A DEMAND-DRIVEN APPROACH FOR EFFICIENT INTERPROCEDURAL DATA FLOW ANALYSIS

by

Evelyn Duesterwald

M.S., University of Pittsburgh, 1991

Submitted to the Graduate Faculty of
Arts and Sciences in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

University of Pittsburgh

1996

A DEMAND-DRIVEN APPROACH FOR EFFICIENT INTERPROCEDURAL DATA FLOW ANALYSIS

Evelyn Duesterwald, PhD

University of Pittsburgh, 1996

Despite the increasing importance of data flow analysis, today's applications in compiler optimizations and software tools still rely on traditional exhaustive analysis algorithms. Exhaustively computing data flow information, especially if interprocedural analysis is involved, is known to be costly. This dissertation develops and experimentally evaluates a new approach to interprocedural data flow analysis that is demand-driven rather than exhaustive. Demand-driven analysis reduces the time and space overhead of exhaustive algorithms by limiting the analysis effort to the collection of information that is actually needed.

A general framework is developed for deriving demand-driven interprocedural analysis algorithms from a standard algebraic description of the data flow problem. This framework models a demand for data flow information as a set of queries. A system of query propagation rules is derived that formally describes the resolution of a query. The framework includes a generic demand-driven algorithm that determines the response to a query by a polynomially bounded number of applications of these rules.

Experimental results are presented that demonstrate the benefits of the demand-driven approach in practice. Experimentation with two analysis problems, namely reaching definitions and copy constant propagation, shows that demand-driven analysis performs well in practice and reduces the time and space requirements when compared with exhaustive analysis. Additional experimentation evaluates the demand-driven approach when used in a specific software engineering application. The experiments show that demand-driven analysis, if used in the context of data flow integration testing, is significantly faster than exhaustive analysis and even outperforms an improved version of the exhaustive analysis that is based on incremental updates.

While experimentation demonstrates that demand-driven analysis can achieve considerable improvements over traditional exhaustive algorithms, the analysis may still include redundant computations. To eliminate these remaining redundancies, the technique of *congruence partitioning* is developed. Congruence partitioning is performed to optimize the performance of data flow analysis in a preparatory phase prior to the actual solution computation. Congruence partitioning prevents redundant computations by directly manipulating and reducing the solution equation system. A general framework for congruence partitioning is presented that can be used to optimize the performance of either exhaustive or demand-driven analysis algorithms.

ACKNOWLEDGEMENTS

My foremost thanks go to my co-advisor Mary Lou Soffa. I thank her for first putting the idea of pursuing a PhD in my mind during my exchange student year and for being my mentor and a continuous source of support and inspiration in the years thereafter. I also would like to thank my co-advisor Rajiv Gupta for his support and guidance throughout my graduate studies. And thanks go to the other members of my committee Jaspal Subhlok and Robert Daley.

I would hardly be where I am right now if it were not for my friends and I thank all of them. Lastly, I want to dedicate this page to Juan Leon in remembrance of a late hour at the Squirrel Hill Cafe.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Static data flow analysis is the process of determining program properties that hold for some or for all executions of a program. Various questions about the way variables and other program objects are used in a program are formulated as *data flow problems*. Solving a data flow problem precisely, without actually executing the program, is an uncomputable problem. Thus, the computed data flow solution necessarily provides only an approximation of the actual program behavior during execution.

Since its introduction in the early 70s, the use of data flow analysis has grown considerably. Data flow analysis was first developed for optimizing compilers to enable efficient register allocation and a large number of machine independent global optimizations [ASU86]. Today's utilization of data flow analysis goes far beyond its initial application in optimizations, and current compilers spend an increasingly large portion of the compilation time in gathering global data flow information. The advent of parallel computer architectures has created new challenges for compiler writers which must be addressed in order to fully exploit the potential benefits of these architectures. The generation of parallel code is heavily dependent on data flow information to perform such tasks as vectorization and parallelization [AK87, PW86, FOW87], partitioning [SH86], and code scheduling [Set76]. In addition, data flow analysis has also become the primary component of many software engineering applications such as editing [RTD83], verification [CC77a], debugging [Wei84, DGS92a], software testing [RW85, HS89a, DGS92b], program integration [HPR89], and parallel program analysis [CS89, EP89, SS88, DS91].

1.1 Current Problems

Along with the growing importance of data flow analysis in today's applications comes an increased concern about the high time and space requirements of computing and maintaining all data flow information that is needed. Computing data flow solutions, especially if interprocedural analysis is involved, is costly. Experimental studies show that performing certain data flow analyses over even medium-sized programs can take several hours

[GN93, Lan92].

The problem with the considerable time and space overhead of data flow analysis becomes even more critical when it is considered that a program is typically analyzed more than once. The need for multiple analyses may result from several sources. First, today's compilers require data flow information for an increasing number of independent tasks, each of which may require a distinct data flow problem to be solved. A program is then analyzed multiple times, each time to provide the solution to a different data flow problem. Second, the solution to a single data flow problem may need to be computed more than once as a result of code transformations. If code transformations are applied to a program, the data flow information in the program changes and previously computed data flow solutions may no longer be valid. Data flow must be either updated or re-computed following the application of code transformations. Furthermore, the order in which various code transformations are applied is often guided by heuristics. Thus, repeated phases of computing data flow and transforming the code may be necessary for certain transformations to be effective. Finally, data flow information may also be invalidated through program edits by the user. During program development, program edits are expected and must be efficiently handled. If it is not possible to correctly update a previously computed data flow solution after a program edit, the respective analysis may have to be repeated to provide the new data flow solution.

In spite of the increasing need for efficient data flow analysis algorithms, current data flow applications typically still rely on traditional exhaustive algorithms for computing data flow solutions. Phrased in the traditional *data flow framework* [KU77], 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: that is, 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.

Exhaustive data flow solutions are not only costly to compute, they are also inappropriate in applications that actually utilize only a part of the data flow information. For example, several code transformations in optimizing compilers are applicable to only certain structures in a program such as loops. Thus, data flow information is needed only for selected portions of the program. Even if optimizations are applicable everywhere in the program, one may want to reduce the overall optimization overhead by restricting their application to only the most frequently executed regions of the program (e.g., frequently called procedures or inner loops). Other applications that require data flow information only selectively are found in software development tools. Interactive software tools that aid in debugging and understanding of complex code require information about various aspects of a program. Typically, the information requested by a user is not exhaustive but selective, i.e., data flow for only selected points of the program is needed. For example, during debugging a user may want to know where a certain value is defined in the program. Also the actual amount of data flow information that is needed to satisfy the user's requests is not fixed before the software tool executes. However, exhaustively precomputing all data flow information that might be requested by the user can be very costly, especially for large programs.

This thesis explores the deficiencies of current exhaustive data flow analyzers and develops two new systematic approaches to improve the performance of data flow analyzers for today's applications. The first approach is a *demand-driven* approach to data flow analysis. A demand-driven approach reduces the analysis overhead by limiting the analysis effort to the collection of information that is actually needed by the application. Although demand-driven analysis can effectively reduce the analysis overhead, the analysis may not be computationally minimal and may still perform redundant computations. Further improvements are possible by optimizing the analysis effort on the lower level of intermediate computations. The second approach that is developed in this dissertation, *congruence partitioning*, addresses the elimination of low-level redundancies in the data flow computation. Together, demand-driven analysis and congruence partitioning enable improvements in the data flow analyzer that go well beyond the capabilities of previous methods to reduce the analysis cost.

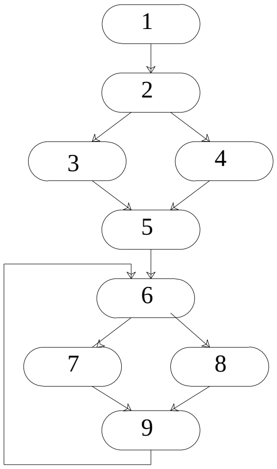
1.2 Previous Approaches for Reducing Analysis Cost

Several approaches have been developed to improve the performance of traditional exhaustive data flow analysis. Consider Figure 1.1 (i) that shows a fragment of a program's control flow graph. A hypothetical but realistic data flow equation system for a forward data flow problem is shown in Figure 1.1 (ii). Each equation $X(n)$ in the system expresses the data flow solution at node n. The operator $\sqcap$ denotes the *meet* operator that is applied to merge equation values at confluence points in the graph.

The exhaustive approach determines the data flow solution by computing the fixed point of all equations in the system simultaneously. Previous approaches to improve this exhaustive fixed point computation fall into one of two classes: approaches that are aimed at optimizing the exhaustive solution computation and approaches that completely depart from exhaustively computing data flow and instead pursue a partial solution computation. Approaches that fall into the former class are parallel data flow analysis and forwarding techniques. Example of the latter class are incremental and demand-driven data flow analyses.

1.2.1 Parallel Data Flow Analysis

If a multiprocessor system is available, data flow analysis can be sped up by parallelizing the data flow computation [GZZ89, GPS90, LMR91, KGS94]. Parallel analysis algorithms



(i)

$$\begin{aligned}
X(1) &= \mathit{init}\\
X(2) &= X(1)\\
X(3) &= f_3(X(2)))\\
X(4) &= X(2)\\
X(5) &= f_5(X(4))\\
X(6) &= X(5) \sqcap X(9)\\
X(7) &= f_7(X(6))\\
X(8) &= f_8(X(6))\\
X(9) &= X(7) \sqcap X(8)
\end{aligned}$$

(ii) exhaustive

$$\begin{aligned}
X(1) &= \mathit{init}\\
X(3) &= f_3(X(1)))\\
X(5) &= f_5(X(1))\\
X(6) &= X(5) \sqcap X(10)\\
X(7) &= f_7(X(6))\\
X(8) &= f_8(X(6))\\
X(9) &= X(7) \sqcap X(8)
\end{aligned}$$

(iii) optimized exhaustive

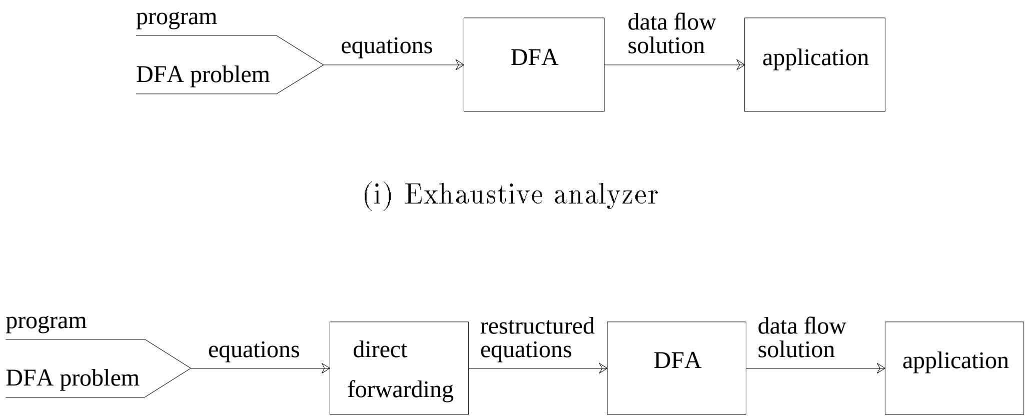
Figure 1.1: A flow graph fragment with initial and optimized exhaustive equation system.

are obtained by decomposing the data flow problem into a series of subproblems that can be solved in parallel. The results obtained for the subproblems are then combined to obtain the complete data flow solution. The various parallel data flow algorithms that have been developed differ in the way they decompose a data flow problem. Although, moderate amounts of parallelism may be detected using these techniques, the parallel analyses do not scale well for large numbers of processors due to insufficient parallelism in the exhaustive data flow solution definition.

1.2.2 Forwarding Techniques

Other sequential approaches achieve performance improvements by reducing the amount of intermediate computation that is performed during the analysis. Some of the intermediate computations can be suppressed by directly forwarding information from the points where it is generated to the points where it is used. Consider the equation system in Figure 1.1 (ii). Instead of repeatedly evaluating the copy equations $X(2)$ and $X(4)$ during the fixed point iteration, their values can be directly forwarded to the equations that need them. The resulting reduced equation system, after forwarding has been applied and the two equations $X(2)$ and $X(4)$ have been eliminated, is shown in Figure 1.1 (iii).

Figure 1.2 illustrates the coupling of forwarding techniques with data flow analysis. Figure 1.2 (i) depicts the traditional analysis design, where data flow analysis is applied in an isolated phase and all analysis results are fed to the application afterwards. Figure 1.2 (ii)



(i) Exhaustive analyzer

(ii) Analyzer an optimizing forwarding preparatory phase.

Figure 1.2: Date flow analyzer (DFA) design.

shows the analysis design with a forwarding phase. Forwarding techniques are applied prior to the actual analysis in order to restructure and optimize the solution definition for a faster fixed point computation.

Forwarding techniques [CFR$^+$91, FOW87, BMO90, DRZ92, JP93, CCF90] use some form of a derived graph representation that provides direct connections (i.e., forwarding edges) between the points that generate information and the points where the information is needed. However, these graph based approaches are limited to certain kinds of data flow problems that can actually take advantage of the introduced direct connection edges. Available expressions is an example of a data flow problem that does not benefit from the direct connection edges provided by these graphs. An exception are the *sparse evaluation graphs* (SEG) [CCF90] that are explicitly constructed for each data flow problem to be solved. Although SEGs are general, they are not equally effective for all problems. For example, the SEG for the problem of live variables is likely to result in little or no improvements unless the live variable problem is broken into a series of N subproblems, one for each program variable, hence requiring the construction of one graph for each program variable.

Furthermore, although forwarding techniques can effectively reduce the amount of intermediate computation during the analysis, the benefits of these techniques are limited by the assumption that the data flow solution is to be computed everywhere in the program. Thus, the analysis may still spend time and space in unnecessary computations if the application actually requires only a subset of the complete data flow solution.

1.2.3 Incremental Data Flow Analysis

Analyses that explicitly depart from exhaustive solution computations have previously been developed in an incremental context to handle evolving or changing software. The goal of incremental data flow analysis [Ros81, Zad84, RP88, PS89, RR91] is to avoid costly re-computations of the exhaustive solution in response to small changes in a program. Instead of fully re-analyzing a program from scratch each time a change is made to the program, a previously computed exhaustive solution, that has become invalid as a result of the change, is incrementally updated. Given a point where a change occurred, incremental analysis techniques identify the portion of the global solution that is invalidated by the change and correct or re-compute data flow information only for the identified portion.

Figure 1.3 illustrates the incremental analysis approach. Figure 1.3 (i) shows the control flow graph fragment from Figure 1.1 but this time assuming that a program change occurred at node 5 that invalidated the previous value of equation $X(5)$. Instead of re-evaluating the complete equation system from Figure 1.1 (i), incremental analysis identifies and re-evaluates only the portion of the equation system that is affected by the change at node 5. The affected equations are shown in Figure 1.3 (ii) and the flow graph portion that corresponds to these affected equations is shown in bold in Figure 1.3 (i).

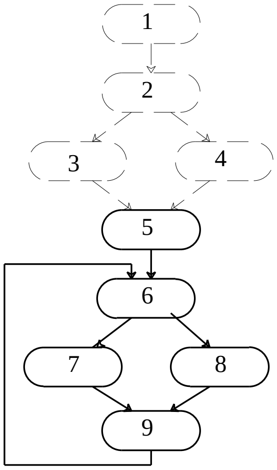
The illustration in Figure 1.3 shows that, unlike forwarding techniques or parallelization, incremental analysis effectively avoids exhaustive computations and leads to a partial analysis. However, incremental data flow analysis can only avoid exhaustive *re*-computations and does not address the problem of having to compute a costly exhaustive solution initially.

1.2.4 Demand-Driven Data Flow Analysis

Developing an approach that, like incremental analysis, effectively avoids exhaustive computations but that is also applicable for the initial solution computation leads to a *demand-driven* analysis design. As shown in Figure 1.5 demand-driven data flow analysis is no longer performed in isolation from the application. Instead, demand-driven analysis is interleaved with the application in such a way that the computation of data flow is performed only if triggered by a request from the application. A demand-driven analysis is partial rather than exhaustive and evaluates only the portion of the data flow solution that is needed to satisfy the actual demands.

Consider Figure 1.4 and assume that only the solution at node 5 is demanded. Since only the value for equation $X(5)$ is of interest, other equations in the system that do not contribute to this value do not need to be evaluated. The portion of the equation system that is actually needed to compute equation $X(5)$ is shown in Figure 1.4 (ii). The portion of the flow graph that corresponds to this partial equation system is shown in bold in Figure 1.4 (i).

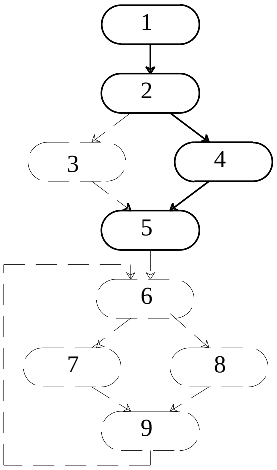
Demand-driven analysis is a promising approach to reduce the analysis overhead in applications that require data flow only selectively (i.e, only at some program points) and/or



$$
\begin{aligned}
X(5) &= f_5(X(4))\\
X(6) &= X(5) \sqcap X(9)\\
X(7) &= f_7(X(6))\\
X(8) &= f_8(X(6))\\
X(9) &= X(7) \sqcap X(8)
\end{aligned}
$$

(i) (ii) incremental

Figure 1.3: The affected equations for incremental update after a change at node 5.



$$
\begin{aligned}
X(1) &= init\\
X(2) &= X(1)\\
X(4) &= X(2)\\
X(5) &= f_5(X(4))
\end{aligned}
$$

$$
\begin{aligned}
X(1) &= init\\
X(5) &= f_5(X(1))
\end{aligned}
$$

(i) (ii) partial (iii) optimized partial

Figure 1.4: The partial and optimzed partial equation systems for a demand at node 5.

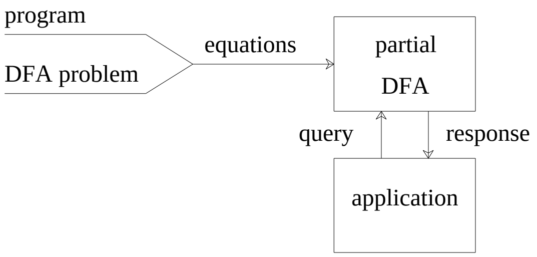


Figure 1.5: Demand-driven analyzer design.

sparsely (i.e., only a subset of the information at each selected point). Furthermore, a demand-driven analysis approach naturally provides the capability to service user requests in interactive software tools such as debuggers where the nature and extent of user queries may vary depending on the user and the program.

Like incremental analysis, demand-driven analysis is an application-directed approach that reduces analysis overhead by computing partial solutions rather than exhaustive ones. However, the resulting partial analyzers are not necessarily computationally minimal. The partial solution computation may still include redundancies, which are revealed when taking a closer look at the intermediate computations. Eliminating these remaining redundancies is the goal of forwarding techniques. In fact, demand-driven analysis and forwarding techniques are orthogonal approaches that are targeted at different types of redundancies in the data flow computation. Consider again Figure 1.4. The partial equation system in (ii) still contains unnecessary intermediate computations in the form of copies that could be eliminated through forwarding. The resulting optimized partial equations after forwarding is shown in Figure 1.4 (iii). Thus, in order to enable aggressive improvements of data flow analyzers, both approaches should be considered.

Recently, two approaches to demand-driven analysis have been presented by Reps, Horwitz and Sagiv [RSH94, RHS95, SRH95a]. In these approaches, a demand-driven analysis is modeled as a certain 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. A disadvantage of the graph-reachability approach is the need to construct an exploded supergraph for each data flow problem to be solved. The size of the exploded supergraph can be substantial and, correspondingly, so can be the time needed for the graph construction. The authors report that 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 [SRH95b]. Although a recent variation of the graph-reachability approach

[SRH95a] results in a more compact version of the exploded supergraph for copy constant propagation, the size of the graph is not reduced for other problems, such as the classical bit vector problems.

Another problem with the graph-reachability framework results from the dependence on a specialized graphical program representation. By departing from the standard fixed point solution definition, the graph-reachability approach makes it difficult to combine the demand-driven analysis with other analysis improvements techniques that are fixed point based, such as the forwarding or parallelization techniques.

1.3 Overview of the Research

This research develops and experimentally evaluates two systematic approaches to improve the performance of traditional data flow analyzers. This first approach, which represents the core of this research, consists of the development of a new demand-driven framework for interprocedural data flow analysis. To avoid the deficiencies of previous approaches aimed at reducing analysis cost, the demand-driven analysis framework satisfies the following design goals:

- *Generality*: The demand-driven approach is applicable to a general class of data flow problems.
- *Practicality*: The demand-driven technique is efficient in practice, which is demonstrated through experimentation.
- *Application-independence*: The developed approach is applicable in classical compiler applications as well as in applications for software tools.
- *Fixed point based*: The approach does not require the construction of a specialized graphical program representation and models the problem based on well-understood fixed point computations.
- *Integratable*: The demand-driven analysis algorithms can easily be integrated with other fixed point based techniques, such as forwarding.
- *Parallelizable*: The developed demand-driven algorithms have a natural parallelization.

Although the practicality and usefulness of the demand-driven approach is established as part of this research through both analytical examinations and experimentation, there is still room for further improvements. To complement the demand-driven approach and to address the elimination of the remaining inefficiencies in the data flow computation, this research also includes the development of a second approach. This second approach consists of a framework for a new generalized forwarding technique: *congruence partitioning*. A

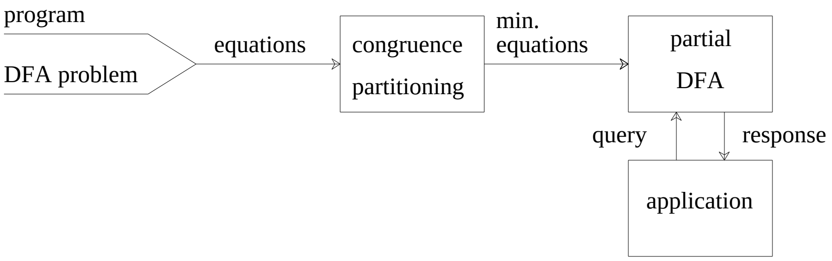


Figure 1.6: Demand-driven analyzer with congruence partitioning.

congruence partitioning is computed to reduce the size of a data flow equations by identifying and eliminating congruent equations from the system. Congruence partitioning is as general as the previously most general approach to forwarding, that is, sparse evaluation graphs [CCF90]. Thus, congruence partitioning is applicable to any monotone data flow problem. However, congruence partitioning is more powerful than previous techniques and enables optimizations of an equation system that can not be achieved using any of the previous methods.

