A Demand-Driven Analyzer for Data Flow Testing at the Integration Level [†]

Evelyn Duesterwald   Rajiv Gupta   Mary Lou Soffa

Department of Computer Science
University of Pittsburgh, Pittsburgh, PA 15260
{duester,gupta,soffa}@cs.pitt.edu

Abstract

Data flow testing relies on static analysis for computing the def-use pairs that serve as the test case requirements for a program. When testing large programs, the individual procedures are first tested in isolation during unit testing. Integration testing is performed to specifically test the procedure interfaces. The procedures in a program are integrated and tested in several steps. Since each integration step requires data flow analysis to determine the new test requirements, the accumulated cost of repeatedly analyzing a program can considerably contribute to the overhead of testing. Data flow analysis is typically computed using an exhaustive approach or by using incremental data flow updates. This paper presents a new and more efficient approach to data flow integration testing that is based on demand-driven analysis. We developed and implemented a demand-driven analyzer and experimentally compared its performance during integration testing with the performance of (i) a traditional exhaustive analyzer and (ii) an incremental analyzer. Our experiments show that demand-driven analysis is faster than exhaustive analysis by up to a factor of 25. The demand-driven analyzer also outperforms the incremental analyzer in 80% of the test programs by up to a factor of 5.

1 Introduction

Since its development for optimizing compilers, static data flow analysis has evolved as a primary component in various software engineering tools, including editors, debuggers and software testers. Many software engineering applications utilize data flow information only selectively in a program. For example, applications such as debugging and software testing often process a program in multiple sessions and each session may utilize data flow information only at selected portions of a program. However, while the utilization of data flow information may be partial, the data flow computation is traditionally performed exhaustively over a program. In the traditional analysis approach, the computation of data flow at one point requires data flow computations at *all* program points. Computing such exhaustive solutions in applications that actually utilize only a part of the data flow information necessarily results in the computation of information that is never used. This paper proposes demand-driven analysis as a more efficient analysis approach for data flow based software engineering applications. To demonstrate the benefits of demand-driven analysis, we consider its application in data flow integration testing and show how the cost of integration testing can be reduced through a demand-driven analysis design.

Data flow testing uses coverage criteria [14, 5] to select the sets of definition-use (def-use) pairs in a program that serve as the test case requirements. Def-use pairs are determined by solving the data flow problem of reaching definitions. The testing of large programs usually takes place at several levels. The individual program units are tested first in isolation during *unit testing*. Then, their interfaces are tested during one or more *integration steps* [8]. Each integration step requires the computation of the def-use pairs that cross the most recently integrated procedure interfaces to establish the new test requirements. Exhaustively recomputing reaching definitions and def-use pairs at the beginning of each integration step is inefficient and may easily result in overly high analysis times.

The problem of avoiding the costly re-computation of exhaustive data flow solutions is not unique to integration testing. It arises in virtually all data flow applications that deal with evolving software. Previously, *incremental data flow* algorithms have been proposed to address this problem [16, 13]. Incremental analysis avoids re-computation by performing the appropriate updates of a previously computed solution. Incremental analysis techniques could also be used in integration testing to extend the solution after each intergration step with the newly established reaching definitions. However, incremental analysis requires the complete reaching definition solution to be maintained between integration steps in addition to the def-use pairs. Moreover, the incremental update of the exhaustive solution at each integration step

[†] Partially supported by National Science Foundation Presidential Young Investigator Award CCR-9157371 and Grant CCR-9402226 to the University of Pittsburgh.

0270-5257/96 $5.00 © 1996 IEEE
Proceedings of ICSE-18

may be costly since information is propagated from the new interfaces throughout the program, including to portions that may have no relevance for the current integration step.

Another approach, and the one advocated in this paper, uses demand-driven analysis. Demand-driven analysis avoids the shortcomings of previous analysis approaches. Exhaustive information propagation is entirely avoided and replaced with a goal-oriented search. We previously presented a general framework for demand-driven analysis [4] and showed that the demand-driven search can formally be modeled as the functional reversal of an originally exhaustive analysis. By its goal-oriented nature, the search can bypass the code regions that are of no interest to the current data flow demands. Importantly, unlike incremental analysis, using a demand-driven analyzer for integration testing does not require the storage of reaching definition solutions between integration steps.

This paper develops a new approach to data flow testing at the integration level and demonstrates that this approach can be efficiently implemented based on demand-driven analysis. We present a demand-driven analyzer to efficiently compute the newly established data flow information at each integration step. We implemented the demand-driven analyzer and experimentally evaluated its performance in the context of integration testing based on bottom-up integration. We compared the accumulated analysis time of the demand-driven analyzer during the integration process with the accumulated analysis times of (i) an exhaustive analyzer and (ii) an analyzer based on incremental updates. Our experiments show that demand-driven analysis is significantly faster than exhaustive analysis by a factor ranging from 2.6 to 25. The demand-driven analyzer even outperforms the incremental analyzer in eight out of ten programs by up to a factor of 5.

Section 2 presents the relevant background in data flow analysis. Our approach to data flow integration testing is presented in Section 3 and the demand-driven analyzer used in this approach is described in Section 4. Section 5 presents the experimental evaluation and Section 6 discusses related work.

2 Data Flow Analysis

A program is represented by a set of control flow graphs, as shown in Figure 1. The nodes in the graph represent statements and edges represent the transfer of control among statements. The nodes *entry* and *exit* represent the unique entry and exit nodes of a procedure. The sets $pred(n)$ and $succ(n)$ contain the immediate predecessors and successors of a node n and the set $call(p)$ contains the nodes that represent a call to procedure p. To distinguish multiple occurrences of the same variable, we use the node number as a subscript, i.e., x_n denotes the reference of variable x at node n.

