A Practical Framework for Demand-Driven Interprocedural Data Flow Analysis

EVELYN DUESTERWALD, RAJIV GUPTA, and MARY LOU SOFFA
University of Pittsburgh

The high cost and growing importance of interprocedural data flow analysis have led to an increased interest in demand-driven algorithms. In this article, we present a general framework for developing demand-driven interprocedural data flow analyzers and report our experience in evaluating the performance of this approach. A demand for data flow information is modeled as a set of queries. The framework includes a generic demand-driven algorithm that determines the response to a query by iteratively applying a system of query propagation rules. The propagation rules yield precise responses for the class of distributive finite data flow problems. We also describe a two-phase framework variation to accurately handle nondistributive problems. A performance evaluation of our demand-driven approach is presented for two data flow problems, namely, reaching-definitions and copy constant propagation. Our experiments show that demand-driven analysis performs well in practice, reducing both time and space requirements when compared with exhaustive analysis.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers;optimization*; D.2.2 [**Software Engineering**]: Tools and Techniques; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*question-answering*

General Terms: Algorithms, Performance, Experimentation

Additional Key Words and Phrases: copy constant propagation, data flow analysis, def-use chains, demand-driven algorithms, distributive data flow frameworks, interprocedural data flow analysis, program optimizations

---

1. INTRODUCTION

Data flow analysis has become a mandatory component of today's optimizing and parallelizing compilers. In addition, data flow analysis is increasingly used to improve the capabilities and performance of software development tools such as editors [Reps et al. 1983], debuggers [Weiser 1984] and testing tools [Duesterwald et al. 1992; Frankl and Weyuker 1988]. Along with the growing use of data flow analysis comes an increased concern about the high time and space requirements of computing and maintaining the data flow information that is needed. Computing data flow solutions, especially if interprocedural analysis is involved, can be very costly [Griswold and Notkin 1993]. Moreover, costly analysis phases may have to be applied more than once for several reasons:

This work was supported in part by National Science Foundation Presidential Young Investigator Award CCR-9157371 and grant CCR-9402226 to the University of Pittsburgh.
A preliminary version of the demand-driven framework was presented in ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1995.
Authors' addresses: E. Duesterwald, Hewlett-Packard Laboratories, 1 Main Street, Cambridge, MA 02142; email: duester@hpl.hp.com; R. Gupta and M.L. Soffa, Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260; email: {gupta; soffa}@cs.pitt.edu.

Multiple Optimizations. Optimizing compilers typically perform a number of independent optimizations, and each optimization may require a distinct data flow analysis to be performed.

Invalidated Information. If code transformations are applied to a program, the data flow in the program changes, and previously computed data flow solutions may no longer be valid. As a consequence, data flow must be either updated or recomputed following the application of code transformations.

User Edits. Data flow information may be invalidated through program edits by the user. During program development, program edits are to be expected and should be efficiently handled. The respective analysis may have to be repeated to provide the new data flow solution.

Although the need for efficient data flow analysis algorithms continues to increase, current data flow applications still rely on exhaustive algorithms. Phrased in the traditional framework [Kam and Ullman 1977], the solution to a data flow problem is expressed as the fixed point of a system of data flow equations. Each equation expresses the solution at one program point in terms of the solutions at immediately preceding (or succeeding) points. As a result, data flow solutions are computed in an inherently exhaustive fashion: information is computed at *all* program points. Such an exhaustive solution definition is likely to result in very large equation systems limiting both the time and space efficiency of even the fastest fixed-point evaluation algorithm.

This article presents an alternative approach to data flow analysis that avoids the costly computation of exhaustive solutions through the demand-driven retrieval of information. Demand-driven analysis provides a promising approach to improve the performance of data flow analyses for several reasons:

Selective Data Flow Requirements. Several code transformations in optimizing compilers are applicable to only certain components of a program, such as loops. Even if optimizations are applicable everywhere in the program, the compilation time overhead can be reduced by optimizing only the most frequently executed regions of the program (e.g., frequently called procedures or inner loops). Demand-driven analysis provides an efficient way for computing all relevant information that affects the code inside the selected code region without having to analyze the complete program.

User Queries in Interactive Tools. In an interactive tool, the user issues specific information requests with respect to selected program points rather than inquiring about the entire program. For example, when debugging, a user may want to know what statements have an impact on the value of a variable at a certain point. The extent of data flow information requested by the user is not fixed before the debugging tool executes, but may vary depending on the user and the program. Demand-driven algorithms compute only what is necessary to service user queries.

Avoiding Incremental Solution Updates. Using an exhaustive analysis approach either requires costly recomputations of the exhaustive solution each time a change is made to the program, or it requires the storage and incremental update of the exhaustive solution throughout program development [Burke 1987; Pollock and Soffa 1989; Ryder 1983]. Maintaining exhaustive solutions throughout program development can be costly but it can be avoided entirely if a demand-driven approach is used to compute data flow information as it is needed, based on the current version of the program.

In this article we present a general framework for deriving demand-driven interprocedural analysis algorithms [Duesterwald et al. 1995]. A demand for a specific subset of the exhaustive solution is formulated as a set of queries. Queries are issued by the application and may be generated automatically (e.g., for compiler optimizations) or manually by the user (e.g., for an interactive tool). A query is a pair $q = \langle y, n \rangle$ that raises the question as to whether a set of data flow facts y is part of the exhaustive solution at a program point n. A response (*true* or *false*) to the query q is determined by propagating q from point n in the reverse direction of the original exhaustive analysis until all points have been encountered that contribute to the determination of the response for q. This query propagation is formally modeled as a partial reversal of the original exhaustive data flow analysis. The framework includes a generic algorithm that implements the partial reversal and provides the demand-driven analysis routine.

The framework is applicable to interprocedural data flow problems with finite lattices. The derived demand-driven algorithms are as precise as their exhaustive counterparts if the data flow problem is distributive. The class of distributive problems includes, among others, the four classical bit vector problems (i.e., reaching-definitions, available expressions, live variables, and very busy expressions), restricted versions of constant propagation (e.g., copy constant propagation), and interprocedural problems such as procedure side-effect analysis [Cooper and Kennedy 1988]. If the problem is monotone but not distributive, precision of the demand-driven algorithm may be lost with respect to a standard exhaustive algorithm. We outline a framework variation that is less efficient but enables the precise handling of nondistributive monontone data flow problems.

The practical benefits of the demand-driven framework are demonstrated through experimentation. An experimental study of demand-driven algorithms for two problems, namely, def-use chain computation based on reaching-definitions and copy constant propagation, was conducted to evaluate the performance of our demand-driven approach. The experimental results for the def-use chain analyzer show that demand-driven analysis is more efficient than exhaustive analysis for computing def-use chains in the majority of test programs. In copy constant propagation, which is a more expensive data flow analysis, demand-driven analysis performs even better and outperforms standard exhaustive analysis in all test programs.

The remainder of this article is organized as follows. Section 2 presents the pertinent background in interprocedural data flow analysis. The demand-driven analysis framework is presented in Section 3. This section first considers procedures without parameters and then discusses extensions to handle procedures with parameters. Section 4 discusses the framework variation for handling nondistributive data flow problems. Section 5 describes instances of the demand-driven analysis framework for reaching-definitions and copy constant propagation. An experimental evaluation of the performance of the demand-driven analyzers is presented in Section 6. Section 7 discusses related work, and concluding remarks are given in Section 8.

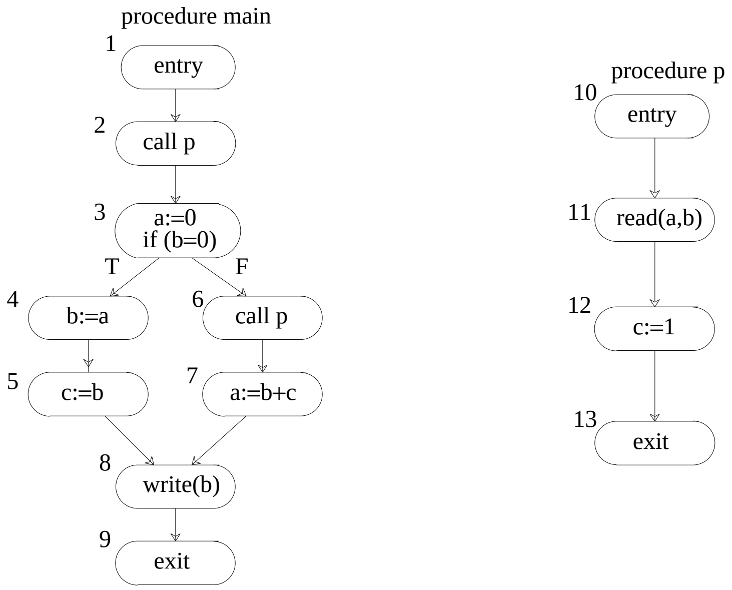

Fig. 1. The ICFG for a sample program.

2. BACKGROUND

A program consisting of a set of possibly recursive procedures $p_1, \ldots, p_k$ is represented by an *interprocedural control flow graph* (ICFG). An ICFG is a collection of distinct control flow graphs $G_1, \ldots, G_k$ where $G_i = (N_i, E_i)$ represents procedure p_i. The nodes in N_i represent the statements in procedure p_i, and the edges in E_i represent the transfer of control among the statements in p_i. Two distinguished nodes $entry_i$ and $exit_i$ represent the unique entry and exit nodes of p_i. The set $E = \cup\{\, E_i \,|\, 1 \leq i \leq k\}$ denotes the set of all edges in the ICFG, and $N = \cup\{N_i \,|\, 1 \leq i \leq k\}$ denotes the set of all nodes. The complexity analysis of our algorithms assumes that $|E| = O(|N|)$. [1] The sets $pred(n) = \{m | (m, n) \in E_i\}$ and $succ(n) = \{m | (n, m) \in E_i\}$ contain the immediate predecessors and successors of node n, respectively. For a call site node s, $call(s)$ denotes the procedure called from s.

We assume that each procedure is reachable from a series of calls starting from the main procedure. Furthermore, we assume the program contains no infinite loops and no interprocedural branching other than procedure calls and returns. A sample program and its ICFG are shown in Figure 1.

An *execution path* is a sequence of nodes $\pi = n_1 \ldots n_k$ such that for $1 \leq i < k$ (i) $(n_i, n_{i+1}) \in E$, where n_i is not a call site (intraprocedural control), (ii) $call(n_i) = p$ for some procedure p and $n_{i+1} = entry_p$ (procedure invocation), or (iii) $n_i = exit_p$ for some procedure p, and there exists an $m \in pred(n_{i+1})$ such that $call(m) = p$

[1] This assumes that switch statements have been transformed into a sequence of conditionals. A similar transformation is assumed for indirect function invocation through function variables.

(procedure return). An execution path that has correctly nested call and return nodes is called a *valid execution path.* To be a valid execution path, a procedure that returns must return to the site of its most recent call. Consider the example in Figure 1. The path $1, 2, 10, 11, 12, 13, 7, 8, 9$ that returns to the incorrect call site after the execution of procedure p is not valid. Invalid paths violate the calling context of procedures and may lead to imprecise information if considered during the analysis. The demand-driven analysis described in this article propagates information only along valid execution paths.

A *finite data flow framework* is a pair $D = (L, F)$, where

—$(L, \sqsubseteq, \bot, \top)$ is a finite lattice. The finite set L represents the universe of program facts with a partial order $\sqsubseteq$, a least *bottom* element $\bot$, and a greatest *top* element $\top$. The partial order $\sqsubseteq$ defines a *meet* operator $\sqcap$ and a dual *join* operator $\sqcup$ as the greatest lower bound and the least upper bound, respectively.

—$F \subseteq \{f : L \mapsto L\}$ is a set of monotone flow functions (i.e., $x \sqsubseteq y \Longrightarrow f(x) \sqsubseteq f(y)$) that contains the identity function and that is closed under composition and pointwise meet (i.e., $f, g \in F \Longrightarrow f \cdot g \in F$ and $f \sqcap g \in F$).

If all functions in F are distributive (i.e., $f(x \sqcap y) = f(x) \sqcap f(y)$), D is called a *distributive finite data flow framework.*

A data flow framework models a particular analysis problem. To describe the analysis problem for a particular program, we consider an instance of the respective framework.

An *instance* of a finite framework $D = (L, F)$ is given by an ICFG $G = (N, E)$ and a mapping that maps each node $n \in N$ in the ICFG to a function $f_n \in F$. The function f_n models the data flow when execution passes through node n. If $x \in L$ holds on entry of a node n, then $f_n(x) \in L$ holds on exit from node n. The demand-driven concepts and techniques presented in this article apply equally well to frameworks with flow functions associated with edges.

According to their analysis direction, data flow problems are classified as either forward or backward problems. In this article we focus on frameworks for forward data flow problems. However, the concepts and techniques apply equally well to backward problems by representing a backward problem on the reverse control flow graph and by replacing call nodes with return nodes [Marlowe and Ryder 1990b].

Consider the computation of the solution for a problem modeled in the preceding framework. During intraprocedural analysis the propagation of information is restricted to the control flow paths within each procedure. *Inter*procedural analysis considers also the propagation of information across procedure boundaries at call and return points. Our demand-driven analysis framework is a demand-driven variant of Sharir and Pnueli's functional approach to interprocedural analysis [Sharir and Pnueli 1981]. Similar to the interprocedural framework by Knoop and Steffen [1992] our extension to the Sharir and Pnueli framework handles programs with local variables and procedures with parameters. We briefly review the Sharir and Pnueli approach in the remainder of this section.