The demand-driven analysis approach and congruence partitioning constitute two complimentary approaches for improving data flow analysis. The two approaches can be pursued individually or in combination. The coupling of congruence partitioning with a demand-driven analyzer is illustrated in Figure 1.6. In this combination congruence partitioning is applied first in order to construct a reduced definition of the solution. Instead of exhaustively evaluating the reduced equation system a demand-driven analyzer is used to compute only what is needed by the application. The demand-driven framework and the framework for congruence partitioning together enable improvements in interprocedural data flow analysis that go well beyond the improvements achievable by previous techniques.

The remainder of this section provides a brief overview of the two approaches presented in this thesis.

1.3.1 A Demand-Driven Analysis Framework

The first approach developed in this thesis consists of an algebraic framework for deriving demand-driven interprocedural analysis algorithms. This framework models the demand-driven analysis of a program as a query system. 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., in compiler optimization) or manually by the user (e.g., in an interactive software tool). A query is a pair $q =< y, n >$ specified by a set of data flow facts y and a program point n. Query $q =< y, n >$ raises the question as to whether the set of facts y is part of the exhaustive solution at 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 a demand-driven query propagation procedure.

The generic demand-driven algorithm is precise for the class of distributive data flow problems with finite lattices. If applied to a monotone problem the algorithm can still be used to provide approximate but safe information. Alternatively, a less efficient but precise framework variation is presented to handle non-distributive data flow problems.

The practical benefits of the demand-driven framework are demonstrated through numerous experiments. An experimental study of demand-driven algorithms for two problems, namely reaching definitions and copy constant propagation, was conducted to evaluate the performance of demand-driven analysis independently of a particular application. To complete the experimental study, a second set of experiments was carried out that evaluates demand-driven analyzers in a specific software engineering application.

As an additional benefit, the developed 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. Unlike previous techniques for data flow analysis parallelization, the amount of parallelism in the data flow computation is independent of the program structure and depends only on the size of the program and the number of generated queries.

1.3.2 Congruence Partitioning

The second approach developed in this thesis consists of a framework for a new generalized forwarding technique: *congruence partitioning*. Congruence partitioning reduces the size of a data flow equation system through the discovery of *congruence relationships* among solution equations. Two equations are *congruent* if their fixed points are equal. Thus, at least one of two congruent equations is redundant and can therefore be eliminated. By repeatedly applying this elimination process, an equivalent but smaller equation system can be constructed that includes only a single equation from each class of congruent equations. The congruence partitioning framework includes efficient partitioning algorithms for computing different kinds of congruence relations.

Congruence partitioning is applicable to both exhaustive and demand-driven analyzers. As with other forwarding techniques, congruence partitioning is applied as a preparatory phase prior to the actual analysis in order to restructure and optimize the data flow solution definition. However, congruence partitioning is more powerful than previous forwarding techniques in that it enables more aggressive equation system reductions.

1.4 Organization of the Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 provides the background in global data flow analysis. An overview of the demand-driven analysis framework along with a discussion of the applications of demand-driven data flow analysis is given in Chapter 3. Chapter 4 provides the technical details of the demand-driven analysis framework and describes the individual framework components.

The experimental evaluation of the demand-driven approach is presented in Chapters 5 through 7. Chapters 5 and 6 consider specific analysis problems and show that the general framework can be efficiently implemented for these problems. Chapter 5 presents and experimentally evaluates demand-driven analyzers for the class of *Gen-Kill* problems. Chapter 6 presents a demand-driven analyzer for copy constant propagation and its experimental evaluation. Chapter 7 examines the demand-driven approach in a software engineering application, namely data flow integration testing. A new and efficient demand-driven approach to data flow integration testing is developed and experimentally evaluated.

The second approach for improving data flow analysis developed in this dissertation is presented in Chapter 8. Chapter 8 presents the formal framework for congruence partitioning and includes a discussion of how congruence partitioning can be coupled with the demand-driven analyzers from the previous sections. The dissertation is concluded in Chapter 9 with a summary and a discussion of future directions.

Chapter 2

Background

Since their introduction in the 70s, data flow analyses have formally been modeled in algebraic frameworks called *data flow frameworks*. The study of data flow frameworks was motivated by the need for a uniform model for the design and development of program analysis techniques. This chapter surveys the pertinent background in global data flow analysis. First, the program representation commonly used in data flow analysis is presented in Section 2.1. Section 2.2 presents the algebraic frameworks for intra- and interprocedural data flow analysis. The algorithms that are commonly used to solve problems formulated in these frameworks are presented in Section 2.3.

2.1 Program Representation

A program consisting of a set of possibly recursive procedures is represented by an *interprocedural control flow graph* (ICFG). An ICFG is a collection of control flow graphs $G_1, \ldots, G_k$, such that $G_i = (N_i, E_i)$ is a control flow graph representing 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. It is assumed that $|E| = O(|N|)$. The sets $pred(n) = \{m|(m,n) \in E_i\}$ and $succ(n) = \{m|(n,m) \in E_i\}$ denote the sets of immediate predecessors and successors of node n, respectively. For a call site node s, $call(s)$ denotes the procedure called from s. A program and its ICFG are shown in Figure 2.1.

During the analysis only *valid execution paths* should be considered. An execution path is a sequence of nodes $\pi = n_1 \ldots n_k$, such that for $1 \leq i < k$ either (i) $(n_i, n_{i+1}) \in E$ (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 $m \in pred(n_{i+1})$ such that $call(m) = p$ (procedure return). Furthermore to be valid, each procedure exit node $exit_p$ in π must be followed by a successor node of the matching call site n with $call(n) = p$

```
procedure main
declare a,b;
begin
  a:=1;
  read(b);
  call p;
end

procedure p
begin
 if (cond) then a:=b
    else b:=1;
         call p;
  endif;
end
```

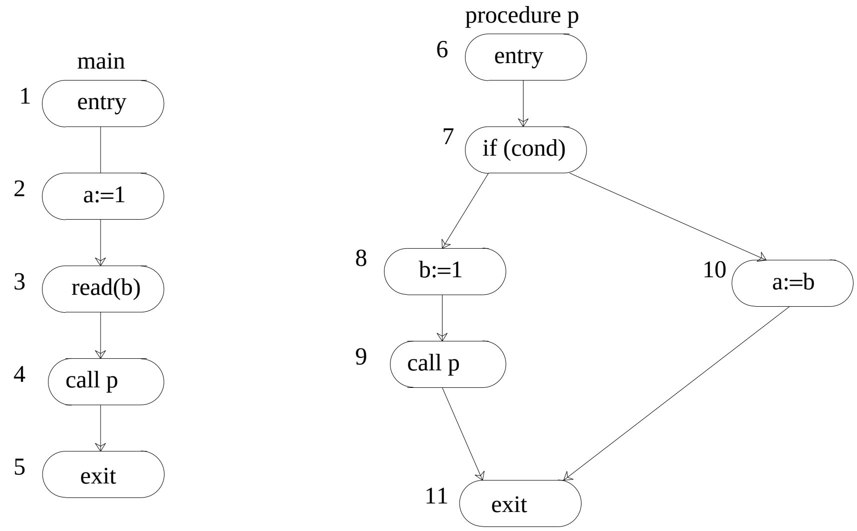


Figure 2.1: A program and its ICFG.

that most recently occurred in π prior to $exit_p$. A path is not valid if it enters a procedure p from one call site but upon reaching p's exit node returns to a different call site.

Consider the example in Figure 2.1. The path $1, 2, 3, 4, 6, 7, 10, 11, 5$ is a valid execution path but the path $1, 2, 3, 4, 6, 7, 10, 11, 11, 5$ is not valid. Invalid paths violate the *calling context* of procedures and may lead to imprecise information if considered during the analysis. The set of interprocedurally valid execution paths from a node n to a node m is denoted by $IVP(n, m)$.

Another structure commonly used in interprocedural analysis is the *call graph* of a program. A call graph is a directed graph $G = (N, E)$, where the nodes in N represent the procedures in the program and there exists an edge $(p, q) \in E$ for each call site in a procedure p that calls a procedure q. Since there may be multiple call sites in p calling q, the call graph G is a multi graph. A path in a call graph is called a *call chain*.

2.2 Data Flow Frameworks

An algebraic framework for data flow analysis was introduced by Kildall [Kil73]. Refinements and extensions of the original framework were suggested by others, including Kam and Ullman [KU77, KU76] and Graham and Wegman [GW76]. Data flow frameworks are also called *monotone frameworks* based on the monotonicity of the information flow functions in the framework. A function f is *monotone* if $x \leq y$ implies $f(x) \leq f(y)$.

A comprehensive overview of data flow frameworks is presented in [MR90].

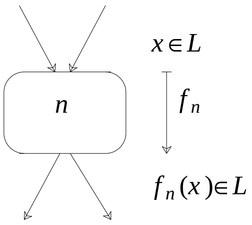


Figure 2.2: Data flow at a node n

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

- $(L, \sqcap, \sqsubseteq, \perp, \top)$ is a complete lattice representing the universe of program facts with a partial order $\sqsubseteq$, a least element $\perp$ (bottom), a greatest element $\top$ (top), and a *meet* operator $\sqcap$, such that for all $x, y, z \in L$:

$$\begin{array}{ll} x \sqcap x = x & \text{(idempotence)} \\ x \sqcap y = y \sqcap x & \text{(commutativity)} \\ x \sqcap (y \sqcap z) = (x \sqcap y) \sqcap z & \text{(associativity)} \end{array}$$

- $F \subseteq \{f : L \mapsto L\}$ is a set of monotone flow functions over L that contains the identity function *id* and that is closed under composition and pointwise meet:

$$\forall f, g \in F : f \cdot g \in F \text{ and if } h(x) = g(x) \sqcap f(x) \text{ then } h \in F.$$

A function $f \in F$ is called $\sqcap$-*distributive* (i.e., distributive with respect to the meet $\sqcap$) if $f(x \sqcap y) = f(x) \sqcap f(y)$. If the meet operator $\sqcap$ is clear from the context, the function is simply called distributive. If all functions in F are distributive, D is called a *distributive data flow framework.*

An *instance* of a framework $D = (L, F)$ is given by a pair $I = (G, m)$, where:

- $G = (N, E)$ is an ICFG,

- $m : N \mapsto F$ is a mapping that maps each node in the ICFG to a function in F.

The function $m(n)$ mapped to a node n (also denoted f_n) models the data flow when execution passes through node n. As illustrated in Figure 2.2, if $x \in L$ holds on entry of a node n then $f_n(x) \in L$ holds on exit from node n. For a given path $p = n_0, n_1, \ldots, n_k$ function application along this path is denoted as: $f_p = f_k \cdot \ldots \cdot f_1 \cdot f_0$.

The problems that can be phrased in a data flow framework have been classified along various dimensions. According to their analysis direction, data flow problems are classified as either *forward* or *backward* problems. In a forward problem information is propagated in the direction of control while in a backward problem information flows in the direction opposite to control. Both forward and backward problems can be uniformly modeled in the above framework by representing a backward problem as a forward problem on the reverse control flow graph. *Bidirectional* problems require information propagated in both a forward and a backward direction. The problem of *partial redundancy elimination* was originally formulated as a bidirectional problem [MR79]. However, recently two new formulations have been presented to solve partial redundancy elimination in a sequence of forward and backward analyses [KS92, DRZ92]. A general method for breaking bidirectional problems into a sequence of uni-directional problems was presented in [DRZ92].

Another common approach is to classify data flow problems as *union* or *intersection* problems. Union and intersection problems are problems in which the lattice L has a powerset structure. In a union problem the meet operator is conventional set union, whereas the meet operator in an intersection problem is set intersection. The meet lattice structure most naturally models intersection problems. However, by duality [Bir84] every union problem can be phrased in a meet lattice.

Data flow problems are formulated as either *intraprocedural* or *interprocedural* problems. An intraprocedural problem only considers the data flow within each procedure. Interprocedural problems analyze, in addition, the interactions among procedures at call sites. Usually, intraprocedural data flow problems (e.g., live variables) also have an interprocedural version. However, there are also problems, such as the alias problem for reference parameters [Coo85] that only exist at the interprocedural level.

Finally, data flow problems are also classifiable in terms of their algebraic complexity. The most general characterization is the class of *monotone problems* with the subclass of *distributive problems*. An important subclass of the distributive problems is the class of *bit vector problems*. In a bit vector problem, there exists a natural mapping from the lattice L to the set $\{0,1\}^n$, such that each element in L can be represented by a bit vector of length n. Bit vector problems have efficient implementations since operations on lattice elements can be implemented as boolean operations on bit vectors. Another particular simple subclass of the distributive problems are the *partitionable problems* (also called *locally separable* in [RHS95]). A problem is partitionable if it can be broken into a sequence of separate simple analyses, one for each program object, such as variables or expressions. A partitionable problem is characterized by a restricted form of flow functions. Each flow function f is of the form: $f(x) = c$ or $f(x) = x \sqcap c$, where c is a constant in L. The four classic data flow problems: reaching definitions (REACH), available expressions (AVAIL), live variables (LIVE) and very busy expressions (BUSY) are examples of partitionable problems. These problems are also called *Gen-Kill* problems since their original flow functions (i.e., the functions prior to partitioning) are of the form: $f(x) = (x \perp Kill) \cup Gen$, where Gen and $Kill$ are constant subsets in L. Note that the four classic problems are also bit vectors problems.

However, not every bit vector problem is partitionable. Faint variables[1] is an example of a problem that can be implemented using bit vectors but that is not partitionable.

2.2.1 The Intraprocedural Solution (Kam/Ullman)

The solution for an instance of a data flow framework is an assignment of lattice elements to program nodes. The optimal solution is the *meet-over-all-paths* (*mop*) solution, which precisely captures the flow of information along each valid execution path. The assignment $mop : N \mapsto L$ is defined as:

$$\begin{aligned} mop(entry_{main}) &= \bot \\ mop(n) &= \bigsqcap_{p \in IVP(entry_{main}, n)} f_p(\bot) \end{aligned} \tag{2.1}$$

The solution *mop* assigns to each node the set of data flow facts (i.e., lattice elements) that hold on node entry for every valid execution path leading to the node. The bottom value $\bot$ at node $entry_{main}$ indicates that no information holds on program entry. Since the *mop* solution is in general undecidable, data flow algorithms provide an approximation to the *mop* solution. Kam and Ullman showed in their standard *intra*procedural framework [KU77] that a unique and decidable approximation to the *mop* solution is computed as the *greatest fixed point* (*gfp*) of the following system of *data flow equations*:

$$\begin{aligned} X(entry_{main}) &= \bot \\ X(n) &= f_n(\bigsqcap_{m \in pred(n)} X(m)). \end{aligned} \tag{2.2}$$

The equation $X(n)$ describes the data flow facts that hold on entry of node n. The greatest fixed point can be computed over the equation system by initialing all equations with the top value $\top$ and then iteratively lowering the initial values by repeatedly evaluating the equations until the system converges.

In a monotone data flow framework D, there always exists a unique *gfp* of the equation system (2.2). The *gfp* solution is always lower in the lattice than the *mop* solution and is therefore a safe approximation. If D is distributive, the *gfp* solution is equal to the *mop* solution. If D is non-distributive then there exists an instance of D in which *gfp* is strictly lower than *mop* [KU77].

2.2.2 The Interprocedural Solution (Sharir/Pnueli)

During intraprocedural analysis the propagation of information is restricted to the control flow paths within each procedure. *Inter*procedural analysis considers in addition the propagation of information across procedure boundaries at call and return points. Several formal

[1] A variable v is a faint variable if v is dead or if v is defined in terms of another variable that is faint.

frameworks for interprocedural analysis have been developed [CC77c, Ros81, JM73, SP81, KS92]. This section presents Sharir and Pnueli's functional approach to interprocedural analysis which also serves as the basis for the demand-driven analysis framework presented in Chapter 3.

Sharir and Pnueli [SP81] presented a two-phase approach to interprocedural analysis that ensures that only valid execution paths are considered and the calling context of each procedure is preserved. During the first phase the data flow effect of each procedure is analyzed independent of its calling context. The results of this phase are procedure *summary functions* as defined in equation system (2.3). 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 $\phi_{(entry_p, n)}$ that maps an element $x \in L$ to the corresponding element $\phi_{(entry_p, n)}(x) \in L$ that holds on entry to node n assuming that x holds upon procedure entry.

$$\begin{aligned}
&\phi_{(entry_p, entry_p)}(x) = x \\
&\phi_{(entry_p, n)}(x) = \bigsqcap_{m \in pred(n)} \begin{cases} f_m \cdot \phi_{(entry_p, m)}(x) & \text{if } m \text{ is not a call site} \\ \phi_{(entry_q, exit_q)} \cdot \phi_{(entry_p, m)}(x) & \text{if } call(m){=}q \end{cases}
\end{aligned} \tag{2.3}$$

The actual calling context of a called procedure is propagated during the second phase based on the summary functions. The data flow solution $X(n)$ at a node n in a procedure p is determined by mapping the solution $X(entry_p)$ that holds on entry of p to node n using the summary function $\phi_{(entry_p, n)}$. The equation system (2.4) defines the data flow solution $X(n)$ that holds on entry of node n.

$$X(entry_{main}) = \bot$$

For each procedure p:

$$X(entry_p) = \bigsqcap_{call(m)=p} X(m)$$

For non-entry nodes n:

$$X(n) = \bigsqcap_{m \in pred(n)} \begin{cases} f_m(X(m)) & \text{if } m \text{ is not a call site} \\ \phi_{(entry_q, exit_q)}(X(m)) & \text{if } call(m){=}q \end{cases} \tag{2.4}$$

For finite lattices Sharir and Pnueli propose an efficient algorithm to solve the equation systems 2.3 and 2.4. If the lattice is finite the equation system 2.3 is finite so that the solution can be computed using a standard iterative fixed point algorithm based on the

Gen and *Kill* sets		
node n	Gen_n	$Kill_n$
1	-	-
2	a_2	a_{10}
3	b_3	b_8
4	-	-
5	-	-
6	-	-
7	-	-
8	b_8	b_3
9	-	-
10	a_{10}	a_2
11	-	-

Summary function for p	
node n	$\phi_{(6,n)}(X)$
6	X
7	X
8	X
9	$(X - \{b_3\}) \cup \{b_8\}$
10	X
11	$X \cup \{b_3, a_{10}\}$

Reaching definitions	
node n	$X(n)$
1	-
2	-
3	a_2
4	a_2, b_3
5	a_2, a_{10}, b_3, b_8
6	a_2, b_3, b_8
7	a_2, b_3, b_8
8	a_2, b_3, b_8
9	a_2, b_3, b_8
10	a_2, b_3, b_8
11	a_2, a_{10}, b_3, b_8

Figure 2.3: Relevant data flow sets for REACH analysis of Figure 2.1.

initial value $\top$ for equations $\phi_{(entry_p,n)}(x)$ for each node n in procedure p and each lattice element x. Once the summary function equation system has been solved, the actual solution equations in system 2.4. are computed during a second phase using the computed summary function values.

To illustrate the definition of the interprocedural analysis framework consider the problem of determining the sets of interprocedural *reaching definitions* (REACH) in a program. The definition of a variable v is a reaching definition at a node n if there exists a valid execution path from the definition to node n that does not re-define v. REACH is the problem of computing at each node the set of definitions that reach that node.

REACH is a union problem for which the lattice is the powerset lattice of the set of definitions (*DEF*) in the program. The meet is set union (”$\cup$”) and $\top = \emptyset$ and $\bot = DEF$. The partial order in this lattice is reverse set inclusion (”$\supseteq$”). The framework for REACH is distributive and an instance is given by an ICFG for a particular program and a mapping of flow functions f_n to each node n such that:

$$f_n(X) = (X \perp Kill_n) \cup Gen_n.$$

$Kill_n$ is the set of definitions that are *killed* at n, that is, the definitions that cannot reach the end of node n because of a re-definition contained in n. Gen_n is the set of *generated* definitions, that is, the definitions that occur in node n without a subsequent re-definition at n. The *Gen* and *Kill* sets for the example in Figure 2.1 are shown in Figure 2.3. To distinguish multiple occurrences of the same variable, definitions of variables are subscripted with the number of the node that contains the definition.

For a set of definitions X, the summary function value $\phi_{(entry_p,exit_p)}(X)$ denotes the set of definitions that reach the exit of procedure p assuming that the definitions in X reach the entry of p. Figure 2.3 shows the solutions to the equation systems for the summary function $\phi_{(6,11)}$ for procedure p from Figure 2.1.

2.2.3 Abstract Interpretation

Data flow frameworks provide a uniform way to model and specify program analyzers. However, they do not include explicit mechanisms to facilitate formal correctness proofs of the modeled program analyses. To formally prove correctness it is necessary to relate the program invariants derived by an analyzer to the program's formal semantics that are assumed to correctly describe the actual program behavior. Cousot and Cousot [CC77a, CC79] developed a variant of the classical data flow framework called *abstract interpretation* that explicitly incorporates formal semantics. An abstract interpretation models data flow analysis as an approximation of the program's formal semantics, which are in general not computable. The formal semantics are expressed by a lattice of sets of possible program states. The simpler lattice actually used in a particular data flow analysis is viewed as an abstraction (i.e., an approximation) of the precise semantics. Transfer between the two lattices is achieved through abstraction and concretization functions. Based on this connection between an abstraction and a concretization function, program facts derived during data flow analysis are proven to be correct approximations of the program's formal semantics.

2.3 Data Flow Analysis Algorithms

Algorithms for global data flow analysis fall into two major classes: iterative algorithms and elimination algorithms. The algorithms in either class deliver the greatest fixed point solution of the respective data flow equation system. In iterative algorithms, the equations are repeatedly evaluated until the evaluation converges to a fixed point. Elimination algorithms compute the fixed point by decomposition and reduction of the control flow graph to obtain subsequently smaller systems of equations. If not stated otherwise, all asymptotic complexities of data flow algorithms presented in this sections are based on intraprocedural analysis.