We consider programs with C-style global and local scoping and value parameter passing. Let *Global* be the set of global variables, let $s \in call(p)$ for some procedure p and let $a_1, \ldots, a_k$ be the actual parameters passed to the formal parameters $f_1, \ldots, f_k$ of p. The mapping of variables from the calling procedure to variables in the procedure called at s is modeled by the function:

$$bind_s(v) = (Global \cap \{v\}) \cup \{f_i | a_i = v\}$$

To avoid imprecision, the calling context of procedure calls must be considered for an execution path to be valid. For example in Figure 1, the sequence 1,2,3,4,13,14,16,5 is a valid execution path while the sequence 1,2,3,4,13,14,16,12 that invokes *proc3* from *proc1* but returns illegally to *proc2* is not valid. Throughout this paper, when referring to a path we assume that the path is a valid execution path.

The def-use pairs in a program are determined by solving the data flow problem of *reaching definitions.* In order to determine interprocedural reaching definitions in a program that consists of multiple procedures, the variable bindings through parameter passing must be considered.

Definition 1 *Let p and q be two procedures. A variable v in procedure p is* **directly bound** *to variable w in procedure q if there exists a call site* $s \in call(q)$ *in p and* $w \in bind_s(v)$. *Variable v in p is* **indirectly bound** *to variable w in q if there exists a sequence of variables* $v_1, \ldots, v_k$, *such that* $v = v_1$ *and* $w = v_k$ *and* v_i *is directly bound to* v_{i+1} *for* $1 \leq i < k$.

We say that variable v is *bound* to variable w if v is either directly or indirectly bound to w. Consider Figure 1. Variable x in *proc1* is bound in *proc3* to itself since x is global and bound to the formal parameter g via the call at node 4. Variable y is not bound to any variable in *proc3* since y is local and not passed as a parameter. However, y is bound to the formal parameter f in *proc2* via the call at node 6.

To formally define the sets of reaching definitions we use the notion of *killing* a definition.

Definition 2 *A definition d of variable v is* **killed** *at node n if node n contains the definition of a variable w and v is bound to w at node n.*

Definition 3 *A path* π *is called a* **def-clear** *path for variable v, if* π *does not contain a node that kills a definition of v.*

We can now define interprocedural reaching definitions and the symmetric interprocedural reachable uses in terms of def-clear paths:

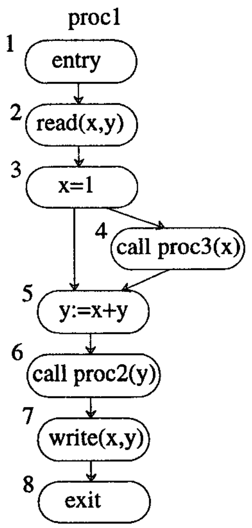
Definition 4 *A definition d of variable v is a* **reaching definition** *at node n if v is bound to some variable w at n and there exists a def-clear path for v from d to node n.*

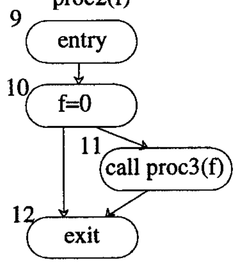
Definition 5 *A use u of variable v is a* **reachable use** *at node n if there exists a variable w at node n that is bound to v and there exists a def-clear path for v from node n to use u.*

```
procedure proc1
local y; /* x is global */
   read(x,y);
   if x=1 then call proc3(x);
   y:=x+y;
   call proc2(y);
   write(x,y);
```

```
procedure proc2(f)
   if f=0 then call proc3(f);
```

```
procedure proc3(g)
   if g=10 then x:=g+1;
```





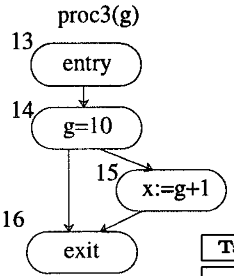


Table 1: Def-use pairs

intraproc.	interproc.	
(x_2, x_3)	(x_2, g_{14})	(y_5, g_{14})
(x_2, x_4)	(x_2, g_{15})	(y_5, g_{15})
(x_2, x_5)	(x_2, x_5)	(x_{15}, x_5)
(y_2, y_5)	(x_2, x_7)	(x_{15}, x_7)
(y_5, y_6)	(y_5, f_{10})	
(y_5, y_7)	(y_5, f_{11})	

Table 2: Call site variables

call(proc2) = {6}		*call(proc3)* = {4, 11}			
$Pres^x_6$	Def^x_6	$Pres^x_4$	Def^x_4	$Pres^x_{11}$	Def^x_{11}
true	$\{x_{15}\}$	*true*	$\{x_{15}\}$	*true*	$\{x_{15}\}$

Table 3: Procedure side effect variables

Procedure *proc2*			Procedure *proc3*		
n	$P^x_{proc2}[n]$	$D^x_{proc2}[n]$	n	$P^x_{proc3}[n]$	$D^x_{proc3}[n]$
9	*true*	$\{x_{15}\}$	13	*true*	$\{x_{15}\}$
10	*true*	$\{x_{15}\}$	14	*true*	$\{x_{15}\}$
11	*true*	$\{x_{15}\}$	15	*false*	$\{x_{15}\}$
12	*true*	$\emptyset$	16	*true*	$\emptyset$

Figure 1: Example program and its control flow graphs.

We define $RD(v, n, P)$ to be the set of definitions of variable v that reach node n in program P. Similarly, $RU(v, n, P)$ is the set of uses of v that are reachable at node n in P.

Definition 6 *Let d be a definition of variable v and let u be a use of variable w such that v is bound to w at the use u. The pair (d, u) is a* **def-use pair** *if d reaches u, or equivalently, if u is a reachable use at d.*

In a program that consists of multiple procedures, def-use pairs may cross procedure boundaries. To determine whether a def-use pair crosses procedure boundaries we examine the def-clear paths associated with the pair. For a given def-use pair (d, u) there may be several distinct def-clear paths from d to u. Some of these paths may cross procedure boundaries while others may be strictly intraprocedural paths. The pairs that cross procedure boundaries are characterized as follows.