Sharir and Pnueli present a two-phase functional approach to interprocedural analysis that ensures that the calling context of each procedure is preserved. During the first phase each procedure is analyzed independently of its calling context. The results of this phase are procedure *summary functions* as defined by the equation

$$\text{For each procedure } p\text{:} \quad X(entry_p) = \bigsqcap_{call(m)=p} X(m)$$

$$\text{For nonentry nodes } n\text{:} \quad X(n) = \bigsqcap_{m \in pred(n)} \begin{cases} f_m(X(m)) & \text{if } m \text{ not a call site} \\ \phi_{(entry_q, exit_q)}(X(m)) & \text{if } call(m)=q \end{cases}$$

(a)

$$\phi_{(entry_p, entry_p)}(x) = x$$

$$\phi_{(entry_p, n)}(x) = \bigsqcap_{m \in pred(n)} \begin{cases} f_m \cdot \phi_{(entry_p, m)}(x) & \text{if } m \text{ not a call site} \\ \phi_{(entry_q, exit_q)} \cdot \phi_{(entry_p, m)}(x) & \text{if } call(m)=q \end{cases}$$

(b)

Fig. 2. Exhaustive interprocedural data flow equation system.

system in Figure 2(b). The summary function $\phi_{(entry_p, exit_p)} : L \mapsto L$ for procedure p maps data flow facts from the entry node $entry_p$ to the corresponding set of facts that hold upon procedure exit. The summary functions are defined inductively by computing for each node n in p the function $\phi_{(entry_p, n)}$ such that if $x \in L$ holds upon procedure entry, the corresponding element $\phi_{(entry_p, n)}(x) \in L$ holds on entry to node n.

The actual calling context of a procedure is propagated during the second phase, based on the summary functions. The data flow solution $X(n)$ at a node n expresses the set of facts that holds on entry to node n. $X(n)$ is computed by mapping the solution $X(entry_p)$, which holds on entry to p, to node n using the summary function $\phi_{(entry_p, n)}$. Figure 2(a) shows the equation system that defines the solution $X(n)$.

For finite lattices, Sharir and Pnueli propose a work list algorithm to solve the equation system in Figure 2 in two phases. The algorithm requires $O(MaxCall \times height(L) \times |L| \times |N|)$ time, where $height(L)$ denotes the height[2] of the lattice L and where $MaxCall$ is the maximal number of call sites calling a single procedure.

3. A FRAMEWORK FOR DEMAND-DRIVEN ANALYSIS

Before formally describing our demand-driven framework we first illustrate the demand-driven approach using the problem of copy constant propagation (CCP). CCP is a distributive variation of constant propagation. Unlike the more general constant propagation analysis, CCP does not evaluate arithmetic expressions. A variable v is a copy constant at a node n if v is assigned a constant value at node n or if v is always assigned the value of another variable that is a copy constant prior

[2]The height of a lattice L denotes the number of elements in the longest chain in L.

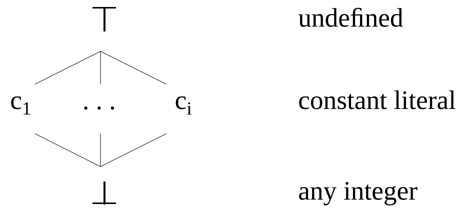

Fig. 3. The lattice for copy constant propagation.

to reaching node n.

The CCP lattice for a program with k variables is the product lattice L^k, where the component lattice L is defined as shown in Figure 3. The component lattice in CCP is finite, since the only possible values for a copy constant are the literals that occur in the program text. Each lattice element is a k-tuple $x = (x_1, \ldots, x_k)$ with a component $x_i \in L$ for variable v_i. The meet operator $\sqcap$ denotes the greatest lower bound of two elements according to the partial order depicted in Figure 3 and is defined pointwise. The dual join operator $\sqcup$ denotes the least upper bound.

A *base element* $[v_i = c]$ is a tuple with a single nonbottom component: $[v_i = c] = (\bot, \ldots, \bot, x_i = c, \bot, \ldots, \bot)$. Furthermore, any element that results as a finite join of base elements is written as $[v_1 = c_1] \sqcup \ldots \sqcup [v_l = c_l] = [v_i = c_i, \ldots, v_l = c_l]$. If the particular constant value is irrelevant, it is omitted, i.e., $[v_i]$ describes that variable v_i has a fixed but unknown constant value.

The distributive flow functions in CCP are defined pointwise for each component. Consider the control flow graph in Figure 1. Each lattice element is a triple (x_a, x_b, x_c) such that the components x_a, x_b, and x_c denote the lattice values for the three variables a,b, and c, respectively. The flow function for node 12 is defined as $f_{12}(x_a, x_b, x_c) = (x_a, x_b, 1)$ indicating that variable c has the constant value 1 after the execution of node 12, and node 12 has no effect on the values of variables a and b.

We now take a close look at how data flow analysis provides the answer to the following sample question: " Is variable b in Figure 1 a copy constant at the write statement in node 8?" Standard analysis provides the answer by an exhaustive forward propagation determining copy constant information for *all* variables at *all* nodes in the graph. At each node n the least solution is computed in a vector $X(n) = (x_a, x_b, x_c)$, where x_a, x_b, and x_c are the lattice values for variables a, b, and c at node n. The solution vectors are computed by propagating a program entry value throughout the program, as described in Section 2. This propagation involves the repeated application of the flow functions to the current solution vector values. Note that during a forward propagation all available information must be collected, since prior to reaching node 8, information at a preceding node cannot be ruled out as irrelevant. In particular, when encountering a call to procedure p (at nodes 2 and 6) the procedure must be fully analyzed in order to ensure that the complete information that may reach node 8 has been collected.

Now consider how a demand-driven analysis determines whether variable b is a copy constant at node 8. Unlike exhaustive analysis, demand-driven analysis is goal-directed. A solution to the question is determined by a partial search that is started at node 8 and proceeds backward along each path that in the forward direction leads to node 8. Note that this direction is the reverse of the direction of the exhaustive analysis. During this backward search only information that is actually relevant for the current query is collected. The search terminates as soon as the gathered information implies a solution to the initial query. In this case, upon encountering the read statement at node 11 (via the call at node 6), the backward search will establish that b cannot be a copy constant at node 8 and terminate.

Generally, there are three ways in which demand-driven analysis avoids unnecessary computations that must be performed in the exhaustive analysis. First, early termination is achieved by terminating the demand-driven analysis algorithms as soon as the relevant information has been obtained, possibly leaving large portions of the program that are not needed to obtain the demanded solution unvisited. Second, when visiting a node, the information that is not needed to resolve the current demand at that node is not collected. Finally, nodes/procedures are analyzed only if needed, i.e., only if it has been determined that information from the node/procedure may affect the demanded solution. For example, in Figure 1, the demand-driven analysis computes summary information for procedure q only with respect to the call site at node 6. Summary information with respect to the call site at node 2, which is never reached during the search, is not considered.

The demand-driven analysis framework provides a generalization of the preceding backward search that formally defines the search characteristics. Such generalization is obtained by providing answers to the following questions:

—What kind of information can be collected in a demand-driven way?

—What are the search operations performed at each node and when does the search terminate?

—Are there efficient algorithms to implement the backward search?

The demand-driven analysis framework contains a component to answer each of these questions. The first component provides the formal definition of a *data flow query* expressing the type of information that is the subject of the search. The second framework component formally models the search procedure as the backward propagation of data flow queries. The third framework component describes a generic iterative query propagation algorithm.

For ease of presentation, we first consider procedures without parameters. The extension to procedures with parameters is described in Section 3.7.

3.1 Data Flow Queries

A data flow query specifies the kind of information that can be determined in a demand-driven fashion at a given program node. Consider an instance $G = (N, E)$ of a finite data flow framework (L, F) with a solution vector X, and let $y \in L$ and $n \in N$. A *data flow query* q is specified by a pair

$$q = \langle y, n \rangle$$

that denotes the truth value of the term: $y \sqsubseteq X(n)$. In other words, query q raises the question as to whether a specific set of data flow facts y is implied by the exhaustive solution at a selected program node n. For example, the question as to whether variable b in Figure 1 is a copy constant at node 8 corresponds to the query $q = \langle [b], 8 \rangle$.

3.2 Query Propagation Rules

Consider now the problem of determining the answer (*true* or *false*) for a query q without completely evaluating the exhaustive solution equation system from Figure 2. Informally, the answer to $q = \langle y, n \rangle$ is obtained by propagating q from node n in the reverse direction of the original analysis until all nodes have been encountered that contribute to the answer for q. This propagation process is modeled as a partial reversal of the original data flow analysis. To define the analysis reversal, the following cases are examined in the propagation of a query $q = \langle y, n \rangle$:

—$q = \langle y, entry_p \rangle$ *for some procedure p (procedure entry node)* Query q raises the question as to whether y holds on entry to every invocation of procedure p. It follows that q can be translated into the Boolean conjunction of queries $\langle y, m \rangle$ for every call site m calling procedure p. If p is the main program then q evaluates to *true* if $y = \bot$, since by definition $X(entry_{main}) = \bot$. Otherwise, q evaluates to *false*.

—$q = \langle y, n \rangle$, *where node n is some arbitrary nonentry node* For simplicity, assume first that n has a single predecessor m. The exhaustive equation system from Figure 2(a) shows that $y \sqsubseteq X(n)$ if and only if $y \sqsubseteq h(X(m))$, where h is either a node flow function or a summary function. In either case h is monotone such that $h(\bot) \sqsubseteq h(X(m)) \sqsubseteq h(\top)$, and the following two special cases result for query q:

$$\begin{aligned} y \sqsubseteq h(\bot) &\implies q \text{ evaluates to } \textit{true} \\ y \not\sqsubseteq h(\top) &\implies q \text{ evaluates to } \textit{false} \end{aligned}$$

If neither of these two cases applies, the query q translates into a new query $q' = \langle z, m \rangle$ for node m. The lattice element z to be queried on entry to node m should be the least element z (i.e., smallest set of facts), such that $z \sqsubseteq X(m)$ implies $y \sqsubseteq h(X(m))$. The appropriate query element z for the new query q' can be determined using the function h^r which is the reverse of function h [Hughes and Launchbury 1992].

3.3 Reverse Flow Functions

Assume a finite lattice L and a monotone function $h : L \mapsto L$. The *reverse function* $h^r : L \mapsto L$ is defined as

$$h^r(y) = \bigsqcap \{x \in L : \ y \sqsubseteq h(x)\}.$$

The reverse function h^r maps an element y to the least element x, such that $y \sqsubseteq h(x)$. Note that if no such element exists, $h^r(y) = \top$ (undefined). Furthermore, the definition of h^r implies $h^r(\bot) = \bot$.

Example. Consider the reverse flow function f_n^r in CCP for a node n. By the distributivity of the reverse functions, it is sufficient to define f_n^r only for base

$(i)\quad \langle \bot, n\rangle \Longleftrightarrow true$
$\qquad \langle \top, n\rangle \Longleftrightarrow false$

(ii) For each procedure p :

$$\langle y, entry_p\rangle \Longleftrightarrow \bigwedge_{call(m)=p} \langle y, m\rangle$$

(iii) For a nonentry node n:

$$\langle y, n\rangle \Longleftrightarrow \bigwedge_{m\in pred(n)} \begin{cases} \langle f^r_m(y), m\rangle & \text{if } m \text{ is not a call site} \\ \langle \phi^r_{(entry_p, exit_p)}(y), m\rangle & \text{if } call(m) = p \end{cases}$$

Fig. 4. Query propagation rules.

elements. [3] The reverse function value $f_n^r([v_i{=}const])$ denotes the least lattice element, if one exists, that must hold on entry to node n in order for variable v_i to have the constant value *const* on exit of n. If $f_n^r([v_i{=}const]) = \bot$, the trivial value $\bot$ is sufficient on entry to node n (i.e., variable v_i always has value *const* on exit). For example, consider node 3 in Figure 1 with the assignment $a := 0$. The reverse function f_3^r is defined such that $f_3^r([a = 0]) = \bot$, indicating that a always has the value 0 on exit of node 3, and $f_3^r([a = 1]) = \top$, indicating that there exists no entry value which would cause variable a to have the value 1 on exit. For a variable v not equal to a and any constant value c, the reverse flow function is simply the identity $f_3^r([v = c]) = [v = c]$.

