The Impact of Interprocedural Analysis and Optimization in the $\mathbb{R}^n$ Programming Environment

Keith D. Cooper
Ken Kennedy
Linda Torczon

Department of Computer Science[1]
Rice University
Houston, TX

Abstract

In spite of substantial progress in the theory of interprocedural data flow analysis, few practical compiling systems can afford to apply it to produce more efficient object programs. To perform interprocedural analysis, a compiler needs not only the source code of the module being compiled, but also information about the side effects of every procedure in the program containing that module, even separately compiled procedures. In a conventional batch compiler system, the increase in compilation time required to gather this information would make the whole process impractical. In an integrated programming environment, however, other tools can cooperate with the compiler to compute the necessary interprocedural information *incrementally* as the program is being developed, decreasing both the overall cost of the analysis and the cost of individual compilations.

A central goal of the $\mathbb{R}^n$ project at Rice University is to construct a prototype software development environment that is designed to build whole programs, rather than just individual modules. It employs interprocedural analysis and optimization to produce high-quality machine code for whole programs. This paper presents an overview of the methods used by the environment to accomplish this task and discusses the impact of these methods on the various environment components. The responsibilities of each component of the environment for the preparation and use of interprocedural information are presented in detail.

Categories and Subject Descriptors: D.2.6 [**Software Engineering**]: Programming Environments; D.3.4 [**Programming Languages**]: Processors — *compilers, interpreters, optimization*.

General Terms: Languages.

Additional Keywords and Phrases: data flow analysis

1. Introduction

Calls to separately compiled procedures are one of the principal remaining impediments to generating efficient object code for programs written in Algol-like languages. Global optimizing compilers exist for many systems; these do an excellent job of generating code for a single subroutine. Unfortunately, the quality of the code they generate tends to decline in the presence of procedure calls. This happens for several reasons. In the absence of better information, the compiler must assume that the called subroutine will use and change every variable accessible to it — in other words, every parameter and every global variable. This assumption inhibits a number of optimizations; for example, the value of an expression computed before a procedure call cannot be used to eliminate an occurrence of the same expression beyond the call if it involves a variable accessible to the called routine.

[1]This work has been supported by the National Science Foundation and IBM Corporation.

Separate compilation also prevents optimizations that would be easy if all the subroutines of a program were compiled together. For example, a common practice in library routines is to pass constants as parameters to procedures, particularly when the parameter is used to select an option. In the LINPACK library, almost every subroutine has a parameter to indicate the stride of indexing for some array. Typically, this stride is passed the integer constant ''1'', information that would be enormously useful to the compiler in optimizing the called routine [19].

If the problems caused by lack of information about separately-compiled procedures are troublesome for traditional compilers, the difficulties are magnified when the compiler is attempting the more sophisticated transformations required to support vector and parallel supercomputers. In such a compiler, lack of knowledge about the program in which a subroutine will execute can make an order of magnitude difference in the running time of certain sections of code [4, 27].

In spite of these problems, no commercially available optimizing compilers employ interprocedural analysis for separately-compiled procedures because the cost of gathering the requisite information is too great in a traditional compiling scheme. Computing the side effects of a procedure call requires detailed knowledge of the internals of both the called procedure and any procedures invoked either directly or indirectly from it. Thus, to determine the side effects of a single procedure call, the compiler potentially needs information about the internals of every procedure in the program, even procedures that are separately compiled. Gathering this information requires examining the source of every procedure in the program - an expensive process. If the compiler cannot store information gathered in one compilation for later use in other compilations, it must re-analyze the entire program in order to compile any of its constituent procedures. This is particularly unfortunate since the primary goal of separate compilation is to reduce the amount of recompilation required in response to changes in an individual procedure.

One of the primary goals of the $\mathbb{R}^n$ programming environment project [26] is to mount a concerted attack on the problems of performing interprocedural analysis and optimization. Work on the environment has included the development of new algorithms and techniques for interprocedural analysis as well as development of a programming environment for Fortran [5] that serves as a test bed for the compiler. While the environment supports Fortran, the algorithms have been designed to support more general Algol-like languages. In fact, $\mathbb{R}^n$ supports a recursive dialect of Fortran that includes several extensions proposed for Fortran 8x [6].

A software development environment like $\mathbb{R}^n$ reorganizes the compilation process in a way that makes computing such information palatable. Since all modules are developed and all programs are defined using tools of the environment, these tools can cooperate to record the information necessary to do a good job of interprocedural analysis and optimization. Whenever the optimizing compiler needs information about possible side effects of a particular procedure, it can simply extract the information from the environment's central database with the assurance that it accurately reflects the current state of the program and its procedures. Because information is retained between compilations, the compiler can use efficient incremental updating techniques to produce correct and consistent interprocedural information. Since the compiler doesn't recompute all the interprocedural information at every compilation, the analysis has a much lower total cost than possible in a traditional batch compiler.

From the outset, the $\mathbb{R}^n$ programming environment has been designed to record and use interprocedural information to support compiler optimization and to help the programmer produce programs with fewer errors. The pervasive influence of the decision to apply interprocedural techniques has been one of the major surprises in the $\mathbb{R}^n$ project. This paper discusses the ways that the various components of the environment implement this design. Section 2 presents an overview of the environment design, highlighting the principal design decisions that led to the current structure. Section 3 summarizes the algorithms used for interprocedural analysis and optimization. Section 4 presents *recompilation analysis*, a technique used to reduce the frequency of spurious recompilations. Section 5 details the responsibilities of each environment component in supporting interprocedural analysis and discusses the applications of the resulting information throughout the system.

2. The IR^n Programming Environment

For the IR^n environment to be effective as a tool for research in the construction of optimizing compilers, it must also be successful as a programming support tool. Since numerical programmers are primarily interested in problem-solving they are unlikely to tolerate a clumsy programming system for the sake of research. Furthermore, a new system must be notably superior to the currently-used programming tools to induce programmers to switch. Therefore, the IR^n environment has been designed as an integrated collection of tools for the development of *whole programs* rather than individual source modules. Of course, IR^n includes a structure editor for developing Fortran source. Additionally, it provides an editor for composing collections of modules into programs. Like most programming environments, IR^n provides an optimizing compiler for individual source modules. Additionally, it includes a compiler that optimizes whole programs. Because all of these tools cooperate in information gathering, the resulting system proves more helpful to the user and the resulting programs more efficient.

The environment is built as a collection of command processors that run cooperatively under the IR^n *monitor*. The monitor controls interactions between the command processors and provides primitives for handling the mouse, bit-mapped screen, keyboard, menus, and windows. The individual command processors record and use information in the IR^n *database*. The database holds information about programs and procedures built and maintained using the environment. It also serves as the primary mechanism for communication between tools.

Within the database, two types of program objects coexist. A *module* is an editable unit of source text. A typical module should contain one or more Fortran subprograms. A *composition* is a structured collection of pointers to modules that defines a composite object. Programs, libraries, and compound modules are all defined by compositions in IR^n.

The environment includes a structure editor for Fortran source text, called the *module editor*. It helps the programmer enter syntactically correct programs by providing templates for the major language constructs. For example, to insert a DO-loop, the programmer need only invoke the DO-loop command and the cursor is replaced by a DO-loop template with place markers in the positions where further text should be entered. The module editor obviates the need for a parser by directly constructing an abstract syntax tree for the module. Such trees are the standard program representation used by tools in the environment. Recognizing that not all programmers will use a structure editor, the environment also provides an alternate path for entering source text; the module is unparsed, edited using the documentation editor, and then reparsed and stored in intermediate form.

The module editor makes use of information stored in the IR^n database to make it easier to construct subprograms that are consistent with the program being developed. For example, when a programmer wishes to insert a call to an external subroutine, the editor queries the database to retrieve the parameter specifications for the called routine. It uses this information to construct a template for the call in which the parameter placeholders identify the name and type of each parameter to be inserted.

In keeping with our commitment to supporting the development of whole programs, the environment must provide some mechanism for defining and modifying the structure of a program. This mechanism is the *composition editor*. The composition editor permits the user to define a structured collection of modules with consistent interfaces. It is a structure editor for a module interconnection and configuration management language. This language is somewhat unusual in that it was never intended for use outside of the composition editor, so it has a display-oriented concrete syntax. The combination of editor and language subsumes the functions of the linkage editor input language in traditional systems and the module interconnection and configuration management languages found in modern programming systems [18, 31, 32, 34, 43].

The composition editor helps the user construct a list of components, each of which is a module or a sublist. Each sublist is itself a composition, exporting and importing names and their associated definitions. Within each list, the composition editor checks for consistent interfaces and completeness with respect to required definitions. The composition editor also provides library search and automatic completion facilities. When the desired composition differs only slightly from an existing one, the composition editor simplifies the specification process by providing the ability to edit an existing composition.

A complete composition with a main entry point constitutes a *program*. To support optimization of whole programs, the $\mathbb{R}^n$ environment must incorporate a tool to manage the process of preparing a program for efficient execution. Hence, the system includes a *program compiler* responsible for directing the construction of an executable image for a program. The program compiler is the principal innovation of the optimization strategy for $\mathbb{R}^n$. By acknowledging that we wish to compile efficient code for whole programs, we are led immediately to the need for a tool to oversee the process.