Definition 7 *A def-use pair (d, u)* **crosses procedure p on entry (on exit)** *if there exists a def-clear path from d to u that includes the entry (exit) node n of p such that d reaches node n and u is a reachable use at node n.*

We say a def-use pair crosses a procedure p if the pair crosses p on entry and/or on exit. For example, in Figure 1 the pair (x_{15}, x_5) crosses *proc3* on exit. The pair (x_2, x_5) crosses *proc3* on entry and on exit. In contrast, the pair (y_2, y_5) does not cross *proc3* since y is local to *proc1* and not passed as a parameter. Also, the pair (y_5, y_7) does not cross *proc2*. Although y is passed to *proc2* as a parameter and y_5, therefore, reaches the entry node of *proc2*, the use y_5 is not a reachable use inside *proc2* [1].

Definition 8 *A def-use pair (d, u) for a variable v is an* **interprocedural pair** *if there exists a procedure p such that the pair crosses p.*

Definition 9 *A def-use pair (d, u) for a variable v is an* **intraprocedural pair** *if either v is a local variable or there exists a def-clear path from d to u that is entirely contained in the same procedure.*

Note that a pair with multiple def-clear paths may be both intra- and interprocedural. Table 1 in Figure 1 shows the complete set of def-use pairs for the program. The first column shows the intraprocedural pairs and the second column shows the interprocedural pairs. For example, the pair (x_2, x_5) is both an

[1] For y_5 to be a reachable use in *proc2* there would have to be a variable v in *proc2* that is bound to y in *proc1*. Such a variable v does not exist unless y is global ($v = y$) or f is a reference parameter ($v = f$).

intraprocedural pair due to the def-clear path (2,3,5) and an interprocedural pair since the pair crosses *proc3*. The pair (y_5, y_7) for the local variable y is not an interprocedural pair since the pair does not cross *proc2*. Hence, (y_5, y_7) is an intraprocedural pair.

3 Integration Testing

In data flow testing, after the def-use pairs in a program have been computed, test cases are generated manually or automatically to exercise def-use pairs according to a selected coverage criterion [14, 5]. For example, the *all-defs* criterion requires that for each definition a path to at least one reachable use is exercised in some test case.

The objective of integration testing is to organize the overall testing effort by explicitly separating the testing of the code within a procedure from the testing of the procedure interfaces. To achieve this separation, data flow testing is divided into several phases, an initial unit testing phase and several integration testing steps.

Unit Testing

During unit testing each procedure is tested in isolation based on only the *intra*procedural def-use pairs within the procedure. When testing an individual procedure p without considering its actual calling context in the program, certain assumptions must be made concerning the interfaces to other procedures. Temporary definitions are inserted to provide initial values for each formal parameter and each global variable that is used in procedure p. Furthermore, if p contains procedure calls, worst case assumptions must be made about the possible side effects of the called procedures. Thus, it is assumed that no def-clear paths exists through a called procedure q for variables that are addressable in q. As the integration proceeds, temporary definitions are removed and actual def-clear paths through called procedure are identified and tested.

Consider the example in Figure 1. During unit testing, two temporary definitions f_{in} and g_{in} are added for the formal parameters f in *proc2* and g in *proc3*, respectively. Each call site is assumed to redefine the value of the global variable x. Table 1 shows the intraprocedural def-use pairs that result in each procedure. In addition the temporary def-use pairs (f_{in}, f_{10}),(f_{in}, f_{11}), (g_{in}, g_{14}) and (g_{in}, g_{15}) are considered during unit testing.

Integration Steps

After the individual units have been tested, the interactions among procedures are tested separately during procedure integration. The integration takes place in several integration steps. During each step, one or more procedures are selected for integration according to an integration strategy, such as bottom-up or top-down integration [11]. Testing at each integration step involves only def-use pairs that cross one of the currently integrated procedures.

```
Algorithm ComputeCross(p, q)
Input: p, q: proc. in a program P prior to integration;
Output: the set Cross(p, q)
Cross := ∅;
let P' be the program after integrating q;
for each call site s in p where $s \in call(q)$ do
    for each variable v such that $binds_s(v) \neq \emptyset$ do
        compute $Def = RD(v, s, P)$;
        compute $Use = \bigcup_{w \in binds_s(v)} RU(w, entry_q, P')$;
        add $\{(d,u) \mid d \in Def, u \in Use\}$ to Cross;
    endfor
    for each variable $v \in Global$ do
        compute $Def = RD(v, exit_q, P)$;
        compute $Use = RU(v, s, P')$;
        add $\{(d,u) \mid d \in Def, u \in Use\}$ to Cross;
    endfor;
endfor;
```

Figure 2: Algorithm *ComputeCross*.

We assume for simplicity that during each step a single procedure q is integrated with one of its calling procedures p. To integrate procedure q with p the temporary definitions for formal and global variables in procedure q are removed and every call site in procedure p that calls q is considered. The def-use pairs tested during the integration step are the interprocedural pairs that are established by the integration of q with p. These newly established pairs are captured in the set $Cross(p,q)$ defined as follows:

Definition 10 *Let p and q be two procedures, such that p calls q. The set Cross(p, q) of* **cross pairs** *consists of the interprocedural pairs (d, u) that cross q such that there exists a call site $s \in call(q)$ in p and either d reaches s or u is a reachable use at s.*

Consider the integration of *proc2* with *proc1* in Figure 1 at the call at node 6 and assume that *proc3* has not yet been integrated. The def-use pairs that cross *proc2* via the call at node 6 are $Cross(proc1, proc2) = \{(x_2, x_7), (y_5, f_{10}), (y_5, f_{11})\}$.

A pair in $Cross(p,q)$ may cross several call sites. However, a def-use pair will not be considered for testing unless there exists a def-clear path that crosses only procedures that have already been integrated. If a def-use pair has multiple def-clear paths that each cross different procedures then the pair may be considered for testing during multiple integration steps.