If the function h is distributive, the following inequalities hold and establish a Galois connection between the function h and its reverse h^r [Cousot 1981; Hughes and Launchbury 1992]:

$$(h^r \cdot h)(x) \sqsubseteq x \quad \text{and} \quad (h \cdot h^r)(x) \sqsupseteq x \tag{1}$$

We now show how the relationship between a flow function and its reverse can be exploited during query propagation. First, consider the following properties of the function reversal. It can be easily shown that the distributivity of h with respect to the meet $\sqcap$ implies the distributivity of the reverse function h^r with respect to the join $\sqcup$:

$$h^r(x \sqcup y) = h^r(x) \sqcup h^r(y) \tag{2}$$

Furthermore, the following properties hold with respect to the composition, the meet, and the join of functions [Hughes and Launchbury 1992]:

$$\begin{aligned} (g \cdot h)^r &= h^r \cdot g^r \\ (g \sqcap h)^r &= g^r \sqcup h^r \end{aligned} \tag{3}$$

Using the reverse functions, the complete set of query propagation rules can be

[3] For each element $[v_1 = c_1, \ldots, v_l = c_l]$ that is obtained as a finite join over base elements, the reverse function is $f^r([v_1 = c_1, \ldots, v_l = c_l]) = f^r([v_1 = c_1]) \sqcup \ldots \sqcup f^r([v_l = c_l])$.

established as shown in Figure 4. The operator $\wedge$ denotes Boolean conjunction. If node m is not a call site, the reverse function f_m^r can be determined by locally inspecting the flow function f_m. Otherwise, if node m calls procedure p, the reverse summary function $\phi^r_{(entry_p, exit_p)}$ is determined.

3.4 Reverse Summary Functions

A straightforward but inefficient way for computing reverse summary functions is to first determine all original summary functions by solving the equation system from Figure 2 and then reversing each function. A more efficient approach is to directly compute the reverse function values. By reversing the order in which summary functions are defined and by taking advantage of properties concerning the meet and composition of reverse functions, the following definition of the reverse summary function $\phi^r_{(entry_p, exit_p)}$ results for each procedure p:

$$\phi^r_{(exit_p, exit_p)}(y) = y$$

$$\phi^r_{(n, exit_p)}(y) = \bigsqcup_{m \in succ(n)} \begin{cases} f_m^r \cdot \phi^r_{(m, exit_p)}(y) & \text{if } m \text{ is not a call site} \\ \phi^r_{(entry_q, exit_q)} \cdot \phi^r_{(m, exit_p)}(y) & \text{if } call(m) = q \end{cases} \tag{4}$$

The propagation rules are now completely specified. Next we consider algorithms that implement these rules in an efficient way.

3.5 Generic Demand-Driven Algorithm

The framework contains, as a third component, a generic demand-driven algorithm. Procedure *Query*, shown in Figure 5, implements the query propagation rules and takes as its input a query and returns the answer *true* or *false*. When a procedure call is encountered, procedure *Compute*ϕ^r (Figure 6) is invoked to provide the reverse procedure summary information for the called procedure. In the worst case where the amount of information demanded is equal to the exhaustive solution, the asymptotic time and space complexities of the demand-driven algorithm are no worse than for the corresponding iterative exhaustive algorithm in the Sharir/Pnueli interprocedural analysis framework [Sharir and Pnueli 1981].

Procedure *Query* uses a work list that is initialized with the node from the input query q. At any step during the computation, the answer to q is equivalent to the Boolean conjunction of the answers to the queries currently in the work list. A variable $query[n]$ is used at each node n to store the queries raised at n. During each step a node n is removed from the work list and the query $\langle query[n], n \rangle$ is translated according to the propagation rule that applies to node n. The new queries resulting from this translation are merged with the previous queries at the respective nodes. A node n from a newly generated query is added to the work list unless the newly generated query was previously raised at node n (lines 9,10,17, and 18). Note that procedure *Query* terminates immediately after a query evaluates to *false*. If a query evaluates to *false*, it is not necessary to evaluate all remaining queries in the work list, since the overall answer to the input query must also be *false*. Thus, procedure *Query* can terminate early and the remaining contents of the work list are simply discarded. Otherwise, procedure *Query* terminates with the answer *true* when the

Procedure $Query(y, n)$
input: a lattice element $y \in L$ and a node n
output: the answer *true* or *false* to the query $\langle y, n \rangle$
begin:
1. **for each** $m \in N$ **do** $query[m] \leftarrow \bot$
2. $query[n] \leftarrow y$; $worklist \leftarrow \{n\}$;
3. **while** $worklist \neq \emptyset$ **do**
4. remove a node m from $worklist$;
5. **case** $m = entry_{main}$:
6. **if** $query[m] \sqsupset \bot$ **return**(*false*);
7. **case** $m = entry_q$ for some procedure q:
8. **for each** call site m' such that $call(m') = q$ **do**
9. $query[m'] \leftarrow query[m'] \sqcup query[m]$;
10. **if** $query[m']$ changed **then** add m' to $worklist$;
11. **endfor**;
12. **otherwise**:
13. **for each** $m' \in pred(m)$ **do**
14. $new \leftarrow \begin{cases} f^r_{m'}(query[m]) & \text{if } m' \text{ is not a call site} \\ \phi^r_{(entry_q, exit_q)}(query[m]) & \text{if } call(m') = q \end{cases}$
15. **if** $(new = \top)$ **then return**(*false*)
16. **else**
17. $query[m'] \leftarrow query[m'] \sqcup new$;
18. **if** $query[m']$ changed **then** add m' to $worklist$;
19. **endif**;
20. **endfor**;
21. **endwhile**;
22. **return**(*true*);
end

Fig. 5. Generic demand-driven analysis procedure.

work list is exhausted and all queries have evaluated to *true*.

To determine the complexity of the query algorithm the number of join operations and reverse function applications is considered. A join/reverse function application is performed at a node n in lines 9, 14, and 17 only if the query at a successor of n has changed (or at the entry node of a procedure p if n is a call site of p), which can happen at most $O(height(L))$ times. Hence, procedure *Query* requires in the worst case $O(height(L) \times |N|)$ join operations and/or reverse function applications.

If the program under analysis consists of only a single procedure (i.e., *intra*-procedural analysis), procedure *Query* provides a complete implementation of the demand-driven data flow analysis. The *inter*procedural case requires an efficient algorithm to compute the reverse summary functions. The algorithm to compute reverse summary function values is obtained as a variation of the Sharir and Pnueli work list algorithm for computing the forward summary functions in the exhaustive framework presented in Section 2. However, here summary functions are computed in the reverse direction. Assuming that (1) the asymptotic cost of a meet and a join are same and that (2) the asymptotic cost of flow function application and of reverse flow function application are the same, the algorithm presented in this section has the same worst-case complexity as Sharir and Pnueli's algorithm for the

Procedure $Compute\phi^r(p, y)$
input: lattice element $y \in L$, procedure p and table M that is initialized with element $\perp$ prior to the first invocation of $Compute\phi^r$
output: the reverse summary function value $\phi^r_{(entry_p, exit_p)}(y)$
begin
  if $M[exit_p, y] = y$ **then** /* *result previously computed* */
    return($M[entry_p, y]$);
  $worklist \leftarrow \{(exit_p, y)\}$; $M[exit_p, y] = y$;
  while $worklist \neq \emptyset$ **do**
    remove a pair (n, x) from $worklist$ and let $z \leftarrow M[n, x]$;
    case n is a call site and $call(n) = q$:
      if $M[exit_q, z] = z$ **then**
        for each $m \in pred(n)$ **do**
          $Propagate(m, x, M[entry_q, z])$;
      else /* *trigger computation of* $\phi^r_{(entry_q, exit_q)}(z)$ */
        $M[exit_q, z] \leftarrow z$ and add $(exit_q, z)$ to $worklist$;
    case $n = entry_q$ for some procedure q:
          /* *Propagate* z *to call sites if needed* */
      for each call site m such that $call(m) = q$ and $M[m, x'] = x$ for some x' **do**
        for each $m' \in pred(m)$ **do** $Propagate(m', x', z)$;
    otherwise:
          /* n *is not a call site and not an entry node* */
        for each $m \in pred(n)$ **do** $Propagate(m, x, f^r_n(z))$;
  endwhile;
  return($M[entry_p, y]$);
end

/* *propagate new to* $M[n, y]$ */
Procedure $Propagate(n, y, new)$
input: a node n, lattice elements y and new
begin
  $M[n, y] \leftarrow M[n, y] \sqcup new$;
  if $M[n, y]$ changed **then** add (n, y) to $worklist$;
end

Fig. 6. Procedure $Compute\phi^r$ to compute reverse summary functions.

original summary functions.

Figure 6 shows procedure $Compute\phi^r$ which is invoked with a pair (p, y) specifying a procedure p and a lattice element y. Procedure $Compute\phi^r$ returns the summary function value $\phi^r_{(entry_p, exit_p)}(y)$ after evaluating the necessary subsystem of the reverse equation system (4). Individual function values are stored in a table $M : N \times L \mapsto L$ such that $M[n, y] = \phi^r_{(n, exit_q)}(y)$, where q is the procedure that contains node n. The table is initialized with the value $\perp$, and its contents are assumed to be preserved between subsequent calls to procedure $Compute\phi^r$. Thus, results of previous calls are reused, and the table is incrementally computed during a sequence of calls. After calling $Compute\phi^r$ with a pair (p, y), the work list is initialized with the pair $(exit_p, y)$. The contents of the work list indicate the table entries whose values have changed but whose new values have not yet been propagated. During each step a pair is removed from the work list; its new value is determined, and all other entries whose values might have changed as a result are added to the work list.

Consider the cost of k calls to $Compute\phi^r$. Storing the table M requires space for $|N| \times |L|$ lattice elements. To determine the time complexity consider the number of join operations (in procedure *Propagate*) and of reverse flow function applications (at the call to *Propagate* in line 16). The loop in lines 4–17 is executed $O(height(L) \times |L| \times |N|)$ times, which is the maximal number of times the lattice value of a table entry can be raised, i.e., the maximal number of additions to the work list. In the worst case, the currently inspected node n is a procedure entry node. Processing a procedure entry node results in calls to *Propagate* for each predecessor of a call site for that procedure. Thus, the k calls to $Compute\phi^r$ require in the worst case $O(\max(k, MaxCall \times height(L) \times |L| \times |N|))$ join and/or reverse function applications, where $MaxCall$ is the maximal number of call sites calling a single procedure. Procedure $Compute\phi^r$ requires $O(|N| \times |L|)$ space to store lattice elements.

3.6 Caching

Processing a sequence of k queries requires k separate invocations of procedure *Query*, which may result in the repeated evaluations of the same intermediate queries. Repeated query evaluation can be avoided by maintaining a cache. Enhancing procedure *Query* to include caching requires only minor extensions. The cache consists of entries $cache[n, y]$ for each node n and lattice element y. Each entry contains the previous result, if any, of evaluating the query $\langle y, n\rangle$. The query propagation is modified such that each time before a newly generated query q is added to the work list, the cache is consulted. The query q is added to the work list only if the answer for q is not found in the cache.

Entries are added to the cache after the termination of a query evaluation in a single pass over the visited nodes. The inclusion of caching has the effect of incrementally building the data flow solution during a sequence of calls to *Query*. Caching does not increase the asymptotic time or space complexity of procedure *Query*. Storing the cache requires $O(|N| \times |L|)$ space, and updating the cache can at most double the amount of work performed during the query evaluation. Moreover, the asymptotic worst-case complexity of k invocations of *Query*, if caching is used, is the same for any k distinct queries.

3.7 Procedures with Parameters

This section extends the query propagation rules to handle procedures with parameters and local variables. For simplicity, we consider programs with global and local scoping. Hence, the address space $Addr(p)$ of a procedure p is defined as

$$Addr(p) = GV \cup LV(p) \cup PV(p),$$

where GV is the set of global variables; $LV(p)$ is the set of variables local to procedure p and $PV(p)$ is the set of formal parameters of p. The parameters of p may be either value or reference parameters.

To describe the data flow effects of parameter passing we define a *binding function* b_s for each call site s. The binding function b_s models the binding of a set of variables V from the calling procedure (caller) to the set of variables $b_s(V)$ in the called procedure (callee) according to the parameters passed at s. The binding function b_s also expresses the implicit binding of every global variable to itself:

$$\forall v \in GV : b_s(\{v\}) = \{v\}$$

Except for variables in GV, the only variables from the address space of a caller that are accessible in a callee are the variables that are explicitly passed as reference parameters. Let ap be an actual parameter that is bound at call site s to the formal reference parameter fp in the called procedure q:

$$b_s(\{ap\}) = \{fp\}$$

Local variables in the caller that are passed by value or that are passed as part of an expression to a value parameter are not bound to any variable in the callee. Hence $b_s(\{v\}) = \emptyset$ if $v \notin GV$ and if v is not passed to a reference parameter at s. The bindings for a set V of variables are determined as the union of the bindings for each variable in V:

$$b_s(V) = \{v \in V \mid b_s(\{v\})\}$$

In addition we need to describe the data flow effects on global and local variables that result after returning from a called procedure. These data flow effects have previously been described by specific *return functions* [Knoop and Steffen 1992]. Similarly, we define for our analysis the *reverse binding* b_s^{-1} that binds variables from the called procedure to the corresponding variables in the calling procedure. Specifically, let *fp* be a formal reference parameter of the callee that was passed at call site s as the actual parameter *ap* from the address space of the caller. Then we have

$$b_s^{-1}(\{fp\}) = \{ap\}.$$

For a global variable $v \in GV$ we have

$$b_s^{-1}(\{v\}) = \{v\}.$$

Finally, for variables v local to the callee, including formal value parameters we define

$$b_s^{-1}(\{v\}) = \emptyset.$$

Binding functions are defined over sets of variables. However, data flow analysis requires the binding of lattice elements at call sites. Thus, for each data flow problem it is assumed that two functions $\tilde{b}_s$ and $\tilde{b}_s^{-1}$ are defined to be the corresponding counterparts of b_s and b_s^{-1} that are applicable to the lattice elements in the data flow problem.