2.3.1 Iterative Algorithms

Iterative algorithms for global data flow analysis originate with Kildall's *worklist iteration* [Kil73]. In worklist iteration, nodes are successively removed from a worklist and the associated equations are evaluated. If, as a result, the lattice value changes at the currently inspected node, all successors of that node are added to the list. Worklist iteration terminates when the list is empty, resulting in a worst-case running time for intraprocedural analysis of $O(|N| \times height(L))$ node visits, where $height(L)$ denotes the height [2] of semi-lattice L. Sharir and Pnueli proposed a worklist algorithm for their interprocedural two-phase equation systems. For finite lattices, the algorithm requires $O(MaxCall \times height(L) \times |L| \times |N|)$ time, where *MaxCall* is the maximal number of call sites calling a single procedure.

Reverse postorder iteration [KU76, HU73] is an iterative algorithm that proceeds in multiple passes over the control flow graph. Kam and Ullman have shown that for *rapid*[3] data flow frameworks, reverse postorder iteration requires $d + 3$ passes, where d is the maximum number of back edges in a cycle-free path (the maximal loop nesting depth in structured programs). In practice, values for d are usually less than or equal to three [Knu71]. Kam and Ullman's notion of rapidity [KU76] is a property of the function space. Intuitively, rapidity implies that all information can be propagated along acyclic paths. A classical problem that is not rapid is constant propagation.

Other iterative algorithms include *node listings* [Ken75], which are specifications of the order in which the nodes in the graph are visited and several variants of iterative algorithms presented by Horwitz, Demers and Teitelbaum [HDT87].

2.3.2 Elimination Algorithms

The key idea in elimination algorithms is to reduce the original system of equations to subsequently smaller systems by structure-driven graph transformations and corresponding substitutions in the equation system. If no further reductions are possible, the resulting equation system is evaluated, and the solution to the original system is obtained through propagation by reversing the reduction process. A comprehensive survey of elimination algorithms for data flow analysis appears in [RP86].

In *interval analysis* [AC77, Coc70] the control flow graph is subsequently partitioned into subgraphs called intervals. An interval is replaced by a single node that contains the information local to the interval. The partitioning continues until the graph is reduced to a single node for which the data flow solution can easily be obtained. An improvement over Allen and Cocke's original interval analysis is presented by Hecht and Ullman called $T_1 \perp T_2$ *analysis* [HU73], based on two graph transformations T_1 and T_2. $T_1 \perp T_2$ analysis proceeds in $O((d + 2) \times |E|)$ time, where d refers to the depths of the graph as described in the previous section. Both Allen and Cocke's interval analysis and $T_1 \perp T_2$ analysis are restricted to *reducible* control flow graphs. An extension that can handle irreducible flow graphs was presented by Graham and Wegman [GW76]. Graham and Wegman's analysis proceeds in $O(|E| \log |E|)$ time for *fast* [4] problems.

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

[3] A data flow problem is called *rapid* if $\forall g, f \in F, \forall x \in L : f \cdot g\,(\perp) \sqsupseteq g(\perp) \sqcap f(x) \sqcap x$.

[4] A data flow problem is *fast* if $\forall f \in F, \forall x \in L : f(x) \sqcap x \sqsubseteq f(f(x))$.

2.3.3 Other Methods

Some approaches to data flow analysis do not assume an algebraic lattice framework. *Symbolic evaluation* techniques derive program invariants by associating symbolic values with program variables and by computing algebraic closed form expressions over these symbolic values. Using a specialized program representation called the *global value graph* [RL77, RT82] efficient techniques for symbolic evaluation have been developed for a restricted class of data flow problems.

Another approach that does not assume a lattice framework is Tarjan's *path algebra* [Tar81b, Tar81a]. The flow of information is computed by parsing a *path expression*, i.e., a regular expression representation of program paths, where each expression operator is associated with an information transfer function.

Recently, Reps, Horwitz and Sagiv presented a new graph oriented approach to interprocedural data flow analysis [RHS95]. In this approach, a data flow problem is transformed into a specialized 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 is applicable to distributive data flow problems with a lattice that is the powerset of a finite set.

Chapter 3

Overview

The core of this research is the development of a formal framework for demand-driven interprocedural data flow analysis. This chapter provides an overview of the demand-driven approach and is organized as follows. Section 3.1 motivates the demand-driven approach using the example of copy constant propagation. An outline of the demand-driven framework and its components is presented in Section 3.2. The chapter concludes with a discussion of the benefits of the demand-driven approach for the parallel execution of data flow analysis.

3.1 Example: Copy Constant Propagation

As an illustration of the demand-driven approach consider the problem of *copy constant propagation* (CCP). CCP is a distributive version of the general (non-distributive) constant propagation analysis [Kil73]. Unlike general constant propagation, CCP does not evaluate arithmetic expressions. A variable v is a *copy constant* if v is assigned a constant value or if v is assigned a copy of another variable that is a copy constant.

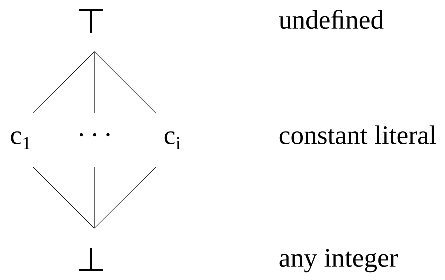
The CCP lattice for a program with k variables is the product lattice L^k, where the component lattice L is defined as shown in Figure 3.1 (i). Note that the component lattice in CCP is finite, since the only possible constant values for a copy constant are the constant literals that occur in the program 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 $\sqcap$ and the dual join operator $\sqcup$ are defined pointwise according to the partial order depicted in Figure 3.1 (ii).

A *base element* in L^k is a tuple $(x_1, \ldots, x_k)$ with a single non-bottom component x_i: $x_i = c$ and $x_j = \perp$ for $j \neq i$. Such a base element is also written as:

$$[v_i{=}c] = (\perp, \ldots, \perp, x_i = c, \perp, \ldots, \perp)$$

Similarly, any element $x \in L^k$ that results as a finite join of base elements is written as:

$$x = [v_1 = c_1] \sqcup \ldots \sqcup \ [v_l = c_l] = [v_i = c_i, \ldots, v_l = c_l]$$



(i)

$\sqcap$	$\bot$	c_i	$\top$
$\bot$	$\bot$	$\bot$	$\bot$
c_j	$\bot$	c_i if $c_i = c_j$ $\bot$ otherwise	c_j
$\top$	$\bot$	c_i	$\top$

(i)

Figure 3.1: The CCP lattice L (i) and the definition of the meet operator (ii).

The distributive flow functions in CCP are defined pointwise for each component:

$$f(x_1, \ldots, x_k) = (f(x_1, \ldots, x_k)_1, \ldots, f(x_1, \ldots, x_k)_k).$$

The component function $f(x_1, \ldots, x_k)_j$ with respect to variable v_j is defined in Table 3.1 for various types of assignments. For example, for a constant assignment $v_i := c$ the component function is defined as $f(x_1, \ldots, x_k)_i = c$ indicating that variable v_i has constant value c after the execution of the assignment. The assignment has no effect on the values of other variables v_j. Thus, their component functions are identity functions, i.e., $f(x_1, \ldots, x_k)_j = x_j$. For the same reason, the component functions for a conditional expression are also identity functions.

The flow functions for CCP are illustrated for the control flow graph shown in Figure 3.2, where each flow function for a non-call node is shown next to the node. 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 variables a,b and c, respectively.

statement at node n	function $f_n(x)_j$, where $x = (x_1, \ldots, x_k)$
const. assignment: $v_i := c$	$f_n(x)_j = \begin{cases} c & \text{if } i = j \\ x_j & \text{otherwise} \end{cases}$
copy: $v_i := v_l$	$f_n(x)_j = \begin{cases} x_l & \text{if } i = j \\ x_j & \text{otherwise} \end{cases}$
expr. assignment: $v_i := expr.$	$f_n(x)_j = \begin{cases} - & \text{if } i = j \\ x_j & \text{otherwise} \end{cases}$
$read(v_i)$	$f_n(x)_j = \begin{cases} - & \text{if } i = j \\ x_j & \text{otherwise} \end{cases}$

Table 3.1: Flow functions for CCP.

3.1.1 Exhaustive Analysis

Consider Figure 3.2 and the question as to whether variable b is a copy constant at the write statement at node 7. Standard analysis provides an answer by an exhaustive forward propagation that provides copy constant information for all variables at all nodes in the graph. At each node n the complete solution vector $X(n) = (x_a, x_b, x_c)$ is computed, where x_a, x_b and x_c are the lattice values that have been determined for variables a, b, and c at node n. The solution vectors are computed by propagating a program entry value throughout the program. This propagation involves the repeated application of the flow functions to the current values at each node. The program entry value is simply $X(entry_{main}) = (\bot, \bot.\bot)$ denoting that no variable has a value upon program entry.

Each time a node is visited during the propagation the complete solution vector is evaluated. Note that when propagating in a forward direction, all available information must be collected since prior to reaching node 7 it cannot be determined that information at a predecessor node is not relevant. In particular, when encountering the call to procedure q in node 1 and the call to procedure p in node 5, the two procedures must be fully analyzed in order to ensure that the complete information that may reach node 7 has been collected.

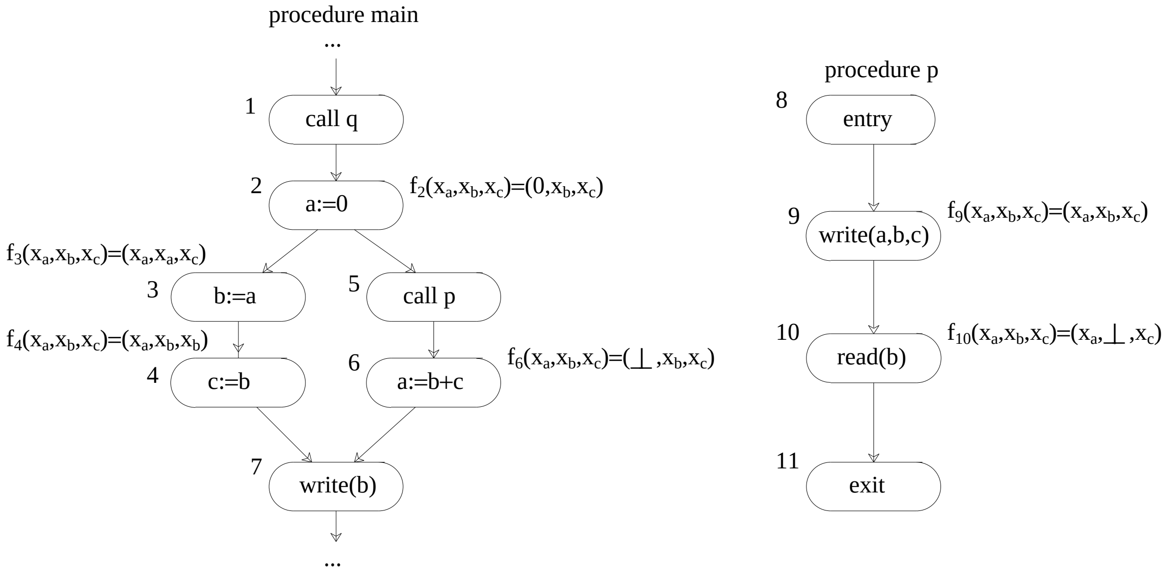

procedure main
...
1
call q
2
a:=0
$f_2(x_a,x_b,x_c)=(0,x_b,x_c)$
$f_3(x_a,x_b,x_c)=(x_a,x_a,x_c)$
3
b:=a
5
call p
$f_4(x_a,x_b,x_c)=(x_a,x_b,x_b)$
4
c:=b
6
a:=b+c
$f_6(x_a,x_b,x_c)=(\bot,x_b,x_c)$
7
write(b)
...
procedure p
8
entry
9
write(a,b,c)
$f_9(x_a,x_b,x_c)=(x_a,x_b,x_c)$
10
read(b)
$f_{10}(x_a,x_b,x_c)=(x_a,\bot,x_c)$
11
exit

Figure 3.2: The ICFG for a sample program.

3.1.2 Demand-Driven Analysis

Now consider how a demand-driven analysis determines whether variable b is a copy constant at node 7. Unlike exhaustive analysis, demand-driven analysis is goal-directed. A solution to the problem is determined by a partial backward search that is started at node 7 and that proceeds in the reverse direction of the exhaustive analysis backwards along each path that leads to node 7. During this backward search only information that is actually relevant for the current problem is collected. The search terminates as soon as a solution has been found. Assume the backward search for the constant value of variable b in Figure 3.2 first proceeds along the left branch in procedure *main* and visits node 4. The computation at node 4 cannot affect the data flow value for variable b. Hence, no information is collected, and the search continues at predecessor node 3. An inspection of node 3 reveals that for variable b to be a copy constant, variable a must be a copy constant at node 3. Hence, the search continues at node 2 with the new problem of determining whether a is a copy constant. Since node 2 contains a constant assignment to variable a, the search terminates successfully with the information that b is a copy constant with the value 0 along the traversed path. Next the remaining path along the right branch in procedure *main* is traversed, starting at node 6. Again, the computation at node 6 cannot affect the value of b, and the search continues at node 5. Since node 5 contains a call to procedure p, the search in *main* is interrupted to collect information about the called procedure p. Next, procedure p is analyzed starting at the exit node 11 and proceeding backwards towards p's entry node. When reaching node 10, it is determined that b cannot be a copy constant at the exit of procedure p. This information is passed back to the call node 5 in *main*. At this point, the overall search terminates with the result that b is not a copy constant at node 7 since a path was encountered along which b is not constant.

The illustration of the demand-driven search procedure shows that there are three ways in which demand-driven analysis avoids unnecessary computations that must be performed in the exhaustive analysis. First, when visiting a node, the information that is not relevant to the current demand does not have to be collected in a demand-driven analysis. Second, nodes that cannot contribute to the demanded solution do not even have to be visited. Third, procedures are analyzed only if needed, that is, only if it has been determined that information from the procedure does affect the demanded solution. For example in Figure 3.2, the demand-driven analysis does not analyze procedure q since the backward search revealed that information from q cannot possibly affect the solution to the problem at node 7.

3.2 The Demand-Driven Analysis Framework

The demand-driven analysis framework developed in this dissertation generalizes and formally defines the backward search from the previous section. The generalized framework is obtained by providing the 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 one component to answer each of these questions:

Component 1: Definition of a data flow query.

Component 2: Set of query propagation rules.

Component 3: Generic iterative query propagation algorithm.

3.2.1 Component 1: Data Flow Queries

Demands for data flow information are modeled by data flow queries. Thus, a data flow query describes the kind of information that can be determined in a demand-driven fashion at a given program node n. The general format of a query is:

$$< y, n >,$$

where y is a lattice element and n is a program node. Query $q =< y, n >$ raises the question as to whether lattice element y is part of the exhaustive solution $X(n)$ at node n, i.e., whether $y \sqsubseteq X(n)$.

Example: Consider again the question as to whether variable b in Figure 3.2 is a copy constant with the constant value 0 at node 7. This question is modeled by the data flow query: $< [b = 0], 7 >$.

3.2.2 Component 2: Query Propagation Rules

The demand-driven analysis describes the search for the solution for a data flow demand that is expressed by a query $q =< y, n >$. This search is fully characterized by a set of *query propagation rules.* Consider again the query $< [b = 0], 7 >$ for the CCP example from Figure 3.2. The query propagation rules describe how the initial query $< [b = 0], 7 >$ is propagated backwards in the program and finally resolved. To specify the query propagation rules in general terms, the second framework component is responsible for the following two functions:

- reversal of the function space and
- procedure summary computation.

The reversal of the function space is needed to generalize and formally define the query propagation through a node. Consider the query $< [b = 0], 3 >$ in the CCP example from Figure 3.2. When propagating the query past node 3, the query changes and the new query raised at the predecessor node 2 is: $< [a = 0], 2 >$. The transformation from the initial query $< [b = 0], 3 >$ to the new query $< [a = 0], 2 >$ simply expresses that for variable b to be a copy constant with value 0 at node 3, variable a has to be a copy constant with value 0 at node 2. To provide a general description of the transformation of a query as it is propagated through the program requires the reversal of the flow functions.

The reversal of the function space provides the means to establish the complete propagation rules for the intraprocedural case. To handle programs with multiple procedures requires additional mechanisms to process procedure calls. Exhaustive analysis, as modeled by the Sharir/Pnueli framework [SP81], analyzes a program with procedure calls through procedure summary computations. The demand-driven framework follows the same approach. However, unlike exhaustive summary computation, the summary computation for the demand-driven analysis is a *partial* and *reverse* summary computation. The reverse procedure summaries express the side effects of procedure calls on the queries that are raised at call sites and are computed only if needed to propagate a query across a call.

Together, the flow function reversal and reverse summary computations enable the specification of a set of query propagation rules that model the complete demand-driven analysis of a program.

3.2.3 Component 3: Generic Analysis Algorithm

The framework contains, as a third component, a generic demand-driven algorithm. A top-level procedure *Query* takes as input a query and returns the answer *true* or *false* to the query. Procedure *Query* implements the query propagation rules based on the reversal of the flow functions at each node. When a procedure call is encountered during the propagation another procedure *Compute*ϕ^r is invoked to provide the reverse procedure summary information for the called procedure. The overall demand-driven algorithm, implemented by the two procedures *Query* and *Compute*ϕ^r, is presented in two versions: a *caching* version, which includes a cache memory for storing intermediate query results to enable fast re-use in future query evaluations, and a *non-caching* version that does not store intermediate results.

In the worst case, in which 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.

3.2.4 Generality

The demand-driven analysis framework is applicable to monotone *inter*procedural data flow problems with finite lattices. If the program under analysis consists of only a single procedure, the analysis algorithm naturally reduces to an *intra*procedural algorithm. For *intra*procedural problems, the finiteness of the lattice is not required and the framework is applicable to all monotone problems. The derived demand-driven algorithms are as precise as their exhaustive counterparts if the data flow problem is distributive. If the problem is monotone but not distributive, precision of the demand-driven algorithms may be lost. The loss of information that is caused by non-distributive flow functions will be examined in detail and it will be shown that the demand-driven algorithms can still be used to provide approximate but safe query responses for non-distributive problems. In addition, a two-phase variation of the framework is presented that is capable of handling non-distributive data flow problems precisely.

The class of distributive and finite data flow problems that can be handled precisely includes, among others, the interprocedural versions of the four classical partitionable bit vector problems: REACH (reaching definitions), AVAIL (available expressions), LIVE (live variables) and BUSY (very busy expressions), as well as to non-partitionable problems, such as copy constant propagation, linear constant propagation [1] or faint variables and to other common interprocedural problems, such as procedure side-effect analysis [CK88].

3.3 Applications

Conceptually demand-driven analysis can be employed to replace traditional exhaustive analysis in any application. However, the efficiency of using demand-driven analysis may vary depending on the nature of the application. Generally, the benefits of using a demand-driven analysis in place of an exhaustive analysis are higher when the demanded fraction of the complete data flow solution is small. While demand-driven analysis algorithms have the same asymptotic worst-case complexities as the corresponding exhaustive analysis algorithms, they may not always be faster in practice. In fact, if an application demands the complete exhaustive solution, demand-driven analysis is likely to perform slower than exhaustive analysis by some constant factor.

The following sections review several applications in compiler optimization and software tools and examine their suitability for using a demand-driven analysis approach.

3.3.1 Compiler Optimizations

The program optimizations that benefit the most from demand-driven analysis are optimizations that utilize data flow information only selectively in a program. There are several situations that give rise to selective data flow utilization in optimization.

Region optimization

Some optimizations are only applicable to certain portions of the program. A common and important example are loop optimizations. Loop optimizations include classical transformations, such as *loop-invariant code motion* [ASU86], as well as parallelizing loop transformations, such as *scalar expansion*[2]. Safely applying a loop optimization requires both the data flow solution that results inside the loop and the data flow from outside that affects the solution at the entry and exit points of the loop. For example, scalar expansion requires information about the liveness of variables at the exit of the loop. Classical algorithms for loop-invariant code motion[ASU86] require information about the liveness of variables referenced by the statements that are to be moved as well as information about variable definitions that reach statements inside the loop from outside.

Even if an optimization is applicable everywhere in a program, one may want to reduce the overall optimization costs by restricting the optimization to the most frequently executed

[1] Linear constant propagation is an extension of copy constant propagation that, in addition to copies, also considers assignments of the form $x := y + c$, where c is a constant.

[2] The purpose of scalar expansion is to create a private copy of a scalar variable accessed in a loop for each loop iteration, thereby removing a dependency among the loop iterations.

portions of the program. For example, *common subexpression elimination* or *code hoisting* may only be applied to the loops in the program, or only to the most frequently called procedures. It is generally possible to optimize a selected code region by performing analysis only over the selected region. However, the worst case assumptions that would have to be made concerning the data flow information that enters the code region from outside could prevent the discovery of otherwise safe optimization opportunities. Demand-driven analysis provides an efficient way for retrieving all relevant information that is external to the selected code region without having to analyze the complete program.

Sparse optimization opportunities

Another source of selective data flow utilization are global optimizations that are likely to be enabled at only a few points in the program. An example of such a global optimization is *copy propagation.* A program is not likely to initially contain a large number of copies; however copy instructions are often created as a result of other transformations, for example, as the result of common subexpression elimination. Instead of performing copy propagation exhaustively and thereby possibly analyzing large portions of the program that do not even contain copies, copies may be propagated on demand directly after their creation. Each time a copy is created as the result of a transformation, demand-driven analysis can be used to retrieve the *available copy* [ASU86] information that is needed to directly propagate the copy.

Sparse data flow usage

Finally, demand-driven analysis is useful in global optimizations that require only a small fraction of the exhaustive solution at each program point. An example is the construction of *def-use chains.* Determining the def-use chains in a program requires the computation of *reaching definitions.* Among other uses, def-use chains are needed to construct *program dependence graphs* [FOW87]. The exhaustive reaching definition solution determines all definitions of all variables that reach each program point. However, the construction of def-use chains requires at each point only the reaching definitions of the variables that are used at that point. Reaching definition information of variables that are not even live at a program point is irrelevant and does neither directly nor indirectly contribute to the construction of def-use chains. The experimental results presented in Chapter 6 demonstrate that demand-driven analysis is more efficient than exhaustive analysis for constructing def-use chains in a program.