The program compiler computes the interprocedural information needed to efficiently compile the individual modules within a program and directs the module compiler to carry out the various interprocedural optimizations. In the tradition of *make* [23] and its successors [44], it also automates the process of reconstruction after a change. Finally, the program compiler directly performs certain optimizations required to make the program construction efficient, such as elimination of duplicate modules specified in the program composition.

To produce code for an individual module, the program compiler invokes the *module compiler*, which comprises the optimization and code generation phases of a typical optimizing compiler. It differs from traditional compilers primarily in its use of interprocedural information provided by the program compiler to do a more thorough job of optimization.

The environment includes two command processors for executing programs. The *execution manager* permits the execution of a compiled program within an $\mathbb{R}^n$ window. The *interpretive debugger* enables the programmer to step through parts of a given program, allowing him to monitor and interrupt execution. By cooperating with the compiler, the debugger supports execution of hybrid programs that consist of both compiled and interpreted modules. During debugging, the programmer can execute compiled versions of stable modules, while interpreting the modules under development. This makes interpretive debugging more practical for large programs because control passes quickly through those parts of the program that are not material to the debugging process, slowing down to bring the full power of the interpretive debugger to bear on those parts of the program where it is needed.

The $\mathbb{R}^n$ environment includes a wide variety of other tools, including a text editor, a data base browser and help processor, but these are not directly related to the subject of this article and will not be further discussed. For additional information on the $\mathbb{R}^n$ programming environment, the reader should see the survey article by Hood and Kennedy [26].

3. Interprocedural Information

The $\mathbb{R}^n$ programming environment computes and uses three distinct types of interprocedural information: potential side effects of procedure calls, sets of variable names that can refer to the same data item, and information about formal parameters that have constant values determinable at compile time. In this section, we introduce each of these interprocedural data flow problems and outline techniques for solving them in a programming environment.

When an optimizing compiler encounters a procedure call, it must account for the effect of executing the call on the values of variables accessible in the calling procedure. If it does not have accurate knowledge about the internals of the called procedure readily available, the compiler must assume that every variable accessible to the called procedure is both used and modified as a side effect of executing the call. This is the most *conservative* assumption possible, since it inhibits any optimizations that might rely on a variable or value being unchanged or unused across a procedure call. In $\mathbb{R}^n$, the environment uses interprocedural data flow analysis to precisely compute the set of variables that may be modified and the set of variables that may be used as a result of executing a procedure call. These *side effect summary sets* are not only useful for improving the optimization of individual compiled modules, they can also permit the editor to give better diagnostics. For example, if the programmer passes a constant to a formal parameter that may be changed, the editor will warn him of the possible side effect.

The program compiler also computes information about aliasing relationships between formal parameters and global variables. Whenever a procedure can access a single storage location using more than one name, those names are said to be *aliases*. It is essential that the module compiler understand which names are potential aliases; in the absence of such information it must treat *every* global variable and *every* formal

parameter as if they were potential aliases. This implies, for example, that their values cannot be retained in registers across statement boundaries.[2]

To understand this, consider the following sequence of assignment statements:

```
a = 10
b = 12
c = a * b
```

In the absence of aliasing, the compiler would probably retain the values of a and b in registers, since they are referenced almost immediately after being assigned values. If, however, the procedure containing this code fragment can be invoked in a manner such that a and b refer to a single storage location, this simple optimization to eliminate apparently redundant loads and stores would result in *incorrectly* assigning the value 120 to c.

The widespread use of constant parameters in programs that incorporate routines from libraries like LINPACK [20] has underscored the need for determining which parameters and global variables are known constants when a procedure is invoked. The $\mathbb{R}^n$ environment computes a conservative approximation to the set of all variables that are known constants on input to each procedure in a given program. This information is not only useful in optimizing a particular procedure, it can also help in the determination of whether it is profitable to perform procedure integration [7, 14, 46].

3.1. General Solution Strategy

The flow of information in the solution of interprocedural data flow problems in the $\mathbb{R}^n$ environment is depicted in Figure 1. The process begins with the module editor, which collects *local information* needed for the development of interprocedural information. This local information includes a specification of the entry points provided by the module, with parameter names and types, a listing of the entry points called from within the module, along with actual parameter names and types for each, and a specification of which variables may be used or modified locally. In addition, the editor performs some preliminary computations to support interprocedural constant propagation, determining which actual parameters at a call site are known constants and which can be expressed as compile time functions of parameters to the calling subprogram. The composition editor provides the remainder of the information required for the analysis.

Using these inputs, the program compiler computes the desired interprocedural information. First, it builds the program's call graph[3] from the specification of which entries are defined and called by each module in the program composition. Then it solves data flow problems over the call graph to compute the

---

Figures are provided on separate pages

Figure 1. Flow of interprocedural information in the environment.

---

[2]Strictly speaking, the Fortran standard permits the compiler to ignore aliasing. The standard contains a restriction that neither of two aliases may be modified in a standard-conforming program [5]. Nevertheless, the $\mathbb{R}^n$ project attempts to trace aliases because information about potential aliases can be useful as a diagnostic aid to the programmer and because we wanted our system to achieve a higher level of predictability than required by the standard.

[3]The call graph is actually a multi-graph, with an edge for every call site. For the sake of consistency with the literature and readability, we refer to it as a graph. Throughout this paper we use *call graph edge* and *call site* interchangeably, where an edge *e=(p,q)* of the call multi-graph represents a call from a site within *p* to procedure *q*.

needed information. Finally, it stores the interprocedural information in the data base for later use by the compiler or by subsequent editing sessions on modules in the program. Note that the solutions can be completely developed *without ever invoking the module compiler* — the module editor and the composition editor provide everything that is needed. This property is an important design goal for the data flow algorithms; if they required input from the module compiler, complex circular dependencies could make the problems intractable.

The program compiler must also be capable of incrementally updating the solution of a data flow problem in response to an editing change. If the interprocedural information is to be useful, it must be updated in response to every change to any module in the program. Thus, whenever a module incorporated in a program is put away after editing, the program compiler uses incremental techniques to update the program's interprocedural information.

The next three subsections introduce the data flow problems that must be solved for each of the three classes of information developed in the $\mathbb{R}^n$ environment and give an overview of the algorithms used for solving them. The last subsection describes the interprocedural optimization techniques we intend to try in $\mathbb{R}^n$.

3.2. Side Effects

The $\mathbb{R}^n$ program compiler computes *side effect summary sets* that precisely describe the possible side effects of each call site in the program. This information consists of two summary sets for each call site e in the call graph:

1) the set MOD(e) of all variables whose values may be changed as a result of executing the procedure call, and

2) the set USE(e) of all variables whose values may be used before control returns from the called procedure.

The MOD and USE sets are said to be *flow insensitive* because they describe events that *may* happen. While the presence of a variable x in MOD(e) implies that executing e can change the value of x, it does not assert that every execution of e will change x. By contrast, the problem of determining which events must always happen as a side effect of a procedure call requires careful analysis of the control flow within each procedure. Such problems are called *flow sensitive* problems. Myers has shown that solving flow sensitive interprocedural data flow problems is Co-NP complete in the presence of aliasing [36]. Because flow insensitive summary sets do not depend on the internal control flow structures of individual procedures, they can be computed more efficiently. The $\mathbb{R}^n$ environment computes only flow insensitive interprocedural information.

Consider the problem of computing MOD(s) for each statement s. For most statements, MOD(s) can be determined by a simple inspection. If, however, s contains a procedure call, the problem is more complex. Any variable that is passed as a parameter to the called procedure or any variable that is global to the called procedure is a candidate for MOD(s).

To compute a reasonably precise MOD(s), we need to identify those variables that might be changed, directly or indirectly, by the procedure called from s. Let GMOD(q) be the *generalized modification* set for a procedure q. It contains precisely those variables that may be modified by executing q, identified by the names under which they are accessible to q. Given GMOD(q) for each procedure q in the program, MOD(e) for a specific invocation of q can be computed by reversing the effects of the parameter binding and name scoping rules at the call site. MOD(e) is simply the set of variables accessible at the call site that can be bound to names in GMOD(q) by the call. Then, MOD(s) can be computed as MOD(e) combined with any variables modified locally in s.

Two distinct mechanisms place variables in GMOD(q). Either q modifies the variable locally, independent of any procedure calls made inside q, or some other procedure invoked by q modifies the variable. Thus, we can view GMOD(q) as having two components:

- the set IMOD(q) of variables that might be modified by statements in q other than procedure calls and
- the set of variables that might be modified as a side effect of executing a procedure call within q.

IMOD(q) is the *immediate modification* set for procedure q. For each procedure q, IMOD(q) can be computed by an inspection of the source for q — it is independent of the other procedures in the program. Hence, it can be computed by the editor and stored with the module that defines q.

The set of variables that might be modified as a side effect of a call to procedure q is completely determined by IMOD(q) and the GMOD(q) sets for procedures called from within q. This leads to the following system of equations:

$$\text{GMOD}(p) = \text{IMOD}(p) \cup \bigcup_{e=(p,q)\in \text{E}} f_e^{-1}(\text{GMOD}(q))$$

where f_e maps an actual parameter at a call site into the corresponding formal parameter of the called procedure. It maps global variables at the call site to the corresponding global variables of the called procedure. These equations can be directly solved as a backwards data flow problem over the program's call graph. Any of the standard algorithms from global data flow analysis can be adapted to this task. The USE sets can be computed by solving a similar system of equations.