Consider again the integration of *proc2* with *proc1* in Figure 1. The pairs (y_5, g_{14}) and (y_5, g_{15}) cross call sites of *proc2* and *proc3*. However, not until both procedures *proc2* and *proc3* are integrated can it be determined that there exists a def-clear path from y_5 to g_{14} crossing both *proc2* and *proc3* via the call sites at nodes 6 and 11. Hence, the two pairs are not considered for testing until both procedures *proc2* and *proc3* have been integrated.

The computation of $Cross(p,q)$ is described in algorithm *ComputeCross* shown in Figure 2. The set $Cross(p,q)$ is computed by considering one call site

$s \in call(q)$ at a time. For each such call site s cross pairs are determined in two steps.

Cross-on-entry: First, the pairs that cross procedure q on entry are determined (lines 6-8) by matching the definitions that reach the call site s with the uses that are reachable at the entry of procedure q. To include the def-use pairs that cross the called procedure both on entry and on exit, the reachable uses are computed in the program with procedure q being integrated.

Cross-on-exit: Next, the pairs that cross procedure q on exit are determined (lines 11-13). These pairs result by matching the definitions that reach the exit of procedure q with the uses that are reachable from the call site in p. Reaching definitions are computed prior to the integration of procedure q to exclude the pairs that cross q both on entry and on exit and avoid their repeated computation.

Consider again the integration of *proc2* with *proc1* at node 6 in Figure 1 assuming that *proc3* has not yet been integrated. The following sets are computed to determined the cross-on-entry pairs:
Pairs for variable x:
$RD(x, 6, P) = \{x_2\}$ and $RU(x, 9, P') = \{x_7\}$
Pairs for variable y:
$RD(y, 6, P) = \{y_5\}$ and $RU(f, 9, P') = \{f_{10}, f_{11}\}$.
The set *Cross*(*proc1*, *proc2*) results as $\{(x_2, x_7), (y_5, f_{10}), (y_5, f_{11})\}$. There are no cross-on-exit pairs prior to the integration of procedure *proc3*.

The efficiency of *ComputeCross* depends primarily on the algorithm that is used to compute the data flow sets RD and RU. We show in the following section that a fast computation of these sets is possible through demand-driven analysis.

4 Def-Use Pairs on Demand

This section presents the demand-driven algorithm for computing the set $RD(v, n, P)$ of interprocedural reaching definitions. A corresponding algorithm for the symmetric problem of computing reachable uses follows in a straightforward way.

In traditional reaching definition analysis, definitions are exhaustively propagated from their generation points to all the points that they reach. Demand-driven analysis avoids exhaustive computations by reversing the original analysis process. Exhaustive forward propagation is replaced with a goal-oriented backward search. The search is triggered by a query for the definitions that reach a selected node and terminates as soon as all nodes that contain a relevant definition have been found. A query is a triple $?RD(v, n, P)$ and represents a request for the computation of the set $RD(v, n, P)$. We use ε to denote the empty query. The resolution of a query is fully described by the two propagation rules shown in Figure 3 (i). A propagation rule describes how a query at a node n is translated into an equivalent union of zero or more new queries at predecessor nodes of n. Associated with each propagation rule is an *action* to guide the collection of definitions.

```
Algorithm Compute_RD $(v, n, P)$
Input: $v$: a variable, $n$: a node in a program $P$
Output: the set $RD(v, n, P)$
worklist := $(v, n)$;
while worklist is not empty do
    remove a tuple $(v, n)$ from worklist;
    if $n = entry_p$ for some proc. $p$
        then apply Rule 1 (with action);
        else apply Rule 2 (with action);
    for each new query $RD(v, m, P)$ do
        add the tuple $(v, m)$ to worklist;
endwhile
return the set of collected definitions;
```

Figure 4: Algorithm *Compute_RD* (v, n, P).

Rule 1: Procedure entry $?RD(v, entry_q, P)$): If v is not local to q then the query is translated into a union of new queries, one at each call site $s \in call(q)$. If v is local or q is the main program, the query is translated into ε. The action associated with this rule is empty since no definition sites are encountered.

Rule 2: Non-entry node $?RD(v, n, P)$): The query is translated into a union of new queries, one at each predecessor $m \in pred(n)$. To determine the new query we define the following variables at each node m:

$$Pres_m^v = \begin{cases} true & \text{if } v \text{ is not re-defined at } m \\ false & \text{otherwise} \end{cases}$$

$$Def_m^v = \begin{array}{l} \text{set of defs. of } v \text{ that may be generated} \\ \text{at } m \text{ and that reach the exit of } m \end{array}$$

If $Pres_m^v = true$ (i.e., v is preserved at m), then the query propagates to predecessors m. If $Pres_m^v = false$ (i.e., v is re-defined at m) the propagation terminates at m and the new query for m is ε. The action is to collect any definitions of v that are generated.

For a node m that does not contain a procedure call, the variables $Pres_m^v$ and Def_m^v are determined by a local inspection of node m. Determining the two variables if m represents a call to a procedure q requires knowledge about the side effects of invoking q. If variable v is not global, the call cannot have any side effects on the reaching definitions for v. Hence, $Pres_m^v = true$ and $Def_n^v = \emptyset$. Otherwise, we perform analysis to compute the side effects of procedure invocation. The side effect analysis is described in detail in the next section.

