSTANTS(p) denotes a run-time constant, but not all run-time constants can be found.

For the purpose of computing CONSTANTS(p) we formulate the problem in a lattice theoretic framework. With each entry point p we associate a function *Val* that maps formal parameters to elements of the usual *constant propagation lattice* L. Each element of L is one of three types:

- a lattice *top* element ,
- a lattice *bottom* element , or
- a constant value c.

The structure of this lattice is defined by the following list of rules for the lattice *meet* operation $\wedge$:

1) $\wedge\, a = a$ for any lattice element a
2) $\wedge\, a =$ for any lattice element a
3) $c \wedge c = c$ for any constant c
4) $c_1 \wedge c_2 =$ if $c_1 \neq c_2$

Figure 3 depicts the constant propagation lattice graphically. While it is an infinite lattice, it has bounded depth. In particular, if a variable is modified by assigning it a new value computed by taking the meet of its old value and some other lattice element, its value can be reduced at most twice.

For any formal parameter x, let $Val(x)$ represent the best current approximation to the value of x on entry to the procedure. After the analysis is complete, if $Val(x) =$, the parameter is known to be non-constant; if $Val(x) = c$, the parameter has the constant value c. The value is used as an initial approximation for all parameters; a parameter retains that value only if the procedure containing it is never called. Once we have computed $Val(x)$ for every parameter x of procedure p, CONSTANTS(p) is simply the set of parameters for which *Val* is equal to some constant value.

To compute the *Val* sets, we associate with each call site s in a given procedure p a *jump function*[3] J_s that gives the value of each parameter at s as a function of the formal parameters of the procedure p[4]. J_s is actually a vector of functions, one for each formal parameter of the procedure invoked at the call site. For each formal parameter y of the procedure called at s, the component function J_s^y computes the best approximation within the lattice L to the value passed to y at call site s, given the values passed to formal parameters of p. The *support* of the function J_s^y is the exact set of formal parameters of p that are used in the computation of J_s^y.

Given these definitions, the interprocedural constant propagation problem can be solved using an analogue of the Wegman-Zadeck method for constant propagation in a single procedure [WeZa 85]. The interprocedural algorithm, shown in Figure 4, assumes that each parameter used in the program has a unique name, so the procedure to which it is a parameter can be uniquely determined from the parameter name. The algorithm uses a worklist of parameters to be processed. It is guaranteed to converge because the lattice is of finite depth and a parameter is only added to the worklist when its value has been reduced. Since the value of any parameter can be reduced at most twice, each procedure parameter can appear on the worklist at most two times.

If $cost(J_s^x)$ is the cost of evaluating J_s^x, the total amount of work done by the computation is proportional to

$$\sum_s \sum_x cost(J_s^x) \cdot |support(J_s^x)|$$

where s ranges over all call sites in the program and x ranges over all formal parameters that are bound by the call at s. This bound is based on the observation that J_s^x is evaluated each time some parameter in $support(J_s^x)$ is reduced in value. Because of the structure of the lattice, a parameter can be reduced in value at most twice. If the cost of evaluating each J_s^x is bounded by a constant, implying that the size of the support is also

[3] The term *jump function* originated with John Cocke and is used here for historical reasons.

[4] For the purposes of this discussion, ignore the issue of global variables used as parameters. These can be treated as an extended set of parameters.