Unfortunately, directly applying standard data flow algorithms to this system of flow equations is inefficient. Because the binding function f_e can arbitrarily bind actual parameters to formal parameters, it introduces a level of complexity to the problem that prevents global data flow algorithms from achieving their fast time bounds on an instance of the problem. Iterative methods, for example, may have to iterate around the call graph many times before the solution converges. For this reason, solution techniques based upon transitive closure or classical data flow analysis techniques [9, 8, 35, 40] achieve a worst case time bound no better than **O**(EN), where E is the number of edges in the call graph and N is the number of vertices [14][4]. As part of designing the data flow analysis phases for the $\mathbb{R}^n$ environment, we have developed an algorithm for computing the GMOD(q) sets in time **O**(Eα(E,N)), where α is a functional inverse of Ackermann's function [16]. This is the best known time bound for this problem.

To achieve this bound, we split the problem into two subproblems, one for global variables and the other for call-by-reference formal parameters. The global variable subproblem gives rise to a very simple system of data flow analysis equations that can be solved using any of the fast algorithms for analysis of a single procedure [25, 24, 30].

The formal parameter subproblem inherits most of the complications that make the summary set computation hard. To efficiently solve it, we first reduce it to the problem of computing a relation map^* *that describes the parameter binding chains that occur in the program call graph. A pair <a,b>* is in map^* *if and only if both a* and *b* are formal parameters and there is a chain of procedure calls in the program that results in *a* being bound to *b*. An actual parameter *w* corresponding to a formal parameter *x* of procedure *p* may be modified if there exists some formal parameter *y* of another procedure *q* such that *x* map^* *y* and $y \in$ IMOD(q). map^* *can be thought of as the reflexive transitive closure of* f_e *taken over all the paths in the program. Fortunately, we can compute* map^* *in a particularly efficient manner.*

Using a result of Tarjan [41], we know that if we can reduce the problem to a single-source path expression problem on the program's call graph we can use any of the standard elimination techniques for intraprocedural data flow analysis [24, 30]. In particular, we can use an algorithm also due to Tarjan that solves single-source path expression problems in time **O**(Eα(E,N)) [42].

The desired reduction is achieved by demonstrating how to compute map^* *for paths described by regular expressions in time proportional to the length of a single bit-vector, which is equal to the total number of formal parameters of procedures in the program. One must show how to compute* map^* *for the union, concatenation, and reflexive transitive closure of paths for which we already know* map^*. *We*

[4]Throughout this paper, the time bounds are expressed in terms of *bit vector* steps [1].

represent map* as a bit matrix whose dimension matches the total number of parameters in the program. If we assume that the number of parameters to any single procedure in the program is bounded by some constant K, the contribution of any single path through the program to map* can be represented by a $K \times K$ matrix.[5] *In this case, all three operations (union, concatenation, and transitive closure) involve multiplying $K \times K$ matrices, which can be done in constant time [16].*

The GMOD sets for each procedure are produced by simply taking the union of the results of the two analyses. For each edge $e = (p, q)$, MOD(e) is formed by applying f_e^{-1} to GMOD(q) and adding in any aliases that can hold on entry to the calling procedure. A more detailed treatment of this algorithm has appeared in other papers [14, 16]. Burke has shown that the same approach works in an interval analysis framework [11].

The summary set computation illustrates the general strategy introduced in Section 3.1. Because the IMOD sets contain purely local side effects, they can be computed in the module editor and stored with the module. Thus, a procedure's IMOD set is computed once per editing change to the module, rather than once per compilation of each program containing the module. The GMOD computation is program specific, so it is done in the program compiler. The resulting information must be stored with the program; even the sets for individual call sites are stored with the program rather than with the module containing the site.

3.3. Aliasing

To avoid generating incorrect code for a module that is passed two different names for the same data location, most optimizing compilers treat all formal parameters and all global variables as if they were potential aliases. If the compiler has precise knowledge about the aliasing patterns that can actually occur in a program, it can restrict the set of variables treated conservatively to those that can actually be aliased. The $\mathbb{R}^n$ environment annotates each procedure p with a set ALIAS(p), that contains all the aliases that may hold on entry to p. Each potential alias is represented by a pair <x,y> of variable names. A pair <x,y> in ALIAS(p) indicates that some chain of call sites in the program leads to an invocation of p where both x and y are names for the same storage location.

To solve the aliasing problem, we break it down into two separate problems: detecting the creation, or *introduction*, of aliases, and following their *propagation* around the call graph. The introduction problem can be solved by examining the source text for call sites. Thus, the module editor annotates each call site e with a set INTRO(e) containing all of the potential aliases introduced at the call site. To create INTRO(e), the editor must detect cases when a single variable is passed in multiple parameter positions or when a global variable is passed as an actual parameter.

The alias propagation problem can be formulated as a forward data flow analysis problem over the program's call graph. The ALIAS sets for a program's procedures can be found by solving the following equations:

$$\text{ALIAS}(q) = \bigcup_{e=(p,q)\in \text{E}} (\text{INTRO}(e) \cup b_e(\text{ALIAS}(p)))$$

where b_e is a function that models the pairwise parameter binding involved in propagating aliases around the call graph. It can be defined as:

$$b_e(X) = \bigcup_{<x,y>\in X} \{ <x',y'> \mid x' = f_e(x) \text{ and } y' = f_e(y) \}$$

where f_e is the binding function for call site e introduced in the previous section. It maps actual parameters to formal parameters at the call site.

These equations can be solved using any of the standard algorithms from global data flow analysis [30, 15]. In particular, the $\mathbf{O}(E\alpha(E,N))$ technique described in the previous section can be used. That

[5] Although we might expect the total number of parameters in a program to grow as the program grows, we should not expect the number of parameters to any single routine to grow without bound. Indeed, many compilers place a limit on this number.

algorithm produces a mapping of formal parameter bindings. We can compute an analogous mapping for pairs of parameters and use it to propagate the alias pairs from a call site's INTRO set directly to every procedure that can inherit both variables in the pair along a single path. Thus, the same basic algorithm can compute ALIAS sets, albeit with a larger constant of proportionality since pairs of parameters are being tracked.

Again, the work involved in computing this interprocedural information is distributed between the module editor and the program compiler, in keeping with the general strategy. The INTRO sets for each call site are the result of purely local properties of the calling procedure so they are computed in the module editor and stored with the individual module. Based on local properties of the call site, the module editor can also detect call sites that are irrelevant to later propagation analysis, effectively pruning the size of the call graph for the problem. Since propagation analysis requires knowledge of the entire program, the program compiler performs it and stores the resulting ALIAS sets with the program.

3.4. Constant Propagation

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 is an approximation; the problem of finding all variables that are constant at run time is undecidable [29] and the approximate constant propagation problem used in typical optimizing compilers is flow sensitive, hence intractable, in an interprocedural setting [36].

For the purpose of computing CONSTANTS(p) we formulate the problem in a slightly different form. Let us associate with each entry point p a function *Val* that maps formal parameters to elements of the usual *constant propagation lattice* L, in which each element is one of three types:

- a lattice *top* element $\top$,
- a lattice *bottom* element $\bot$, 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) $\top \wedge a = a$ for any lattice element a
2) $\bot \wedge a = \bot$ for any lattice element a
3) $c \wedge c = c$ for any constant c
4) $c_1 \wedge c_2 = \bot$ if $c_1 \neq c_2$

The constant propagation lattice is depicted graphically in Figure 2. 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 $Val(x) = \bot$, the parameter is known to be non-constant; if $Val(x) = c$, the parameter has the constant value c. The value $\top$ 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

---

Figures are provided on separate pages

Figure 2. The constant propagation lattice.

Figures are provided on separate pages

Figure 3. The constant propagation algorithm

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*[6] J_s that gives the value of each parameter at s as a function of the formal parameters of the procedure p[7]. J_s is actually a vector of functions, one for each formal parameter of the procedure invoked at the call site. If y is a formal parameter 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 [46]. In this approach, the call graph plays the role of the use-definition chain graph and the jump functions play the role of the individual instructions. The Wegman-Zadeck method cannot be applied directly to the whole program because constructing interprocedural use-definition chains requires the solution of a flow sensitive data flow problem. As stated earlier, such problems are intractable in the interprocedural setting. Thus, some approximate technique is required. The jump functions provide a range of approximations of different precisions.

The interprocedural algorithm, shown in Figure 3, 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 bounded by a constant, the cost is proportional to the sum over each edge in the call graph of the number of parameters bound by that edge.

The tricky part of this method is the construction of the jump functions J_s. Our approach to constructing these functions is based upon the following principles:

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

[7]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.

- For any parameter x that can be determined by inspection of the source of the procedure containing s to be constant at call site s, set $J_s^x = c$ where c is the known constant value. This implies that $support(J_s^x) = \emptyset$.
- 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 $\bot$. For example, x might be passed a value that is read in from an external medium. Again, this implies that $support(J_s^x) = \emptyset$.
- For any parameter x at call site s that is neither constant nor bottom, 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 module 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 $J_s^x = \bot$. Torczon discusses three strategies of different complexity for developing jump functions [45] — 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 [38].