With respect to any optimization, demand-driven analysis has the advantage that it bypasses the incremental solution update problem. An exhaustively computed solution becomes invalid after code transformations are applied to the program. Thus, in order to continuously apply optimizations, the exhaustive solution must be updated after each code transformation using incremental data flow techniques[3]. After complex code transformations, the update of the data flow solution may be costly and may even result, in the worst case, in a complete re-computation of the solution.

Although demand-driven analysis appears to be a promising and efficient approach in a large number of optimizations, there are also optimizations that do not favor a demand-driven approach. Whether demand-driven analysis is suitable for an optimization depends primarily upon the way data flow information is required at a node. Recall that data flow queries formulate information requests with true/false answers. This true/false type query format provides a natural formulation for questions asking for membership of a selected candidate in the exhaustive solutions. For example, a query such as: "Is variable v live at this point?". There are optimizations whose information requests are not naturally satisfiable with true/false type queries. Using true/false type queries in these optimizations may be inefficient and may result in querying the complete exhaustive solution at a node.

For example consider the question: "What are the aliases of variable v at node n?". If no additional information is available every variable may be a potential alias of v. True/false type queries retrieve alias information for requests of the form "Are v and w aliases at node n?". Thus, using true/false type queries may require one query for each potential alias of v to find all of v's actual aliases. Another example is the data flow problem of *partial redundancy elimination* (PRE) [MR79, KRS92]. A part of PRE is the optimal placement of each expression. The optimal placement of an expression is determined as the *earliest* placement that is *safe* for the expression [KRS92]. Prior to solving the data flow problem any program point may be a candidate for optimal placement. Thus, the points at which data flow information is actually needed is not fixed prior to the analysis. The use of demand-driven analysis would require the issuing of queries exhaustively at every point in order to find the optimal placement.

In general, if an application, like the alias problem or PRE, requires an exhaustive number of queries to be raised, an exhaustive algorithm is likely to be more efficient in practice.

3.3.2 Software Tools

Data flow analysis is often used to improve the capabilities and performance of software tools. Examples of data flow based software tools are editors [RTD83], debuggers [Wei84] and tools for software testing [FW88, DGS92b]. Furthermore, data flow analysis has been proven especially useful in tools for the software maintenance stage [GS92, GHS92]. Demand-driven analysis is a suitable approach to improve the performance of software tools for several reasons:

[3] See also the discussion on incremental data flow analysis in Chapter 1.

Service user requests on-line

Software tools are often interactive. In an interactive tool, the user issues specific information requests with respect to one or more selected program points rather than inquiring information at all points. For example, when debugging, a user may want to know what data values reach a use of a variable at a certain program point or what statements impact 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. Unlike an exhaustive analysis approach, a demand-driven approach enables control over the analysis effort through the amount of information actually requested by the user.

Avoiding incremental updates

As in compiler optimizations, using demand-driven analysis in software tools bypasses the incremental update problem. Many tools are used while the program is under development and thus changes in the program are expected and must be efficiently handled. Using an exhaustive analysis approach either requires costly re-computations of the exhaustive solution each time a change is made to the program or it requires the storage and maintenance of the exhaustive solution throughout the program development. Maintaining the exhaustive solution throughout the program development may be costly in that the solution must be updated in response to each program change using incremental data flow analysis techniques [Ros81, Ryd83, RC87, Bur87, PS89]. In contrast, using a demand-driven analysis requires neither storage of exhaustive information nor does it require solution updates. Instead, each time data flow information is needed in the tool, the information is computed in a demand-driven mode based on the latest version of the program.

3.4 Parallelizing Demand-Driven Data Flow Analyses

In order to reduce the overhead of performing data flow analysis, research has been directed towards the parallelization of the data flow analysis algorithms. Several parallel versions of exhaustive analysis algorithms have been developed. Typically, the parallelization of exhaustive analysis algorithms requires a separate preparatory phase in order to discover and exploit the parallelism available in the data flow computation [GZZ89, GPS90, LMR91, KGS94]. Based on the discovered parallelism, parallel versions of data flow algorithms are obtained by decomposing the data flow problem into a series of subproblems which can be solved in parallel.

In contrast to previous work, the query propagation algorithm for demand-driven analysis is naturally parallelizable and does not require a separate phase to reveal the available parallelism. The individual queries in a program can be analyzed independently and in parallel. The parallelization simply uses a dispatcher process to distribute the queries among the participating processors. The processors then analyze their assigned queries in parallel. Unlike previous work in analysis parallelization, the degree of parallelism in demand-driven analysis does not depend on the program structure but solely on the size of the program and the number of queries.

Chapter 4

A Framework for Demand-Driven Data Flow

This chapter presents a general lattice based framework for demand-driven interprocedural data flow analysis and is organized as follows. The demand-driven framework and its components that model the query propagation through analysis reversal are presented in Section 4.1. The framework components are first described for programs with parameterless procedures. Framework extensions to handle procedures with parameters including the handling of aliasing introduced by reference parameters are discussed in Section 4.2. Section 4.3 presents several strategies to parallelize the demand-driven analysis. The framework provides a precise interprocedural analysis algorithm for data flow problems that are distributive. Section 4.4 discusses how to effectively handle non-distributive data flow problems. This chapter concludes with a discussion of related work in Section 4.5.

4.1 Framework Components

A general framework for demand-driven data flow analysis is obtained by formulating the analysis problem as a problem of resolving *data flow queries* with respect to the exhaustive solution. The solution equation system according to the Sharir-Pnueli exhaustive analysis framework[1] is restated in Figure 4.1.

A data flow query q raises the question as to whether a specific set of data flow facts $y \in L$ is a safe approximation of the exhaustive solution at a selected program node n. A lattice element y is a safe approximation of the solution $X(n)$ if y is lower in the lattice than $X(n)$.

Definition 4.1 (Data flow query) *Let $y \in L$ and $n \in N$. A* **data flow query** *q is specified by a pair $q =< y, n >$ and denotes the truth value of the term: $y \sqsubseteq X(n)$.*

[1] See also Section 2.2.2.

Summary function equations:

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

$$\phi_{(entry_p,n)}(x) = \bigsqcap_{m\in pred(n)} \begin{cases} f_m \cdot \phi_{(entry_p,m)}(x) & \text{if } m \text{ is not a call site} \\ \phi_{(entry_q,exit_q)} \cdot \phi_{(entry_p,m)}(x) & \text{if } call(m){=}q \end{cases} \tag{4.1}$$

Solution equations:

$$X(entry_{main}) \quad = \quad \bot$$

For each procedure p:

$$X(entry_p) \quad = \quad \bigsqcap_{call(m)=p} X(m) \tag{4.2}$$

For non-entry nodes n:

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

Figure 4.1: Equation systems in the exhaustive Sharir-Pnueli framework.

Example: Recall the informal discussion of data flow queries in Chapter 3 that considered the question as to whether a variable b is a copy constant at a particular node n. The least lattice element that expresses that b has some arbitrary but fixed constant value c is the element $[b{=}c] = (x_a{=}\bot,\ x_b = c,\ x_c{=}\bot)$ (i.e., variables a and c may assume any value). Thus, the question corresponds to the query $q = <[b{=}c], n>$.

Consider now the problem of determining the answer (*true* or *false*) for a query q without having to exhaustively evaluate the equation systems 4.1 and 4.2 from Figure 4.1. Informally, the answer to $q{=}<y, n>$ 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 $q = < y, n >$:

- $q = < y, entry_{main} >$ (program entry node): No further propagation of query q is possible. Since $X(entry_{main}) = \bot$ by definition, it follows that q evaluates to *true* if $y = \bot$ and to *false* otherwise.

- $q = < y, entry_p >$ for some procedure p (procedure entry node): Query q raises the question as to whether y holds on entry of every invocation of procedure p. It follows that q can be translated into the boolean conjunction of queries $< y, m >$ for every call site m calling procedure p.

- $q = < y, n >$, where node n is some arbitrary non-entry node: For simplicity, assume first that n has a single predecessor m. Equation system 4.2 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 if m represents a call site. In either case h is monotone, so 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 none of these two special cases apply, the query q translates into a new query $q' = < z, m >$ for node m. The lattice element z to be queried on entry of 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 [HL92].

Definition 4.2 (Reverse function) *Given a complete lattice L and a monotone function $h : L \mapsto L$. The* **reverse function** *$h^r : L \mapsto L$ is defined as:*

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

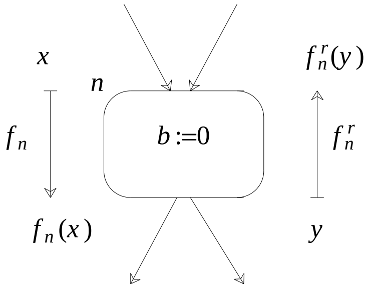


Figure 4.2: A node flow function and its reverse function at a node n

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).

Example: The mappings of a flow function and its reverse function are illustrated in Figure 4.2. Consider the example of CCP and assume that y in Figure 4.2 denotes the lattice element $[a = 0, b = 0]$ expressing that variables a and b have value 0. The reverse function value $f_n^r([a = 0, b = 0])$ describes the least lattice element that has to hold on entry of node n for a and b to have value 0 on exit of node n. There are several entry lattice elements that are sufficient to establish the element $[a = 0, b = 0]$ on node exit. For example, the two values $[a = 0]$ and $[a = 0, b = 0, c = 0, d = 0]$ since $[a = 0, b = 0] \sqsubseteq f([a = 0])$ and $[a = 0, b = 0] \sqsubseteq f([a = 0, b = 0, c = 0, d = 0])$. The reverse function maps element $[a = 0, b = 0]$ to the least sufficient entry element. Thus, $f_n^r([a = 0, b = 0]) = [a = 0]$.

If the function h is $\sqcap$-distributive then the following relationship holds between the function h and its reverse h^r [Cou81, HL92]:

$$y \sqsubseteq h(x) \iff h^r(y) \sqsubseteq x \tag{GC1}$$

The relationship (GC1) uniquely determines the reverse function and defines a *semi-dual Galois connection* [Bir84] between h and its reverse h^r. The relationship GC1 can equivalently be expressed by the following two inequalities:

$$h^r \cdot h(x) \sqsubseteq x \quad \text{and} \quad h \cdot h^r(x) \sqsupseteq x \tag{GC2}$$

Note that the $\sqcap$-distributivity is necessary for establishing this relationship. Thus, unless otherwise stated, it is assumed that flow functions are $\sqcap$-distributive. Next, the relationship between a flow function and its reverse function will be examined to determine how it can be exploited during the query propagation.

First, consider the following properties of the function reversal. It can be easily shown that the $\sqcap$-distributivity of h implies the $\sqcup$-distributivity of the reverse function h^r:

$$h^r(x \sqcup y) = h^r(x) \sqcup h^r(y)$$

Furthermore, the reverse functions are *strict*:

$$h^r(\bot) = \bot$$

The following lemma states the relevant properties with respect to the composition, the meet and the join of functions.

Lemma 4.1 ([HL92]) *Let g and h be two $\sqcap$-distributive functions.*
(i) $(g \cdot h)^r \;=\; h^r \cdot g^r$
(ii) $(g \sqcap h)^r \;=\; g^r \sqcup h^r$

Proof: (i) By (GC1) the following two equivalences hold:

$$g(x) \sqsupseteq y \Longleftrightarrow g^r(y) \sqsubseteq x \text{ and } h(x) \sqsupseteq y \Longleftrightarrow h^r(y) \sqsubseteq x.$$

Substituting $h(x)$ for x in the first equivalence and substituting $g^r(y)$ for y in the second equivalence yields:

$$g(h(x)) \sqsupseteq y \Longleftrightarrow g^r(y) \sqsubseteq h(x) \text{ and } h(x) \sqsupseteq g^r(y) \Longleftrightarrow h^r(g^r(y)) \sqsubseteq x.$$

Combining the two equivalences yields therefore:

$$g(h(x)) \sqsupseteq y \Longleftrightarrow h^r(g^r(y)) \sqsubseteq x,$$

which shows by (GC1) that $(h^r \cdot g^r)$ is the reverse function of $(g \cdot h)$.

Consider now property (ii) and assume $(g \sqcap h)(x) \sqsupseteq y$. By the distributivity of the meet $\sqcap$ we obtain: $g(x) \sqsupseteq y$ and $h(x) \sqsupseteq y$.
Hence, applying (GC1) yields: $g^r(y) \sqsubseteq x$ and $h^r(y) \sqsubseteq x$.
Therefore also: $(g^r \sqcup h^r)(y) = g^r(y) \sqcup h^r(y) \sqsubseteq x$
which shows that $(g^r \sqcup h^r)$ is the reverse function of $(g \sqcap h)$. □

Example: Table 4.1 shows the definition of the reverse flow functions for CCP. By the $\sqcup$-distributivity of the reverse functions, it is sufficient to define f_n^r for only the base elements in the lattice L^k. 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 value results as $f^r([v_1 = c_1, \ldots, v_l = c_l]) = f^r([v_1 = c_1]) \sqcup \ldots \sqcup f^r([v_l = c_l])$. The reverse function value $f_n^r([v_i{=}c])$ denotes the least lattice element, if one exists, that must hold on entry of node n in order for variable v_i to have the constant value c on exit of n. If $f_n^r([v_i{=}c]) = \bot$, the trivial value $\bot$ is sufficient on entry of node n (i.e., variable v_i always has value c on exit). For example, $f_n^r([v_i{=}c]) = \bot$ if node n contains the assignment $v_i := c$. The value $f_n^r([v_i{=}c]) = \top$ indicates that there exists no entry value which would cause variable v_i to have the value c on exit. The value $f_n^r([v_i{=}c]) = \top$ results, for example, if node n contains a constant assignment to variable v_i but the assigned constant value differs from c.

statement at node n	reverse flow function $f_n^r([v_i = c_1])$
constant assignment $v_j{:=}c_2$	$f_n^r([v_i{=}c_1]) = \begin{cases} - & \textit{if } i{=}j \textit{ and } c_1{=}c_2 \\ \top & \textit{if } i{=}j \textit{ and } c_1{\neq}c_2 \\ [v_i{=}c_1] & \textit{otherwise} \end{cases}$
copy: $v_j{:=}v_l$	$f_n^r([v_i{=}c_1]) = \begin{cases} [v_l{=}c_1] & \textit{if } i{=}j \\ [v_i{=}c_1] & \textit{otherwise} \end{cases}$
expr. assignment $v_j := \mathit{expr.}$	$f_n^r([v_i{=}c_1]) = \begin{cases} \top & \textit{if } i{=}j \\ [v_i{=}c_1] & \textit{otherwise} \end{cases}$
$\mathit{read}(v_j)$	$f_n^r([v_i{=}c_1]) = \begin{cases} \top & \textit{if } i{=}j \\ [v_i{=}c_1] & \textit{otherwise} \end{cases}$

Table 4.1: Reverse flow function for CCP.

The following theorem states the rules that are used to translate and propagate queries by reversing the data flow at each node. The operator $\wedge$ denotes boolean conjunction.

Theorem 4.1 (Query Propagation) *Let $D = (L, F)$ be a $\sqcap$-distributive data flow framework and let $q =< y, n >$ be a data flow query in D. The following equivalences hold for the propagation of query q:*

(i)

$$< \bot, n > \iff \textit{true}$$

$$< \top, n > \iff \textit{false}$$

(ii) *For each procedure p*

$$< y, entry_p > \iff \begin{cases} \textit{false} & \textit{if } p \textit{ has no call sites} \\ \bigwedge_{call(m)=p} < y, m > & \textit{otherwise} \end{cases}$$

(iii) *For a non-entry node n:*

$$< y, n > \iff \bigwedge_{m \in pred(n)} \begin{cases} < f_m^r(y), m > & \textit{if } m \textit{ is not a call site} \\ < \phi^r_{(entry_p, exit_p)}(y), m > & \textit{if } call(m) = p \end{cases}$$

Proof: By definition the query can be rewritten based on the solution definition $X(n)$ from equation system 4.2 as: $< y, n > \iff y \sqsubseteq X(n)$. Thus, rule (i) follows immediately. Consider the remaining two rules:
(ii) If $n = entry_p$ and p has no call sites then rule (ii) follows immediately. Otherwise,

$$\begin{aligned} < y, entry_p > &\iff y \sqsubseteq X(entry_p) \\ &\iff y \sqsubseteq \bigsqcap_{call(m)=p} X(m) \\ &\iff y \sqsubseteq X(m) \text{ for all } m \text{ with } call(m) = p \\ &\iff \bigwedge_{call(m)=p} < y, m >, \end{aligned}$$

which implies rule (ii).
(3) If n is a non-entry node we obtain:

$$< y, n > \iff y \sqsubseteq \bigsqcap_{m \in pred(n)} \begin{cases} f_m(X(m)) & \text{if } m \text{ is not call site} \\ \phi_{(entry_q, exit_q)}(X(m)) & \text{if } call(m) = q \end{cases}$$

$$\iff \begin{cases} y \sqsubseteq f_m(X(m)) \text{ for all } m \in pred(n) \text{ that are not call sites and} \\ y \sqsubseteq \phi_{(entry_q, exit_q)}(X(m)) \text{ for all } m \in pred(n) \text{ with } call(m) = q. \end{cases}$$

By applying condition (GC1) we obtain:

$$\iff \begin{cases} f_m^r(y) \sqsubseteq X(m) \text{ for all } m \in pred(n) \text{ that are not call sites and} \\ \phi^r_{(entry_q, exit_q)}(y) \sqsubseteq X(m) \text{ for all } m \in pred(n) \text{ with } call(m) = q. \end{cases}$$

$$\iff \begin{cases} \bigwedge_{m \in pred(n)} < f_m^r(y), m > \text{ if } m \text{ is not a call site and} \\ \bigwedge_{m \in pred(n)} < \phi^r_{(entry_q, exit_q)}(y), m > \text{ if } call(m) = q. \end{cases}$$

Clearly, it is impossible that $y \sqsubseteq X(n)$ if there exists a predecessor $m \in pred(n)$, such that $f_m^r(y) = \top$ or $\phi^r_{(entry_q, exit_q)}(y) = \top$. Furthermore, the monotonicity of the functions f_m^r and $\phi^r_{(entry_q, exit_q)}$ implies that $y \sqsubseteq X(n)$ if for all $m \in pred(n)$: $f_m^r(y) = \bot$ or $\phi^r_{(entry_q, exit_q)}(y) = \bot$. Thus, rule (iii) follows. □.

The query propagation as described by Theorem 4.1 requires the application of reverse functions. 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 a procedure p the reverse summary function $\phi^r_{(entry_p,exit_p)}$ is determined. The next section presents the query propagation algorithm assuming that all necessary reverse summary functions are available. The determination of reverse summary functions is discussed in Section 4.1.2.

4.1.1 A Query Propagation Algorithm

The demand-driven algorithm that implements the query propagation is shown in Figure 4.3. Procedure *Query* takes as input a query q and returns the answer for q after a finite number of applications of the propagation rules. Procedure *Query* uses a worklist that is initialized with the node n from the input query $q =< y, n >$. A variable $query[n]$ is used at each node n to store the queries raised at n. 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 worklist. During each step a node n is removed from the worklist and the query $< query[n], n >$ 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 worklist unless the newly generated query was already previously raised at node n (lines 9-10 and 17-18). 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 worklist since the overall answer to the input query must also be *false*. Thus, procedure *Query* can terminate early and the remaining contents of the worklist are simply discarded. Otherwise, if no query evaluates to *false*, procedure *Query* terminates with the answer *true* if the worklist 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 counted. 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 $O(height(L))$ 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 (the *intra*procedural case), procedure *Query* provides a complete implementation of the demand-driven data flow analysis. The *inter*procedural case requires an efficient method to compute the reverse summary functions.

4.1.2 Reverse Summary Functions

This section discusses an algorithm to compute individual reverse summary function values in order to extend procedure *Query* to the interprocedural case. A straightforward but inefficient way to compute reverse summary functions is to first determine all original summary functions by evaluating the summary function equation system 4.1 from Figure

```
**Procedure** $Query(y, n)$
**input:**  a lattice element $y \in L$ and a node $n$
**output:** the answer $true$ or $false$ to the query $< y, n >$
**begin:**
**for each** $m \in N$ **do** $query[m] \leftarrow -$
$query[n] \leftarrow y$; $worklist \leftarrow \{n\}$;
**while** $worklist \neq \emptyset$ **do**
    remove a node $m$ from $worklist$;
    **case** $m = entry_{main}$:
        **if** $query[m] \sqsupset -$ **return(**$false$**);**
    **case** $m = entry_q$ for some procedure $q$:
        **for each** call site $m'$ such that $call(m') = q$ **do**
            $query[m'] \leftarrow query[m'] \sqcup query[m]$;
            **if** $query[m']$ changed **then** add $m'$ to $worklist$;
        **endfor**;
    **otherwise**:
        **for each** $m' \in pred(m)$ **do**
            $new \leftarrow \begin{cases} f^r_{m'}(query[m]) & \text{if } m' \text{ is not a call site} \\ \phi^r_{(entry_q, exit_q)}(query[m]) & \text{if } call(m')=q \end{cases}$
            **if** $(new = \top)$ **then return(** $false$ **)**
            **else if** $(new \sqsupset -)$ **then**
                $query[m'] \leftarrow query[m'] \sqcup new$;
                **if** $query[m']$ changed **then** add $m'$ to $worklist$;
            **endif**;
        **endfor**;
**endwhile**;
**return(**$true$**)**;
**end**
```

Figure 4.3: Generic demand-driven analysis procedure.

```
**Procedure** $Compute\phi^r(y, p)$
**input:**  a lattice element $y \in L$ and a procedure $p$
**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**
```

Figure 4.4: Procedure *Compute*ϕ^r to compute reverse summary functions.

4.1 and then reverse each function. This section describes a more efficient algorithm that directly computes the reverse functions. This algorithm mirrors the operations performed in Sharir and Pnueli's worklist algorithm for evaluating equation system 4.1 [SP81], except that summary functions are computed in reverse direction. Assuming that (i) the cost of a meet and a join are same and that (ii) the 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 original summary functions. As in Sharir and Pnueli's algorithm the tabulation strategy requires the lattice to be finite.