Example. We illustrate the use of the binding functions in CCP for the program fragment shown in Figure 7. Consider the query that raises the question as to whether variable f is a copy constant at the write statement. How this query is propagated across the recursive call depends upon whether f is a value parameter and thus local to the caller or whether it is a reference parameter for which a binding relation to a variable in the called procedure exists.

First assume that f is a value parameter and thus that $b_s(\{f\}) = \emptyset$, expressing that no summary information for the callee is needed and that the query can simply

```
procedure main                procedure p(f)
begin                         begin
    call(p(x));                   if (cond) then
end                               f:=5;
                                  else
                                      read(f);
                                      call p(f);
                                      write(f);
                                  endif;
                              end
```

Fig. 7. A recursive procedure with parameters.

be copied across the call. At the read statement the query can then be falsified, establishing that f is not a copy constant.

Now assume that f is a reference parameter. In this case $b_s(\{f_{caller})\} = \{f_{callee}\}$. Note that this equation refers to two distinct instances of variable f as noted by the subscripts. This binding relation indicates that summary information concerning the formal parameter f is required. Thus, a procedure summary computation is triggered to determine the procedure entry query that results when querying whether the formal f is a copy constant on procedure exit. The summary computation determines along one branch of the if statement that the query resolves to *true* with the value 5. This value will eventually stabilize at the procedure entry node indicating that the formal parameter f will always have value 5 on procedure exit independently of the procedure entry value of f. Using this summary information at the recursive call causes the initial query to resolve to *true* indicating that every instance of the formal f always has constant value 5 at the write statement.

3.8 Aliasing

The presence of reference parameters causes an additional complication for data flow analysis by introducing the potential of aliasing. As for standard exhaustive analysis, ignoring the potential of aliasing may lead to unsafe information during query propagation.

Two variables x and y are *aliases* in a procedure p if x and y may refer to the same location during some invocation of p. One source of aliasing in a program results from reference parameters. Reference parameters may introduce aliases through the binding mechanisms between actual and formal parameters. For example, if a global variable x is passed to a formal parameter f, then the alias pair (x, f) is created in the called procedure. Similarly, passing the same variable to two distinct formal parameters f_1 and f_2 creates the alias pair (f_1, f_2) in the called procedure.

Approximate alias information can be expressed in the form of the two summary relations *MayAlias*(p) and *MustAlias*(p) for each procedure p [Cooper 1985]. A pair (x, y) is contained in *MayAlias*(p) if x is aliased to y in some invocation of p. A pair (x, y) is in *MustAlias*(p) if x is aliased to y in all invocations of p. The computation of alias relations induced by reference parameters can be modeled as a data flow problem over a program's call graph [Cooper 1985]. An exhaustive algorithm for computing the alias sets iterates over the program's call graph to compute the potential alias pairs for all procedures prior to the analysis [Cooper 1985]. However, the data flow problem to compute the alias sets $MayAlias(p)$ and $MustAlias(p)$ is a distributive problem with a finite lattice. Thus, the demand-driven analysis concepts from the previous sections can be employed to compute the alias pairs as needed during query propagation.

The presence of aliasing is handled in data flow analysis by refining the flow functions to safely reflect the potential alias relationships. Consider, for example, how alias information is used to refine CCP analysis. A variable x is considered a constant at node n if either x or one of x's *must* aliases is assigned a constant value. A potential constant value of variable x is assumed to be destroyed if x or any of x's *may* aliases is assigned a nonconstant expression. For example, consider a variable v and the refinement of the CCP flow function component f_v (i.e., the projection of the flow function f to the component for v) at an assignment statement $w := 0$. Let l be a CCP lattice value, and let l_v be the component of l that denotes the lattice value for variable v:

$$f_v(l) = \begin{cases} 0 & \text{if } (v, w) \in MustAlias(p) \\ l_v \sqcap 0 & \text{if } (v, w) \in (MayAlias(p) - MustAlias(p)) \\ l_v & \text{otherwise} \end{cases}$$

The refined reverse flow function for the assignment is defined as

$$f^r([v{=}const]) = \begin{cases} \bot & \text{if } (v, w) \in MustAlias(p) \text{ and } const = 0 \\ \top & \text{if } (v, w) \in MayAlias(p) \text{ and } const \neq 0 \\ [v{=}const] & otherwise. \end{cases}$$

The analysis refinements are also applicable and safe if aliasing results from sources other than reference parameters. Other sources of aliasing in a program include pointer variables and array references. For example, the execution of the C statement $a := \&b$ creates the alias pair $(*a, b)$. Several techniques have been developed to approximate alias information in programs with pointer variables [Choi et al. 1993; Emami et al. 1994; Landi and Ryder 1992]. Aliasing introduced by pointer variables is more complicated than reference parameter aliasing since the alias relationships change intraprocedurally, and aliases may be introduced by arbitrary levels of indirection. As long as approximate but safe alias information for pointer variables is available, the information can be used to refine the analysis as described in this section.

4. A FRAMEWORK VARIATION FOR NONDISTRIBUTIVE PROBLEMS

The demand-driven framework from the previous section assumes that data flow problems are distributive. The distributivity of the flow functions is necessary to ensure that the query propagation rules yield as precise information as the original exhaustive analysis does. This section considers demand-driven analysis for data flow problems with nondistributive flow functions. First, we show that the demand-driven framework is approximate if applied to a nondistributive data flow problem. Section 4.2 outlines a two-phase framework variation that provides precise query responses, even for nondistributive problems.

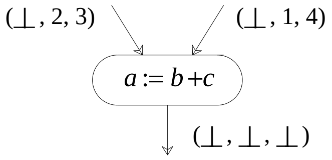

Fig. 8. Expression node in constant propagation.

4.1 Approximate Demand-Driven Analysis

If applied to distributive data flow problems, the query propagation rules are precise; given a data flow query $q = \langle y, n \rangle$, q evaluates to *true* if and only if element y is part of the solution at node n. In the presence of nondistributive flow functions information may be lost during query propagation. If a flow function f is monotone but not distributive, the relation between f and its reverse f^r is weaker than in the distributive case.

Recall that the distributivity of the flow functions establishes a relationship (1) between a function and its reverse that is expressed by the following two inequalities:

$$(f^r \cdot f)(x) \sqsubseteq x \text{ and } (f \cdot f^r)(x) \sqsupseteq x$$

The relationship can equivalently be expressed for lattice elements x and y as [Cousot 1981; Hughes and Launchbury 1992]:

$$y \sqsubseteq f(x) \Longleftrightarrow x \sqsupseteq f^r(y)$$

If the function f is not distributive and merely monotone, the preceding relationship between f and its reverse may no longer hold. Specifically, if f is monotone and not distributive, only the following can be established:

$$(f^r \cdot f)(x) \sqsubseteq x$$

And equivalently only one direction of the above equivalence holds:

$$y \sqsubseteq f(x) \Longrightarrow x \sqsupseteq f^r(y)$$

Example. We illustrate the loss of precision that results from nondistributive flow functions using the example of constant propagation for the expression node shown in Figure 8. Unlike CCP, regular constant propagation (CP) includes the evaluation of arithmetic expressions. Consider the flow function for the node in Figure 8. Each lattice element is a triple (x_a, x_b, x_c) with one component for each of the three variables a, b, and c. The flow function f_{cp} for the assignment is of the form:

$$f_{cp}(x_a, x_b, x_c) = \begin{cases} (x_b + x_c, x_b, x_c) & \text{if both } x_b \text{ and } x_c \text{ denote constant values} \\ (\perp, x_b, x_c) & \text{otherwise} \end{cases}$$

We first show that f_{cp} is not distributive; and then we show that relationship (1) does not hold for f_{cp} and its reverse f^r_{cp}.

Claim 1: Function f_{cp} is Not Distributive. Consider the situation where element $(\perp, 2, 3)$ is propagated to the node along the left incoming branch and element $(\perp, 1, 4)$ is propagated along the right incoming branch. Applying the flow function f_{cp} to each incoming value in isolation yields $f_{cp}(\perp, 2, 3,) = (5, 2, 3)$ and $f_{cp}(\perp, 1, 4,) = (5, 1, 4)$. Thus, with respect to each branch the lattice value on exit of the node indicates correctly that variable a has the constant value 5: $f_{cp}(\perp, 2, 3,) \sqcap f_{cp}(\perp, 1, 4,) = (5, \perp, \perp)$. However, if the information that reaches the node along the two incoming paths is merged prior to applying the flow function, it will not be discovered that variable a has value 5: $f_{cp}((\perp, 2, 3,) \sqcap (\perp, 1, 4,))$ $= f_{cp}(\perp, \perp, \perp) = (\perp, \perp, \perp)$. Hence, f_{cp} is not distributive.

Claim 2: Relationship (1) is Violated for f_{cp} and Its Reverse f_{cp}^{-r}. Consider the reverse function f_{cp}^{r}, and assume it is applied to the lattice element $[a = 5]$ that denotes that variable a has value 5. By definition $f_{cp}^{r}([a = 5])$ is the meet over all elements (x_a, x_b, x_c) such that $f_{cp}(x_a, x_b, x_c) \sqsupseteq [a = 5]$. There are infinitely many values for variables b and c that would result in a constant value 5 for variable a when executing the assignment $a := b + c$. Since the meet over this infinite set of possible constant values is bottom, it follows that $f_{cp}^{r}([a = 5]) = (\perp, \perp, \perp)$, incorrectly suggesting that the value $(\perp, \perp, \perp)$ on node entry is sufficient for a to have value 5 on node exit. Thus, the relationship $f_{cp} \cdot f_{cp}^{r}(x) \sqsupseteq x$ does not hold for the nondistributive function f_{cp}.

As a result of the weaker relationship between a nondistributive flow function and its reverse, the query propagation rules no longer provide equivalent translations. One direction of the equivalence in the propagation rule definition is violated. Specifically, consider the propagation rule for translating a query at node n to a corresponding query at n's predecessor m assuming n has only a single predecessor. From the rules in Figure 4 we obtain for a distributive function f_m:

$$\langle y, n \rangle \iff \langle f_m^{r}(y), m \rangle$$

If f_m is not distributive the implication

$$y \sqsubseteq f(x) \impliedby x \sqsupseteq f^{r}(y)$$

may not hold. As a result of this, the implication "$\impliedby$" in the preceding query propagation rule may not hold, and only the following implication remains:

$$\langle y, n \rangle \implies \langle f_m^{r}(y), m \rangle$$

With only an implication in the query translation the rules no longer provide reliable answers. The preceding implication still ensures that if query $q = \langle y, n \rangle$ evaluates to *false*, then it must be that $y \not\sqsubseteq X(n)$. However, if the propagation rules yield a *true* answer nothing can be said. If appropriate worst-case assumptions are made for *true* responses in the query propagation, the query algorithm may still be used to provide approximate information in the presence of nondistributive functions.

4.2 Framework Variation

We extend demand-driven analysis algorithms to nondistributive problems by departing from the concepts of precise analysis reversal. The loss of information that results from reversing a nondistributive function f occurs if the inequality $f \cdot f^{r}(x) \sqsupseteq x$ is violated. In this case a query q cannot be safely propagated across the node because it is no longer possible to translate q into an equivalent set of new queries at preceding nodes. However, it may be possible to "guess" the new queries that would be sufficient to provide the answer for q. The new queries must be chosen such that, given their answers, the answer for q can be found. However, unlike the distributive case, the relationship between the answers for the new queries and the answer for q is left unspecified. To illustrate this strategy consider again the assignment statement $a := b + c$ in constant propagation. Assume that the query $q = \langle [a = 0], n \rangle$ is raised on exit of this statement. The answer for q directly results once it has been determined whether the two operands b and c are constants. Thus, the new queries generated at predecessors m of n are of the form $q' = \langle [b, c], m \rangle$. Note that these new queries are merely approximate, since no specific constant values for variables b and c are established. Thus, the lattice element $[b, c]$ expresses that variables b and c have constant but unknown values. After all guessed queries for constants have been identified during the backward propagation, an additional analysis phase is performed in a forward direction to determine the actual constant values for the identified queries. Thus, if b and c are indeed constants, the second phase provides their values, and the original query q can be resolved.

The complete two-phase analysis variation operates as follows. The first phase, called the *marking phase*, is a backward analysis during which all guessed queries are marked. The guessed queries are marked by identifying the portion of the solution at each node that would be sufficient to answer the current query. The marked portion of the solution is then formulated as a corresponding query for the predecessor node. In the constant propagation example the marking phase identifies at each node the variables for which a solution would be needed in order to answer the input query. The marked queries describe the set of data flow queries whose answers provide an answer for the original input query. If during the marking phase a procedure call is encountered, marking within the called procedure is performed by a summary computation in the same style as described for the distributive case (i.e., by means of explicitly computed summary functions).

The second phase is essentially a partial version of the original exhaustive forward analysis that is a result of performing the forward analysis on only the solution elements marked during the first marking phase. In the constant propagation example, the second phase performs a limited version of traditional constant propagation analysis. However, unlike exhaustive constant propagation analysis that starts at program entry and propagates all constants throughout the entire program, the second phase only propagates the subset of the constants needed to resolve previously guessed queries by considering only the marked portion of the program.