Based on the availability of the variables $Pres_n^v$ and Def_n^v we can use a simple worklist algorithm for the demand-driven evaluation of a query as outlined in Figure 4. Algorithm *Compute_RD* proceeds by iteratively applying the propagation rules to a worklist of maintained queries until no more new queries can be generated. At this point all requested reaching definitions have been encountered and the evaluation terminates. Since each variable can be queried at most

(Rule 1) Procedure Entry Node:

$$?RD(v, entry_p, P) \Longleftrightarrow \left\{ \begin{array}{ll} \varepsilon & \text{if } v \text{ is local to } p \\ \bigcup\limits_{s \in call(p)} ?RD(w, s, P) & \text{if } v \text{ is not local and } v \in bind_s(w) \end{array} \right\} \{\mathit{Action{:}\ none}\}$$

(Rule 2) Non-Entry Node:

$$?RD(v, n, P) \Longleftrightarrow \bigcup_{m \in pred(n)} \left\{ \begin{array}{ll} ?RD(v, m, P) & \text{if } Pres_m^v = true \\ \varepsilon & \text{otherwise} \end{array} \right\} \{\mathit{Action{:}\ collect}\ Def_n^v\}$$

(i)

Side Effect Variables:

$$\begin{array}{lll} P_q^v[exit_p] & = true & \\ P_q^v[n] & = \bigvee\limits_{m \in succ(n)} \left\{ \begin{array}{ll} P_q^v[m] \wedge Pres_m^v & \text{if } m \text{ is not a call} \\ P_q^v[m] \wedge P_r^v[entry_r] & \text{if } m \in call(r) \end{array} \right. \end{array}$$

$$\begin{array}{lll} D_q^v[exit_p] & = \emptyset & \\ D_q^v[n] & = \bigcup\limits_{m \in succ(n)} \left\{ \begin{array}{ll} D_q^v[m] \cup Def_m^v & \text{if } m \text{ is not a call and } P_q^v[m] = true \\ D_q^v[m] \cup D_r^v[entry_r] & \text{if } m \in call(r) \text{ and } P_q^v[m] = true \\ D_q^v[m] & \text{otherwise} \end{array} \right. \end{array}$$

(ii)

Figure 3: Query propagation rules (i) and side effect variable definition (ii).

once at each node, the worst case number of generated queries during each execution of *Compute_RD* is $D \times N$, where N is the number of nodes and D is the number of definitions of the variables in the largest address space of any one procedure in the analyzed program. Assuming that $|pred(n)|$ is bounded by a small constant, the processing of each generated query requires the inspection of at most c nodes, where $c = max\{\ |call(q)|\ |\ q \text{ is a procedure in } P\ \}$. Hence, the overall worst case running time is $O(c \times D \times N)$.

4.1 Computing Side Effect Variables

We consider now the computation of the side effect variables $Pres_s^v$ and Def_s^v for a call site s. We follow the two phase approach to interprocedural data flow analysis that accurately accounts for the calling context of each procedure [19, 2, 8]. During the first phase the side effects of (possibly recursive) procedures are analyzed independent of their calling contexts. The second phase determines the effect of a call by appropriately adjusting the side effect information to the context of the call. Unlike previous approaches that compute side effects exhaustively for all procedures [2, 8], we compute the side effect variables in a demand-driven fashion as they are needed during the query evaluation.

Phase 1: The side effect of a procedure q on a global variable v is computed in a boolean variable $P_q^v[n]$ and a set $D_q^v[n]$ for each node n in q as defined in Figure 3 (ii), where $\wedge$ and $\vee$ denote boolean conjunction and disjunction, respectively. $P_q^v[n]$ is set to *true* (preserved) if there exists a def-clear path for v from n to the exit of q. Otherwise, $P_q^v[n]$ is *false*. $D_q^v[n]$ is the set of definitions that reach the procedure exit along some path starting at n.

Phase 2: The procedure side effects are fully described by the values on procedure entry: $P_q^v[entry_q]$ and $D_q^v[entry_q]$. Using these values we determine the side effects for a global variable v at a call site s with $s \in call(q)$ as $Pres_s^v = P_q^v[entry_q]$ and $Def_s^v = D_q^v[entry_q]$.

Note that $Pres_s^v$ can be directly set to *true* and Def_s^v to $\emptyset$, without having to evaluate side effect equations, if it is known that neither q nor a procedure directly or indirectly called by q contains a textual reference to v.

Figure 1 shows the side effect variables for the global variable x in Tables 2 and 3. For instance, the entry $Pres_4^x = P^x_{proc3}[entry_{proc3} = 13] = true$ expresses that the value of x may be preserved throughout the execution of procedure *proc3* and $Def_4^x = D^x_{proc3}[entry_{proc3} = 13] = \{x_{15}\}$ is the set of definitions of x that reach the exit of procedure *proc3*.

The solution of the equations P_q^v and D_q^v is the least fixed point based on the initial values $P_q^v[n] = false$ and $D_q^v[n] = \emptyset$. When the values for $P_q^v[entry_q]$ and $D_q^v[entry_q]$ are requested a worklist is initialized with the triple $(v, exit_q, q)$ to trigger the computation. During each step a triple (v, n, P) is removed from the worklist, the corresponding equations $P_q^v[n]$ and $D_q^v[n]$ are evaluated and if their values have changed, the triple for each dependent equation that may be affected by the change is added to the worklist. Each invocation of the worklist algorithm results in only a partial evaluation of the equation system. The value of each of the $O(D \times N)$ equations can change at most once and each change can result in the inspection of at most c other nodes. Thus, assuming that the equation values are maintained between k subsequent side effect variable requests, the accumulated worst case running time for the k requests is $O(\max(k, c \times D \times N))$.

Example: We illustrate the query evaluation for the program in Figure 1 with the query $?RD(x, 5, P)$ which requests the definitions of variable x that reach node 5. When the query is propagated across the call *proc3*(x) at node 4, the side effect variables $Pres_4^x$ and Def_4^x are computed. $Pres_4^x = P^x_{proc3}[13] = true$ indicates that the query is preserved. Hence the new query $?RD(x, 4, P)$ is generated and the definition $Def_4^x = D^x_{proc3}[13] = \{x_{15}\}$ is collected. Next, the query $?RD(x, 4, P)$ is propagated through node 3 resulting in the new query $?RD(x, 3, P)$. Since node 2 contains definition x_2, applying rule 2 yields ε as the new query and causes the collection of definition x_2. As no new queries have been generated, the search terminates with the set of reaching definitions $\{x_2, x_{15}\}$.