First, an inductive definition of the reverse summary functions is derived from equation system 4.1. Reversing the order in which summary functions are constructed and applying Lemma 4.1 yields the following definition of the reverse summary function $\phi^r_{(entry_p,exit_p)}$ for each procedure p:

Reverse summary function equations:

$$\begin{aligned}
&\phi^r_{(exit_p,exit_p)}(y) = y \\
&\phi^r_{(n,exit_p)}(y) = \bigsqcup_{m \in succ(n)} \begin{cases} f^r_m \cdot \phi^r_{(m,exit_p)}(y) & \text{if } m \text{ is not call site} \\ \phi^r_{(entry_q,exit_q)} \cdot \phi^r_{(m,exit_p)}(y) & \text{if } call(m) = q \end{cases}
\end{aligned} \tag{4.3}$$

Figure 4.4 shows an iterative worklist algorithm *Compute*ϕ^r that, if invoked with a pair (p, y), returns the value $\phi^r_{(entry_p,exit_p)}(y)$ after a partial evaluation of the reverse equation system 4.3. Individual function values are stored in a table $M : N \times L \mapsto L$ such that $M[n, y] = \phi^r_{(n,exit_p)}(y)$. The table is initialized with the value $\perp$ and its contents are assumed to be preserved between subsequent calls to procedure *Compute*ϕ^r. Thus, results of previous calls are re-used and the table is incrementally computed during a sequence of calls. After calling *Compute*ϕ^r with a pair (p, y) a worklist is initialized with the pair $(exit_p, y)$. The contents of the worklist indicate the table entries whose values have changed but the new values have not yet been propagated. During each step a pair is removed from the worklist, its new value is determined and all other entries whose values might have changed as a result are added to the worklist.

Consider the cost of k calls to *Compute*ϕ^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 worklist. 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*ϕ^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

```
Procedure $EnterCache(cache, q, val)$
input:  $cache$, query $q =< y, n >$, query result $val \in \{true, false\}$,
        and the set of nodes $S$ at which the propagation terminated.
output: updated $cache$
begin
if $val = false$ ‘bf then
    eliminte all nodes from $S$ at which the propagation terminated with $true$;
endif
for each node $m$ reachable from some node in $S$ do
    if $query[m] \sqsupseteq -$ /* $query[m] \sqsubseteq -$ serves as visited flag */
        enter $cache[m, query[m]] = false$;
    endif
endfor
end
```

Figure 4.5: Procedure *EnterCache* for updating the cache.

calling a single procedure.

Assuming that each access to a reverse summary function in procedure *Query* is replaced by an appropriate call to $Compute\phi^r$, the total cost of procedure *Query* is $O(MaxCall \times height(L) \times |L| \times |N|)$ join and reverse node flow function applications and $O(|N| \times |L|)$ space to store lattice elements.

4.1.3 Caching

This section discusses a variant of procedure *Query* that uses a cache memory to improve the performance of the query evaluation over a sequence of queries. Processing a sequence of k queries requires k separate invocations of procedure *Query*, which may result in the repeated evaluation 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. The entry $cache[n, y]$ contains the previous result, if any, of evaluating the query $< y, n >$. Otherwise, if query $< y, n >$ has not yet been evaluated, the entry $cache[n, y]$ is marked empty.

The query propagation is modified such that each time before a newly generated query q is added to the worklist, the cache is consulted. The query q is added to the worklist only if the answer for q is not found in the cache. Entries are added to the cache after each terminated query evaluation as described in procedure *EnterCache* shown in Figure 4.5.

First, consider the case that the query evaluation terminates early with a *false* answer. Let n be the node at which the propagation terminates with *false*. Recall, that during the propagation a query is translated into an equivalent conjunction of queries at predecessor nodes. Hence, a *false* evaluation for the query at node n implies a *false* value for the queries that were generated at nodes that are reachable from n, Thus, the cache entry $cache[m, query[m]] = \mathit{false}$ is added (line 4) at all reachable nodes m. Note that at this point the entries $cache[m, z]$ for all elements z such that $z \sqsupseteq query[m]$, could also be set to *false*. The query propagation may have traversed additional paths that ended in a node with a *true* answer. However, since the query propagation terminated early with a *false* answer it cannot be safely assumed that *true* is indeed the final answer along the true terminating paths. Thus, no cache entries other than the *false* entries are added to the cache.

Now consider the case that the query evaluation completes with an exhausted worklist and a *true* answer along each traversed path. The fact that all traversed paths lead to a *true* evaluation implies that the query value at each visited node must be *true*. Thus, the cache is updated at every visited node m such that: $cache[m, query[m]] = true$ (line 6). In addition, all entries $cache[n, z]$ for $z \sqsubseteq query[n]$ can also be set to *true*.

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 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 worst case complexity of k invocations of *Query* with result caching is the same for any number k, where $1 \leq k \leq |L| \times |N|$ and $|L| \times |N|$ is the number of distinct queries.

4.2 Procedures with Parameters

The demand-driven analysis concepts developed in the previous sections have not considered parameter binding mechanisms at call sites. When executing a procedure call, the value of each actual parameter in the address space of the calling procedure is bound to a corresponding formal parameter in the address space of the called procedure. To handle procedures with parameters, these bindings among variable values must also be reflected during data flow analysis. In the same way as values of actual parameters are bound to formal procedure parameters during execution, the lattice elements that refer to actual parameters are bound to the corresponding lattice elements for formal parameters during data flow analysis. This section shows how to refine Theorem 4.1 and the equation system 4.3 for computing reverse summary functions to correctly account for parameter bindings at procedure calls.

4.2.1 Binding Functions

As programs with global and local scoping are considered, the address space $Addr(p)$ of a procedure p is defined as:

$$Addr(p) = Global \cup Formal(p) \cup Local(p),$$

where $Global$ is the set of global variables, $Formal(p)$ is the set of formal parameters of procedure p and $Local(p)$ is the set of variables local to p. The set $Formal(p)$ of formal parameters is further divided into a set $Formal_{inout}(p)$ of reference parameters and a set $Formal_{in}(p)$ of value parameters, i.e.:

$$Formal(p) = Formal_{inout}(p) \cup Formal_{in}(p)$$

Binding functions are defined to model how the values of variables in the address space of a calling procedure are bound to variables in the address space of a called procedure. A binding function b_s is defined for each call site s to map the value of a variable v from the calling procedure to the set of variables $b_s(v)$ in the called procedure to which the value of v is bound at s. To analyze the data flow when control returns to a calling procedure, it will also be necessary to consider the reverse binding b_s^{-1} that binds a value from the called procedure to the corresponding variable in the calling procedure.

Definition 4.3 (Binding functions) *Let $s \in call(q)$ be a call site in a procedure p that passes the actual parameters $(ap_1, \ldots ap_j)$ to the formal parameters $(fp_1, \ldots fp_j)$ in procedure q. Let $v \in Addr(p)$ and let $w \in Addr(q)$. The* **binding function** *b_s for call site s is defined as:*

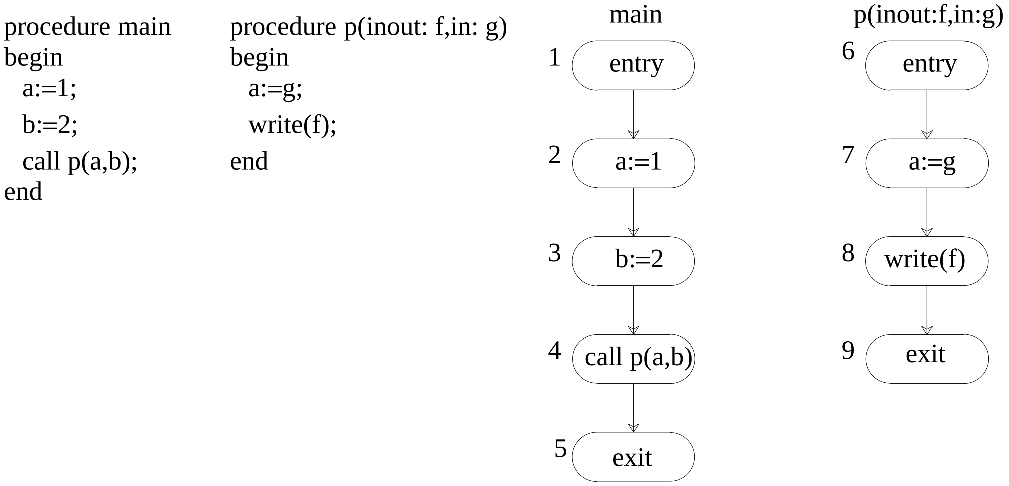
$$b_s(\{v\}) = (\{v\} \cap Global) \cup \begin{cases} \{fp_i\} & \text{if } v = ap_i \\ \emptyset & \text{otherwise} \end{cases}$$

The **reverse binding function** *b_s^{-1} for s is defined as:*

$$b_s^{-1}(\{w\}) = (\{w\} \cap Global) \cup \begin{cases} \{ap_i\} & \text{if } w = fp_i \text{ and } fp_i \in Formal_{inout}(q) \\ \emptyset & \text{otherwise} \end{cases}$$

The bindings for a set of variables is computed as $b_s(V) = \bigcup_{v \in V} b_s(\{v\})$ and analogously $b_s^{-1}(V) = \bigcup_{v \in V} b_s^{-1}(\{v\})$. Note that the reverse binding function b_s^{-1} only binds the values of global variables and reference parameters to variables in the calling procedures. The values of local variables or value parameters are no longer accessible after control has returned to the calling procedure.

Example: Consider the program in Figure 4.6. Variables a and b are global, and procedure p has one reference parameter f and one value parameter g. The bindings of variables at



```
procedure main          procedure p(inout: f,in: g)
begin                   begin
  a:=1;                   a:=g;
  b:=2;                   write(f);
  call p(a,b);          end
end
```

Figure 4.6: Program with reference and value parameter passing and its ICFG.

the call site at node 4 in procedure *main* are: $b_4(\{a\}) = \{a, f\}$ and $b_4(\{b\}) = \{b, g\}$. The reverse bindings are: $b_4^{-1}(\{f\}) = b_4^{-1}(\{a\}) = \{a\}$, $b_s^{-1}(\{b\}) = \{b\}$ and $b_4^{-1}(\{g\}) = \emptyset$.

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 natural counterparts of b_s and b_s^{-1} that apply to the lattice elements in the data flow problem. Hence, function $\tilde{b}_s$ maps a lattice element from the calling procedure to a corresponding lattice element in the called procedure according to the value binding described by function b_s. Analogously, function $\tilde{b}_s^{-1}$ maps lattice elements from the called procedure to the corresponding lattice element in the calling procedure. For a node n that is not a call site the binding functions $b_n, b_n^{-1}, \tilde{b}_n$ and $\tilde{b}_n^{-1}$ are simply the identity function.

Example: Consider the binding functions in CCP. For each lattice base element of the form $[v_i = c]$ the two functions are defined as:

$$\begin{aligned}
\tilde{b}_s([v_i{=}c]) &= \bigsqcup_{v_j \in b_s(\{v_i\})} [v_j{=}c] \\
\tilde{b}_s^{-1}([v_i = c]) &= \begin{cases} [v_j = c] & \text{if } \{v_j\} = b_s^{-1}(\{v_i\}) \\ \perp & \text{otherwise} \end{cases}
\end{aligned}$$

Based on the binding functions $\tilde{b}_s$ and $\tilde{b}_s^{-1}$, the demand-driven analysis framework can be refined to handle programs with reference and value parameters. Refining the framework

Refined reverse summary equations using binding funcions:

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

$$\phi^r_{(n,exit_p)}(y)= \bigsqcup_{m\in succ(n)} \begin{cases} f^r_m \cdot \phi^r_{(m,exit_p)}(y) & \text{if } m \text{ is not a call site} \\ (\tilde{b}_m^{-1} \cdot \phi^r_{(entry_q,exit_q)} \cdot \tilde{b}_m) \cdot \phi^r_{(m,exit_p)}(y) & \text{if } call(m) = q \end{cases}$$

(i)

Refined propagation rules:

$$\begin{array}{lrcl} (i) & < \bot, n > & \Longleftrightarrow & true \\ & < \top, n > & \Longleftrightarrow & false \end{array}$$

$$(ii) \quad < y, entry_p > \;\Longleftrightarrow\; \begin{cases} false & \text{if } p \text{ has no call sites or } \tilde{b}_m^{-1}(y) = \emptyset \\ \bigwedge_{call(m)=p} < \tilde{b}_m^{-1}(y), m > & \text{otherwise} \end{cases}$$

$$(iii) \quad < y, n > \;\Longleftrightarrow\; \bigwedge_{m\in pred(n)} \begin{cases} < f^r_m(y), m > & \text{if } m \text{ is not a call site n} \\ < (\tilde{b}_m^{-1} \cdot \phi^r_{(entry_p,exit_p)} \cdot \tilde{b}_m)(y), m > & \text{if } call(m) = q \end{cases}$$

(ii)

Figure 4.7: Analysis refinements for reference and value parameter passing.

requires refinement of the query propagation rules from Theorem 4.1 and refinement of the equation system 4.3 for the reverse summary functions by appropriately incorporating flow binding functions when propagating data flow information between procedures. The resulting refined equations are shown in Figure 4.7.

4.2.2 Aliasing

The previous section discussed the necessity for reflecting the bindings among variables that result from parameter passing during the analysis. The presence of reference parameters causes an additional complication for data flow analysis by introducing the potential of aliasing.

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. Reference parameters may introduce aliases through the binding mechanisms between actual and formal parameters. There are two ways in which an alias pair may be created by reference parameters. First, if a global variable x is passed to a formal parameter f then the alias pair (x, f) is created in the called procedure. Second, 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.

Example: In Figure 4.6, the call at node 4 in procedure *main* passes the global variable a to the reference parameter f creating the alias pair (a, f) in procedure p.

Ignoring the potential of aliasing may lead to unsafe query responses. Consider again Figure 4.6 and the example of CCP. If it is not known that (f, a) is an alias pair in procedure p then the re-definition of the value of f at node 7 through the alias a will be missed. As a consequence, variable f would be incorrectly reported to still have the value 1 at the write statement in node 8 although the value of f at node 8 is actually 2 through the assignment of the alias a at node 7.

This section discusses how separately computed information about the potential aliases in a program is used to refine the query propagation and ensure safe query responses. Alias information is typically computed in form of the two summary relations $\mathit{MayAlias}(p)$ and $\mathit{MustAlias}(p)$ for each procedure p [Coo85]. A pair (x, y) is contained in $\mathit{MayAlias}(p)$ if x is aliased to y in some invocation of p. A pair (x, y) is in $\mathit{MustAlias}(p)$ if x is aliased to y in all invocations of p. The sets $\mathit{MayAlias}(x, p) = \{y \mid (x, y) \in \mathit{MayAlias}(p)\}$ and $\mathit{MustAlias}(x, p) = \{y \mid (x, y) \in \mathit{MustAlias}(p)\}$ denote the sets of *may* aliases and *must* aliases of variable x, respectively. Furthermore, for a node n contained in a procedure p: $\mathit{MayAlias}(x, n) = \mathit{MayAlias}(x, p)$ and $\mathit{MustAlias}(x, n) = \mathit{MustAlias}(x, p)$.

The precise determination of alias sets is an *NP*-complete problem [Mye81]. Therefore, alias sets are necessarily approximative in practice. The most conservative but safe approximation of the two alias sets are as follows.

$$\begin{aligned} MayAlias(x,p) &= \{x\} \cup Global \cup Formal(p) \\ MustAlias(x,p) &= \{x\} \end{aligned}$$

More precise estimates of the actual alias relations in a program are determined through additional analysis. The computation of alias relations induced by reference parameters can be modeled as a data flow problem over a program's call graph [Coo85]. Moreover, the data flow problem to compute $MayAlias(p)$ 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 the query propagation. Alternatively, an exhaustive algorithm for computing the alias sets that iterates over the program's call graph may be used to compute the potential alias pairs for all procedures prior to the analysis [Coo85]. The analysis refinements presented in this section assume that the sets $MayAlias(p)$ and $MustAlias(p)$ are available without making any assumptions on their accuracy or on the method used to compute them.

Consider now how the alias information is used to refine CCP analysis. A variable x is considered a constant at a 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 killed at a node n if x or any of x's *may* aliases is assigned a non-constant expression. Table 4.2 and Table 4.3 display the refined flow functions and the refined reverse flow functions for CCP.

Example: Consider again Figure 4.6 and the flow function for node 7 in procedure p. The refined function results as: $f_7(x)_a = f_7(x)_f = x_g = 2$ since f is a must alias of a. The reverse function at node 7 is defined as: $f_7^r([f = c]) = f_7^r([a = c]) = [g = c]$.

The analysis refinements described in this section 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 statement $a := \&b$ in a C program creates the alias pair $(*a, b)$. Several techniques have been developed to approximate alias information in programs with pointer variables [LH88, CWZ90, LR92, CBC93, Deu94, EGH94, WL95]. The results of these alias analyses can be used to establish the two relations *MayAlias* and *MustAlias* and enable the refinements described in this section.

4.3 Parallelizing Demand-Driven Data Flow Analyses

As an additional benefit, the demand-driven analysis concepts developed in this thesis provide a novel approach to the parallelization of data flow analysis. Unlike standard exhaustive data flow analysis algorithms, the query propagation algorithm for demand-driven analysis is naturally parallelizable. Since individual queries are propagated through

statement at node n	refined flow function $f_n(x)_j$, where $x = (x_1, \ldots, x_k)$
$v_i := c$	$f_n(x)_j = \begin{cases} c & \text{if } v_j \in \mathit{MustAlias}(v_i, n) \\ x_j \sqcap c & \text{if } v_j \in (\mathit{MayAlias}(v_i, n) - \mathit{MustAlias}(v_i, n)) \\ x_j & \text{otherwise} \end{cases}$
$v_i := v_l$	$f_n(x)_j = \begin{cases} x_l & \text{if } v_j \in \mathit{MustAlias}(v_i, n) \\ x_j \sqcap x_l & \text{if } v_j \in (\mathit{MayAlias}(v_i, n) - \mathit{MustAlias}(v_i, n)) \\ x_j & \text{otherwise} \end{cases}$
$\mathit{read}(v_i)$ or $v_i := \mathit{expr}.$	$f_n(x)_j = \begin{cases} - & \text{if } v_j \in \mathit{MayAlias}(v_i, n) \\ x_j & \text{otherwise} \end{cases}$

Table 4.2: Refined flow functions for CCP.

statement at node n	refined reverse flow function f_n^r
$v_j := c_2$	$f_n^r([v_i{=}c_1]) = \begin{cases} - & \text{if } v_i \in \mathit{MustAlias}(v_j, n) \text{ and } c_1 = c_2 \\ \top & \text{if } v_i \in \mathit{MayAlias}(v_j, n) \text{ and } c_1 \neq c_2 \\ [v_i{=}c_1] & \mathit{otherwise} \end{cases}$
$v_j := v_l$	$f_n^r([v_i{=}c_1]) = \begin{cases} [v_l{=}c_1] & \mathit{if}\ v_i \in \mathit{MustAlias}(v_j, n) \\ [v_i{=}c_1,\ v_l{=}c_1] & \text{if } v_i \in (\mathit{MayAlias}(v_j, n) - \mathit{MustAlias}(v_j, n)) \\ [v_i{=}c_1] & \mathit{otherwise} \end{cases}$
$\mathit{read}(v_j)$ or $v_j := \mathit{expr}.$	$f_n^r([v_i{=}c_1]) = \begin{cases} \top & \mathit{if}\ v_i \in \mathit{MayAlias}(v_j, n) \\ [v_i{=}c_1] & \mathit{otherwise} \end{cases}$

Table 4.3: Refined reverse flow functions for CCP.

the program independently, the propagation of several queries can be performed in parallel. Thus, a parallelization of the demand-driven analysis algorithm results naturally by simply distributing the set of generated queries among the available processors.

Several parallel versions of the demand-driven algorithm can be implemented based on different degrees of information sharing among the participating processors. This section considers three different parallelization strategies.

- **No Cache/Private Cache Model**

In the first strategy the participating processors operate in isolation either using a private cache or using no cache. This parallelization strategy is straightforward and requires no communication among the processors. However, since the participating processors operate in isolation no information can be shared. Hence, the same intermediate query results may be computed by several processors.

Consider the estimated parallel analysis time if no cache is used based on p processors and a program with q queries assuming that each processor is assigned $\lceil \frac{q}{p} \rceil$ queries. Let $av(T_d^1)$ be the average demand-driven single-query analysis time over all queries in a program using no cache. The average parallel analysis time $P_{av}(p,q)$ without caching for q queries using p processes results as:

$$P_{av}(p,q) = \frac{q \times av(T_d^1)}{p}.$$

Note, that P_{av} is an optimistic estimate that assumes perfect load balancing among the processors. To determine a pessimistic worst case estimate of the parallel analysis time, let $max(T_d^1)$ be the maximal demand-driven single-query analysis time over all queries in the program. The maximal parallel analysis time $P_{max}(p,q)$ without caching for q queries and p processes results as:

$$P_{max}(p,q) = \frac{q \times max(T_d^1)}{p}.$$

P_{max} is a worst case estimate that can only result if a single processor is assigned only queries that require the maximum single-query analysis time.

Let T_{seq} be the analysis time for the program if a sequential algorithm is used. The average speedup $S_{av}(p,q)$ of the parallel execution over the sequential analysis is given by:

$$S_{av}(p,q) = \frac{T_{seq}}{P_{av}(p,q)}.$$

The corresponding guaranteed least speedup $S_{min}(p,q)$ that results is given by:

$$S_{min}(p,q) = \frac{T_{seq}}{P_{max}(p,q)}.$$

Note that for a fixed number of queries the speedup grows continuously with the number of processors and the number of queries.

- **Shared Cache Model**

The second parallelization strategy avoids duplication of computed results through the use of a shared cache. The participating processors communicate through the shared cache and cooperatively process the complete query sequence. The use of a shared cache is transparent in each processor (except for possible access delays). Importantly, the cache management is particularly simple. The analysis results determined by different processors for the same cache entry must be identical since all processors analyze the same program. Thus, there cannot be contention for read and/or write accesses to the cache.

The average parallel execution time that results if a shared cache is used can be estimated as:

$$P_{av}(p, q) = \frac{q \times av(T^1_{d/c})}{p}$$

where $T^1_{d/c}$ is the single-query analysis time based on using a cache. Note that the average single-query analysis time over a sequence of queries, if caching is used, may vary with the length of the sequence. A preliminary experimental examination of the average single-query analysis time for query sequences of different lengths revealed no clear correlation between sequence length and average single-query analysis time.