The preceding strategy of using a preparatory backward analysis in order to reduce the analysis effort of the original forward analysis is not limited to the problem of constant propagation. The preparatory marking phase can be thought of as a filter that is applied in order to identify some portions of the solution that are guaranteed to be irrelevant and therefore can be ignored during the actual analysis (second phase). However, in general, it cannot be guaranteed that the preparatory phase actually leads to a reduction in the second phase. In the worst case, the entire program is marked during the first phase, in which case the complete exhaustive original analysis would be performed during the second phase. Thus, if the data flow problem is distributive, the demand-driven approach of choice is the reversal-based analysis framework developed in Section 3.

5. APPLICATIONS

We presented a framework for demand-driven data flow analysis that includes a generic demand-driven algorithm. It remains to be shown that demand-driven algorithms have efficient implementations in practice. Since the generic algorithm is expressed in general terms, a straightforward implementation may not be the most efficient one for a given data flow problem. This section considers two analysis problems – namely, reaching-definition analysis and copy constant propagation – and shows that it is possible to efficiently implement the generic algorithm by exploiting the specific properties of these data flow problems.

5.1 Demand-Driven Reaching-Definition Analysis

Reaching-definition analysis is one of the classical bit vector problems. The algebraically simple definition of bit vector problems allows for efficient implementations of the lattice elements and lattice operations using Boolean operations on bit vectors.

A *definition* of a variable v is any statement that assigns a value to a. A definition d is a *reaching-definition* at a program point n if there exists a valid execution path from definition d to n along which the defined variable is not redefined. The lattice elements in reaching-definition analysis are subsets of the set Def of definitions in the program, and the meet operator is set union ($\cup$). We use $Def(v)$ to denote the definitions of variable v.

A specialized instance of the general framework for computing reaching-definitions is obtained by specializing the individual components of the framework: (1) the query definition, (2) the query propagation rules, and (3) the generic analysis algorithm. The specialization presented here assumes C-style programs with local and global scoping and procedures with value parameters. Extensions for handling reference parameters are straightforward and based upon the handling of procedure parameters as described in Section 3.7.

5.2 Specialized Queries and Propagation Rules

The generic framework from Section 3 assumes queries in a *true/false* format. While it is generally possible to gather reaching-definition information using *true/false* queries only, it may not be the most efficient way. Consider, for example, the problem of determining the set of definitions that reach a selected variable at a selected point. Using *true/false* queries, a separate query would have to be raised for each definition of the selected variable. The propagation of multiple queries for definitions of the same variables can easily be combined and resolved during a single propagation. To accommodate such a combined query propagation, we consider a specialized format for reaching-definition queries of the form $q = \langle v, n \rangle$ asking for *all* reaching-definitions of variable v that reach node n.

Adjusting the query propagation rules to resolve the specialized reaching-definitions queries results in a number of simplifications. Unlike the general case, the reverse flow functions for reaching-definitions can be determined statically based on locally determined information about the definitions contained at each node. This information is expressed in two variables $Pres_n(v)$ and $Def_n(v)$

—$Pres_n(v) = true$ if the execution of node n *preserves* the value of variable v, i.e, For each procedure p:

$$\{\langle v, entry_p\rangle\} = \begin{cases} \{true\} & \text{if } b_m^{-1}(v) = \emptyset \\ \bigcup\limits_{call(m)=p} \langle b_m^{-1}(v), m\rangle & \text{otherwise} \end{cases}$$

For a nonentry node n that is not a call site:

$$\{\langle v, n\rangle\} = \bigcup_{m \in pred(n)} \left\{ \begin{array}{ll} \{\langle v, m\rangle\} & \text{if } Pres_n(v) = true \\ \{true\} & \text{otherwise} \end{array} \right\}$$

$$\left\{ \begin{array}{l} Action: \\ Solution = Solution \cup Def_m(v) \end{array} \right\}$$

(a)

$$Pres^r_{(exit_p, exit_p)}(v) = true$$

$$Pres^r_{(n, exit_p)}(v) =$$

$$\bigvee_{m \in succ(n)} \left\{ \begin{array}{ll} Pres^r_{(entry_q, exit_q)}(b_m(v)) \wedge Pres^r_{(m, exit_p)}(v) & \text{if } m \in call(q) \\ Pres_m(v) \wedge Pres^r_{(m, exit_p)}(v) & \text{otherwise} \end{array} \right\}$$

$$\{Action: \quad if\ Pres^r_{(m, exit_p)}(v) = true\ then\ Def^r_p(v) = Def^r_p(v) \cup Def_m(v)\}$$

(b)

Fig. 9. (a) Specialized propagation rules and (b) summary computation for reaching-definitions.

if every definition of v that reaches the entry to node n also reaches the exit of node n.

—$Def_n(v)$ is the set of definitions of variable v that are generated when executing node n. Note that unless node n is a call site, $Def_n(v)$ is a singleton.

Figure 9(a) shows the resulting propagation rules. Note that the specialized query format $\langle v, n\rangle$ describes a set of queries and therefore no longer represents a single truth value. To accommodate the new format the propagation rules in Figure 9 are displayed using set notation. During the propagation of a query $\langle v, n\rangle$ all definitions that are encountered are collected in the solution set.

To determine $Pres_n(v)$ and $Def_n(v)$ if node n contains a call site we define the specialized instances of the procedure summary functions. Corresponding to the general summary function $\phi^r_{(entry_p, exit_p)}$ from Section 3, a Boolean summary function $Pres^r_{(entry_p, exit_p)}$ is defined for each procedure p as shown in Figure 9(b) such that

—$Pres^r_{(n,exit_p)}(v) = true$ if there exists a path in procedure p from node n to procedure exit that preserves the definitions of variable v, that is, a path that does not contain a definition of variable v.

While computing these reverse summary functions, the definitions that are encountered and that reach procedure exit are collected in a set $Def^r_p(v)$ as stated in the equation system in Figure 9(b). Thus, $Def^r_p(v)$ contains the definitions of variable v that are generated along some path from the entry to the exit of procedure p without being subsequently redefined. Note that $Def^r_p(v)$ not only contains the definitions directly contained in p but may also contain definitions that are generated by procedures subsequently called from p. Based on the summary functions, the query propagation rules can be extended to include the propagation across a call site $s \in call(p)$ by setting the variables $Pres_s(v)$ and $Def_s(v)$ as follows: $Pres_s(v) = Pres^r_{(entry_p,exit_p)}(v)$ and $Def_s(v) = Def^r_p(v)$. To illustrate the definitions of these variables, we show in Table II the definition sets for the program example from Figure 1.

5.3 Demand-Driven Algorithm

The instance of the generic demand-driven procedures *Query* and *Compute*ϕ^r for reaching-definition analysis can be derived from the specialized propagation rules in a straightforward way. The query algorithm takes as its input a query of the form $\langle v, n\rangle$ for a variable v and a node n and returns the set of definitions of v that interprocedurally reach node n.

The query format limits the number of queries that may be raised to one query per variable at each node. The maximal number of queries that may be generated when processing an input query $q = \langle v, n\rangle$ depends upon whether v is a local variable, a global variable, or a formal parameter. In the worst case, v is a formal parameter such that the initial query may change when propagating it through a procedure entry node to call sites. The initial query can generate additional queries with respect to all other variables. Hence, up to $MaxVar \times |N|$ queries may be generated, where $MaxVar$ is the maximal size of the address space in any procedure and where each generation of a query results in the inspection of at most $MaxCall$ other queries. The costs of propagating k queries, excluding the cost of computing summary information, is therefore $O(max(k, MaxVar \times |N| \times MaxCall))$. Next, consider the cost of k requests for summary computations. Unlike the general case in which reverse summary computations may be triggered for each lattice element, the specialized summary computation from Figure 9 only considers summary computation with respect to individual variables. Thus, at most $|GV| \times |N|$ entries may be generated, and each entry may require the inspection of at most $MaxCall$ other entries. The overall cost of k summary requests is $O(\max(k, |GV| \times |N|) \times MaxCall)$.[4] It follows that the total worst-case time for processing k queries is $O(\max(k, MaxVar \times MaxCall \times |N|))$.

[4]In programs that contain reference parameters, summary information is needed for global variables and formal reference parameters. The asymptotic complexity for k requests changes to $O(\max(k, (|GV| + MaxFP) \times |N| \times MaxCall))$ where $MaxFP$ is the maximal number of formal parameters in any procedure.

Table I. Data Flow Variables for Reaching-Definitions in Figure 1

		$Pres_n(v)$			$Def_n(v)$		
Proc.	Node n	a	b	c	a	b	c
main	2	*false*	*false*	*false*	a_{11}	b_{12}	c_{12}
	3	*false*	*true*	*true*	a_3	-	-
	4	*true*	*false*	*true*	-	b_4	-
	5	*true*	*true*	*false*	-	-	c_5
	6	*false*	*false*	*false*	a_{11}	b_{12}	c_{12}
	7	*false*	*true*	*true*	a_7	-	-
	8	*true*	*true*	*true*	-	-	-
p	10	*true*	*true*	*true*	-	-	-
	11	*false*	*false*	*true*	a_{11}	b_{11}	-
	12	*true*	*true*	*false*	-	-	c_{12}
	13	*true*	*true*	*true*	-	-	-

Table II. Summary Computation for Procedure p in Figure 1

$Pres^r_{(n,13)}(v)$				$Def^r_p(v)$		
node n	a	b	c	a	b	c
10	*false*	*false*	*false*	a_{11}	b_{11}	c_{12}
11	*false*	*false*	*false*			
12	*true*	*true*	*false*			
13	*true*	*true*	*true*			

5.4 Demand-Driven Copy Constant Propagation

Copy constant propagation (CCP) analysis is more complex than reaching-definition analysis. Like reaching-definitions, CCP is a distributive problem. However, unlike reaching-definitions, CCP cannot be broken into a separate analysis for each variable. This property of reaching-definitions analysis, which simplifies the analysis algorithm, was termed *partitionable* [Zadeck 1984] or *locally separable* [Reps et al. 1995].

5.5 Specialized Queries and Propagation Rules

An instance of the demand-driven framework for CCP is obtained by specializing the three framework components: (1) the query definition, (2) the query propagation rules, and (3) the generic analysis algorithm.

According to the general framework, a query requests a specific lattice element, i.e., a specific constant value c of a variable v at a node n. For example, the query $q = \langle [v = 0], n\rangle$ raises the question "Is variable v a copy constant at node n with value 0?" Using this *true/false* query format, queries with respect to each constant literal may be necessary to determine whether a variable is a constant. Generating such a potentially high number of queries is not only costly, it is actually unnecessary. The propagation of multiple queries with respect to the same variable that only differ in their constant value is identical except for the response upon termination. Thus, as in reaching-definition analysis, these queries can be combined into a single query of the form $\langle [v], n\rangle$ = "Is variable v a copy constant at node n?" As in the example of reaching-definitions, the query propagation is enhanced

(i) $\langle \bot, n\rangle \iff true$ and $\langle \top, n\rangle \iff false$

(ii) For each procedure p:

$$\langle [v], entry_p\rangle \iff \begin{cases} false & \text{if } b_m^{-1}(v) = \emptyset \\ \bigwedge_{call(m)=p} \langle [b_m^{-1}(v)], m\rangle & \text{otherwise} \end{cases}$$

(iii) For each nonentry node n:

$$\langle [v], n\rangle \iff \bigwedge_{m \in pred(n)} \begin{cases} \langle f_m^r([v]), m\rangle & \text{if } m \text{ is not a call site} \\ \langle \phi^r_{(entry_q, exit_q)}([v]), m\rangle & \text{if } call(m) = q \end{cases}$$

$$\{Action\text{: } Solution = Solution \sqcap Val_n(v)\ \}$$

(a)

$$\phi^r_{(exit_p, exit_p)}([v]) = [v]$$

$$\phi^r_{(n, exit_p)}([v]) = \bigsqcup_{m \in succ(n)} \begin{cases} f_m^r \cdot \phi^r_{(m, exit_p)}([v]\ ) & \text{if } m \text{ is not a call site} \\ \phi^r_{(entry_q, exit_q)}([v]) \cdot \phi^r_{(m, exit_p)}([v]\ ) & \text{if } call(m) = q \end{cases}$$

$$\{Action\text{: } if\ \phi^r_{(m, exit_p)}([v]) \notin \{\bot, \top\}\ then\ Val_p^r(v) = Val_p^r(v) \sqcap Val_m(v)\}$$

(b)

Fig. 10. (a) Specialized propagation rules and (b) reverse summary functions for CCP

with actions to collect information during the query resolution. The information collected are the actual constant values that are encountered when resolving the query as stated in the rules in Figure 10(a). Similar to the $Def_n(v)$ sets in reaching-definitions, we define a set $Val_n(v)$ that contains the constant value, if any, that v is assigned locally when executing node n.

The definition of the reverse summary functions for CCP is shown in Figure 10 (ii). The reverse summary function provides the necessary information to determine the sets $Val_s(v)$ at call sites $s \in call(p)$ such that

$$Val_s(v) = \begin{cases} Val_p^r(v) & if\ \phi_{(entry_p, exit_p)}([v]) \neq \top \\ \bot & otherwise. \end{cases}$$

The reverse flow functions and summary functions are illustrated in Tables III and IV for the program example from Figure 1.