4.2 Reference Parameters and Aliases

Two variables x and y are *aliases* in a procedure q if x and y may refer to the same location during some invocation of q. Alias pairs may be introduced by reference parameter passing. For example, if the same actual parameter is passed to two formal parameters f_1 and f_2, then (f_1, f_2) is an alias pair in the called procedure. Ignoring aliasing during the analysis may lead to unsafe query responses; some of the def-use pairs may be missed. In [4] we discussed analysis refinements for safely handling aliasing in constant propagation using separately computed alias information [3]. We can use the same approach for safely refining the propagation rules from Figure 3 to handle reference parameters. According to these refinements $Pres_n^v$ is set to *true* if v or any alias of v is preserved during the execution of n. Analogously, the set $Def_n{}^v$ is determined by collecting the definitions not only for v but also for any alias of v. Further details can be found in [4].

Table 4: Test suite

program	lines	N	P	calls	pairs	steps
queens	89	150	4	4	119	4
cat	240	377	5	4	165	4
calendar	352	731	10	14	236	9
getopt	395	739	5	6	268	4
linpack	564	686	12	30	1160	14
patch	753	1316	14	13	599	12
gzip	1387	3024	38	123	1461	68
grep	1488	2906	32	72	1048	47
sort	1528	3554	35	145	1570	80
dc	1576	3298	67	230	1958	153

4.3 Caching

The query evaluation of algorithm *Compute_RD* can result in at most $D \times N$ requests for side effect variables. Thus, the overall running time including the time for computing side effects is $O(c \times D \times N)$ and $O(D \times N)$ space is needed to store the queries and side effect variables.

The response time to a sequence of k queries can be improved by storing intermediate query results in a cache memory for fast reuse in future query evaluations. The cache memory is inspected prior to generating a new query and the new query is generated only if no previous results for the query are stored in the cache. Entries are added to the cache after each terminated query evaluation such that a definition that was collected at a node n is added to the cache entry at all reachable visited nodes. Caching does not increase the asymptotic cost of the algorithm. The worst case time complexity for $k \leq D \times N$ queries using caching is $O(c \times D \times N)$ and the space requirements remain $O(D \times N)$. Importantly, the worst case complexity is no worse than for a standard exhaustive algorithm for interprocedural reaching definitions based on the Sharir/Pnueli framework [19].

5 Experiments

We implemented the demand-driven algorithm presented in the previous section in the context of bottom-up integration testing. The procedures in a program are integrated in depth-first (bottom-up) order of the program's *call graph*. The call graph contains one node for each procedure and there is an edge (q, p) if procedure q calls procedure p. During each integration step one edge (q, p) in the call graph is processed and the new def-use pairs are determined as described by algorithm *ComputeCross* from Figure 2. To evaluate the performance of the demand-driven analyzer during the integration we also implemented two other analyzer versions: an exhaustive analyzer and an incremental analyzer. The different analyzer types vary in the way they compute the data flow sets that are accessed in *ComputeCross*:

- **The (caching) demand-driven analyzer**
The demand-driven analyzers for both interprocedural reaching definitions and reachable uses are based on the algorithm presented in Section 4. Two versions of each analyzer were implemented: a caching and a non-caching version.

- **The exhaustive analyzer**
The exhaustive interprocedural reaching definition analysis is based on a standard iterative fixed point algorithm of Sharir and Pnueli's functional approach to interprocedural analysis [19]. The exhaustive analyzer, that is implemented based on bitvectors, recomputes the reaching definitions in the program from scratch at the beginning of each integration step. However, we optimized the computation by performing exhaustive analysis only over the procedures that are *affected* by the current integration step. If the call edge (q, p) is currently being integrated, only procedures that are connected to q or p in the call graph along previously integrated edges are affected and included in the exhaustive re-analysis of the current step.

- **The incremental analyzer**
We also implemented an incremental version of the exhaustive analyzer using bitvectors. The incremental analyzer maintains the complete exhaustive solution between subsequent integration steps. The number of established reaching definitions and def-use pairs increases as the integration proceeds. Thus, the reaching definition solution that was valid at a previous integration step may be incomplete for the current step but does not contain any false reaching definitions. Hence, the incremental update problem is particularly simple and requires no deletions. The previous solution can be extended by simply using it as the initial value to re-start the fixed point iteration for the current integration step.

We conducted two sets of experiments to evaluate the performance of the three analyzer types when used for data flow integration testing. The performance of each analyzer was measured on a Sparcstation 5 for a test suite of C programs, shown in Table 4. Except for program queens, the programs are core routines of Unix utility sources ranging from 89 to 1576 code lines. Table 4 shows the number of nodes N, the number of procedures P, the number of calls, the total number of def-use pairs and the number of integration steps for each program. All reported analysis times are cpu times in seconds.

5.1 Experiment I

In the first set of experiments we compared the performance of the demand-driven analyzer (without and with caching) with the performance of the exhaustive analyzer. We measured the analysis times during the integration and determined for each test program the accumulated analysis times shown in Table 5, where:

Table 5: Analysis times (in seconds)

program	T_{ex}	T_{dd}	T_{dd}^{cache}	T_{incr}
queens	0.17	0.06	0.09	0.08
cat	0.52	0.20	0.22	0.29
calendar	0.78	0.20	0.21	0.32
getopt	3.80	0.98	0.99	1.43
linpack	3.95	0.49	0.57	1.25
patch	17.01	5.76	5.27	3.51
gzip	96.87	15.53	23.88	14.85
grep	57.86	5.50	4.69	6.44
sort	193.76	9.22	7.58	15.00
dc	66.48	2.58	2.17	13.38