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 any formal parameter to q and any global variable is changed upon return from q and, hence any jump function that depends upon one of those variables must be set to $\bot$.

However, through side effect analysis, we can 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.

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

It is important to note that the MOD sets for the context program are likely to change between the time the module 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. Hence, if the program compiler performs side effect analysis before constant propagation, the jump function can read information about side effects from the database and produce a more precise solution to the constant propagation problem. Without the ability to build jump functions that are conditional on MOD information in this way, the editor would be forced to make $J_s^x = \bot$. A detailed discussion of different jump function implementation techniques may be found in the paper by Callahan, Cooper, Kennedy and Torczon [12].

3.5. Interprocedural Optimization

The $\mathbb{R}^n$ programming environment will capitalize on the presence of interprocedural information and module source code in the database to perform optimizations that span procedure boundaries. The goal of interprocedural optimization is to tailor the code generated for a procedure to execute more efficiently by taking into account, at compile time, facts about the run-time environment in which the procedure will be executed. The module compiler uses stored knowledge about the calling procedure and the values of its variables at the call site, along with information about the side effects of any call sites in the procedure being compiled, to generate less general, more efficient code.

3.5.1. In-line Substitution

The most straightforward interprocedural optimization is in-line substitution. While this type of optimization has been discussed in the literature since the mid-1960's [21], few practical compiling systems

have actually implemented it. To perform in-line substitution, the compiler simply treats the procedure body as if it were a macro definition and the call site as an instantiation of the macro. Actual parameters are used in place of formal parameters in the expanded body, and local storage for the called procedure is merged with the local storage for the calling procedure.[8] This transformation completely eliminates the overhead associated with making the procedure call. When followed by a decent global optimization pass on the expanded procedure, in-line substitution results in optimizing both procedures together, producing the best possible code for them within the given compiler.

For non-recursive procedures called from only one place, or those whose compiled body is smaller than the linkage code, in-line substitution always produces an improvement. In the general case, however, the optimization benefits must be weighed against the increased object code size resulting from multiple copies of a single procedure.

3.5.2. Linkage Tailoring

Strictly speaking, in-line substitution is a type of linkage tailoring. However, there are a number of less radical forms. For example, the compiler might use *semi-closed* or *semi-open* procedure linkages [2]. In a semi-closed linkage, the compiler moves some, but not all, of the implementation of the called procedure into the calling procedure. By moving more of the linkage code into the calling procedure, it can be better tailored to the environment at each call site. A semi-open linkage entails generating a private copy of a procedure for a routine that calls it from multiple sites. The activation records can be merged, register allocation performed jointly, and an inexpensive linkage used — in short, it provides many of the benefits of inline substitution with a smaller space penalty. Either of these schemes moves code across the call site boundary. This allows the compiler to do a better job of optimization, since it can optimize that code with respect to a specific call site. For call sites embedded in loops, standard code motion will relocate any loop invariant code outside the loop. Parts of the prologue code that establish operand addressability are likely to fall in this category.

The program compiler will attempt to discover places where *cloning* a procedure can lead to improved optimization. When the interprocedural environment in which a procedure is to be executed differs radically between two sets of call sites, the program compiler can clone the procedure, tailoring the interface in different ways to the different call sites. To illustrate the usefulness of this technique, consider a procedure from a library that employs an option parameter to specify which of two different ways the procedure is to be used. Frequently, the option is specified by a constant parameter at the point of call. For one of the two option values, the procedure reduces to a small fragment of code suitable for in-line substitution; for the other, the reduced code is much longer. The program compiler might clone this procedure into two copies, one called from all sites in which there is a constant parameter specifying the short code fragment and the other called from the remaining sites.

3.5.3. Other Techniques

Finally, a number of interesting optimization opportunities arise because the compiler operates inside a programming environment, where it has easy access to the text for each procedure and records of previous compilations. An environment can perform transformations which are difficult to implement in a traditional separate compilation system. One example is *cross jumping*, where a simple pattern matching process is used to unify identical instances of procedure epilogue code. Extraneous copies of this code are replaced by branches into the remaining instance, with a concomitant saving in object code size.

4. Recompilation Analysis

Using interprocedural analysis and optimization to improve the quality of code generated by the compiler introduces a new problem, called the *recompilation problem*. After changes to one or more modules

[8] Of course, the compiler must carefully avoid name conflicts and ensure that the parameter binding semantics are preserved.

incorporated into a program, the system must decide which modules to recompile. Although on first consideration this might seems simple, the use of interprocedural information in module compilations makes it a complex problem. Certainly, each module whose source has been changed must be recompiled. However, because a change to the source of one module may invalidate the assumptions about interprocedural data flow that were used in compiling a second module, the second module may need to be recompiled as well. Thus, a single change might necessitate recompiling every module at which any of the interprocedural data flow sets have changed, a potentially enormous set of recompilations.

In a system where the compiler uses interprocedural information to make compile-time decisions, the most conservative recompilation strategy would require recompiling every procedure in response to each editing change. This approach completely eliminates the savings of separate compilation. In order to preserve some of the economic benefits of separate compilation, the $\mathbb{R}^n$ environment employs a systematic *recompilation analysis* to reduce the number of spurious recompilations caused by changes to source modules [45, 17, 10].

In $\mathbb{R}^n$, we are experimenting with several strategies for recompilation analysis. All of the techniques apply the same test to determine when a procedure must be recompiled. The test compares the actual interprocedural information describing a program against *annotation sets*. These sets contain those interprocedural facts that can be true without invalidating the procedure's previous compilation. The three methods differ in the precision with which they assign values to these annotation sets. The basic tradeoff lies between the expense of computing values for the sets and the precision of the resulting test.

Editing changes can mandate that a procedure be recompiled to ensure the correctness of code previously generated for it. The tests presented in this section detect recompilations for correctness. They do not consider the question of recompiling to achieve improved optimization after an editing change. This reflects a basic philosophy: $\mathbb{R}^n$ assumes that compilations are expensive. Accordingly, the recompilation analyzer will not slate a procedure for recompilation simply to improve its level of optimization.

For recompilation analysis, we attach the following sets to the program's call graph.

(1) *MayBeAlias*(p), for each procedure p — the set of alias pairs that are allowed without forcing a recompilation. If a change adds a pair to ALIAS(p) that is not in *MayBeAlias*(p), p must be recompiled.

(2) *MayMod*(e), for each call graph edge e — the set of variables that may be modified as a side effect of the call without forcing a recompilation. If a change adds a variable to MOD(e) that is not in *MayMod*(e), p must be recompiled.

(3) *MayUse*(e), for each call graph edge e — the set of variables that may be used as a side effect of the call without forcing a recompilation. If a change adds a variable to USE(e) that is not in *MayUse*(e), p must be recompiled.

(4) *MustBeConstant*(p), for each procedure p — the set of constant pairs that must hold on entry to procedure p if recompilation is to be avoided. If there exists a pair <x,v> in *MustBeConstant*(p) that is not in CONSTANTS(p), p must be recompiled.

Given these sets, a procedure p must be recompiled if

(1) ALIAS(p) - *MayBeAlias*(p) $\neq \emptyset$, or

(2) MOD(e) - *MayMod*(e) $\neq \emptyset$, for any call site e in p, or

(3) USE(e) - *MayUse*(e) $\neq \emptyset$, for any call site e in p, or

(4) *MustBeConstant*(p) - CONSTANTS(p) $\neq \emptyset$.

Set subtraction is defined so that $a \in$ (X-Y) if and only if a is a member of X and not Y.

To construct a list of procedures needing recompilation, the analyzer first initializes the list to include every procedure where editing has produced a semantic change since its last compilation. Next, it applies incremental techniques to update the program's ALIAS, MOD, USE, and CONSTANTS sets. Whenever this update changes the value of an interprocedural set, the compiler applies the appropriate test. If the test

indicates that the procedure must be recompiled, the analyzer adds it to the list. Because the analyzer only tests sets that change during the incremental update, the test requires a number of set operations proportional to the size of the region of changed data flow.

To help develop intuition about the tests, consider the following assignment of values. Let *MayBeAlias*, *MayMod*, and *MayUse* be set uniformly to $\emptyset$. Assign all of the *MustBeConstant* sets a special set consisting of a pair $<x,\Omega>$ for each formal parameter or global variable x, where Ω is a constant value that appears nowhere in the program. With this assignment of values, the test will indicate recompilations for every procedure where either the source text or some associated interprocedural set has changed. Already, this is an improvement over recompiling the entire program — it recompiles only procedures where the compile-time situation has changed.

The effectiveness of the testing procedure depends entirely on the values assigned to *MayBeAlias*, *MayMod*, *MayUse*, and *MustBeConstant*. Improving the precision of the test involves expanding *MayBeAlias*, *MayMod*, and *MayUse* to include more allowed facts, or shrinking *MustBeConstant* to exclude more facts. The next three subsections present different methods of computing values for these sets. The methods are presented in increasing order of complexity; each successive method provides a more precise result from the previous test.