Table III. CCP Reverse Flow Functions for Figure 1

Proc. *main*	$f_n^r([v])$			Proc. p	$f_n^r([v])$		
Node n	a	b	c	Node n	a	b	c
2	$-$	$-$	$-$	11	$\top$	$\top$	$[c]$
3	$\perp$	$[b]$	$[c]$	12	$[a]$	$[b]$	$\perp$
4	$[a]$	$[a]$	$[c]$				
5	$[a]$	$[b]$	$[b]$				
6	$-$	$-$	$-$				
7	$\top$	$[b]$	$[c]$				
8	$[a]$	$[b]$	$[c]$				

Table IV. CCP Summary Computation for Procedure p in Figure 1

$\phi^r_{(n,13)}([v])$				$Val_p^r(v)$		
Node n	a	b	c	a	b	c
10	$\top$	$\top$	$\perp$	-	-	1
11	$\top$	$\top$	$\perp$			
12	$[a]$	$[b]$	$\perp$			
13	$[a]$	$[b]$	$[c]$			

5.6 Demand-Driven Algorithm for CCP

The CCP instance of the generic query algorithm *Query* takes as input a query of the form $\langle [v], n\rangle$. The propagation of each query of the form $\langle [v], n\rangle$ combines the propagation of the set of queries (i.e., $\{\langle [v=0], n\rangle, \langle [v=1], n\rangle, \ldots\}$) by keeping track of all constant values that are encountered as specified by the propagation rules from Figure 10. The maximal number of queries that can be generated at each node is *MaxVar*. Thus, a total of $O(\mathit{MaxVar} \times |N|)$ queries can be generated, each requiring the inspection of at most *MaxCall* other nodes. Now consider the cost of computing the reverse summary functions as defined in Figure 10(b). Summary table entries are computed only for the base elements and are needed only with respect to one specific constant value c, since the result for one constant value implies the results for other values. Thus, at most $|GV| \times |N|$ entries are computed.[5] Each entry may contain a set of base elements and is therefore of size *MaxVar*. The computation of table entries requires in the worst case $O(|GV| \times \mathit{MaxVar} \times |N|)$ table updates, and each table update may trigger up to *MaxCall* other table updates. Assuming join and reverse function applications are performed pointwise, each join or function application requires $O(|\mathit{MaxVar}|)$ time resulting in the total time of $O(\max(k, \mathit{MaxCall} \times |GV| \times \mathit{MaxVar}^2 \times |N|))$ for k requests of reverse summary function evaluations. Thus, the overall time requirements are $O(\mathit{MaxCall} \times |GV| \times \mathit{MaxVar}^2 \times |N|)$.[6]

[5] In programs that contain reference parameters, summary information is needed for both global variables and formal reference parameters resulting in $O((|GV| + \mathit{MaxFP}) \times |N|)$ entries, where *MaxFP* is the maximal number of formal parameters in any procedure.

[6] For programs with reference parameters the overall time requirements are $O(\mathit{MaxCall} \times \mathit{MaxVar}^3 \times |N|)$.

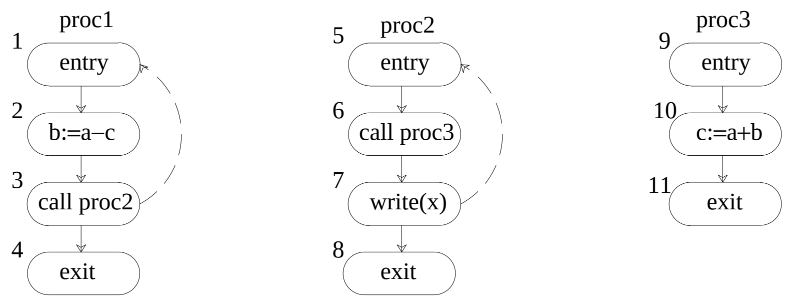

Fig. 11. Query advancing.

5.7 Optimization

This section describes a simple but effective optimization of the query propagation algorithm through *query advancing*. We illustrate query advancing using the example of reaching-definitions in Figure 11. The same advancing optimizations may be used for shortening the propagation paths of CCP queries.

There are two types of opportunities for query advancing:

—*Advancing across call:* Consider the query for reaching-definitions $\langle x, 7\rangle$ requesting the reaching-definitions of the global variable x at node 7 in Figure 11. Propagating the query across the call site at node 6 would require the computation of summary information about the called procedure. However, if it is known that the called procedure p (and procedures subsequently called from p) does not contain a node with a definition of variable x (or any of x's aliases) no summary computation is necessary, since all definitions of x must be preserved. Thus, the summary computation can be skipped, and the query $\langle x, 7\rangle$ can directly be forwarded across the procedure call as shown by the dashed arrow.

—*Advancing to entry:* Consider the propagation of the query $\langle x, 5\rangle$ from the entry of procedure *proc2* to the call site in procedure *proc1*. Based on the propagation rules from Section 3 the query would be translated into a new query at node 3. However, if it is known that procedure *proc1* (and any procedure subsequently called from *proc1*) does not contain a node with a definition of variable x (or any of x's aliases), it is not necessary to propagate the new query through procedure *proc1*. Instead the query can be directly forwarded to the entry node of *proc1* as shown by the dashed arrow.

The additional information needed for query advancing are the *flow-insensitive* procedure summary sets *Mod*(p) [Cooper and Kennedy 1988]. *Mod*(p) contains the variables that may be modified by the execution of procedure p because they are modified either directly in p or in a procedure subsequently called by p. Cooper and Kennedy presented a simple iterative work list algorithm that operates on the program's call graph to compute the *Mod* sets [Cooper and Kennedy 1988].

The summary information *Mod*(p) is called *flow-insensitive*, since determining this information does not require flow analysis and the control flow within each procedure can be ignored. In contrast, the summary information expressed by the reverse functions ϕ^r is called *flow-sensitive*, since it does require the control flow in each procedure to be analyzed.

6. PERFORMANCE EVALUATION

An experimental study was carried out to evaluate the practical benefits of the demand-driven approach. The study's primary objective was to compare the performance of demand-driven analysis algorithms with that of standard exhaustive algorithms. Additional experiments were carried out to evaluate the benefits of caching. The study examined analyzers for two analysis problems:

(1) interprocedural def-use chains based on reaching-definitions and

(2) interprocedural copy constant propagation as described in Section 5.

Computing def-use chains is a fundamental problem in most compiler optimizations. A def-use chain connects the definition of a variable with one of the uses of the defined value. Using reaching-definitions information, def-use chains are computed by pairing each use of a variable with each of the variable's definitions that reach the use. The demand-driven def-use chains analysis is based on the reaching-definitions analyzer from Section 5.

For each analysis problem three analyzer versions were implemented:

—a caching version of the demand-driven analysis algorithm,

—a noncaching version of the demand-driven analysis algorithm, and

—an exhaustive analysis algorithm. The exhaustive algorithms for reaching-definitions and copy constant propagation are based on the functional approach to interprocedural analysis by Sharir and Pnueli [1981]. Since the Sharir/Pnueli framework also serves as the basis for the demand-driven analysis framework, it provides a natural exhaustive counterpart to the demand-driven algorithms.

The three algorithms were implemented in C as part of the PDGCC compiler project at the University of Pittsburgh. The PDGCC system contains a C front end that provides statement-level control flow graphs. The implemented algorithms assume programs that are free of pointer-induced aliasing. Pointer references in C are handled by assuming that the address operator "&" destroys the value of the variables to which it is applied. Future versions will incorporate separately computed alias information into the analysis as described in Section 3 to safely handle aliasing without having to make overly conservative worst-case assumptions. An important aspect of the compiler front end that has direct implications on analysis performance is the treatment of temporary variables. The PDGCC front end generates single-assignment temporary variables. The use of single-assignment temporaries avoids the creation of artificial data dependencies among statements which may be beneficial for tasks such as register allocation. However, the generation of single-assignment temporaries also increases the size of the address space. Single-assignment temporaries are typically used in a fairly controlled way such that their uses and definitions are in nearby statements. Thus, temporaries may not actually require global analysis and could instead be analyzed locally. For example, it may be possible to determine the def-use chains for a temporary variable immediately

Table V. The Test Suite

Program	**Lines**	**Nodes**	**Proc.**	**Calls**	*MaxVar*
queens	84	150	4	4	38 (8)
heapsort	99	173	2	1	72 (24)
nsieve	115	192	2	2	40 (18)
cat	240	377	5	4	61 (15)
calendar	352	731	10	14	53 (10)
getopt	395	739	5	6	80 (12)
linpack	564	686	12	30	140 (17)
diff	818	899	12	33	211 (41)
patch	753	1316	14	13	141 (38)
tar	1214	1451	27	68	182 (49)
gzip	1245	2495	37	97	173 (48)
grep	1488	2906	32	72	127 (29)
sort	1528	3554	35	145	151 (19)
dc	1576	3298	67	230	91 (19)

after the temporary has been created. However, if the program changes, the locality property of references to temporaries may be destroyed, and a subsequent reanalysis of the program may have to consider temporary variables. In order to avoid a bias in the experimental results toward a particular strategy for handling temporary variables, the experiments are conducted in two versions. One version considers the complete address space in each procedure, including all compiler-generated temporaries, and the other version considers only source-level variables in the analysis.

The experiments were run on a SUN SPARCstation 5 with 32MB of RAM. Table V lists the 14 C programs that were used during the study and shows for each program the number of code lines, the number of control flow graph nodes, the number of procedures and call sites and the maximal number of variables in any one procedure. Parentheses indicate the number of source-level variables. In the following tables we use parentheses to indicate the results with respect to the source-level variable space that excludes compiler-generated temporaries.

Except for the first three, the programs are core routines of Unix utility sources. All reported analysis times are user CPU times in seconds determined using the Unix library routine *getrusage*. The reported analysis times reflect the mean value over five test runs. If query advancing was enabled in the demand-driven analyzer, the measured analysis times include the time to compute the *Mod* sets. All reported space measurements include only the amount of memory that is allocated for data flow vectors, cache memory, and other auxiliary structures that are needed for analysis purposes, such as the storage of the *Mod* sets if query advancing was enabled.

6.1 Experiment 1: Demand-Driven Versus Exhaustive Analysis

The first experiment compares the performance of the exhaustive analyzers with the performance of the caching demand-driven analyzers. The exhaustive analysis time T_{ex} and space consumption S_{ex} for each program are listed in Table VI for def-use chains and in Table VII for CCP. The caching demand-driven analyzers for

Table VI. Exhaustive Analysis Times for Def-Use Chains

Program	Time T_{ex} in sec.		Space S_{ex} in Kbytes	
queens	0.04	(0.02)	15.348	(13.548)
heapsort	0.09	(0.06)	22.756	(18.604)
nsieve	0.03	(0.04)	20.196	(20.196)
cat	0.20	(0.08)	43.424	(34.376)
calendar	0.17	(0.08)	82.164	(64.620)
getopt	0.98	(0.39)	105.372	(78.768)
linpack	0.53	(0.30)	227.312	(171.872)
diff	6.85	(2.26)	311.135	(180.012)
patch	2.05	(0.75)	230.424	(167.256)
tar	4.28	(2.12)	326.072	(220.712)
gzip	1.64	(0.91)	525.136	(405.376)
grep	4.56	(1.36)	437.704	(333.088)
sort	5.91	(1.85)	531.744	(361.720)
dc	1.11	(0.66)	416.508	(337.356)

Table VII. Exhaustive Analysis Times for Copy Constant Propagation (CCP)

Program	Time T_{ex} in Sec		Space S_{ex} in Kbytes	
queens	0.10	(0.04)	145.324	(41.068)
heapsort	0.71	(0.10)	382.340	(69.220)
nsieve	0.33	(0.18)	282.756	(125.780)
cat	0.25	(0.10)	334.800	(110.400)
calendar	0.26	(0.09)	463.432	(119.144)
getopt	2.44	(0.71)	2,003.272	(338.648)
linpack	2.80	(0.61)	3,556.032	(469.520)
diff	3.88	(0.99)	5,344.124	(1,342.140)
patch	93.53	(51.96)	24,459.728	(7,691.752)
tar	13.44	(6.91)	9,518.464	(4,813.808)
gzip	62.16	(31.55)	49,777.040	(21,696.240)
grep	3.45	(1.52)	4,277.056	(1,679.680)
sort	19.93	(7.20)	12,541.632	(2,854.096)
dc	14.35	(9.69)	10,296.716	(3,673.300)

both def-use chains and CCP analysis were executed with query advancing for a set of queries that contains one query for each use of a variable. Thus, the query responses of the def-use chain analyzer provide the set of all interprocedural def-use chains in the programs, and the responses of the CCP analyzer determine copy constant information at every use of a variable. The queries were generated in random order over the program.