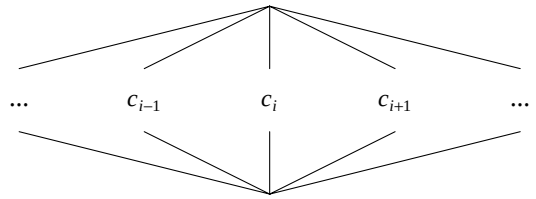

Figure 3. The constant propagation lattice.

```
procedure PropagateConstant;
    var worklist : set;
    begin
        { Initialize }
        for each procedure p in the program do
            for each parameter x to p do
                Val(x) := ;
        for each call site s in the program do
            for each formal parameter y
                that receives a value at s do
                Val(y) := Val(y)∧J_s^y;
        worklist := ∅;
        for each procedure p in the program do
            for each parameter x to p do
                worklist := worklist ∪ {x};
        { Iterate }
        while worklist ≠ ∅ do begin
            let y be an arbitrary parameter in worklist;
            worklist := worklist - {y};
            let p be the procedure containing y;
            { Update worklist }
            for each call site s ∈ p and parameter z
                such that y ∈ support(J_s^z) do begin
                oldVal := Val(z);
                Val(z) := Val(z)∧J_s^z;
                if Val(z) < oldVal then
                    worklist := worklist ∪ {z};
            end
        end
    end
```

Figure 4. General algorithm

bounded by a constant, the cost is proportional to the sum over each edge in the call graph of the number of parameters passed along that edge.

The tricky part of this method is the construction of the jump functions J_s. The next section describes three methods for implementing them.

4. Jump Functions

In developing algorithms for the construction of jump functions, we should keep the following principles in mind.

- For any parameter x at a call site s that can be determined to be constant by inspection of the source of the procedure containing s, set $J_s^x = c$ where c is the known constant value. This implies that $support(J_s^x) = \varnothing$.
- For any parameter x at call site s that cannot be determined as a function of input parameters to the procedure containing s, set J_s^x to be . For example, x might be passed a value that is read in from an external medium. Again, this implies that $support(J_s^x) = \varnothing$.
- For any parameter x at call site s that is determinable at compile time but not a constant, J_s^x is a function of parameters to the calling procedure and local constants of the calling procedure. In this case, $support(J_s^x)$ is non-empty.

Taken together, these principles gives rise to a range of strategies for constant propagation. The boundary between these three classes of values depends on the sophistication of the techniques used in the source editor to determine jump functions. If the complexity of computing a particular jump function exceeds the capabilities of the editor, it can simply give up and assign $J_s^x =$. Torczon discusses three strategies of different complexity for developing jump functions [Torc 85] — one that only finds jump functions that are constant or bottom, one that also discovers jump functions that pass a parameter through to a call site without change, and a third that employs a sophisticated symbolic interpretation algorithm such as the one proposed by Reif and Lewis [ReLe 82]. We describe each of these in the following sections.

```
procedure main;
  α: call joe(10,100,1000);
end;

procedure joe(i, j, k);
      l = 2000;
      m = 1000;
  β: call ralph(l, m, k);
      o = m * 2;
      q = 2;
  γ: call ralph(o, q, k);
      write q, m, o, l;
end;

procedure ralph(a, b, c);
      b = a * c/2000;
end;
```

Figure 5. After all-or-nothing analysis.

4.1. All or Nothing

A particularly easy to implement technique would have the source editor employ a single-procedure constant propagation technique, such as the Wegman-Zadeck algorithm to determine which variables are constant at each call site, under the assumption that none of the parameters to the routine containing the call site are constant. Then each jump function J_s^x is set to a constant value if the analysis determines that x must be constant at s and to otherwise. This approach would find constants that can be propagated over a single call. It would miss all constants that must be propagated completely through an intermediate procedure.

In the example of Figure 1,this method would discover that i, j and k are constant on entry to *joe* and that b is constant at call site γ although not at call site β. Because procedure *joe* is analyzed to produce J_β^k before the interprocedural propagation takes place, it cannot discover that k is constant at call site β, even though it is directly passed from the entry to β. The overall effect of the propagated information after optimization is shown in Figure 5.

4.2. Pass Through

One way to enhance the all-or-nothing technique is to recognize those situations where a variable is directly passed through a procedure to a call site. This case arises in our example because k is passed by *joe* to call site β. In other words, we would determine that

$$J_\beta^c = k.$$

This is an easy extension to implement because the determination can be based on DEF-USE chains [AhUl 77, Kenn 78], which are required by the single-procedure constant propagation methods. If we trace back from a call site s to all definition

points for variable y, passed to x at s, and find that the only such point is the procedure entry, we may safely set $J_s^x = y$.