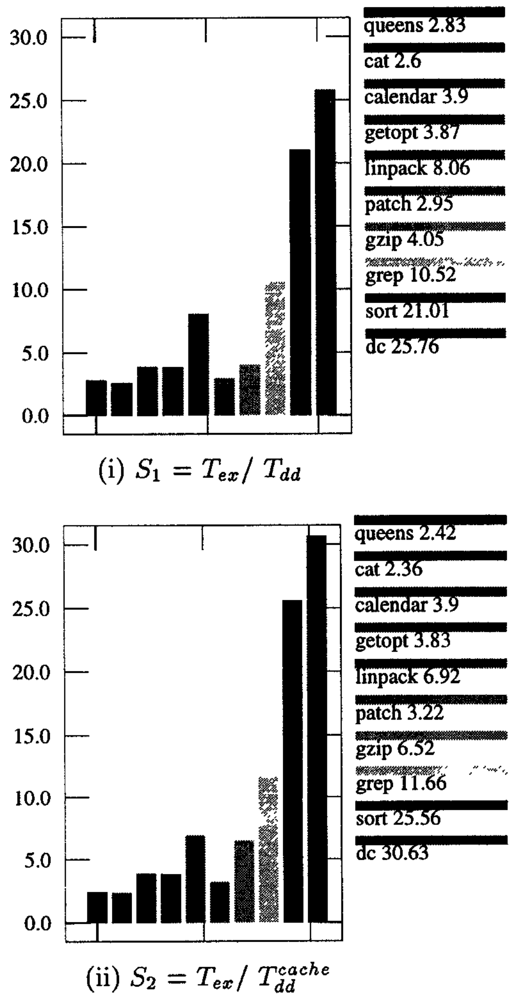
$T_{ex} =$ acc. time of exhaustive analysis.
$T_{dd} =$ acc. time of demand-driven analysis without caching
$T_{dd}^{cache} =$ acc. time of demand-driven analysis with caching

We calculated the speedups: $S_1 = T_{ex}/\, T_{dd}$ and $S_2 = T_{ex}/\, T_{dd}^{cache}$ of the demand-driven analyzer (without caching and with caching) over the exhaustive analyzer. Figure 5 (i) and (ii) display the two speedups S_1 and S_2. The demand-driven analyzer without caching is significantly faster than the exhaustive analyzer by a factor ranging from 2.6 to 25.7. As shown in Figure 5 (ii), adding caching resulted in similar speedups ranging from 2.3 up to 30. Compared to the non-caching version, caching increased the speedup for the 5 larger programs, but did not pay off for the 5 shorter programs since the number of cache hits was too small to compensate for the overhead of allocating and maintaining the cache. In larger programs where an individual query is more expensive, the savings from cache hits quickly outweigh the cache overhead.

Figure 5 (i) and 5 (ii) show that the speedups of the demand-driven analyzer tend to grow with increasing program size (in terms of code lines).

5.2 Experiment II

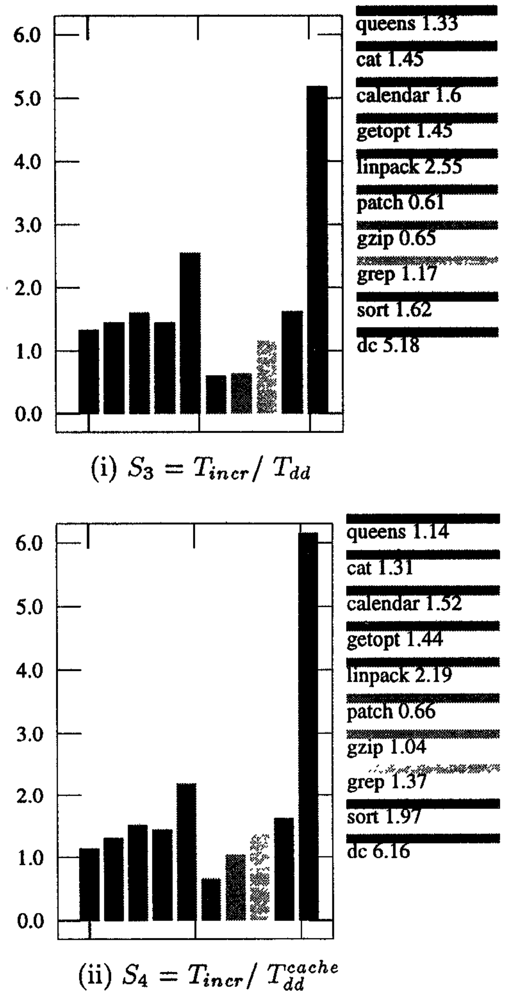
We conducted a second set of experiments to compare the performance of the demand-driven analyzer with the performance of an incremental analyzer. We ran the integration system using the incremental analyzer and measured the accumulated analysis time T_{incr}. The results are shown in Table 5. Figures 6 (i) and 6 (ii) display the speedups: $S_3 = T_{incr}/\, T_{dd}$ and $S_4 = T_{incr}/\, T_{dd}^{cache}$ of the demand-driven analyzer (without and with caching) over the incremental analyzer. Except for two programs (patch and gzip) the demand-driven analyzer without caching is faster than the incremental analyzer up to a factor of 5.18. The demand-driven analyzer without caching has an additional advantage over the incremental analyzer in that no storage of information other than the def-use pairs



(i) $S_1 = T_{ex} / T_{dd}$

(ii) $S_2 = T_{ex} / T_{dd}^{cache}$

Figure 5: Speedups: demand-driven over exhaustive.



(i) $S_3 = T_{incr} / T_{dd}$

(ii) $S_4 = T_{incr} / T_{dd}^{cache}$

Figure 6: Speedups: demand-driven over incremental.

is required between integration steps. In contrast, the incremental analyzer maintains the complete reaching definition solution in addition to the def-use pairs throughout the integration.