The demand-driven analysis time T^{opt}_{cache} accumulated over all queries is shown in Table VIII for def-use chains and in Table IX for CCP analysis. The tables show the accumulated demand-driven analysis time and space (T^{opt}_{cache} and S^{opt}_{cache}). Tables VIII and IX also shows the *cache fill*, which is the percentage of the exhaustive solution that has been accumulated in the cache at the end of the demand-driven analysis. Thus, the cache fill indicates the portion of the exhaustive solution that is actually needed to answer all queries. The cache fill values show that actually only a small portion of the exhaustive solution is relevant. The remaining portion of

Table VIII. Demand-Driven Versus Exhaustive Analysis with Caching for Def-Use Chains

Program	Time (secs) T^{opt}_{cache}		Space S^{opt}_{cache}	Cache Fill %		Speedup $\frac{T_{ex}}{T^{opt}_{cache}}$		Space $\frac{S^{opt}_{cache} \times 100}{S_{ex}}$
queens	0.05	(0.04)	15.492	17	(41)	0.8	(0.5)	100.9
heapsort	0.12	(0.10)	22.436	25	(49)	0.75	(0.6)	98.6
nsieve	0.05	(0.04)	21.152	17	(28)	0.6	(1.0)	104.7
cat	0.09	(0.07)	41.924	16	(35)	2.2	(1.1)	96.5
calendar	0.08	(0.03)	69.900	7	(19)	2.1	(2.6)	85.1
getopt	0.32	(0.28)	99.988	16	(44)	3.1	(1.3)	94.9
linpack	0.49	(0.33)	222.716	13	(51)	1.1	(0.9)	97.9
diff	0.60	(0.48)	249.712	7	(18)	11.4	(4.7)	80.3
patch	1.01	(0.93)	201.884	19	(31)	2.0	(0.8)	87.6
tar	1.29	(1.20)	266.684	17	(23)	3.3	(1.7)	81.8
gzip	1.82	(1.62)	419.292	15	(20)	0.9	(0.5)	79.8
grep	0.84	(0.67)	365.152	12	(21)	5.4	(1.7)	83.4
sort	1.03	(0.94)	443.880	13	(35)	5.7	(1.9)	83.5
dc	0.71	(0.61)	373.460	9	(14)	1.6	(1.1)	89.7

Table IX. Demand-Driven Versus Exhaustive Analysis with Caching for CCP

Program	Time (secs) T^{opt}_{cache}		Space S^{opt}_{cache}	Cache Fill %		Speedup $\frac{T_{ex}}{T^{opt}_{cache}}$		Space $\frac{S^{opt}_{cache} \times 100}{S_{ex}}$
queens	0.07	(0.03)	92.904	12	(43)	1.4	(1.3)	63.9
heapsort	0.18	(0.13)	238.672	18	(54)	3.9	(0.7)	62.4
nsieve	0.08	(0.06)	101.672	15	(37)	4.1	(3.0)	35.9
cat	0.11	(0.09)	192.308	10	(28)	2.2	(1.1)	57.4
calendar	0.15	(0.04)	324.276	4	(16)	1.7	(2.2)	69.9
getopt	0.80	(0.29)	1,608.344	4	(25)	3.0	(2.4)	80.2
linpack	0.80	(0.47)	1,577.688	6	(50)	3.5	(1.2)	44.3
diff	1.03	(0.62)	3,108.972	4	(17)	3.7	(1.5)	58.1
patch	2.11	(1.16)	4,206.108	5	(12)	44.3	(44.7)	17.1
tar	1.89	(1.16)	4,180.120	8	(13)	7.1	(5.9)	43.9
gzip	4.97	(2.83)	9,333.080	4	(7)	12.5	(11.1)	18.7
grep	1.21	(0.78)	2,483.448	6	(16)	2.8	(1.9)	58.0
sort	1.45	(0.82)	3,252.552	5	(21)	13.7	(8.7)	25.9
dc	1.58	(0.74)	2,944.576	5	(12)	9.0	(13.1)	128.5

the solution consists of useless reaching-definitions or copy constant information for variables that are no longer live in the current procedure. Demand-driven analysis naturally suppresses the computation of the useless information of dead variables, since this information is not queried. [7]

For the complete variable space including temporaries, the relevant portion ranges from 7% to only 25%. As expected, when temporaries are excluded, the relevant portion is higher, ranging from 14% to 51%. Temporary variables are likely to generate large portions of unneeded information, since temporary variables are usually defined and used at nearby points and are dead in the remaining portion of the containing procedure. However, Tables VIII and IX show that even after excluding temporaries from the analysis, on an average more than half of the solution is not needed.

Tables VIII and IX also display the speedups (T_{ex}/T^{opt}_{cache}) of the demand-driven analyzer with caching over the exhaustive analyzer. The demand-driven analyzer computes def-use chains faster than the exhaustive analyzer by factors ranging from 1.1 up to 11.4 in 10 out of 14 test programs. In CCP, the demand-driven analysis outperforms the exhaustive analysis in all programs with speedup factors ranging from 1.4 up to 44.3. The exclusion of temporaries causes a larger portion of the exhaustive solution to be computed (i.e., a higher cache fill) and therefore results in slightly lower speedups.

Tables VIII and IX also show the space savings of the demand-driven analyzer as a percentage of the exhaustive space. The worst-case space requirements of the demand-driven analysis are higher than for standard exhaustive analysis by a small constant amount, since in addition to data flow solutions a "visited" flag needs to be stored at every node in the graph. However, the tables show that in practice demand-driven analysis requires less space in almost all programs. The lower space requirements are expected, since demand-driven analysis computes less information than exhaustive analysis. The space savings are primarily due to the fact that demand-driven analysis permits the suppression of unnecessary procedure summary computations.

6.2 Experiment 2: Noncaching Versus Exhaustive

A second experiment was conducted to determine the effect of caching on the performance of the demand-driven analyzers. The noncaching demand-driven analysis (with query advancing) was executed with the same set of queries as in the first experiment. The accumulated analysis times T^{opt} and space consumption S^{opt} are shown in Table X for def-use chains and in Table XI for CCP. The tables show the accumulated analysis time T^{opt} and the amount of space used (S^{opt}) for the noncaching analyzers. Tables X and XI also show the speedup (T^{opt}/T^{opt}_{cache}) of

[7]Note that the same redundancies in the exhaustive solution would result if, instead of reaching-definition analysis, the directional dual live-use analysis were used to compute the def-use chains. In exhaustive live-use analysis, the uses of variables may be propagated past the points where the respective variable is live. Avoiding the propagation of live-use information through the program portion where the respective variables are dead would require dynamically changing the bit vector sizes during the analysis each time a variable becomes dead. The overhead of changing bit vector sizes is likely to quickly outweigh the savings that may result from avoiding the useless information propagation.

Table X. The Benefits of Caching for Def-Use Chain Analysis

Noncaching Demand-Driven Analysis			**Caching vs. Noncaching**	
Program	Time (secs) T^{opt}	Space S^{opt}	Speedup $\frac{T^{opt}}{T^{opt}_{cache}}$	Space % $\frac{(S^{opt}_{cache} \times 100)}{S^{opt}}$
queens	0.06 (0.02)	14.612	1.2 (0.5)	106.0
vheapsort	0.17 (0.15)	20.196	1.4 (1.5)	111.0
nsieve	0.10 (0.05)	19.304	2.0 (1.2)	109.6
cat	0.11 (0.10)	37.060	1.2 (1.4)	113.1
calendar	0.07 (0.05)	64.684	0.9 (1.6)	108.1
getopt	0.64 (0.60)	82.468	2.0 (2.1)	121.2
linpack	0.86 (0.73)	203.756	1.8 (2.2)	109.3
diff	1.09 (0.90)	184.856	1.8 (1.8)	135.1
patch	1.58 (1.43)	159.548	1.6 (1.5)	126.5
tar	1.87 (1.75)	200.204	1.5 (1.4)	133.2
gzip	2.84 (2.52)	336.348	1.5 (1.5)	124.6
grep	1.16 (1.05)	289.552	1.4 (1.3)	126.1
sort	1.27 (1.08)	352.600	1.2 (1.1)	125.9
dc	1.04 (0.80)	326.940	1.5 (1.3)	114.2

the demand-driven analyzer with caching over the demand-driven analyzer without caching and the space usage of the caching demand-driven analyzer as a percentage of the space used by the non-caching analyzer. Except for one of the short programs (queens) in the def-use chain analyzer, adding the caching capability resulted in moderate speedup factors of up to 2.2. The analysis of program queens resulted in too few cache hits, causing the savings to be less than the overhead of the cache management.

6.3 Experimentation Summary

The experimental results demonstrate that demand-driven analysis performs well in practice. The first experiment indicated that demand-driven analysis computes def-use chains and copy constant information faster and uses less space than exhaustive analysis in the majority of cases. Importantly, the speedups and space savings of the demand-driven analysis over the exhaustive analysis result even when def-use chains or copy constant information are computed over the entire program. Naturally, the benefits of using a demand-driven approach would be even higher if the amount of requested information were lower. The second experiment showed that, except for very short programs, demand-driven analysis benefits from caching. Again, the reported benefits result if demand-driven analysis is used to service demands that are raised throughout the program. If fewer demands are raised, caching is likely to be less beneficial since there may not be enough cache hits to compensate for the cache management overhead.

An additional inspection of the benchmark programs with the highest and lowest

Table XI. The Benefits of Caching for CCP

Noncaching Demand-Driven Analysis			**Caching vs. Noncaching**	
Program	Time (secs) T^{opt}	Space (Kbytes) S^{opt}	Speedup $\frac{T^{opt}}{T^{opt}_{cache}}$	Space % $\frac{(S^{opt}_{cache} \times 100)}{S^{opt}}$
queens	0.09 (0.03)	87.448	1.2 (1.0)	106.2
heapsort	0.20 (0.14)	232.336	1.1 (1.1)	102.7
nsieve	0.12 (0.09)	88.680	1.5 (1.5)	114.6
cat	0.17 (0.09)	161.396	1.5 (1.0)	119.1
calendar	0.14 (0.08)	289.360	0.9 (2.0)	112.0
getopt	1.10 (0.55)	1,438.120	1.3 (1.8)	111.8
linpack	1.09 (0.69)	1,418.448	1.3 (1.4)	111.2
diff	1.39 (0.94)	2,360.244	1.3 (1.5)	131.7
patch	2.55 (1.55)	3,887.372	1.2 (1.3)	108.1
tar	2.70 (1.57)	3,734.432	1.4 (1.3)	111.9
gzip	5.14 (3.04)	8,815.544	1.0 (1.1)	105.8
grep	1.52 (1.08)	1,899.440	1.2 (1.3)	130.7
sort	1.99 (0.96)	2,378.416	1.3 (1.1)	136.7
dc	1.71 (0.90)	2,606.608	1.1 (1.2)	112.9

speedups was carried out in order to identify the program characteristics that affect the analyzers' performance. In general, the speedups of the demand-driven analyzer over the exhaustive analyzer are highest if the lengths of the propagation paths for the individual queries are the shortest. The length of query propagation paths depends primarily on reference locality properties. If variables are defined and used in nearby statements, the propagation paths are short. A number of program characteristics may have a direct impact on the length of propagation paths and analyzers' performance. These characteristics include program size, nesting depth of control structures, number of global variables, number and size of procedures, and the structure and density of the call graph. Exceptionally high speedups of demand-driven analysis over exhaustive analysis result in programs that combine several of these program characteristics.

7. RELATED WORK

Numerous analysis algorithms have been presented that utilize some form of delayed or demand-driven evaluation to allow for efficient analysis implementations. Among others, the concepts of deriving data flow information by backward propagation of assertions was used in a debugging system for Pascal-like programs with higher-order functions [Bourdoncle 1993] and in an analysis for discovering linked conditions in programs [Stoyenko et al. 1993]. A types-based framework was developed the analysis of functional programs provides efficient analysis algorithms through the demand-driven "lazy" evaluation of properties [Hankin and LeMetayer 1994].

Our algorithm to compute the relevant reverse summary function values is essentially a reversed version of Sharir and Pnueli's tabulation algorithm [Sharir and Pnueli 1981] to compute the original forward summary functions. The table-driven approach for computing procedure summaries pioneered by Sharir and Pnueli has also been used in various other interprocedural analyses [Landi and Ryder 1992; Marlowe and Ryder 1990a; Reps et al. 1995; Steensgaard 1996; Wilson and Lam 1995]. A similar partial fixed-point computation of only relevant equations was also described in the chaotic iteration algorithms [Cousot and Cousot 1978] and the minimal function graphs for applicative programs [Jones and Mycroft 1986]. Reverse flow functions have previously been discussed to demonstrate that an abstract interpretation may be performed in either a forward or a backward direction [Cousot 1981; Hughes and Launchbury 1992].

7.1 Demand-Driven Analysis

Closely related to our demand-driven framework are the approaches to demand-driven interprocedural analysis presented by Reps and Reps et al. [Reps 1994; Horwitz et al. 1995a; Reps et al. 1995; Sagiv et al. 1995]. In the first approach by Reps [1994], a limited class of data flow problems, the *locally separable problems*, is encoded as logic programs. Demand algorithms are then obtained by utilizing fast logic program evaluation techniques developed in the logic-programming and deductive-database communities. In more recent work [Horwitz et al. 1995a; Reps et al. 1995], the first approach is generalized to the larger class of interprocedural finite distributive subset (IFDS) problems. The class of IFDS problems is the subset of the distributive data flow problems that contains only problems with a *finite subset lattice*, i.e., the lattice is the powerset of a finite set. In this second approach, a data flow problem is transformed into a special kind of graph-reachability problem. The graph for the reachability problem, the *exploded supergraph*, is obtained as an expansion of a program's control flow graph by including an explicit graphical representation of each node's flow function.