This enhancement is only slightly more difficult to implement than the all-or-nothing method, yet it finds constants that are passed through several procedures, a common practice in code based upon libraries like LINPACK. In our example, this method would discover that k was constant at call site β but would set k to at call site γ because the single-procedure analysis must assume that *ralph* could change k as a side effect at call site β. We will discuss this problem shortly.

4.3. Symbolic Interpretation

Suppose we take a much more aggressive approach and build jump functions by symbolic interpretation. A simple way to view the construction of a symbolic J_s^x is as follows.

1) Make a fresh copy of the procedure containing s.
2) Replace all input statements by an assignment of to each of the variables read.
3) Replace each call site other than s by an assignment of to each of the actual parameters.
4) Eliminate all statements that cannot affect the value of the actual parameter passed to x at s using a traditional dead code eliminator based on DEF-USE chains such as the one described by Kennedy [Kenn 81].

In practice, we would use a much more efficient technique for constructing these jump functions, such an an adaptation of the Reif and Lewis symbolic interpretation algorithm [ReLe 82].

Using this technique on the example in Figure 1, we would get the following jump functions.

$$J_\beta^a = 2 * k$$
$$J_\beta^b = \textbf{if } j = 100 \textbf{ then } 1000 \textbf{ else } i \textbf{ fi}$$
$$J_\beta^c = k$$

$$J_\gamma^a =$$
$$J_\gamma^b = 2$$
$$J_\gamma^c =$$

When the algorithm is applied with these jump function, it will discover that a is passed the constant 2000 and c is passed the constant 1000 at call site β. Unfortunately, the information at γ is not as good. Both J_γ^a and J_γ^c are unknown because of the possible side effects at call site β. The problem is that we know nothing about the computations performed by *ralph*. The next section discusses the possibility of using information from the solution of other interprocedural data flow problems to sharpen our analysis.

4.4. Side Effect Information

A particularly interesting aspect of the computation of jump functions is their dependence on the solution of other interprocedural data flow problems. Suppose the procedure p contains a call to another procedure q on every path through p to call site s. How does this affect the jump function for s? In the absence of better information, we must assume that every formal parameter to q and every global variable is changed upon return from q. Hence, any jump function that depends upon one of those variables must be set to .

However, we can use the results of interprocedural side effect analysis to obtain better information. Suppose we can formulate jump functions to conditionally depend on whether certain variables may be modified by some procedure invocation. For example, suppose the value passed to a parameter at call site s depends on whether or not another variable is modified at call site t.

$$J_s^x = \textbf{if } a \in \text{MOD}(t) \textbf{ then } \textbf{ else } a + b \textbf{ fi}$$

where MOD(t) is the set of variables that may be modified as a side effect of invoking the procedure called at t. In the environment, the program compiler develops a complete collection of MOD sets for the program based on information gathered by the source editor and the program composition editor [CoKe 84, CoKT 85, CoKT 86]. This information could be very useful for interprocedural constant propagation. In our example, the jump functions for a and b at γ become:

$$J_\gamma^a = \quad \textbf{if } m \in \text{MOD}(\beta) \textbf{ then}$$
$$\textbf{else if } j = 100 \textbf{ then } 2000 \textbf{ else } 2 * i \textbf{ fi}$$
$$J_\gamma^c = \quad \textbf{if } k \in \text{MOD}(\beta) \textbf{ then } \textbf{ else } k \textbf{ fi}$$

This would lead to the discovery that c was constant at γ and hence $c = 1000$ on entry to *ralph*. In Figure 6, we show the code resulting from these improvements. The assignment in *ralph* has been simplified and, since we know that parameters a

```
procedure main;
   α:  call joe(10,100,1000);
end;

procedure joe(i, j, k);
       m = 1000;
   β:  call ralph(2000, m, 1000);
       o = m * 2;
       q = 2;
   γ:  call ralph(o, q, 1000);
       write q, m, o, 2000;
end;

procedure ralph(a, b, c);
       b = a/2;
end;
```

Figure 6. Using symbolic jump functions.

and c to *ralph* are not modified, it is safe to propagate constant values into the corresponding actual parameter positions at call sites β and γ. Notice also that we can eliminate all references to l in joe by propagating its constant value into the write statement.