4.1. Most Recent Compilation

Our first approach to computing the annotation sets simply remembers the values of ALIAS, MOD, USE and CONSTANTS used in the most recent compilation of the procedure. In other words, whenever we recompile a procedure p, we set

(1) *MayBeAlias*(p) = ALIAS(p),

(2) *MayMod*(e) = MOD(e), for each call site e in p,

(3) *MayUse*(e) = USE(e), for each call site e in p, and

(4) *MustBeConstant*(p) = CONSTANTS(p).

This set of assignments provides further insight into the principles underlying the recompilation test. The test relies on an understanding of the interprocedural information and the manner in which it can be used by the optimizer. As discussed in Section 3, the summary and aliasing sets are flow insensitive; they describe events which can occur but do not always occur. Because of this fundamental fact about the nature of the ALIAS, MOD, and USE sets, the compiler can only depend on what is *not* in these sets. If the optimizer applies a transformation when a variable is present in one of these sets, removing it from the set cannot invalidate the safety of that decision. The compiler must have already considered the case when the specified data flow event does not occur. Changes in flow insensitive information necessitate recompilation only when they involve additions to those sets. While a deletion from a flow insensitive set can open up a new opportunity for optimization, it cannot invalidate the correctness of a previous compilation. This principle motivates tests (1), (2), and (3).

The CONSTANTS(p) set approximates a flow sensitive set, so changes in its value must be handled differently. Flow sensitive sets assert that a fact holds true on every path leading to some point in the program, so the compiler can rely directly on the fact to justify a transformation. For example, if a pair $<x,v>$ is in CONSTANTS(p), the compiler can rely on x having the value v on entry to p, folding the value v into references to x along any paths where x is unmodified. If later changes in the program remove $<x,v>$ from CONSTANTS(p), the changes invalidate the constant folding optimization. Thus, for flow sensitive sets, recompilations arise only in response to deletions. Adding an element to a flow sensitive set can open up new opportunities for optimization, but cannot invalidate previously applied optimizations. This rationale underlies test (4), which recompiles a procedure only if an element is deleted from its CONSTANTS set.

4.2. Reference Information

Directly using information from the most recent compilation produces a recompilation test that is a significant improvement over the naive approach. Unfortunately, it fails to take advantage of the

availability of the source text for the procedure under consideration. In particular, it results in recompiling a procedure if a variable is added to one of its interprocedural sets, even if that variable is not actually referenced anywhere in the procedure. Examining which variables are actually referenced in a given procedure leads immediately to an improved test.

To describe the annotation sets for this improved test, we must define three additional sets. For a procedure p, REF(p) is the set of variables either used or modified inside p. REF(p) can be computed as IMOD(p) $\cup$ IUSE(p). Define REF$^+$(p) as the set of all variables referenced in p or some procedure invoked as a result of executing p. Thus, REF$^+$(p) can be computed as GMOD(p) $\cup$ GUSE(p). The set AliasREF(p) describes pairs of variables, one of which is referenced locally in p and the other referenced in p or one of the procedures that can be executed as a result of invoking p. This set is defined as

$$\text{AliasREF}(p) = \{ < x, y > \mid x \in \text{REF}(p) \text{ and } y \in \text{REF}^+(p) \}$$

Finally, the complement of a set REF(p) is denoted as $\overline{\text{REF}(p)}$. Given these definitions, the improved annotation sets can be computed as follows:

(1) *MayBeAlias*(p) = ALIAS(p) $\cup$ $\overline{\text{AliasREF}(p)}$

(2) *MayMod*(e) = MOD(e) $\cup$ $\overline{\text{REF}(p)}$

(3) *MayUse*(e) = USE(e) $\cup$ $\overline{\text{REF}(p)}$

(4) *MustBeConstant*(p) = CONSTANTS(p) $\cap$ REF(p)

Computing the annotation sets from these definitions eliminates spurious recompilations that arise from information about irrelevant variables. In practice, this is important because procedures often contain declarations for global variables that they never reference. In Fortran, this happens with COMMON statements; in other languages, widespread use of include files achieves the same result.

The equations are presented in this form to convey the underlying ideas; an actual implementation should avoid instantiating sets like $\overline{\text{REF}(p)}$ and $\overline{\text{AliasREF}(p)}$. A careful refactoring of the equations leads directly to a much more efficient implementation.

4.3. Compiler Cooperation

While the use of reference information should eliminate many unnecessary recompilations, some can still occur because the compiler often cannot use every interprocedural fact. Thus, a fact judged to mandate recompilation by the reference test may actually be irrelevant, simply because the compiler was unable to capitalize on it to justify an optimization during the most recent compilation. To compute more precise annotation sets requires cooperation between the optimizer in the module compiler and the recompilation analysis in the program compiler. In such a scheme, the module compiler computes a set, for each call site and each optimization applied, containing those interprocedural facts that can invalidate the optimization. Using this information, the module compiler can compute exact annotation sets.

Computing exact annotation sets places a significant burden on the optimizer. For every optimization it applies, it must record those interprocedural facts that affected the safety decision. To convey the intricacy of this process, we present the analysis required for a single optimization, redundant store elimination. If the compiler discovers a point where the value of a local variable of a procedure exists in a register and that value cannot be used later in the procedure, it need not store the value back into memory. To perform this optimization, called *eliminating unnecessary stores*, the compiler must recognize the last use of a variable in a procedure.

This requires solving a global data flow analysis problem called the *live* problem. In global, or intraprocedural, data flow analysis, a procedure p is represented by its data flow graph, $G = (N,E,n_0)$. The nodes of G represent *basic blocks*, sequences of statements with no control flow branches. The edges $e = (m,n) \in E$ represent *control flow* between two basic blocks. Control enters the procedure through its entry node n_0.

A variable is *live* at a point in a procedure if there exists a control flow path from that point to some use of the variable and that path contains no assignments to the variable. Live analysis associates a set LIVE(b) with each block b. LIVE(b) contains all the variables that are live on entry to b. LIVE sets can be computed by solving the following backward data flow problem:

$$\text{LIVE(b)} = \text{IN(b)} \cup \bigcup_{a \in S(\text{b})} (\text{THRU(b)} \cap \text{LIVE}(a))$$

In this equation, $S(b)$ is the successor set of b. IN(b) is the set of variables used in b before being redefined. Variables in IN(b) are live on entry to b. THRU(b) is the set of variables not redefined in b.

Without summary information about procedure calls, the compiler must assume that a procedure call uses any variables visible to it. This assumption can extend the live ranges of variables, inhibiting the application of register store elimination. Interprocedural USE sets can reduce the set of variables assumed LIVE because of a call site. Because MOD(e) says nothing about the ordering of uses and definitions, MOD information is not pertinent to the computation of LIVE information.

Register store optimizations are invalidated when the life of a variable is extended by addition of a variable use after the current last use. Thus, any call sites between the eliminated store and the end of the procedure can potentially invalidate a register store optimization. Adding a variable to the USE set of such a call site would make the eliminated store necessary for correct execution of the program.

Assume the existence of a set CALLS_AFTER(b) for each block b, containing the set of call sites in the procedure containing b that can be executed after execution of b. To construct a recompilation test which precisely characterizes the use of interprocedural information in the register store optimization, we must compute a larger *MayUse*(e) set. Given this set, *MayUse*(e) can be computed as follows:

(1) *MayUse*(e) = ALLVARS, the set of all actual parameters and global variables, for each call site e in p;

(2) Whenever a store of a variable v is eliminated, the optimizer removes v from *MayUse*(e) for each call site e in CALLS_AFTER(b) and each call site inside b occurring after the optimization.

This results in *MayUse* sets that precisely capture the recompilation dependences for this optimization.

To compute CALLS_AFTER(b), the following system of data flow equations must be solved:

$$\text{CALLS\_AFTER(b)} = \bigcup_{a \in S(\text{b})} (\text{LOCAL\_CALLS}(a) \cup \text{CALLS\_AFTER}(a))$$

where LOCAL_CALLS(a) is the set of call sites in basic block a. CALLS_AFTER(b) is the set of call sites that occur after basic block b. This calculation is *rapid* in the sense of Kam and Ullman [28].

Computing exact annotation sets is significantly more complex than the reference information approach. We have shown the analysis required for a single optimization; similar methods would be needed for each optimization applied by the module compiler. Other examples of this analysis can be found in Torczon's dissertation [45, 17]. Actual experimentation will be required to determine whether the improved precision of the resulting recompilation test justifies the expense of employing this level of participation by the module compiler.

5. Environmental Impact

Having introduced the fundamental methods used to perform interprocedural analysis in the $\mathbb{R}^n$ environment, we will now examine how collecting and using interprocedural information affects the overall design of the system and the design of each of its components. In the environment, we have tried to distribute the work of collecting interprocedural information over the entire program development process. In the resulting system, the editors and compilers jointly hold responsibility for collecting interprocedural information. At the same time, we have tried to use interprocedural information to simplify and improve the individual tools whenever possible.

Central to the design of the $\mathbb{R}^n$ programming environment is its database. We begin by discussing the items that must be stored in the database to support the exchange of information required to develop interprocedural information. Then we discuss the responsibilities each tool has in the computation of interprocedural information and the ways in which it uses such information. The discussion pays special attention to the role of the program compiler, which is responsible for overseeing the compilation process. The flow of information between the components of the environment is depicted in Figure 4, which can serve as a map for the remainder of the paper.