As in experiment I, adding caching did not significantly impact on the speedups. Compared to the non-caching version caching improved the speedups for the five larger programs and results in a slight slowdown for the five shorter programs. As shown in Figure 6 (ii), the caching demand-driven analyzer achieved speedups over the incremental analyzer in all but one program (patch) up to a factor of 6.16.

We examined the programs patch and gzip and found that they have a high percentage of global variables. Queries for global variables may require much longer propagation paths than queries for locals, which explains why demand-driven analysis does not perform as well.

6 Related Work

The demand-driven algorithm presented in this paper is a specialized and optimized instance of our formal framework for demand-driven interprocedural analysis [4]. Other general frameworks for demand-driven analysis were presented by Reps, Horwitz and Sagiv [15, 10, 17]. Their recent approach [10, 17] transforms a data flow problem 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. The major difference between their approach and our work is the necessity to construct for each data flow problem an exploded supergraph, whose size can be substantial. During experimentation with the graph-reachability analyzer for copy constant propagation, the analyzer ran out of virtual memory for some C programs of about 1,300 lines [18]. Although their two-phase variation of the initial graph-reachability algorithm [17] resulted in a more compacted version of the exploded supergraphs for copy constant propagation, the size of the graph remains the same in problems such as reaching definitions or reachable uses.

A framework for partial data flow analysis by Gupta and Soffa [7] yields search algorithms that are similar to our demand-driven algorithm, but are limited to *intra*procedural analysis. Def-use pairs also play an important role in program slicing [21]. Interprocedural def-use pairs are implicitly determined as part of an interprocedural program slice [21, 9]. However, previous precise interprocedural algorithms [9] are not suitable for computing interprocedural def-use pairs since their computation is not explicit but interleaved with the slice construction.

Incremental data flow analysis [16, 13] addresses the problem of updating an existing exhaustive solution in response to program changes. Thus, in contrast to demand-driven analysis, incremental analysis requires the computation and maintenance of an exhaustive solution. As pointed out earlier, the incremental update problem that arises during intergration testing is particularly simple. To handle more general types of program changes the incremental algorithms in [16, 13] perform additions, deletions and structural updates of the solution.

Data flow testing at the integration level was previously discussed by Harrold and Soffa [8]. The authors presented an interprocedural data flow analysis to compute def-use pairs (exhaustively) over the complete program.

Another related field is regression testing. The analysis task in regression testing is to determine the test requirements for a modified program to ensure that no errors are introduced into previously tested code. Selective regression testing [12, 20, 6, 1] attempts to re-test only those def-use pairs that are affected by the modification.

Note that the integration of a procedure could be viewed as a program modification. However, unlike general program edits that give rise to regression testing, the integration of a procedure does not invalidate previous tests. On the contrary, it is precisely the intent of integration testing to test procedure interfaces in isolation while assuming that the code within each procedure has already been satisfactorily tested.

References

[1] H. Agrawal, J. Horgan, E. Krauser, and S. London. Incremental regression testing. In *Conf. on Software Maintenance*, pages 348–357, Sept. '93.

[2] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *SIGPLAN '88 Conf. on Programming Design and Implementation*, pages 47–56, June '88.

[3] K. Cooper. Analyzing aliases of reference formal parameters. In *12th ACM Symp. on Principles of Programming Languages*, pages 281–290, '85.

[4] E. Duesterwald, R. Gupta, and M.L. Soffa. Demand-driven computation of interprocedural data flow. In *22nd ACM Symp. on Principles on Programming Languages*, pages 37–48, Jan. '95.

[5] P.G. Frankl and E.J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, SE-14(10):1483–1498, Oct. '88.

[6] R. Gupta, M.J. Harrold, and M.L. Soffa. An approach to regression testing using slicing. In *Conf. on Software Maintenance*, pages 299–308, Nov. '92.

[7] R. Gupta and M.L. Soffa. A framework for partial data flow analysis. In *Int. Conf. on Software Maintenance*, pages 4–13, Sept. '94.

[8] M.J. Harrold and M.L. Soffa. Interprocedural data flow testing. In *3rd Testing, Analysis and Verification Symp.*, pages 158-167, Dec. '89.

[9] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[10] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *3rd ACM SIGSOFT Symp. on the Foundations of Software Engineering*, Oct. '95.

[11] G.J. Myers. *Software reliability: principles and practices.* Wiley-Interscience, New York, '76.

[12] T.J. Ostrand and E.J. Weyuker. Using dataflow analysis for regression testing. In *6th Annual Pacific Northwest Software Quality Conf.*, pages 233–247, Sept. '88.

[13] L. Pollock and M.L. Soffa. An incremental version of iterative data flow analysis. *IEEE Trans. on Software Engineering*, 15(12):1537–1549, Dec. '89.

[14] S. Rapps and E. Weyuker. Selecting software test data using data flow information. *IEEE Trans. on Software Engineering*, 11(4):367–375, Apr. '85.

[15] T. Reps. Solving demand versions of interprocedural analysis problems. In *5th Int. Conf. on Compiler Construction*, pages 389–403. Springer Verlag, LNCS 786, Apr. '94.

[16] B.G. Ryder and M.C. Paull. Incremental data flow analysis algorithms. *ACM Trans. Programming Languages and Systems*, 10(1):1–50, '88.

[17] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *FASE 95: Colloquim on Formal Approaches in Software Engineering*, pages 651–665. Springer Verlag, LNCS 915, May '95.

[18] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. Technical Report TR-1284, Computer Science Department, University of Wisconsin, Madison, WI, Aug. '95.

[19] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice-Hall, '81.

[20] A.M. Taha, S.M. Thebut, and S.S. Liu. An approach to software fault localization and revalidation based on incremental data flow analysis. In *COMPSAC'89*, pages 527–534, Sept. '89.

[21] M. Weiser. Program slicing. *IEEE Trans. on Software Engineering*, SE-10(4):352–357, Jul. '84.