The graph-reachability approach and our approach are closely related, as both provide an extension and variant of the classical functional approach to interprocedural analysis pioneered by Sharir and Pnueli [1981]. However, there are a number of important distinctions. Our approach is more general than the graph-reachability approach because our framework is precise for any distributive problem with a finite lattice, while their approach requires the lattice to be a finite subset lattice. Furthermore, our framework can still provide approximate information for monotone nondistributive problems, while their approach is not applicable to nondistributive problems.

Furthermore, their approach is graph-based, while our framework models demand-driven analysis using fixed-point computations. As a result the two approaches differ in how and when flow functions are evaluated. The graph-based approach explicitly represents the flow functions as part of the exploded supergraph. Thus, flow functions are evaluated as the graph is constructed prior to the actual analysis. In our approach flow functions are evaluated during the actual analysis traversal as needed.

The graph-reachability approach yields polynomial-time analysis algorithms even if the lattice is of exponential size, as is the case for a powerset lattice. The analysis examples presented in this article show how our framework is used to provide efficient specializations for problems with powerset lattices (i.e., reaching-definitions and copy constant propagation), also yielding polynomial-time algorithms.

An important overhead of the graph-reachability approach results from the need to construct a complete exploded supergraph for each data flow problem, independently of the actual number of demands that need to be satisfied. The overhead of constructing the exploded supergraph can be substantial, since the graph is of size $O(E \times D^2)$, where E is the number of control flow graph edges, and D is the number of finite data flow elements (e.g., the number of variables in copy constant propagation). Sagiv et al. [1995b] report that during experimentation with the graph-reachability analyzer for CCP, the analyzer exceeded the virtual memory limit when run on a Sun SPARCstation 20 with 64MB of RAM for some C programs of about 1300 lines.

Experiments for the graph-reachability approach have been conducted for problem of comparable complexity as reported in this article (i.e., live variables and truly live (nonfaint) variables, uninitialized variables, and constant predicates) [Horwitz et al. 1995b]. The performance of the demand-driven analysis algorithm is evaluated by comparison with an exhaustive algorithm that also operates on the exploded supergraph. Similar to our experimental results, their performance study makes a strong point for demand-driven analysis in that speedups of the demand-driven analyzer over the exhaustive analyzer could be achieved in most cases.

Recently, Sagiv et al. [1995a] presented a new variation of the graph-reachability approach that uses a two-phase algorithm. This new approach can handle a larger class of distributive data flow problems than our framework in that it also permits infinite lattices if the distributive function space does not contain infinite descending chains. This new variation also results in a more compact version of the exploded supergraph for CCP. However, for the classical *Gen-Kill* problems the size of the exploded supergraph is the same as in their previous approach.

The utility of demand-driven analysis has also been demonstrated in a number of algorithms that have been developed for specific analysis problems. A more general demand-driven algorithm was presented for *intra*procedural live variable analysis based on attribute grammars [Babich and Jazayeri 1978]. A demand-based analysis for typestate checking was presented in [Strom and Yellin 1993]. The authors experimentally demonstrated that a goal-directed backward analysis is more efficient than an eager forward analysis for typestate checking. Question propagation, a phase in the algorithm for global value numbering [Rosen et al. 1988], performs a demand-based backward search in order to locate redundant expressions. A demand-driven algorithm for range propagation was presented in [Blume and Eigenmann 1995]. Range propagation is performed only over the portion of the program that is relevant for the current information request. In *procedure cloning* [Cooper et al. 1992], procedure clones are created during the analysis on demand whenever it is found that an additional clone will lead to more accurate information. An algorithm for the incremental incorporation of alias information into static single-assignment (SSA) form was presented in [Cytron and Gershbein 1993]. The actual optimization problem to be performed on the SSA form triggers the expansion of the SSA form to include only the necessary alias information. Similar ideas have also been implemented in the demand-based expansion algorithm of factored def-def chains

[Choi et al. 1992].

8. CONCLUDING REMARKS

We presented in this article a new demand-driven approach which has been developed through a general framework. The framework is applicable to the class of distributive and finite data flow problems. To precisely handle the nondistributive case, this work also outlines a two-phase framework variation. The practical benefits of the demand-driven approach have been demonstrated through experimentation.

Using a demand-driven approach to data flow analysis in compilers and software tools results in a considerable change in their overall design, since demand-driven analysis does not obey the classical, strict, phased design of a compiler or software tool. In the phased design, data flow analysis is performed in isolation independently of its context and, in particular, independently of the application phase that follows the analysis. While such a strict separation into phases may simplify the overall design and implementation of a compiler or software tool, it also limits the information available to each individual phase and may thereby render the phases unnecessarily inefficient. Since nothing is known about the actual information demands of the application, the analysis must consider all possibly relevant data flow facts and is therefore necessarily exhaustive. In contrast, demand-driven analysis is directly interleaved with the application such that analysis is performed only if triggered by a demand. If caching is used, repeated invocations of the demand-driven analyzer result in the subsequent accumulation of the data flow solution. Thus, if exhaustively many demands are issued by the application, the demand-driven analyzer eventually accumulates the complete exhaustive solution.

As an additional benefit, our demand-driven algorithms have a natural parallelization. Individual queries can be propagated and resolved in parallel without requiring a separate phase to explicitly uncover parallelism [Duesterwald 1996]. Future investigation will further consider the utility of our demand-driven algorithms for the parallelization of data flow analysis.

ACKNOWLEDGMENTS

The authors gratefully thank the anonymous referees for their detailed comments and suggestions, which have helped improve both the contents and presentation of this article.

REFERENCES

scaled 833BABICH, W. AND JAZAYERI, M. 1978. The method of attributes for data flow analysis: Part II. Demand analysis. *Acta Inf. 10,* 3 (Oct.), 265–272.

BLUME, W. AND EIGENMANN, R. 1995. Demand-driven symbolic range propagation. In *Proceedings of the Workshop on Languages and Compilers for Parallelism.* Lecture Notes in Computer Science, vol. 1033, Springer Verlag, Berlin, 141–160.

BOURDONCLE, F. 1993. Abstract debugging of high-order imperative languages. In *Proceedings of the SIGPLAN 1993 Conference on Programming Language Design and Implementation.* ACM, New York, 36–45.

BURKE, M. 1987. An interval analysis approach toward exhaustive and incremental data flow analysis. Tech. Rep. RC 12702, IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y.

CHOI, J., CYTRON, R., AND FERRANTE, J. 1992. On the efficient engineering of ambitious program analysis. *IEEE Trans. Softw. Eng. 20,* 2 (Feb.), 105–114.

CHOI, J.-D., BURKE, M., AND CARINI, P. 1993. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages.* ACM, New York, 232–245.

COOPER, K. 1985. Analyzing aliases of reference formal parameters. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages.* ACM, New York, 281–290.

COOPER, K., HALL, M., AND KENNEDY, K. 1992. Procedure cloning. In *Proceedings of the IEEE 1992 International Conference on Computer Languages.* 96–105.

COOPER, K. AND KENNEDY, K. 1988. Interprocedural side-effect analysis in linear time. *Proceedings of the SIGPLAN 1988 Symposium on Compiler Construction, published in SIGPLAN Not. 23,* 7 (June), 57–66.

COUSOT, P. 1981. Semantic foundations of program analysis. In *Program Flow Analysis: Theory and Applications*, S. Muchnick and N. Jones, Eds. Prentice-Hall, Englewood Cliffs, N.J., 303–342.

COUSOT, P. AND COUSOT, R. 1978. Static determination of dynamic properties of recursive procedures. In *Proceedings of the IFIP Conference on Programming Concepts*, E. Neuhold, Ed. North-Holland Pub. Co., 237–277.

CYTRON, R. AND GERSHBEIN, R. 1993. Efficient accommodation of may-alias information in SSA form. In *Proceedings of the SIGPLAN 1993 Conference on Programming Language Design and Implementation.* ACM, New York, 36–45.

DUESTERWALD, E. 1996. A demand-driven approach for efficient interprocedural data flow analysis. Ph.D. Thesis, Univ. of Pittsburgh, Pittsburgh, Pa.

DUESTERWALD, E., GUPTA, R., AND SOFFA, M. 1992. Rigorous data flow testing through output influences. In *Proceedings of the 2nd Irvine Software Symposium.* 131–145.

DUESTERWALD, E., GUPTA, R., AND SOFFA, M. 1995. Demand-driven computation of interprocedural data flow. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages.* ACM, New York, 37–48.

EMAMI, M., GHIYA, R., AND HENDREN, L. 1994. Context-sensitive interprocedural points-to analysis on the presence of function pointers. In *Proceedings of the SIGPLAN 1994 Conference on Programming Language Design and Implementation.* ACM, New York, 242–256.

FRANKL, P. AND WEYUKER, E. 1988. An applicable family of data flow testing criteria. *IEEE Trans. Softw. Eng. SE-14,* 10 (Oct.), 1483–1498.

GRISWOLD, W. AND NOTKIN, D. 1993. Automated assistance for program restructuring. *ACM Trans. Softw. Eng. 2,* 3 (July), 228–269.

HANKIN, C. AND LEMETAYER, D. 1994. A type-based framework for program analysis. In *Proceedings of the 1st International Static Analysis Symposium.* 380–394.

HORWITZ, S., REPS, T., AND SAGIV, M. 1995a. Demand interprocedural dataflow analysis. In *Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering.* ACM, New York.

HORWITZ, S., REPS, T., AND SAGIV, M. 1995b. Demand interprocedural dataflow analysis. Tech. Rep. 1283, Computer Science Dept., Univ. of Wisconsin, Madison, Wisc. Aug.

HUGHES, J. AND LAUNCHBURY, J. 1992. Reversing abstract interpretations. In *Proceedings of the 4th European Symposium on Programming.* Lecture Notes in Computer Science, vol. 582, Springer Verlag, Berlin, 269–286.

JONES, N. AND MYCROFT, A. 1986. Data flow analysis of applicative programs using minimal function graphs. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages.* ACM, New York, 296–306.

KAM, J. AND ULLMAN, J. 1977. Monotone data flow analysis frameworks. *Acta Inf. 7,* 3 (July), 305–317.

KNOOP, J. AND STEFFEN, B. 1992. The interprocedural coincidence theorem. In *Proceedings of the 4th International Conference on Compiler Construction.* Lecture Notes in Computer Science, vol. 641, Springer Verlag, Berlin, 125–140.

LANDI, W. AND RYDER, B. 1992. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the SIGPLAN 1992 Conference on Programming Language Design and Implementation*. ACM, New York, San Francisco, Calif., 56–67.

MARLOWE, T. AND RYDER, B. 1990a. An efficient hybrid algorithm for incremental data flow analysis. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*. ACM, New York, Orlando, Fla., 184–196.

MARLOWE, T. AND RYDER, B. 1990b. Properties of data flow frameworks, a unified model. *Acta Inf. 28,* 2 (Dec.), 121–163.

POLLOCK, L. AND SOFFA, M. 1989. An incremental version of iterative data flow analysis. *IEEE Trans. Softw. Eng. 15,* 12 (Dec.), 1537–1549.

REPS, T. 1994. Solving demand versions of interprocedural analysis problems. In *5th International Conference on Compiler Construction*. Lecture Notes in Computer Science, vol. 786, Springer Verlag, Berlin, 389–403.

REPS, T., HORWITZ, S., AND SAGIV, M. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*. ACM, New York, 49–61.

REPS, T., TEITELBAUM, T., AND DEMERS, A. 1983. Incremental context-dependent analysis for language-based editors. *ACM Trans. Program. Lang. Syst. 5,* 3 (July), 449–477.

ROSEN, B., WEGMAN, M., AND ZADECK, F. 1988. Global value numbers and redundant computations. In *15th ACM Symposium on Principles of Programming Languages*. ACM, New York, San Diego, Calif., 12–27.

RYDER, B. 1983. Incremental data flow analysis. In *9th ACM Symposium on Principles of Programming Languages*. ACM, New York, 167–176.

SAGIV, M., REPS, T., AND HORWITZ, S. 1995. Precise interprocedural dataflow analysis with applications to constant propagation. In *FASE 95: Colloquium on Formal Approaches in Software Engineering*. Lecture Notes in Computer Science, vol. 915, Springer Verlag, Berlin, 651–665.

SHARIR, M. AND PNUELI, A. 1981. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, S. Muchnick and N. Jones, Eds. Prentice-Hall, Englewood Cliffs, N.J., 189–234.

STEENSGAARD, B. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*. ACM, New York, 32–41.

STOYENKO, A., MARLOWE, T., HALANG, W., AND YOUNIS, M. 1993. Enabling efficient schedulability analysis through conditional linking and program transformations. *Control Eng. Pract. 1,* 1, 85–105.

STROM, R. AND YELLIN, D. 1993. Extending typestate checking using conditional liveness analysis. *IEEE Trans. Softw. Eng. 19,* 5 (May), 478–485.

WEISER, M. 1984. Program slicing. *IEEE Trans. Softw. Eng. 10,* 4 (July), 352–357.

WILSON, R. AND LAM, M. 1995. Efficient context-sensitive pointer analysis for c programs. In *SIGPLAN 1995 Conference on Programming Language Design and Implementation*. ACM, New York, 1–12.

ZADECK, F. 1984. Incremental data flow analysis in a structured program editor. In *SIGPLAN Symposium on Compiler Construction*. ACM, New York, 132–143.

Received August 1996; revised May 1997; accepted August 1997