- **Hybrid Cache Model**

Finally, there is also the possibility of a hybrid cache model, where in addition to a shared cache, also a private cache is maintained in each processor. Access to the shared cache is only necessary as a result of a cache miss in the private cache. Again, the cache management is simple since the values in each private cache and the shared cache must be identical if they are present in both caches.

4.4 Non-Distributive Frameworks

The demand-driven framework assumes that data flow problems are distributive. The distributivity of the flow functions in the problem is necessary to ensure that the query propagation rules from Theorem 4.1 yield as precise information as the original exhaustive analysis does. This section considers demand-driven analysis for data flow problems with non-distributive flow functions. First, the developed framework is shown to be applicable to non-distributive data flow problems but at the cost of reduced precision. Section 4.3.2 discusses a two-phase framework variation that provides precise query responses for non-distributive problems.

4.4.1 Approximate Demand-Driven Analysis

If applied to distributive data flow problems, the query propagation rules from Theorem 4.1 are precise and decidable. Given a data flow query $q =< y, n >$, query q evaluates

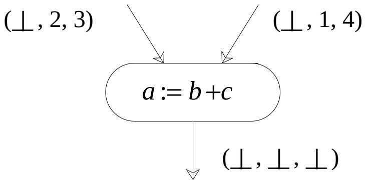


Figure 4.8: Expression node in constant propagation.

to *true* or *false* after a finite number of applications and q evaluates to *true* if and only if element y is part of the solution at node n, i.e., if and only if $y \sqsubseteq X(n)$. If the original analysis framework is monotone but not distributive, information may be lost during the query propagation. Specifically, the information loss occurs during the reversal of non-distributive flow functions. Recall the relationship (GC1) between a distributive function f and its reverse function f^r:

$$y \sqsubseteq f(x) \iff f^r(y) \sqsubseteq x. \tag{GC1}$$

If the function f is monotone but not distributive, then the relation between f and its reverse f^r is weaker than in the distributive case; only the following implication holds:

$$y \sqsubseteq f(x) \implies f^r(y) \sqsubseteq x.$$

As a result of this weaker relationship the query propagation rules no longer provide equivalent translations. Based on the weaker set of propagation rules that results in the presence of non-distributive functions h, queries are only semi-decidable. If a query $q =< y, n >$ evaluates to *false* then $y \not\sqsubseteq X(n)$. However, nothing can be said if q evaluates to *true*.

If appropriate worst case assumptions are made for *true* responses, the query algorithm can still be used to provide approximate information in the presence of non-distributive flow functions.

Example: To illustrate the loss of precision that results from non-distributive flow functions consider the example of constant propagation. Unlike copy constant propagation, regular constant propagation includes the evaluation of arithmetic expressions. Consider the flow function for the assignment statement shown in Figure 4.8. Assume there are only three variables in the program, such that 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 constant propagation associated with the assignment is of the form:

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

Note that f_{cp} is not distributive. To illustrate the non-distributivity consider the situation depicted in Figure 4.8, where the element $(\perp, 2, 3)$ is propagated to the node along the left incoming branch and the 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.

Now consider the reverse function f_{cp}^r if applied to the lattice element $[a = 5]$ denoting that a has the constant value 5. By definition $f_{cp}^r([a = 5])$ is the meet over all elements (x_a, x_b, x_c) such that $f_{cp}^r(x_a, x_b, x_c) \sqsubseteq [a = 5]$. There are infinitely many values for variables b and c such that the execution of the assignment $a := b + c$ yields the constant value 5. Since the meet over an infinite set is by definition $\perp$, it follows that $f_{cp}^r([a = 5]) = (\perp, \perp, \perp)$.

4.4.2 Framework Variation

This section presents a framework variation that enables precise query evaluation even in the presence of non-distributive functions. Theoretically, non-distributivity in the function space could be handled by making the analysis distributive as described in [CC79]. At additional analysis cost, a non-distributive framework can be transformed into a distributive one by operating on sets of lattice elements as opposed to operating on individual lattice elements. In this formulation the meet operator corresponds to a union of lattice elements rather than some form of merge operation. The union operation ensures that no information is lost and that the flow functions are distributive. In the example from Figure 4.8 the meet of the two lattice elements $(\perp, 2, 3)$ and $(\perp, 1, 4)$ would be the collection $\{(\perp, 2, 3), (\perp, 1, 4)\}$. If the flow function f_{cp} for the assignment statement is applied to this collection, the fact that a is a constant after the assignment will be recognized: $f_{cp}(\{(\perp, 2, 3), (\perp, 1, 4)\}) = \{(5, 2, 3), (5, 1, 4)\}$. However, the expansion of the original lattice into a powerset structure can quickly result in an exponential explosion during the analysis, rendering this approach too costly to be of practical use.

An alternative more practical strategy to deal with non-distributivity results by slightly departing from concepts of precise analysis reversal. The loss of information as a result of function reversal can always be recognized. The first time information is lost during the propagation of a query $q =< y, n >$ through a node n happens only if the relationship (GC2) that defines a semi-dual Galois connection between f_n and its reverse f_n^r is violated such that:

$$f_n \cdot f_n^r(y) \not\sqsubseteq y \tag{4.4}$$

Consider for example the constant propagation function f_{cp} for the assignment statement from Figure 4.8. For the element $y = [a = c]$: $f_{cp}(f_{cp}^r([a = c])) = f_{cp}((\bot, \bot, \bot)) = (\bot, \bot, \bot)$, which violates (GC2).

If (GC2) is violated at a node n, information has been lost and the query $q =< y, n >$ cannot safely be propagated across the node. Specifically, 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 at preceding nodes that would be sufficient to provide an answer for q. Guessing sufficient queries differs from translating a query into equivalent queries. The guessed queries are sufficient if, assuming an answer for the guessed queries was available, an answer for q could be found. However, the relationship between the answers for the guessed queries and the answer for q is left unspecified. To illustrate this strategy consider again the assignment statement in constant propagation:

$$a := b + c$$

Assume that the query $q =< [a = 0], n >$ is raised on exit of this statement. Since the flow function for the statement is not distributive, the query cannot be translated into an equivalent new query for the entry of the statement. However, it is possible to guess a sufficient query. The answer for q results directly once it has been determined whether the two operands b are c are constants. Thus, the new query generated at the predecessors m of n is $q' =< [b =?, c =?], m >$. Note that the new query is merely approximate since no specific constant values for variables b and c can be established. Thus, the lattice element $[b =?, c =?]$ expresses that variables b and c have unknown constant values. After all guessed queries 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 algorithm is shown in procedure *IsConstant* shown in Figure 4.9. The first phase is a backward analysis during which all queries that are guessed as sufficient are marked. The marked queries describe the set of data flow queries whose answers provide an answer for the original input query. Procedure *Mark_CP* implements the first marking phase in an iterative worklist algorithm that terminates when no more queries can be marked as sufficient. If during the marking phase a procedure call is encountered, marking

```
Procedure $IsConstant(v, n)$
input:  a variable $v$ and a node $n$
output: if $v$ is constant at $n$ then constant value $c$, otherwise $false$
begin
   $Mark\_CP(v, n)$; /* Phase 1 */
   Perform constant propagation over only the marked portion of the program; /* Phase 2: */
end
```

```
Procedure $Mark\_CP(v, n)$
input:  a variable $v$ and a node $n$
begin
   for each $m \in N$ do $mark[m] \leftarrow \emptyset$;
   $mark[n] \leftarrow \{v\}$; $worklist \leftarrow \{n\}$;
   while $worklist \neq \emptyset$ do
      remove a node $m$ from $worklist$;
      case $m = entry_{main}$:
         if $mark[m]$ not empty then return($false$);
      case $m = entry_q$ for some procedure $q$:
         for each call site $m'$ such that $call(m') = q$ do
            add $mark[m]$ to $mark[m']$;
            if $mark[m']$ changed then add $m'$ to $worklist$;
         endfor;
      otherwise:
         for each $m' \in pred(m)$ do
            if $m$ is a call site and $call(m) = q$ then
               $New \leftarrow SummaryMark\_CP(w, q)$;
            else if a variable in $mark[m]$ is defined at $m$ then
               add variables used at $m$ to $mark[m']$;
            else
               add $mark[m]$ to $mark[m']$;
            if $mark[m']$ changed then add $m'$ to $worklist$;
         endfor;
   endwhile;
end
```

Figure 4.9: Demand-driven analysis algorithm variation for CP.

```
**Procedure** $SummaryMark\_CP(v, p)$
**input:** a variable $v$ and a procedure $p$
**begin**
if $v \in M[exit_p, v]$ then return($M[entry_p, v]$); /* previously marked already */
worklist $\leftarrow \{(exit_p, v)\}$; $M[exit_p, v] = \{v\}$;
while worklist $\neq \emptyset$ do
    remove a pair $(n, w)$ from worklist and let $X \leftarrow M[n, w]$;
    case $call(n) = q$: /* n is a call site */
        for each variable $u \in X$ do
            if $u \in mark[exit_q, u]$ then for each $m \in pred(n)$ do
                $PropagateMark(m, w, M[entry_q, u])$;
            else /* trigger marking called procedure */
                $M[exit_q, u] \leftarrow \{u\}$ and add $(exit_q, u)$ to worklist;
    case $n = entry_q$ for some procedure $q$: /* n is entry node; propagate X to call sites if needed */
        for each call site $m$ such that $call(m) = q$ and $w \in M[m, u]$ for some variable $u$ do
            for each $m' \in pred(m)$ do $PropagateMark(m', u, X)$;
    otherwise: /* n is not a call site and not an entry node */
        if $w$ is defined at $n$ then $New \leftarrow$ set of variables used at $n$;
        else $New \leftarrow \{w\}$ ;
        for each $m \in pred(n)$ do $PropagateMark(m, w, New)$;
endwhile;
return($M[entry_p, v]$);
**end**
```

```
**Procedure** $PropagateMark(n, v, New)$ /* propagate new to mark[n, v] */
**input:** a node $n$, a variable $v$ and a set of variables $New$
**begin**
    $M[n, v] \leftarrow M[n, v] \cup New$;
    if $M[n, v]$ changed then add $(n, v)$ to worklist;
**end**
```

Figure 4.10: Procedure *SummaryMark* called by *Mark_CP*.

within the called procedure is performed as well by invoking procedure *SummaryMark_CP* shown in Figure 4.10. The second phase consists of subsequently resolving all marked queries by accordingly combining the identified constant values. Note that this second phase corresponds to a regular forward constant propagation analysis. However, unlike exhaustive constant propagation analysis that starts at program entry and propagates all constant information throughout the entire program, the second phase only propagates the constants to resolve previously guessed queries by considering only the marked portion of the program.

This two-phase approach is not limited to constant propagation. The strategy of using a preparatory backward analysis in order to reduce the analysis effort of the original forward analysis provides a general variation of the demand-driven approach to handle any monotone data flow problem. However, the two-phase approach is less efficient than the direct analysis reversal. In the worst case, the entire program is marked sufficient 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 the previous sections.

4.5 Related Work on Demand-Driven Analysis

A number of variations on the notion of demand-driven analysis and the notion of a partial backward propagation of information have appeared in the literature. The concepts of deriving data flow information by backward propagation of assertions was described using operational semantics by Cousot [Cou81] and later developed and implemented in a debugging system for higher-order functions [Bou93]. The analysis for discovering linked conditions in programs described in [SMHY93] is also based on backward propagation of assertions starting from test sites in conditionals.

One component of the developed demand-driven approach is the tabulation procedure *Compute*ϕ^r from Figure 4.4 for computing reverse summary function values. Procedure *Compute*ϕ^r implements the demand-driven evaluation of the relevant equation values of the summary function equation system. The algorithm is essentially a reversed version of Sharir and Pnueli's tabulation algorithm [SP81] to compute the original forward summary functions. A similar partial fixed point computation of only relevant equations was also described in the chaotic iteration algorithms [CC77c] and the minimal function graphs for applicative programs [JM73].

Reverse flow functions, which are used in the query propagation rules in Theorem 4.1, have previously been discussed in [HL92] to demonstrate that an abstract interpretation may be performed in either a forward or a backward direction. The relationship between forward and backward directions of an analysis was also discussed by Cousot [Cou81].

The previous work most closely related to the demand-driven framework developed in this thesis are the three approaches to demand-driven *inter*procedural analysis presented by Reps [Rep94] and Reps, Horwitz and Sagiv [RSH94, RHS95, SRH95a]. In the first approach by Reps [Rep94], a limited class of data flow problems, the *locally separable problems*, are 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 a more recent work by Reps et al. [RHS95], the first approach is generalized to a larger class of problems. 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. While the second approach by Reps, Horwitz and Sagiv [RHS95] is closely related to the demand-driven framework developed in this thesis, there are a number of important differences. Unlike the demand-driven approach by Reps, Horwitz and Sagiv, which is a graph based approach, this thesis presents a framework that models demand-driven analysis based on fixed point computations. Fixed point computations are well understood and many efficient algorithms for computing fixed points are available. A drawback of the graph-reachability approach is the need to construct an exploded supergraph for each data flow problem to be solved. The size of the exploded supergraph, and therefore also its construction time, can be substantial. The authors report that during experimentation with the graph-reachability analyzer for CCP, the analyzer ran out of virtual memory for some C programs of about 1,300 lines [SRH95b]. In comparison, the representation of the CCP flow functions using the algorithm developed in this chapter requires only a constant number of pointers to the symbol table entries of the variables defined and used at each (intermediate code) statement.

A further difference from the framework developed in this thesis concerns the applicability of the approaches. The graph-reachability approach imposes more restrictions on the class of problems that can be handled than the approach developed in this thesis. Specifically, the graph-reachability approach is limited to the class of distributive problems with a lattice that is a powerset over a finite domain set. Although distributive functions are also necessary in the approach developed in this thesis to ensure precise data flow solutions, the developed algorithms still provide approximate information in the presence of non-distributive functions. Furthermore, the framework presented in this chapter is less restrictive with respect to the lattice structure in that it is applicable to any finite lattice. Moreover, the restriction to a finite lattice does not even apply for *intra*procedural analyses.

Recently, Sagiv, Reps and Horwitz presented a new variation of the graph-reachability approach that uses a two phase algorithm [SRH95a]. This new approach can handle a larger class of distributive data flow problems than the framework developed in this chapter in that it also permits infinite lattices if the distributive function space does not contain infinite decreasing 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. Babich and Jazayeri [BJ78] presented a demand-driven algorithm for *intra*procedural live variable analysis based on attribute grammars. Strom and Yellin [SY93] presented a demand based analysis for typestate checking. The authors experimentally demonstrate that a goal-directed backward analysis is more efficient than a forward analysis for typestate checking that eagerly collects all available information that may or may not be of relevance. Question propagation, a phase in the algorithm for global value numbering [RWZ88], performs a demand-based backward search in order to locate redundant expressions. This backward search, like our query algorithm, performs the analysis from the points of interest (i.e., the points where an expression is suspected to be redundant) and it also uses early termination to end the search. Blume and Eigenmann presented a demand-driven algorithm for range propagation [BE95]. Range propagation is performed over only the portion of the program that is of relevance for the current information request. This relevant portion is extracted in a separate initial phase in a demand-driven fashion. In *procedure cloning* [CHK92], procedure clones are created during the analysis on demand whenever it is found that an additional clone will lead to more accurate information. Cytron and Gershbein [CG93] described an algorithm for the incremental incorporation of alias information into SSA form. 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 [CCF92].

Chapter 5

A Demand-Driven Analyzer for Gen-Kill Problems

In the previous chapter a framework for demand-driven data flow analysis was presented based upon which a generic demand-driven analysis algorithm was developed. It remains to show that the demand-driven analysis algorithm has efficient implementations in practice. Since the generic demand-driven algorithm is expressed in very general terms, a straightforward implementation may not be the most efficient one for a given data flow problem. It may be possible to improve the algorithm's efficiency by exploiting the specific properties of the data flow problem under consideration.

This chapter presents an efficient specialization of the generic demand-driven algorithm for Gen-Kill problems. The class of Gen-Kill problems includes the four classical problems REACH (reaching definitions), AVAIL (available expressions), LIVE (live variables), and BUSY (very busy expressions). A Gen-Kill problem is characterized by algebraically simple flow functions that allow for efficient implementations of exhaustive analyses based on bit vectors. This chapter demonstrates that characteristics of Gen-Kill problems also enable particularly efficient implementations of demand-driven analysis framework. It will be shown that specializing the demand-driven framework to the class of Gen-Kill problems results in a significant reduction of the asymptotic cost of the demand-driven algorithm.

To demonstrate the practicality of the specialized Gen-Kill framework, an experimental evaluation of a demand-driven REACH analyzer is presented. The demand-driven REACH analyzer is evaluated in the context of computing *du-chains* in a program. The computation of du-chains is required for most data flow applications in optimizations and software engineering tools. A du-chain connects a program point that defines a variable with a point that uses the defined value.

The demand-driven REACH analyzer was implemented along with a standard exhaustive analyzer as part of an experimental system. Experimental results are presented for two versions of the demand-driven analyzer: a caching version that uses a cache memory to store intermediate results for fast reuse, and a non-caching version that does not store intermediate results. Experimentation was performed to compare the performance of computing du-chains for a program using the caching and non-caching versions of the demand-driven analyzer and the exhaustive analyzer.

This chapter is organized as follows. The class of Gen-Kill problems is defined in Section 5.1. Section 5.2 presents the specialization of the demand-driven framework for Gen-Kill problems and discusses its asymptotic cost. Section 5.3 presents an instance of Gen-Kill framework for REACH analysis and shows how the resulting demand-driven REACH analyzer is used to construct the du-chains in a program. An optional optimization of the demand-driven analysis that can further reduce the analysis effort is described in Section 5.4. Section 5.5 presents the experimental study.

5.1 Gen-Kill Problems

A Gen-Kill problem describes data flow facts that are subsets of a finite set D of program objects. Consider the four classical Gen-Kill problems. In problem REACH the set D of program objects is the set of definitions in the program, in LIVE the set D is the set of variables in the program, and in AVAIL and BUSY the set D is the set of expressions that occur in the program.

Gen-Kill problems are characterized by a simple definition of their flow functions. Given a node n, the flow function f_n in a Gen-Kill problem is of the form:

$$f_n(X) = (X \perp Kill_n) \cup Gen_n,$$

where $Kill_n$ and Gen_n are constant subsets of D that depend entirely on the node n. Given a program object $d \in D$, the bit valued components of the sets Gen_n and $Kill_n$ with respect to d can be defined for each node n as follows:

$$Gen_n^d = \begin{cases} \mathit{true} & \text{if } d \in Gen_n \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$Kill_n^d = \begin{cases} \mathit{true} & \text{if } d \in Kill_n \\ \mathit{false} & \text{otherwise} \end{cases}$$

Gen-Kill problems are either intersection or union problems. In an intersection problem (e.g., AVAIL, BUSY) the meet operator $\sqcap$ corresponds to set intersection and the dual join operator $\sqcup$ corresponds to set union. In a union problem (e.g., REACH, LIVE) the roles of the meet and the join operators are interchanged.

Example: REACH is a union problem, where D is the set of definitions in the program. The set Gen_n is the set of definitions that are generated at node n and $Kill_n$ is the set of definitions of variables that are redefined at n (i.e., killed at n). Hence, the flow function $f_n(X) = (X \perp Kill_n) \cup Gen_n$ expresses that a definition $d \in D$ reaches the exit of node n if d is generated at n (i.e., $d \in Gen_n$) or if d reaches the entry of node n and is not killed at n (i.e., $d \notin Kill_n$).

If the program objects in D relate to program variables, as in the four classical Gen-Kill problems, the set D is structured according to the structure of the variable space. According to the usual scoping rules, the set of variables addressable in a procedure p is given by:

$$Addr(p) = Global \cup Formal(p) \cup Local(p).$$

Thus, if a program object $d \in D$ relates to a variable $v \in Addr(p)$, then d is a global, a formal or a local program object. The set $D(p)$ of program objects in a procedure p can be analogously structured:

$$D(p) = D_{global} \cup D_{formal}(p) \cup D_{local}(p).$$

Example: Consider the problem REACH, where the set D is the set of definitions in the program. Given a procedure p, the set $D(p)$ of objects visible in procedure p is given by the sets:

$$\begin{aligned} D_{global} = &\quad \{d \in D \mid d \text{ is a definition of a variable } v \in Global\} \\ D_{formal}(p) = &\quad \{d \in D \mid d \text{ is a definition of a variable } v \in Formal(p)\} \\ D_{local}(p) = &\quad \{d \in D \mid d \text{ is a definition of a variable } v \in Local(p)\} \end{aligned}$$

5.2 A Framework Instance for Gen-Kill Problems

Exploiting the algebraically simple definition of Gen-Kill problems leads to an efficient specialization of the general analysis framework from Chapter 4. A specialized instance of the general framework 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 are based upon the handling of procedure parameters as described in Chapter 4.

5.2.1 Specialized Queries and Propagation Rules

Consider first the specialized definition of a query. A query q in a Gen-Kill problem is of the form $q =< d, n >$, where $d \in D$ is a program object and n is a program node. Note that the general query definition from Chapter 4 allows the first component of a query to be any element of the powerset lattice over D.

(i) For each procedure p:

$$< d, entry_p > \quad \Longleftrightarrow \begin{cases} \mathit{false} & \text{if } p \text{ has no call sites or } d \in D_{local}(p) \\ \bigwedge\limits_{call(m)=p} \; < \tilde{b}_m^{-1}(d), m > & \text{otherwise} \end{cases}$$

(iii) For a non-entry node n that is not a call site:

$$< d, n > \quad \Longleftrightarrow \quad \bigwedge_{m \in pred(n)} \begin{cases} \mathit{true} & \text{if } Gen_m^d = \mathit{true} \\ \mathit{false} & \text{if } Kill_m^d = \mathit{true} \text{ and } Gen_m^d = \mathit{false} \\ < d, m > & \text{otherwise} \end{cases}$$

Figure 5.1: Specialized propagation rules for Gen-Kill problems.

Specializing the query propagation rules results in a number of simplifications. Unlike the general case, the reverse flow function in a Gen-Kill problem can be determined statically based on the local $Kill_n^d$ and Gen_n^d sets. The specialized query propagation rules for a query involving a program object $d \in D$ are shown in Figure 5.1. Note that rule (iii) in Figure 5.1 does not include the case that node n represents a procedure call. An extension of rule (iii) to include procedure calls is obtained by extending the definition of the variables $Kill_n^d$ and Gen_n^d. If node n represents a call site then variables $Kill_n^d$ and Gen_n^d express summary information about the execution of the called procedure. Specifically, $Kill_n^d$ and Gen_n^d express whether d is killed or generated as a result of executing the procedure called at node n.