5.1. The Database

The database is a repository for all of the component parts of programs managed by the environment. It is the common structure for storing information. This includes objects used by programmers, like the source text of a procedure or its documentation, as well as information used to manage the compilation of programs, for example, the annotation sets described in section 4.

There are two major entity types in the database: *modules* and *programs*[9]. A *module* is the smallest editable or compilable unit of source. It corresponds to the source file on [10] systems. A module *provides* the collection of entry points it defines and *requires* the entry points called from within its body. The *specifications* for a module consist of the set of entry points it provides along with the number of parameters for each entry point and the type of each parameter[11].

A *program* is a structured collection of modules that form an executable entity. A program is *consistent* if the entry points called from modules in the collection match, in number and types of parameters, the corresponding entry points provided within the collection. It is *complete* if every entry point required within the collection is matched by an entry point provided[12].

Most of the information visible to the end users of the environment is stored as attributes of programs and modules. For example, a program has, among others, a *composition* or list of modules incorporated, a *call graph*, an *executable image*, and an *entry table* that maps entry points into the modules that implement them. A module has *source*, a list of *entries called*, and numerous *annotations* generated by the editors and compilers.

Figures are provided on separate pages

Figure 4. Flow of information in the environment.

---

[9] In point of fact, the environment also supports the notions of *module-versions*, which are different implementations of the same set of module specifications, and *program-versions*, which are different implementations of the same program specifications. These notions are intended to enhance the usability of the system and are not essential to the discussion in this paper. Hence, for simplicity of presentation we will use the terms ‘‘module’’ and ‘‘module version’’ and the terms ‘‘program’’ and ‘‘program version’’ interchangeably.

[10] is a trademark of AT&T Technologies.

[11] This is clearly a very limited form of specification. It is our intention to experiment with more sophisticated specifications in the future.

[12] The $\mathbb{R}^n$ environment also supports module collections that are not programs but libraries or composite modules. These entities are not essential to the discussion at hand.

5.1.1. Program Dependence

One of the most interesting implications of performing interprocedural optimization is that the compiled code for a module is a function not only of the module source but also of the *program in which that module is to be incorporated*. Because compilations depend on interprocedural information, the code generated is specific to the program in which the compilation is performed. As a result, a single module incorporated into six programs could have six different object modules, each stored as an attribute of the appropriate program.

In general, there are two types of information that the environment must deal with: *module specific* and *program specific*. The division must be carefully drawn in the design of the environment. For example, consider the information sets involved in interprocedural analysis, as described in Section 3. In each case, a substantial component of the analytical process is module specific. Thus, local information like IMOD, IUSE, and INTRO can be computed in the module editor and stored once, with the module. The results of the interprocedural propagations, however, are program specific and must therefore be stored with the program. This includes sets like GMOD, GUSE, ALIAS and CONSTANTS.

5.1.2. Version Control

The decision to perform interprocedural analysis and optimization has a subtle effect on the support for version control provided in $\mathbb{R}^n$. Because the interprocedural information must be updated for every program containing module *m* whenever *m* is changed, it is not desirable to incorporate the same module in many different programs. This makes it unattractive to differentiate between *minor versions* of a module.[13]

To test a new version of a module in the $\mathbb{R}^n$ environment, the programmer must make a completely new program to contain it. This insures that the testing process does not invalidate a working program. However, this policy increases the number of programs containing each of the unchanged modules, complicating the task of updating interprocedural information in response to minor changes in a shared module. Thus we encourage the programmer to create a new version when he begins to make a set of modifications and to use the same version until a logically complete set of modifications has been made and tested, regardless of the number of editing sessions involved. This approach decreases the proliferation of modules, at the expense of losing historical data about the sequence of revisions leading to the current version.

5.2. The Program Compiler

The program compiler is responsible for managing the computation of interprocedural information and the construction of an executable image for the program. It must perform these tasks in an incremental fashion. That is, it must be able to reconstruct the interprocedural information and an executable image efficiently after a change to one or more modules incorporated in the program. This may involve directing the module compiler to produce a new executable image for one or more modules in the program.

In many ways, the $\mathbb{R}^n$ program compiler resembles the *make* utility [23]. It must discover each module whose source has been changed since its last object version was produced and see that it is recompiled. However, its responsibilities go far beyond that. Specifically, after a change to one or more modules or to the program composition, the program compiler must:

- reconstruct the call graph for the program,
- update the interprocedural summary and aliasing information,
- perform interprocedural constant propagation,
- assess the prospects for interprocedural optimizations and
- determine which modules must be recompiled and invoke the module compiler on them.

[13] Systems like SCCS [39] create a new minor version of a module at each editing session.

The last task depends quite heavily on recompilation analysis, discussed in Section 4.

The program compiler has principal responsibility for constructing the interprocedural information used by the other components of the IR^n environment. It uses the program composition constructed by the composition editor along with the information about which entries are called from each module to construct a call graph for the entire program. It then uses that call graph and the initial information computed by the module editor to determine solutions to the interprocedural data flow analysis problems described in Section 3. For a completely new program, it applies the batch versions of the analysis algorithms. If the call graph has already been analyzed, it uses incremental techniques [16] to update the information. In performing this analysis, the program compiler frequently uses information computed in previous stages. For example, the algorithm for constant propagation uses information about side effects and aliasing to compute a more precise set of constants.

The program compiler also has the responsibility to direct the recompilation of modules within the program in response to an editing change. To do this, it builds a list of all modules slated for recompilation. It initializes the list with those modules whose source has been modified since the module's last compilation. To this list it adds every module that has been added to the composition since the last program compilation. After the interprocedural information has been updated, the program compiler applies the recompilation test described in Section 4 to determine which modules must be recompiled because of changes to their interprocedural sets since their most recent compilation. It can, of course, ignore any modules where the interprocedural information has not changed. Similarly, it need not consider modules already marked for recompilation by virtue of local editing changes.

A final responsibility of the program compiler is to locate opportunities for interprocedural optimization. The module compiler performs several types of interprocedural optimizations, such as linkage tailoring, in-line substitution, cloning and cross jumping (see Section 3.5). The program compiler's role in this process is to locate likely sites for such optimizations, to direct the module compiler to perform the optimizations, and to remember which modules have been optimized together. This last point is crucial since some interprocedural optimizations, like linkage tailoring and cross jumping, require subsequent recompilation analysis to treat the two as a unit. Similarly, procedure cloning introduces complications into the recompilation analysis, since it changes the mapping of names to implementations. An editing change might indicate recompilation of a cloned procedure when simply switching one call site from the optimized to the unoptimized version might correct the problem.

5.3. The Module Editor

The module editor is the primary mechanism for creating and modifying the source code for modules under the environment's control. Thus, it is usually the first tool to examine the contents of any module and it is natural that the general scheme employed by the IR^n environment to generate interprocedural information (see Section 3) relies on the module compiler to construct the initial data from which that information is developed. Specifically, the module editor must compute five types of information:

- *Call Graph Components*: information about the contribution of each entry point in the module to the call graph of a program that incorporates it. This information includes a specification of each edge, where an edge corresponds to a call site, that must be added to the call graph whenever the entry point is added to a composition, along with the binding function, f_e, for each call site. When a module is being edited in the context of a given program, the call graph for that program must be annotated with a new edge when a new call is inserted in the module source.
- *Side Effect Summary Sets*: the initial information required by the program compiler to compute the global summary information about side effects of procedures. This includes the IMOD and IUSE sets for each entry point in the module (see Section 3.2).
- *Aliasing Information*: the information needed by the program compiler to compute the sets of alias pairs at each entry in a program. This includes the INTRO sets for each call site, used to compute the alias *introduction* effects, and a mapping b_e from pairs of formal parameters in the calling procedure to pairs of actual parameters at the call site, reflecting the *propagation* effects (see Section 3.3). The module editor can also differentiate between call sites that are important to the analysis and those that

are irrelevant to it.

- *Constant Propagation Information*: jump functions for each call site in the module to support interprocedural constant propagation (see Section 3.4). The amount of work involved in computing the initial information will vary with the specific technique used to implement jump functions. It may involve an operation as simple as scanning each call site to detect literal constants and formal parameters used as actual parameters or as complex as symbolic evaluation to detect local constants passed as actual parameters, constant valued global variables, and actual parameters whose values are always expressible as a function of the parameters to the calling procedure [45, 12].
- *Recompilation Information*: an indication whether this module must be recompiled because of local changes. The recompilation algorithms used in the program compiler rely on the editor to mark modules that have been semantically changed [45]. Depending on the specific strategy used to compute the annotation sets, additional information may be needed (see Section 4). The REF set is one type of initial information used in these algorithms.

The module editor capitalizes on its understanding of Fortran to sharpen the local recompilation analysis. By noting which editing changes alter the meaning of a module, it avoids marking a module for recompilation based on local changes if those changes are irrelevant to compilation. For example, adding comments to a procedure need not cause a recompilation. When a semantic change is made to a declaration or defined constant that is shared between modules, the editor determines which of those modules require recompilation because of the change and marks them appropriately. This latter feature is similar to the analysis performed by Tichy and Baker to limit recompilations based on include files [44].