It is important to note that the MOD sets for the context program are likely to change between the time the source editor creates the jump function and the time the constant propagation problem is solved. Care must be taken to ensure that the constant propagation phase uses the current value of MOD rather than the old value. If the compiling system performs side effect analysis before constant propagation, up-to-date information about side effects will be available when the jump functions are evaluated. Without the ability to build jump functions that are conditional on MOD information in this way, the editor would be forced to make $J^a_\gamma = J^c_\gamma =$.

4.5. Returned Constants

The inability of the constant propagation algorithm to determine the value of the parameter returned by *ralph* through formal parameter b illustrates a further problem. This situation could be improved by creating the analogue of jump functions for values returned by a procedure call. Let R^x_p be a function, defined over the same lattice as the jump functions, that provides a value for the output parameter x of procedure p in terms of the input parameters to p. We shall call this a *return jump function*. A method based on symbolic interpretation would discover that

$$R^b_{ralph} = a * c/2000.$$

If we then permitted J^a_γ for example 1 to be redefined as

$$J^a_\gamma = \quad \textbf{if } m \in \text{MOD}(\beta) \textbf{ then } R^b_{ralph}(J^a_\beta, J^b_\beta, J^c_\beta) * 2 \textbf{ else if } j = 100 \textbf{ then } 2000 \textbf{ else } 2 * i \textbf{ fi}$$

the technique would discover that a is passed the constant value 2000 at γ as well as β. Hence $a = 2000$ on entry to *ralph* and it always sets $b = 1000$ on exit. Figure 7 depicts the example program after each individual routine has been optimized in the light of interprocedural constants and return jump functions. Since the values of q, m, o and l have been substituted in the write statement, all other statements in *joe* can be eliminated. This result is pleasingly close to the one for inline substitution.

It may occur to the reader that return jump functions might be defined in terms of other return jump functions. There is no reason that this should not permitted as long as the system is careful to insure against infinite invocation loops. In the implementation discussed in the next section, we permit an arbitrary use of return jump functions, relying on the acyclic call graph to limit the

```
procedure main;
   α:  call joe(10,100,1000);
end;

procedure joe(i, j, k);
       write 1000, 1000, 2000, 2000;
end;

procedure ralph(a, b, c);
       b = 1000;
end;
```

Figure 7. Using return jump functions.

number of invocations.

Clearly the return jump functions can also be very useful in optimizing code for a single module. They provide a clean representation of the effect of executing a procedure in the presence of values specific to the call site. Because of this, they can be used to provide some of the benefits of inline expansion. We plan to evaluate their usefulness in the optimizing module compiler for the
environment.

5. Implementation

We have implemented essentially the full symbolic constant propagation system using both side effect and returned constant information in the Rice vectorizing compiler system, called PFC [AlKe 84]. In this section we describe some of the details of that implementation.

PFC is a monolithic system which accepts all the modules of a program and analyzes them in two passes. The first pass builds the call graph and records the local information needed to support side effect analysis, alias analysis and constant propagation. The next pass computes interprocedural information by solving data flow problems on the program's call graph. Finally, each procedure is individually vectorized using the computed interprocedural information.

In propagating constants, the jump functions are represented by expression trees, in which interior nodes are labeled with arithmetic operators and leaves are labeled with constants and variables. The variables are either scalar formal parameters or scalar COMMON variables. These expressions are initially built after parsing but before interprocedural side effect analysis information or aliasing information is available.