The summary information for a procedure q is determined by computing vectors $K_q^d[n]$ and $G_q^d[n]$ for nodes n in q. Hence, the vectors $K_q^d[n]$ and $G_q^d[n]$ are the specialized instances of the reverse procedure summary functions ϕ^r from Chapter 4. In an intersection problem, these vectors are defined as follows:

$K_q^d[n] = \mathit{true}$ if, for *some* successor m of n, data flow fact d is killed along *some* path leading to m (i.e., $K_q^d[m] = \mathit{true}$) or d is directly killed at the successor m (i.e., $Kill_m^d = \mathit{true}$).

$G_q^d[n] = \mathit{true}$ if, for *each* successor m of n, data flow fact d is generated and not subsequently killed along *each* path leading to m (i.e., $G_q^d[m] = \mathit{true}$) or d is directly generated and not killed at the successor m (i.e., $Gen_m^d = \mathit{true}$).

The analogous definitions for a union problem are:

$$
\begin{aligned}
K_q^d[exit_p] &= \mathit{false} \\
K_q^d[n] &= Meet_{m \in succ(n)} \begin{cases} K_q^d[m] \vee Kill_m^d & \text{if } m \text{ is not a call} \\ K_q^d[m] \vee K_r^d[entry_r] & \text{if } m \in call(r) \text{ and } d \in D_{global} \\ K_q^d[m] & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
G_q^d[exit_p] &= \mathit{false} \\
G_q^d[n] &= Join_{m \in succ(n)} \begin{cases} G_q^d[m] \vee (Gen_m^d \wedge \neg(K_q^d[m])) & \text{if } m \text{ is not a call} \\ G_q^d[m] \vee (G_r^d[entry_r] \wedge \neg(K_q^d[m])) & \text{if } m \in call(r) \text{ and } d \in D_{global} \\ G_q^d[m] & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 5.2: Specialized procedure summary computation for Gen-Kill problems.

$K_q^d[n] = \mathit{true}$ if, for *each* successor m of n, data flow fact d is killed along *each* path leading to m or d is directly killed at the successor m.

$G_q^d[n] = \mathit{true}$ if, for *some* successor m of n, data flow fact d is generated and not subsequently killed along *some* path leading to m or d is directly generated and not killed at the successor m.

The formal definition of the vectors is shown in Figure 5.2, where the definition of the operators *Meet* and *Join* depends upon whether the current data flow problem is an intersection problem or a union problem. In an intersection problem, *Meet* is defined as the boolean conjunction ($\wedge$) and *Join* is defined as the boolean disjunction ($\vee$). In a union problem, the roles of *Meet* and *Join* are interchanged, such that $Meet = \vee$ and $Join = \wedge$. In an intersection problem the initial values for the equations in Figure 5.2 are: $G_p^d[n] = \mathit{true}$ and $K_p^d[n] = \mathit{false}$, and in a union problem: $G_p^d[n] = \mathit{false}$ and $K_p^d[n] = \mathit{true}$.

The summary information at a call site $s \in call(p)$ for a program object $d \in D$ is fully described by the vector of values on procedure entry: $K_p^d[entry_p]$ and $G_p^d[entry_p]$. Based on this summary information, the query propagation rules are complete and include the propagation across a call site $s \in call(p)$ by setting the variables $Kill_s^d$ and Gen_s^d such that $Kill_s^d = K_p^d[entry_p]$ and $Gen_s^d = G_p^d[entry_p]$.

5.2.2 Demand-Driven Algorithm for Gen-Kill Problems

The specialized framework components lead to a simplified version of the demand-driven algorithm. Procedure *Query_GenKill* shown in Figure 5.3 is derived from the generic procedure *Query* and evaluates the propagation rules from Figure 5.1. The generic summary computation procedure *Compute*ϕ^r specializes to procedure *GenKill*ϕ^r for computing the vectors K_q^d and G_q^d, as shown in Figure 5.4. Both procedures, *Query_GenKill* and *GenKill*ϕ^r assume an intersection problem. The corresponding versions of the procedures for union problems can be similarly developed.

To create the instances of the two demand-driven analysis procedures *Query_GenKill* and *GenKill*ϕ^r for a particular Gen-Kill problem, such as REACH, it is sufficient to specify the following parameters:

- The set of program objects D.
- The meet and join operators $\sqcup$ and $\sqcap$ (either set union or set intersection).
- The local variables Gen_n and $Kill_n$ at every node n that is not a call site.

5.2.3 Asymptotic Cost

Consider the asymptotic cost of the two procedures *Query_GenKill* and *GenKill*ϕ^r. *GenKill*ϕ^r determines the summary values for the vectors K_q^d and G_q^d as the fixed point of the equation system in Figure 5.2. A worklist is initialized with the triple $(d, exit_q, q)$ to trigger the computation of $K_q^d[entry_q]$ and $G_q^d[entry_q]$. During each step a triple (d, n, q) is removed from the worklist, the corresponding equations $K_q^d[n]$ and $G_q^d[n]$ 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. In programs with value parameters, each initial request for summary information refers to a global variable and can only trigger further summary variable requests for the same global variable. Thus, at most $O(|N|)$ equations are evaluated and the evaluation of each equation can result in the inspection of at most *MaxCall* other equations, where *MaxCall* is the maximal number of call sites calling a procedure. It follows that a single request for summary information requires $O(MaxCall \times |N|)$ time. The cost of k requests is $O(min(k, |D_{global}|) \times MaxCall \times |N|)$ and $O(|D_{global}| \times |N|)$ space is needed to store the summary vectors.[1]

Next, consider the maximal number of queries generated in procedure *Query_GenKill* based on the input query $q =< d, n >$. If d relates to a local variable, at most *MaxN* queries are generated, where *MaxN* is the maximal number of nodes in any procedure. If d relates to a global variable, the maximal number of generated queries is $|N|$ since one query for the global may be generated at every node. Finally, if d relates to a formal parameter, the initial query may change when propagating it through a procedure entry node to the call sites. In the worst case, the initial query generates additional queries with respect to all other

[1] In programs that contain reference parameters, summary information is needed for both, global variables and formal reference parameters. The asymptotic complexity for k requests changes to $O((|D_{global}| + MaxD_{formal}) \times MaxCall \times |N|)$ time and $O((|D_{global}| + MaxD_{formal}) \times |N|)$ space, where $MaxD_{formal}$ is the maximal number of program objects in any procedure that relate to formal parameters.

```
**Procedure** $Query\_GenKill(d, n)$
**input:**   a program object $d \in D$ and a node $n$
**output:** the answer $true$ or $false$ to the query $< d, n >$
**begin**
  **for each** $m \in N$ **do** $query[m] \leftarrow \emptyset$
  $query[n] \leftarrow \{d\}$; $worklist \leftarrow \{(d, n)\}$;
  **while** $worklist \neq \emptyset$ **do**
     remove a pair $(y, m)$ from $worklist$;
     **case** $m = entry_{main}$:
        **if** $query[m]$ is not empty **then then return**($false$);
     **case** $m = entry_p$ for some procedure $p$:
        **for each** call site $m'$ such that $call(m') = p$ **do**
             **if** $\tilde{b}_{m'}^{-1}(\{y\}) \neq \emptyset$ **then** /* if $y$ does not relate to a variable local to $p$ */
                 **if** $\tilde{b}_{m'}^{-1}(\{y\}) \notin query[m']$ **then**
                    add $\tilde{b}^{-1}(\{y\})$ to $query[m']$;
                    add $(\tilde{b}^{-1}(\{y\}), m')$ to $worklist$;
                 **endif**;
        **endfor**;
     **otherwise**:
        **for each** $m' \in pred(m)$ **do**
             **if** $m'$ is a call site **then**
                 compute summaries $Gen_{m'}^{y}$ and $Kill_{m'}^{y}$;
             **if** $Kill_{m'}^{y} = 1$ **then return**($false$);
             **if** $Gen_{m'}^{y} = 0$ **then** /* continue */
                 **if** $y \notin query[m']$ **then**
                    add $y$ to $query[m']$ and add $(y, m')$ to $worklist$;
        **endfor**;
  **endwhile**;
  **return**($true$);
**end**
```

Figure 5.3: Specialized demand-driven analysis algorithm for Gen-Kill problems.

```
Procedure $GenKill\phi^r(d, p)$
input:  a program object $d$ and a procedure $p$
output: the summary variables $G_p^d[entry_p]$ and $K_p^d[entry_p]$
begin
if $visited[entry_p, d] \leftarrow 1$ then return; /* result previously computed */
worklist $\leftarrow \{(d, exit_p, p)\}$;
$G_p^d[exit_p] \leftarrow 0$ and $K_p^d[exit_p] \leftarrow 0$;
while $worklist \neq \emptyset$ do
    remove a triple $(y, n, q)$ from $worklist$;
    $visited[n, y] \leftarrow 1$;
    case $n$ is a call site with $call(n) = r$:
        if $visited[entry_r, y] = 1$ then
            for each $m \in pred(n)$ do
                $Propagate(m, d, (K_q^y[n] \vee K_r^y[entry_r]), (G_q^y[n] \vee (G_r^y[entry_r] \wedge \neg(K_q^y[n])));$
        else /* trigger computation of summaries for r */
            $G_r^y[exit_r] \leftarrow 0$ and $K_r^y[exit_r] \leftarrow 0$;
            add $(y, exit_r, r)$ to $worklist$;
    case $n = entry_q$:
                /* Propagate to call sites if needed */
        for each call site $m$ in a procedure $r$, such that $call(m) = q$ and $visited[m, y] = 1$ do
            for each $m' \in pred(m)$ do
                $Propagate(m', y, (K_q^y[m] \vee K_p^y[entry_p]), (G_q^y[m] \vee (G_p^y[entry_p] \wedge \neg(K_q^y[m])));$
    otherwise:
            /* n is not a call site and not an entry node */
            for each $m \in pred(n)$ do
                $Propagate(m, y, (K_p^y[n] \vee Kill_n^y), (G_p^y[n] \vee (Gen_n^y \wedge \neg(K_p^y[n])));$
endwhile;
return;
end
```

```
Procedure $Propagate(p, n, y, kill, gen)$ /* propagate kill and gen to $K_p^y[n]$ and $G_p^y[n]$ */
input:  a procedure $p$, a node $n$, a variable $y$, and bit values $kill$ and $gen$
begin
        $K_p^y[n] \leftarrow K_p^y[m] \vee kill$;
        $G_p^y[n] \leftarrow G_p^y[n] \wedge gen$;
        if $K_p^y[n]$ of $G_p^y[n]$ changed then add $(y, n, p)$ to $worklist$;
end
```

Figure 5.4: Procedure $GenKill\phi^r$ to compute Gen-Kill procedure summaries.

variables. Hence, up to $MaxD \times |N|$ queries may be generated, where $MaxD$ is the maximal size of the program object set D in any procedure. Each generation of a query results in the inspection of at most $MaxCall$ other queries. Thus, the total cost of generating queries in procedure *Query_GenKill*, including the cost of $GenKill\phi^r$, is $O(MaxD \times MaxCall \times |N|)$ time and $O(MaxD \times |N|)$ space to store the generated queries and summary vectors.

In comparison, the asymptotic complexities of the general framework from Chapter 4 are $O(height(L) \times |L| \times MaxCall \times |N|)$ time and $O(|L| \times |N|)$ space. In a Gen-Kill problem the lattice L corresponds to the powerset of the program object set D. Hence, a straightforward implementation of the general framework for a Gen-Kill problem would have resulted in exponential cost in the size of object set D: $O(|D| \times 2^{|D|} \times MaxCall \times |N|)$ time and $O(2^{|D|} \times |N|)$ space.

5.3 Application: Demand-Driven DU-Chain Analyzer

This section illustrates an instance of the demand-driven Gen-Kill framework for REACH analysis. The resulting demand-driven REACH analyzer is considered in the context of du-chain computation. Informally, a du-chain is a pair (d, u) that connects a definition d of a variables with a use u of the defined value.

The program in Figure 5.5 is used to illustrate the du-chain computation. To distinguish multiple definitions and uses of the same variable, the node number is used as a subscript, i.e., x_n denotes the reference of variable x at node n.

5.3.1 Interprocedural REACH Analysis

The computation of interprocedural du-chains and interprocedural reaching definitions is complicated since definitions may reach uses across procedure boundaries. Moreover, in programs with parameter passing, definitions and uses of the same value may refer to the value using different variable names. To correctly establish reaching definitions across procedure boundaries, the variable bindings that result from parameter passing must be considered. To formally describe the bindings that occur along an execution path, the sequence of active procedures whose execution has not yet terminated along the path is considered.

Definition 5.1 (Active Call Sequence) *Let π be a valid execution path. The* **active call sequence** *for π is obtained from π by eliminating all nodes except the call sites of procedures that have not terminated when execution reaches the end of π.*

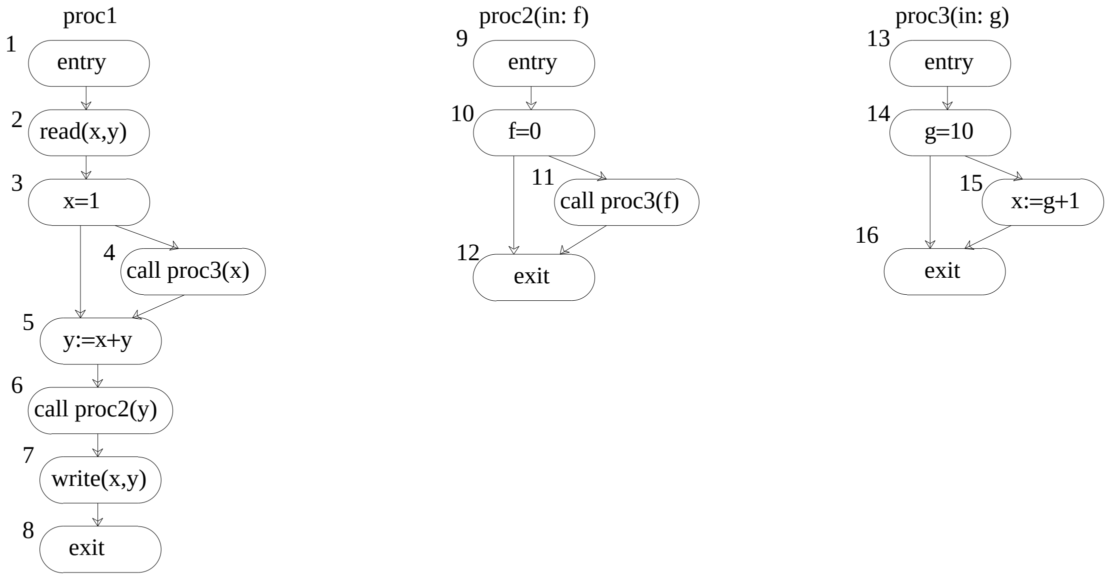
The value of a variable may be bound to variables in a calling or a called procedure as defined below.

Definition 5.2 (Binding) *Let p be a procedure and let s be a call site in p calling a procedure q, i.e., $s \in call(q)$. Furthermore, let v and w be variables in the address spaces of*

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

```
procedure proc2(in: f)
begin
    if f=0 then call proc3(f);
end
```

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



Data flow sets for REACH analysis

procedure $p = proc1$

n	Gen_n	$Kill_n$	$RD(n)$
1	-	-	-
2	x_2, y_2	x_{15}, y_5	-
3	-	-	x_2, y_2
4	x_{15}	-	x_2, y_2
5	y_5	y_2	x_2, x_{15}, y_2
6	x_{15}	-	x_2, x_{15}, y_5
7	-	-	x_2, x_{15}, y_5
8	-	-	x_2, x_{15}, y_5

procedure $p = proc2$

n	Gen_n	$Kill_n$	$RD(n)$	$G_p^{x_2}$	$G_p^{x_{15}}$	$K_p^{x_2}$	$K_p^{x_{15}}$
9	-	-	x_2, x_{15}, y_2, y_5	*false*	*true*	*false*	*false*
10	-	-	x_2, x_{15}, y_2, y_5	*false*	*true*	*false*	*false*
11	x_{15}	-	x_2, x_{15}, y_2, y_5	*false*	*true*	*false*	*false*
12	x_{15}	-	x_2, x_{15}, y_2, y_5	*false*	*true*	*false*	*false*

procedure $p = proc3$

n	Gen_n	$Kill_n$	$RD(n)$	$G_p^{x_2}$	$G_p^{x_{15}}$	$K_p^{x_2}$	$K_p^{x_{15}}$
13	-	-	x_2, x_{15}, y_2, y_5	*true*	*true*	*false*	*false*
14	-	-	x_2, x_{15}, y_2, y_5	*false*	*true*	*false*	*false*
15	x_{15}	x_2	x_2, x_{15}, y_2, y_5	*false*	*true*	*false*	*false*
16	x_{15}	x_2	x_2, x_{15}, y_2, y_5	*false*	*true*	*false*	*false*

Figure 5.5: Program with data flow sets for REACH analysis.

p and q, respectively.

(i) The value of v in p is **bound to** *w* **in** *q* **via** *s if $w \in b_s(v)$.*

(ii) The value of w in q is **bound to** *v* **in** *p* **via** *s if $v \in b_s^{-1}(w)$.*

Let $S = s_1, \ldots, s_k$ be an active call sequence of a valid execution path π, such that s_1 is contained in a procedure p_1 and s_i calls a procedure p_{i+1} for $1 \leq i < k$. Furthermore, let v and w be variables in the address spaces of p_1 and p_{k+1}, respectively.

(iii) The value of v in p_1 is **bound to** *w* **in** *p_{k+1}* **via** *S if there exists a sequence of variables $v_1, \ldots, v_{k+1}$ with $v_1 = v$ and $v_{k+1} = w$, such that the value of v_i is bound to v_{i+1} via s_i for $1 \leq i < k$.*

(iv) The value of w in p_{k+1} is **bound to** *v* **in** *p_1* **via** *S if there exists a sequence of variables $v_1, \ldots, v_{k+1}$ with $v_1 = v$ and $v_{k+1} = w$ such that the value of v_{i+1} is bound to v_i via s_i for $1 \leq i < k$.*

The value of a variable is always bound to the variable itself via the empty call sequence.

Example: Consider the program in Figure 5.5, where all parameters are passed by value. The value of variable x is bound to parameter g in *proc3* via the call at node 4. The value of variable y is not bound to any variable in *proc3* since y is local and not passed as a parameter. However, the value of y is bound to the parameter f in *proc2* via the call at node 6.

Note that the notion of binding differs from that of aliasing, which was previously discussed in Chapter 4. Binding refers to the binding of *values* to new variables upon procedure invocation. Aliasing refers to the binding of variables references, such that two variables may be bound to the same memory location. For example, if a global variable x is passed to a parameter f by reference, then x and f are aliases and the value of x is bound to f. However, if x is passed to f by value then the value of x is also bound to f but x and f are not aliases of one another.

A formal definition of reaching definitions is based on the notion of *killing*, or alternatively, *kill-free* nodes as defined below.

Definition 5.3 (Kill-free) *A node n is called* **kill-free** *for variable v if node n does not contain a definition of a variable that is a must-alias of v at node n. A path π is called* **kill-free** *for variable v if every node in π is kill-free for v.*

Interprocedural reaching definitions and the symmetric problem of interprocedural *reachable uses* can now be defined as follows.

Definition 5.4 (Interprocedural reaching definition) *A definition d of a variable v is a* **reaching definition** *of a variable w at node n if:*

(i) there exists a valid execution path π from d to node n with an active call sequence $S = s_1, \ldots, s_k$ such that variable v is bound to a sequence of variables $v_1, \ldots, v_k = w$ via S and

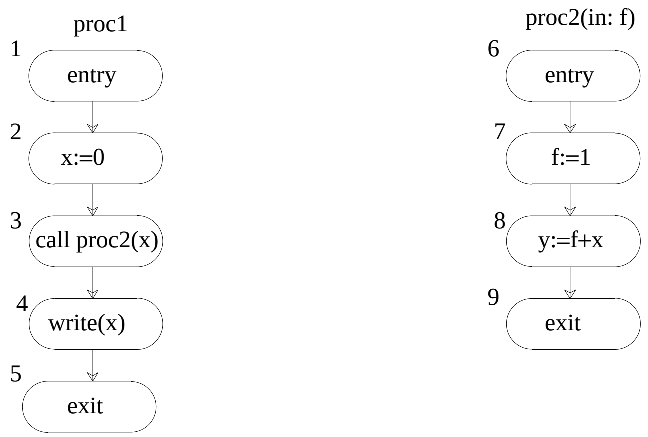


Figure 5.6: Interprocedural du-chains with global variables x and y.

(ii) the following subpaths in π are kill-free: the subpath from d to s_1 is kill-free for v, the subpaths $s_i \ldots s_{i+1}$ are kill-free for v_i for $1 \leq i < k$, and the subpath $s_k \ldots n$ is kill-free for w.

Definition 5.5 (Interprocedural reachable use) *A use u of a variable v is a* **reachable use** *of a variable w at node n if:*

(i) there exists a valid execution path π from n to the use u with an active call sequence $S = s_1, \ldots, s_k$ such that variable w is bound to a sequence of variables $v_1, \ldots, v_k = v$ via S and

(ii) the following subpaths in π are kill-free: the subpath $n \ldots s_1$ is kill-free for w, the subpaths $s_i \ldots s_{i+1}$ are kill-free for v_i for $1 \leq i < k$, and the subpath from s_k to the use u is kill-free for v.

The sets of reaching definitions and reachable uses of a variable v at a node n are denoted $RD(v,n)$ and $RU(v,n)$. Furthermore, the sets of reaching definitions and reachable uses of any variable at a node n, where n is contained in a procedure p, are defined as:

$$RD(n) = \bigcup_{v \in Addr(p)} RD(v,n)$$

$$RU(n) = \bigcup_{v \in Addr(p)} RU(v,n)$$

Example: Consider the example in Figure 5.6 and the question as to whether the definition f_7 of formal value parameter f at node 7 reaches node 4 in procedure *proc1* after the call to *proc2*. Since f is a parameter, its scope ends with the end of procedure *proc2*. Moreover, since f is not a reference parameter, its value is not bound to any variable in procedure *proc1*. Thus, definition f_7 does not reach a node along any path outside procedure *proc2*. Now, consider definition y_8 at node 8. Since y is a global variable, its scope extends beyond procedure *proc2* and the value assigned to y in procedure *proc2* is still bound to y at nodes outside *proc2*. Thus, the definition y_8 reaches nodes 4 and 5 in procedure *proc1*.