The module editor uses interprocedural data flow information to help the programmer develop better code. One of the most important applications of such information is detecting and reporting semantic anomalies in a module. These anomalies may indicate programming errors. For example, when the programmer enters a constant as an actual parameter, if the call site's MOD set contains that parameter position, the editor can indicate that the parameter might be changed. This approach to program diagnosis was pioneered by Fosdick and Osterweil in the DAVE system [37] and used in an interactive mode by Masinter in the MASTERSCOPE facility of INTERLISP [33]. Conradi's FORTVER system also employs interprocedural analysis to uncover *data flow anomalies* in whole programs [13]. However, none of these systems provides the information interactively as the program is being developed. This is a special benefit of the approach to whole program analysis used in $\mathbb{R}^n$.

Zadeck has proposed the use of global data flow information in the editor to point out data flow anomalies to the programmer, suggesting that the precision of the diagnostic information can be improved by using interprocedural knowledge [47]. In considering this application, we must be careful to remember the sensitivity of interprocedural information to specific call graphs. The side effect sets of a procedure call are a function of the entire body of the called procedure, including procedure calls embedded in it. Thus, the summary sets that describe that call depend on specific details of the called procedure and any other procedure that can be invoked indirectly by the call. Since the program's composition controls the binding of procedure entry point names to implementations, any interprocedural sets computed for a program are are valid only for that specific composition. In other words, if interprocedural information is used in the editor to augment or refine intraprocedural data flow analysis, the resulting information will be correct *only* for the specific program for which the interprocedural analysis was performed. When a module is included in multiple programs, conflicting diagnostic information may be reported when different programs are considered.

This is a general problem associated with the use of interprocedural information in the editor — the information is a property of the program, not the module, so the user must be careful to specify the correct program as the context for editing. Nevertheless, there is great promise for a diagnostic system based on data flow information in the $\mathbb{R}^n$ programming environment and we intend to pursue experimentation with such a facility.

5.4. The Composition Editor

The composition editor is the primary vehicle for programming-in-the-large in the $\mathbb{R}^n$ environment. It allows a user to specify the collection of modules to be included in the program and provides facilities for checking the consistency and completeness of the resulting program. For example, it ensures that the actual parameters of a procedure call match the formal parameters of the called procedure in number and type. If the composition is *complete*, that is, it includes a main procedure and an implementation for every needed entry point, then it can be be used to generate an executable image for the program it represents. If a composition is both consistent and complete, the editor marks it eligible for compilation.[14]

The composition editor is used to create programs. In this role, it has several responsibilities for interprocedural analysis. As it builds a new composition, it collects the information from which the program compiler constructs the call graph. Whenever it updates a composition, it must annotate the existing call graph with a list of changes since the last program compilation to alert the program compiler to check for corresponding changes to the interprocedural data flow information.

The concerns of interprocedural analysis affect the design of a command set for manipulating compositions. A case in point is the library search mechanism. Because the editor treats a composition as a mapping from entry points to their implementations, it can use existing programs as libraries to be searched for unresolved entry points. The library search mechanism encourages the user to create new programs by specifically including the new versions of changed modules and then letting the composition editor complete the program automatically by employing library search on the previous version of the program containing these modules.

Elegant though this strategy may be, it has the disadvantage of increasing the amount of work required to construct call graphs and interprocedural information for the new program. Under the library search paradigm, each module is copied individually, with the result that most interprocedural information is lost and a complete re-analysis of the program is required. However, if the user instead copies the old composition and then edits it, replacing the old versions of changed modules with new ones, most of the call graph will be preserved, allowing much of the interprocedural information associated with the old composition to be re-used. Fast incremental update algorithms [16] can then be used to compute the interprocedural information for the new program, just as if the modules had been edited in place. This observation led us to redesign the composition editor to encourage users to build new compositions by incrementally editing old ones.

The composition editor also provides a mechanism for defining a single new module from a collection of modules. This presents special problems for interprocedural analysis. First, the fundamental quantities used by the program compiler in the interprocedural computations must be determined for the new module group. In the case of the MOD problem of Section 3.2, this means computing IMOD(q) for every entry q provided externally by the group. Computing this set requires solving an interprocedural data flow analysis problem on the call graph of the module group. Second, there must be a mechanism for updating the interprocedural information on edges of the call graph internal to the module group, given a change externally. This requires extending the incremental updating algorithms to a hierarchical form. Research on this problem is in progress.

5.5. The Module Compiler

The module compiler is designed to generate highly optimized object code for a source module in the context of a given program. To achieve this goal, it makes use of interprocedural information for the program and follows the optimization directives from the program compiler.

---

[14] In fact, it is possible to compile and execute a program that is not complete. The system maps each missing entry point onto a single default entry point that reports the run-time error. If the program contains both compiled and interpreted modules, it will invoke the interpreter to find an implementation of the module.

In the module compiler, interprocedural summary information is used to improve the precision of the computed intraprocedural information. For instance, in the absence of MOD information about a call site, the module compiler must assume that the call results in modifications to every variable that is accessible to the called procedure. This includes every global variable and every actual parameter at the call site. With interprocedural MOD information, the set of variables assumed to be modified can be greatly reduced. This should lead to significantly improved optimization around procedure calls. For example, an expression involving a variable in COMMON might be available at the point of call to a procedure, but in the absence of better information, it must be assumed to be unavailable immediately after the call, since the global variable may have been changed.

Similarly, the module compiler uses information about interprocedural constants as input to its own constant propagation analysis. This allows the intraprocedural analyzer to recognize constants that are inherited from the calling environment. In practice, important constants such as array dimensions and loop strides are likely to be detected by the program compiler; this information can open up new opportunities for applying purely intraprocedural optimizations.

Finally, information about interprocedural side effects not only helps produce better optimized code, it can also reduce the amount of analysis required in the module compiler. Our experience with an advanced vectorizer [3] shows that the number of use-definition chains constructed by the compiler can be drastically reduced through the use of interprocedural analysis.

The module compiler's major responsibility is to compute annotation sets for the recompilation analysis described in Section 4. For the most recent compilation and reference information schemes, the burden is limited to storing information sets provided by the program compiler. If, however, the exact annotation set scheme is used, the module compiler must record those facts actually used in optimizing the module. Typically, this requires that, for each optimization, the module compiler track the contributions of specific call sites to the global data flow information used in determining the safety of a transformation. Thus, the compiler produces, for each optimization and each call site, a set containing those interprocedural facts that could invalidate the optimization. As an example, Section 4.3 shows the analysis required for eliminating unneeded register stores.

The desire to perform linkage tailoring has a direct impact on the choice of intermediate representations used in the module compiler. If the compiler considers only strictly open and strictly closed linkages, then it may be possible to perform linkage tailoring on a high-level representation such as an abstract syntax tree. In generating either semi-open or semi-closed linkages, however, the compiler will almost certainly introduce constructs that have no reasonable representation in a high-level intermediate form. This will likely necessitate use of a relatively low-level representation to accommodate optimizations like moving loop invariant procedure prologue code out of a semi-open call inside a loop.

5.6. The Interpretive Debugger

A major goal of the debugger design is to support hybrid execution of interpreted and compiled modules. Interprocedural information can simplify the interface between compiled code and the interpretive debugger. Similarly, the design of the debugger may preclude the application of some interprocedural optimizations.

Consider a session in the debugging interpreter. The programmer might ask the interpreter to report, after each statement executes, which variables have had their values changed. For statements that do not include procedure calls, this is a simple task. If, however, the statement involves a call to a compiled procedure, the interpreter must compare the values of each variable accessible outside the calling procedure against its value before the call in order to compute the debugging information. For any non-trivial program, this is a prohibitively expensive proposition.

If, however, the interpreter has access to interprocedural summary information, it can use the summary information to determine which variables *might* change as a result of the call, and the search for changed variables can be restricted to a much smaller set. This same technique can be used to improve the support for *reversible execution*, since implementation of this feature requires dynamic checkpointing of variables that *might* change as a result of a call to a compiled procedure.

The requirement for hybrid interpreted and compiled execution seems likely to rule out the application of certain optimizations. For example, the preceding discussion assumes that the interpreter understands the mapping from variable names to storage locations for compiled code. For the compiler and interpreter to cooperate, the interpreter must be able to reproduce the compiler's assignment of variables to memory locations. This simple assumption may prohibit the compiler from applying sophisticated storage optimizations like those proposed by Fabri [22]. If the compiler used such techniques, either it would need to record the resulting storage map for the interpreter or the interpreter would be forced to duplicate the analysis. Either scenario could add significantly to the complexity of executing hybrid code.

Finally, hybrid execution of interpreted and optimized code has implications for the design of the interpreter. It is easy to imagine a programmer changing the value of a variable inside the interpreter during a debugging session. In a separate compilation environment without interprocedural information, this is a relatively safe action. Once the compiler has used interprocedural information as a basis for optimization decisions, though, the possibility arises that changing the value of a variable inside one procedure has implications for the correctness of the code already compiled for other procedures in the currently executing program. Thus, the interpreter needs to understand the manner in which the compiler uses interprocedural information in order to allow safe and correct responses to user requests during execution.