The expressions also contain two types of ''dummy'' operator nodes. The first type represents a point where the ''value'' computed by the subtree of the dummy node was passed to a subroutine call. It consists of the index of an edge in the call graph representing that call site and an integer indicating which parameter position it held. The presence of such an operator indicates that the interprocedural MOD information must be interrogated to determine whether that value is modified as a side effect of the call.

The second type of dummy operator indicates that a parameter or COMMON variable was used in constructing the expression tree. Expressions containing this operator can be ''invalidated'' (or made nonconstant) if later passes reveal that the variable is aliased. Even though the jump functions are constructed as a part of a monolithic process that examines the whole program at once, it is useful to arrange the evaluation of the jump function to interrogate the interprocedural MOD and alias information because it avoids an extra pass over the source of the individual modules. If this were not done, PFC would need to reread the source after computing MOD and alias information to construct the jump functions. Hence, the method designed for use in a programming environment is also suitable for a more traditional compiling framework.

In addition to jump functions that map from constants available upon procedure entry to values available at call sites, return jump functions are also constructed. In addition to the advantages already stated, this permits a natural treatment of BLOCK DATA subroutines in Fortran. If special dummy calls to the BLOCK DATA routines are inserted at the beginning of each procedure that uses the COMMON blocks they initialize, return jump functions provide a rich source of constants to be propagated throughout the program. Since BLOCK DATA subroutines are the only way to initialize common blocks in Fortran, this is a particularly useful technique for that language.

Once MOD information is available, all of the expression trees are traversed and if the first type of dummy node is encountered, then the call site and parameter position are checked to see if the actual parameter could be modified by the call site. If so, that subtree is replaced by the output expression of the routine with the variables in the output expression replaced with appropriate input values at the call site. A side benefit of this approach is that if a subroutine initializes some constants, those constant values will be detected and propagated to other routines.

In this implementation, strongly-connected regions in the call graph are identified to support the algorithm for computing interprocedural side effects. For most Fortran 77 programs, the call graph will be acyclic, permitting the procedures to be processed in reverse topological order, sometimes called *reverse invocation order* [Alle 74], to derive a solution to the side effect problem in

linear time. The use of reverse invocation order also benefits constant propagation because jump functions need not be checked any further when a dummy node is replaced with the output expression corresponding to a procedure invocation. Parameters modified inside a recursive region are assumed to be variant. Any variable involved in an alias with a variable which is modified is assumed to be nonconstant.

Once the effects of aliasing and the side effects of external procedure calls have been incorporated into the expression trees representing the jump functions, the actual propagation is straightforward. The strongly connected regions are visited in topological order and each procedure in each region is processed. If every incoming edge assigns a variable the same constant value, then that variable is deemed constant with that value at procedure invocation. Otherwise the variable is assigned the special value . Finally, each edge in the procedure is visited and the expression trees evaluated to yield values for the associated parameters.

This implementation differs from the general algorithm of Section 3 in that it will not identify constants that are passed into a recursive region then passed around a set of recursive calls. This is because the implementation was undertaken before we discovered the ‘‘optimistic’’ algorithm of Figure 4. A straightforward revision of the implementation to use the optimistic algorithm is underway. In any case, this deficiency is less important for Fortran, because the current standard precludes dynamic recursion.

We have not yet tried the system on a large variety of programs, so it would be premature to report any empirical evidence about the value of interprocedural constant propagation. We plan to report these results as a part of a general study of interprocedural data flow analysis in PFC.

6. Conclusions

We have presented an efficient procedure for computing interprocedural constants in a programming environment. The method is based on an algorithm adapted from the single-procedure methods of Reif and Lewis [ReLe 82] and Wegman and Zadeck [WeZa 85]. The approach is linear in the size of the call graph if we assume that the number of input parameters to each procedure is bounded, that the jump functions all have bounded support, and that the evaluation time for each jump function is bounded by a constant.

An experimental implementation in the Rice vectorization system PFC has established the practicality of the approach. We are currently implementing the algorithm in the program compiler of the programming environment. We believe that the technique presented here will approach the method of inline substitution for effectiveness. We expect to evaluate this algorithm in two ways: directly in our own compiler and by using as a preprocessor to generate modified source for compilation by existing optimizing compilers like IBM’s VS Fortran and DEC’s VMS Fortran.