6. Current Implementation Status

A preliminary implementation of the $\mathbb{R}^n$ programming environment is running on SUN[15] Microsystems workstations under their version of . It includes stable versions of the monitor, the module editor, the composition editor, and the execution monitor. The execution monitor uses the native f77 compiler to create executables; a version is being designed that will allow compilation and execution on remote systems. A single-user database has been in use for nearly two years; a multi-user database has been available for six months. The interactive debugger, including hybrid execution, is being tested. Implementations of the module compiler and the program compiler are underway. A preliminary implementation of the interprocedural data flow algorithms for the program compiler is in use and the information computed by this phase is being used for diagnostic purposes in the editor. Finally, the current system contains a number of ancillary command processors. including a calculator, a terminal emulator, a HELP processor, and a documentation editor.

7. Summary and Conclusions

We believe that a sophisticated software development environment provides a practical setting for performing optimization based on interprocedural analysis. The consequences of incorporating interprocedural analysis and optimization are far-reaching. The required analysis substantially effects the design of every component of the environment. The $\mathbb{R}^n$ project should be viewed as a demonstration that a programming environment is a natural place to collect and use interprocedural information.

8. Acknowledgements

Both Tom Reps and our referees provided critical insights on the form and content of this paper. William LeFebvre provided invaluable assistance with the text and picture processing software. The $\mathbb{R}^n$ implementation team has provided a marvelous research vehicle for experimenting with new ideas about interprocedural analysis and optimization. To all of these people go our heartfelt thanks.

9. References

1. Aho, A.V., Hopcroft, J.E. and Ullman, J.D. *The Design and Analysis of Computer Algorithms*. Addison-Wesley (1974).

[15]SUN is a trademark of SUN Microsystems, Incorporated.

2. Allen, F.E. and Cocke J. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*, Rustin, R., Ed., Prentice Hall, Englewood Cliffs, N.J. (1972), 1-30.

3. Allen, J.R. and Kennedy, K. PFC: a program to convert Fortran to parallel form. In *Supercomputers: Design and Applications*, Hwang, K., Ed., IEEE Computer Society Press (1984), 186-205.

4. Allen, J.R. and Kennedy, K. A parallel programming environment. *IEEE Software 2*, 4 (July 1985), 22-29.

5. American National Standards Institute. *American National Standard Programming Language Fortran, X3.9-1978* (1978).

6. American National Standards Institute. Proposals approved for Fortran 8x. X3J3/S6.80 (1981).

7. Ball, J.E. Predicting the effects of optimization on a procedure body. *Proceedings of the SIGPLAN 79 Symposium on Compiler Construction, SIGPLAN Notices 14*, 8 (Aug. 1979), 214-220.

8. Banning, J.P. An efficient way to find the side effects of procedure calls and the aliases of variables. *Proceedings of the Sixth Annual ACM Symposium on Principles of Programming Languages* (Jan. 1979), 29-41.

9. Barth, J.M. A practical interprocedural data flow analysis algorithm. *Communications of the ACM 21*, 9 (Oct. 1972), 613-640.

10. Burke, M. Private communication. (Nov. 1983).

11. Burke, M. An interval analysis approach toward interprocedural data flow. Report RC 10640, IBM T.J. Watson Research Center, Yorktown Heights, N.Y. (July 1984).

12. Callahan, D., Cooper, K.D, Kennedy, K. and Torczon, L.M. Interprocedural constant propagation. *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*, (June 1986).

13. Conradi, R. Inter-procedural optimization of object code. Tech Report 25/83, Univ. of Trondheim, Div. of Computer Science, Trondheim-NTH, Norway (1983).

14. Cooper, K.D. Interprocedural data flow analysis in a programming environment. Ph.D. Dissertation, Rice Univ., Dept. of Mathematical Sciences, Houston TX (May 1983).

15. Cooper, K.D. Analyzing aliases of reference formal parameters. *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages* (Jan. 1985), 281-290.

16. Cooper, K.D. and Kennedy, K. Efficient computation of flow insensitive interprocedural summary information. *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction, SIGPLAN Notices, 19*, 6 (June 1984), 247-258.

17. Cooper, K.D., Kennedy, K. and Torczon, L. Interprocedural optimization: eliminating unnecessary recompilation. *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*, (June 1986).

18. DeRemer, F. and Kron, H.H. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering SE-2*, 2 (June 1976), 80-86.

19. Dongarra, J. LINPACK working note #3: FORTRAN BLAS timing. Tech. Report ANL-80-24, Argonne National Laboratory (Feb. 1980).

20. Dongarra, J. J., Bunch, J.R., Moler, C.B, and Stewart, G.W. *LINPACK users' guide*, SIAM, Philadelphia, PA (1979).

21. Ershov, A. ALPHA - an automatic programming system of high efficiency. *Journal of the ACM 13*, 1 (Jan. 1966), 17-24.

22. Fabri, J. Automatic storage optimization. *Proceedings of the SIGPLAN 79 Symposium on Compiler Construction, SIGPLAN Notices, 14*, 8 (Aug. 1979), 83-91.

23. Feldman, S. Make - a computer program for maintaining computer programs. *Software Practice and Experience 9* (1979), 255-265.

24. Graham, S.L. and Wegman, M. A fast and usually linear algorithm for global flow analysis. *Journal of the ACM 23*, 1 (Jan. 1976), 172-202.

25. Hecht, M.S. and Ullman, J.D. A simple algorithm for global data flow analysis problems. *SIAM J. Computing 1*, 2 (Dec. 1975), 519-532.

26. Hood, R.T. and Kennedy, K. A programming environment for Fortran. *Proceedings of the Eighteenth Annual Hawaii International Conference on Systems Sciences*, (Jan. 1985), 625-637.

27. Hood, R.T. and Kennedy, K. Programming language support for supercomputers. In *Frontiers of Supercomputing*, Metropolis, Sharp, Worlton and Ames, Eds., Univ. of California Press, Berkeley, CA (1986), 282-311.

28. Kam, J. and Ullman, J. Global data flow analysis and iterative algorithms. *Journal of the ACM 23*, 1 (1976), 158-171.

29. Kam, J. and Ullman, J. Monotone data flow analysis frameworks. *Acta Informatica 7*, Springer-Verlag (1977), 305-317.

30. Kennedy, K. A survey of data flow analysis techniques. In *Program Flow Analysis: Theory and Applications*, Muchnick, S.S. and Jones, N.D., Eds., Prentice-Hall (1981), 5-54.

31. Lampson, B.W. and Schmidt, E.E. Organizing software in a distributed environment. *SIGPLAN 83 Symposium on Programming Language Issues in Software Systems*, (June 1983), 1-13.

32. Leblang, D.B. and McLean, G.D. Configuration management for large-scale software development efforts. *Workshop on Software Engineering Environments for Programming-in-the-large*, Harwichport, MA (June 1985).

33. Masinter, L. Global program analysis in an interactive environment. Tech. Report SSL-80-1, Xerox Palo Alto Research Center, Palo Alto, CA (Jan. 1980).

34. Mitchell, J., Maybury, W. and Sweet, R. Mesa language manual. Tech. Report CSL-79-3, Xerox Palo Alto Research Center, Palo Alto, CA (April 1979).

35. Myers, E.W. A precise and efficient algorithm for determining existential summary data flow information. Tech Report CU-CS-175-80, Univ. of Colorado Dept. of Computer Science, Boulder, CO (March 1980).

36. Myers, E.W. A precise inter-procedural data flow algorithm. *Proceedings of the Eighth Annual ACM Symposium on Principles of Programming Languages*, (Jan. 1981), 219-230.

37. Osterweil, L.J. and Fosdick, L.D. Dave - a validation, error detection and documentation system for FORTRAN programs. Tech. Report CU-CS-071-75,Univ. of Colorado, Dept. of Computer Science, Boulder, CO (June 1975).

38. Reif, J.H. and Lewis, H.R. Symbolic evaluation and the global value graph. Tech. Report 37-82, Harvard Univ., Aiken Computation Laboratory (1982).

39. Rochkind, M.J. The source code control system. *IEEE Transactions on Software Engineering SE-1*, 4 (Dec. 1975), 364-370.

40. Rosen, B.K. Data flow analysis for procedural languages. *Journal of the ACM 26*, 2 (April 1979), 322-344.

41. Tarjan, R.E. A unified approach to path problems. *Journal of the ACM 28*, 3 (1981), 577-593.

42. Tarjan, R.E. Fast algorithms for solving path problems. *Journal of the ACM 28*, 3 (1981), 594-614.

43. Tichy, W.F. A data model for programming support environments and its applications. In *Automated Tools for Information Systems Design*, Schneider, H-J. and Wasserman, A.L, Eds., North-Holland (1982).

44. Tichy, W.F. and Baker, M.C. Smart recompilation. *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, (Jan. 1985), 236-244.

45. Torczon, L.M. Compilation dependences in an ambitious optimizing compiler. Ph.D. Dissertation, Rice Univ., Dept. of Computer Science, Houston, TX (May 1985).

46. Wegman, M. and Zadeck, F.K. Constant propagation with conditional branches. *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, (Jan. 1985), 291-299.

47. Zadeck, F.K. Incremental data flow analysis in a structured program editor. Ph.D. Dissertation, Rice Univ., Dept. of Mathematical Sciences, Houston TX (Oct. 1983).