There is one important extension to the technique presented here. The constant propagation lattice can be replaced by any lattice of bounded depth, yielding an algorithm that might be used for other purposes such as propagating inequalities. Such information could be used to improve the performance of vectorization systems like PFC, in which the knowledge that a parameter is greater than ‘‘0’’ or greater than ‘‘1’’ might make an otherwise unsafe transformation possible [AlKe 84].

7. References

[AhUl 77] A. Aho and J. Ullman. Principles of Compiler Design. Addison-Wesley. 1977.

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

[AlKe 84] J. R. Allen and K. Kennedy. PFC: a program to convert Fortran to parallel form. *Supercomputers: Design and Applications (K. Hwang, ed.). IEEE Computer Society Press, 1984.*

[Ball 79] J. E. Ball. Predicting the effects of optimization on a procedure body. *Proceedings of SIGPLAN ’79 Symposium on Compiler Construction*, SIGPLAN Notices, 14(8). 1979.

[Coop 83] K. D. Cooper. Interprocedural information in a programming environment. Ph.D. Dissertation, Department of Mathematical Sciences, Rice University, Houston,

TX. May 1983.

[Coop 85] K. D. Cooper. Analyzing aliases of reference formal parameters. *Proceedings of Twelfth POPL*. 1985.

[CoKe 84] K. D. Cooper and K. Kennedy. Efficient computation of flow insensitive interprocedural summary information. *Proceedings of SIGPLAN '84 Symposium on Compiler Construction*, SIGPLAN Notices, 19(6). 1984.

[CoKT 85] K. D. Cooper, K. Kennedy, and L. Torczon. The impact of interprocedural analysis and optimization on the design of a software development environment. *Proceedings of SIGPLAN '85 Symposium on Language Issues in Programming Environment*, SIGPLAN Notices, 20(7). July 1985.

[CoKT 86] K. D. Cooper, K. Kennedy and L. Torczon. Optimization of compiled code in the programming environment. *Proceedings of the Nineteenth Annual Hawaii International Conference on Systems Sciences*. January, 1986.

[Dong 80] J. Dongarra. LINPACK working note #3: FORTRAN BLAS timing. Technical Report ANL-80-24, Argonne National Laboratory, February 1980.

[DBMS 79] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK Users' Guide*. SIAM, Philadelphia. 1979.

[HoKe 85] R. T. Hood and K. Kennedy. A programming environment for Fortran. *Proceedings of the Eighteenth Annual Hawaii International Conference on Systems Sciences*, 1985.

[KaUl 77] J. Kam and J. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7. 1977.

[Kenn 78] K. Kennedy. Use-definition chains with applications. *J. Computer Languages*, 3(3). 1978.

[Kenn 81] K. Kennedy. A survey of data flow analysis techniques. *Program Flow Analysis: Theory and Applications (S.S. Muchnick and N.D. Jones, eds.)*. Prentice-Hall. 1981. pp. 5-54.

[Kild 73] G. Kildall. A unified approach to global program optimization. *Proceedings of First POPL*. 1973.

[Myer 81] E. W. Myers. A precise interprocedural data flow algorithm. *Proceedings of Eighth POPL*. 1981.

[ReLe 82] J. H. Reif and H. R. Lewis. Symbolic evaluation and the global value graph. TR 37-82, Aiken Computation Laboratory, Harvard University. 1982.

[Sche 77] R. W. Scheifler. An analysis of inline substitution for a structured programming language. *Communications of the ACM*, 20(9). 1977.

[Torc 85] L. Torczon. Compilation dependences in an ambitious optimizing compiler. Ph.D. Dissertation, Department of Computer Science, Rice University, Houston, TX. May 1985.

[WeZa 85] M. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *Proceedings of Twelfth POPL*. 1985.