The definition of interprocedural reaching definitions and reachable uses also provide a formal characterization of the du-chains in a program.

Definition 5.6 (Du-chain) *Let d be a definition of variable v at a node n and let u be a use of variable w at node m. The pair (d, u) is a* **du-chain** *if $d \in RD(w, m)$ (or equivalently, if $u \in RU(v, n)$).*

Example: Consider Figure 5.6 and the questions as to whether (x_2, f_8) establish a du-chain. The value of the global x is bound to the formal f but node 7 is not kill-free for f. Hence, x_2 is not a reaching definition of f at node 8 and (x_2, f_8) is not a du-chain. However, every node in the path from node 2 to node 8 is kill-free for variable x. Thus, x_2 is a reaching definition for variable x at node 8 and the pair (x_2, x_8) is a du-chain.

In a program that consists of multiple procedures, du-chains may cross procedure boundaries. To determine whether a du-chain crosses procedure boundaries, the kill-free paths associated with the du-chain are examined. For a du-chain (d, u) there may be several distinct kill-free paths from d to u. Some of these paths may cross procedure boundaries while others may remain intraprocedural paths. For a du-chain to be an interprocedural chain it should have at least one kill-free path that crosses procedure boundaries. However, the existence of a kill-free path that crosses procedure boundaries is not sufficient to make a du-chain interprocedural. Consider for example the du-chain (y_2, y_5) for the local variable y in Figure 5.5. There are two kill-free paths for the chain, one path in *proc1*: 2,3,5 and one path across procedure *proc3*. Despite the existence of a kill-free path across *proc3*, the du-chain (y_2, y_5) does not qualify as an interprocedural chain since variable y is local to *proc1* and not passed as a parameter. Intuitively, a du-chain is interprocedural if the chain extends across a procedure invocation that potentially affects the value of the pair. To formalize this notion *potential kill nodes* of a variable are introduced.

Definition 5.7 (Potential kill) *A node n is called a* **potential kill** *node for variable v if variable v has at least one must-alias in the procedure that contains node n.*

Clearly, for a global variable any node is a potential kill node. However, for a local variable of a procedure p that is not passed as reference parameter no node outside p can be a potential kill node. To qualify as an interprocedural du-chain, the chain must pass through at least one potential kill node that lies in a different procedure.

Definition 5.8 (Interprocedural du-chain) *Let (d, u) be a du-chain, where d is a definition of a variable v contained in a procedure p. The chain (d, u) is an* **interprocedural**

definition	du-chains	
	intraprocedural	interprocedural
x_2	$(x_2, x_3), (x_2, x_4), (x_2, x_5)$	$(x_2, x_5), (x_2, x_7), (x_2, g_{14}), (x_2, g_{15})$
x_{15}		$(x_{15}, x_5), (x_{15}, x_7)$
y_2	(y_2, y_5)	
y_5	$(y_5, y_6), (y_5, y_7)$	$(y_5, f_{10}), (y_5, f_{11}), (y_5, g_{14}), (y_5, g_{15})$

Table 5.1: Du-chains for the example from Figure 5.1.

du-chain *if the definition d reaches the use u along a kill-free path that contains at least one node outside p that is a potential kill node for v.*

Definition 5.9 (Intraprocedural du-chain) *Let (d, u) be a du-chain, where d is a definition of a variable v contained in a procedure p. The chain (d, u) is an* **intraprocedural du-chain** *if the definition d reaches the use u along a kill-free path that contains no nodes outside p that are potential kill nodes for v.*

Example: Table 5.1. shows the complete sets of intra- and interprocedural du-chains for the program in Figure 5.5. The pair (x_{15}, x_5) is an interprocedural pair since node 5 is a potential kill node for the pair that is contained in a different procedure than the definition x_{15}. The pair (x_2, x_5) is also interprocedural since every node in *proc3* is a potential kill node for the pair. In contrast, the pair (y_2, y_5) is not interprocedural. Since y is local, no node outside *proc1* can be a potential kill node.

Note that a du-chain with multiple kill-free paths may be both intra- and interprocedural. For example, the pair (x_2, x_5) in Figure 5.5 is an intraprocedural du-chain due to the kill-free path 2,3,5. The pair (x_2, x_5) is also an interprocedural du-chain due to the kill-free path 2,3,4,13,14,15,16,5 with the potential kill nodes 13, 14, 15 and 16.

5.3.2 DU-Chains on Demand

The standard approach for computing du-chains is to first compute exhaustive sets of reaching definitions at each program point. The actual du-chains are then established in a second phase by selecting the appropriate definitions from the computed sets at every use of a variable. At each point, the definitions of variables that are not live at that point are useless and need not be computed. The computation of these useless reaching definitions can be avoided if a demand-driven approach is used. In a demand-driven approach, reaching definitions are computed only at the program points, where they are needed, that is, at

```
**Procedure** *DU-Chains(P)*
**input:**   a program $P$
**output:** the set of du-chains for $P$
**begin**
1.  **for each** node $n$ in $P$ **do**
2.      **if** $n$ contains the use $u$ of a variable $v$ **then**
3.              compute $RD(v, n)$;
4.              collect the du-chains: $\{(d, u) \mid d \in RD(v, n)\}$
5.      **endif**
6.  **endfor**
**end**
```

Figure 5.7: Demand-driven du-chain computation.

every use of a variable. The algorithm to compute du-chains on demand is outlined in Figure 5.7. To collect the required sets of reaching definitions in line 3 the instance of the demand-driven GEN-KILL framework for REACH is used.

The Gen-Kill framework instance for REACH is created by specifying the framework parameters as follows:

- D = set of definitions in the program
- $\sqcap = \cap$ (set intersection) and $\sqcup = \cup$ (set union)
- Gen_n = set of definitions generated at node n
- $Kill_n$ = set of definition killed at node n

Using this framework instance for REACH, the set $RD(v, n)$ of reaching definitions can be computed by issuing a corresponding query for algorithm *Query_GenKill*.

Example: Consider the demand-driven REACH analysis of the program example in Figure 5.5. The variables Gen_n and $Kill_n$ at each node n are shown in the table in Figure 5.5. The results of the summary vectors G_p^d and K_p^d for the two procedures *proc2* and *proc3* are also shown in the table. For example, the entry $K^{x_{15}}_{proc3}[13]$ = *false* expresses that definition x_{15} is not killed during the execution of *proc*3 and $G^{x_{15}}_{proc3}[13]$ = *true* expresses that definition x_{15} reaches the exit of procedure *proc3*. Note that *proc1* is not called by another procedure and, therefore, requires no summary information.

Using the summary vector, the values for Gen_n and $Kill_n$ at the call sites at nodes 4, 6 and 11 result as:

$$Gen_4 = \{x_{15}\}, \quad Kill_4 = \emptyset,$$

$$Gen_6 = \{x_{15}\}, \quad Kill_6 = \emptyset,$$

$$Gen_{11} = \{x_{15}\}, \quad Kill_{11} = \emptyset.$$

Consider now a request for the set $RD(x,5)$ of reaching definitions of variable x at the entry of node 5. This request is expressed by the two queries $?(x_2,5)$ and $?(x_{15},5)$. Since these two queries request definitions of the same variable, they can be resolved simultaneously during a single query propagation. Thus, the combined query $?(x,5)$ for any definitions of variable x is considered. Since $x \in Global$, propagation across the call $proc3(x)$ at node 4 requires computation of the summary variables $K^{x2}_{proc3}[13] = \mathit{false}$ and $K^{x15}_{proc}[13] = \mathit{false}$, which indicate that the query is not killed. The summary variable $G^{x15}_{proc3}[13] = \mathit{true}$ denotes that definition x_{15} is generated during the execution of procedure *proc3*. Hence, definition x_{15} can be collected as a reaching definition and the search for the remaining reaching definitions continues with the new query $?(x,4)$. Note that, at this point, there is only one further potential reaching definition for variable x left (i.e., definition x_2). Next, the new query is propagated through node 3, where $Kill_3^{x2} = \mathit{false}$ and $Kill_3^{x15} = \mathit{false}$. Since node 2 contains definition x_2, the search terminates with the set of reaching definitions $\{x_2, x_{15}\}$.

5.4 Query Advancing

This section describes a simple but effective optimization of the query propagation. The optimization is applicable to Gen-Kill problems in which the Gen and/or Kill points are based on definitions or uses of variables, i.e., REACH and LIVE. Based on some additional information about procedure execution, the duration of the query propagation is shortened by skipping program portions through *query advancing.* For simplicity, this section assumes that parameters are passed by value only.

Example (Advancing across call): Consider the example in Figure 5.8 and a query for the reaching definitions of the global variable x at node 7, i.e., $< x, 7 >$. Propagating query $< x, 7 >$ across the call site at node 6 would require the computation of summary information about the called procedure *proc3*. However, if it is known that procedure *proc3*, and any procedure subsequently called by *proc3*, does not contain a node with a definition of variable x, the summary information can be determined without analysis: no definitions of x can be killed or generated. Thus, the summary computation can be skipped and the query $< x, 7 >$ can be directly forwarded across the procedure call as shown by the dashed arrow.

Example (Advancing to entry): Next, consider the propagation of the new query $<$ $x, 5 >$ to the call site in procedure *proc1.* Based on the propagation rules from Chapter 4,

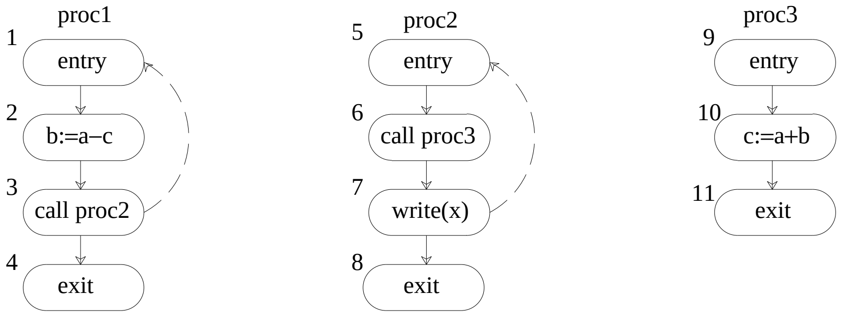


Figure 5.8: Query advancing in REACH analysis.

the query $< x, 5 >$ would be translated into a new query at node 3. However, if it known that procedure *proc1* (and any procedure subsequently called by *proc1*) does not contain a node with a definition of variable x, 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 in Figure 5.8.

The additional information needed for these two advancing optimizations are the *flow-insensitive* procedure summary sets *Mod*(p) for procedures p [CK88]. The set *Mod*(p) contains the variables that may be modified by the execution of procedure p because either p directly modifies the variable or the variable is modified by a procedure subsequently called by p. Let *DMmod*(p) be set of variables that are directly modified in procedure p. The set $Mod(p)$ is then defined by the following equation:

$$\mathit{Mod}(p) = \mathit{DMod}(p) \cup (\bigcup_{p \text{ calls } q} \mathit{Mod}(q))$$

The set *DMod*(p) is locally determined by a simple inspection of the definitions that are contained in procedure p (e.g., during parsing). The local sets *DMod*(p) are then propagated by iterating over the program's call graph using a simple iterative worklist algorithm. In the absence of recursion, an iterative algorithm computes the sets *Mod*(p) in time linear in the number of edges in the call graph (i.e., number of procedure calls). Recursion creates cycles in the call graph which increases the worst case cost of the iterative algorithm to be quadratic in the size of the call graph.

The summary information $Mod(p)$ is called *flow-insensitive* since determining the information does not require flow analysis and the control flow within each procedure can be ignored. In contrast, summary information, such as the vectors G_q^d and K_q^d in the Gen-Kill framework, is called *flow-sensitive*. Flow-sensitive summary information requires the control flow within a procedure to be analyzed. Consider for example the information in the vector G_q^d in REACH. The value $G_q^d[n]$ expresses that there exists a control flow path from node n to the exit of procedure q along which definition d is generated and not subsequently killed.

Using the $Mod(p)$ sets the two jump optimizations are defined as follows:

- **Advancing across calls**
  Propagating a query $q =< d, n >$ for the definition d of a variable x across a call site $m \in call(p)$ requires summary information only if $x \in Mod(p)$. Otherwise, q can be directly forwarded across the call:

$$< d, n > \Longleftrightarrow \bigwedge_{\substack{m \in pred(n),\\ m \in call(p)}} \begin{cases} false & \text{if } x {\in} Mod(p) \wedge Gen_m^d = true \\ true & \text{if } x {\in} Mod(p) \wedge Kill_m^d {=} true \wedge Gen_m^d {=} false \\ < d, m > & \text{otherwise} \end{cases}$$
$$\bigwedge_{\substack{m \in pred(n),\\ m \text{ not a call}}} \begin{cases} false & Gen_m^d = true \\ true & Kill_m^d = true \text{ and } Gen_m^d = false \\ < d, m > & \text{otherwise} \end{cases}$$

- **Advancing to entry**
  When propagating a query $q =< d, entry_p >$ for the definition d of a variable x into a procedure r that calls p then q can be directly forwarded to $entry_r$ if $x \notin Mod(r)$.

$$< d, entry_p > \quad \Longleftrightarrow \begin{cases} false & \text{if } x \in Local(p) \\ \bigwedge_{\substack{call(m) = p, \text{ where}\\ m \text{ is in procedure } r}} & \begin{cases} < \tilde{b}_m^{-1}(d), entry_r > & \text{if } x \notin Mod(r) \\ < \tilde{b}_m^{-1}(d), m > & \text{otherwise} \end{cases} \end{cases}$$

The two query advancing optimizations are not limited to REACH analysis but are applicable to any Gen-Kill problem in which the generation and killing of a program object are based on definitions and uses of variables. To apply query advancing in problems in which generation and/or killing is based on variable uses simply requires a different kind of flow-insensitive summary information. Instead of the sets $Mod(p)$, the similar flow-insensitive sets $Ref(p)$ are used. $Ref(p)$ contains the variables that are referenced as a use in procedure p or in any procedure subsequently called by p:

$$Ref(p) = DRef(p) \cup (\bigcup_{p \text{ calls } q} Ref(q)),$$

where *DRef*(p) is the set of variables that are directly used in procedure p.

Note that exhaustive analysis cannot be optimized by in a similar way. Query advancing is enabled through the goal-directed search of demand-driven analysis. Exhaustive analysis is not goal-directed and collects all data flow facts that are generated. Unless a procedure generates no data flow facts (i.e., contains neither definitions nor uses of variables), its analysis cannot be skipped.

5.5 Experiments

An experimental study was conducted to evaluate the practical benefits of computing du-chains based on the demand-driven REACH analyzer. The study's primary objective was to compare the performance of the demand-driven du-chain algorithm with that of a standard exhaustive algorithm. Additional experiments were carried out to evaluate the trade-off between the benefits and overhead of (i) the caching capability and (ii) query advancing in the demand-driven algorithm.

The experiments are based on implementations of the following three algorithms:

(CACHE) Caching demand-driven du-chain algorithm as described in the previous section based on a caching version of the demand-driven REACH analyzer.

(DD) A non-caching version of the demand-driven du-chain algorithm (CACHE).

(EX) An exhaustive du-chain algorithm. The exhaustive algorithm computes du-chains by standard exhaustive reaching definition analysis based on the interprocedural Sharir/Pnueli framework [SP81]. 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 (CACHE) and (DD).

The exhaustive analysis is implemented using efficient bit vector representations of data flow sets. Note that bit vector operations cannot be used to implement the demand-driven analyzers since analysis is performed with respect to individual queries, i.e., individual bit vector positions. The demand-driven algorithms (CACHE) and (DD) optionally include query advancing.

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 a statement level control flow graph of each procedure based on a three-address intermediate representation of the program. The current versions of the three algorithms perform analysis over scalar variables only. The inclusion of structured variables is the subject of future extensions. The implemented algorithms assume programs that are free of pointer induced aliasing. Pointer references in C programs are handled by assuming that the address operator "&" destroys the value of the variables to which it is applied. This treatment of pointer references may not be safe in the presence of aliasing. To guarantee safety, a write via a pointer should consider every variable that may be pointed to as overwritten. To avoid overly conservative worst case estimates of the set of variables that may be pointed to by a pointer, additional alias information must be collected and incorporated into the analysis as described in Chapter 4.

An aspect of a compiler front end with implications on analysis performance is the treatment of temporary variables. The PDGCC front end was designed to generate single-assignment temporary variables. Thus, temporaries are not recycled for future re-use. 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 may also increase the size of the address space. Large address spaces impact on the performance of data flow algorithms whose complexity depends on the number of addressable variables. Furthermore, single-assignment temporaries are typically used in a fairly controlled way. The uses and definitions of temporaries are likely to be 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 du-chains for a temporary variable immediately after the temporary has been created. However, if the program changes and re-analysis is to be expected, the locality property of references to temporaries may be destroyed and a subsequent re-analysis of the program may have to consider temporary variables.

In order to avoid a bias in the experimental results towards a particular strategy for handling temporary variables, the experiments are conducted in two versions. One version that considers the complete address space in each procedure, including all compiler generated temporaries, and one version that only includes source-level variables into the analysis.

The experiments were run on a SUN SPARCstation 5 with 32 MB of RAM. Table 5.2 shows the 17 C programs that were used during the study. Programs 7-17 are core routines of Unix utility sources. Table 5.2. shows for each program the number of source code lines, the number of nodes in the control flow graph, the number of procedures, the number of calls, and parameters concerning the size of the variable space. The column indicated as *MaxVar* shows the maximal number of variables in any one procedure including temporaries. The maximal number of source-level variables (excluding temporaries) is shown in parentheses. The last column shows the number of global variables.

All reported analysis times are user cpu times in seconds. The cpu times were 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 *GMod* vectors. Analysis times do not include the time for the preparatory pass over the program to set up analysis specific parameters that are required in both the exhaustive and the demand-driven analyses. These parameters, that include the collection of local *Gen* and *Kill* sets, could also be determined during parsing.

Benchmarks							
No.	program	#code lines	#nodes	#procedures	#calls	*MaxVar*	#globals
1	bubble	64	105	5	4	29 (9)	6
2	quicksort	65	141	6	7	32 (12)	5
3	hanoi	69	117	3	5	28 (11)	3
4	queens	84	150	4	4	38 (8)	3
5	heapsort	99	173	2	1	72 (24)	0
6	nsieve	115	192	2	2	40 (18)	6
7	cat	240	377	5	4	61 (15)	7
8	calendar	352	731	10	14	53 (10)	4
9	getopt	395	739	5	6	80 (12)	4
10	linpack	564	686	12	30	140 (17)	0
11	diff	818	899	12	33	211 (41)	27
12	patch	753	1316	14	13	141 (38)	27
13	tar	1214	1451	27	68	182 (49)	45
14	gzip	1245	2495	37	97	173 (48)	45
15	grep	1488	2906	32	72	127 (29)	16
16	sort	1528	3554	35	145	151 (19)	10
17	dc	1576	3298	67	230	91 (19)	10

Table 5.2: Benchmark programs. Parentheses indicate measurements that exclude temporaries.

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 *Gmod* sets if query advancing was used.

5.5.1 Experiment 1: Caching Demand-Driven versus Exhaustive

The first experiment compares the performance of the exhaustive du-chain analyzer with the performance of the caching demand-driven analyzer when used to compute *all* du-chains. The exhaustive du-chain algorithm was performed over all test programs. The resulting exhaustive analysis time T_{ex} and space consumption S_{ex} for each program are listed in Table 5.3.

The caching demand-driven analyzer with query advancing was executed for a set of queries that enables the construction of all du-chains in a program. This set contains one query for the reaching definitions at each use of a variable, that is, at most two queries at each node which corresponds to $O(|N|)$ queries. These queries were generated in random order over the program. Table 5.3 shows the number of generated queries and the total number of du-chains in each program.

The demand-driven analysis time T^{opt}_{cache} accumulated over all queries is shown in Table

Exhaustive Analysis (Du-chain)

program	time (secs) T_{ex}	space (Kbytes) S_{ex}
bubble	0.03 (0.02)	10.500 (9.240)
quicksort	0.03 (0.03)	14.572 (12.880)
hanoi	0.02 (0.02)	12.216 (10.812)
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)

Queries (Du-chain)

program	queries	du-chains
bubble	69 (31)	91 (55)
quicksort	94 (49)	160 (119)
hanoi	60 (36)	72 (41)
queens	105 (56)	119 (60)
heapsort	118 (77)	190 (147)
nsieve	105 (72)	145 (112)
cat	165 (64)	215 (104)
calendar	236 (51)	275 (70)
getopt	268 (159)	1059 (929)
linpack	1160 (668)	1543 (1002)
diff	555 (185)	685 (268)
patch	599 (307)	1075 (740)
tar	578 (260)	847 (500)
gzip	1161 (543)	2068 (1350)
grep	805 (331)	1048 (508)
sort	1065 (421)	1570 (815)
dc	1271 (553)	1958 (1011)

Table 5.3: Exhaustive analysis time (T_{ex}) and space (S_{ex}) and the number of queries and du-chains for each benchmark. Parentheses indicate measurements that exclude temporaries.

5.4. The analysis T^{opt}_{cache} is based on the caching version of the demand-driven algorithms with query advancing. Table 5.4 also shows the accumulated space consumption and the *cache fill.* The cache fill is the percentage of the exhaustive reaching definition 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 construct all du-chains. The cache fill values in Table 5.4 show that actually only a small portion of the exhaustive solution is relevant. The remaining portion of the solution consists of useless reaching definitions that are propagated beyond the points where the defined variable is live in the current procedure. Demand-driven analysis naturally suppresses the computation of these useless reaching definitions.[2]

[2] 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 du-chains. In exhaustive live-use analysis, the use of a variable may be propagated past the points where the variable is live. Avoiding the propagation of useless live-use information 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 useless propagation.

Caching Demand-Driven Analysis (Du-chain)				**Savings**	
program	time (secs)	space (Kbytes)	cache fill	speedup	space : % of S_{ex}
	T^{opt}_{cache}	S^{opt}_{cache}		$\frac{T_{ex}}{T^{opt}_{cache}}$	$\frac{S^{opt}_{cache} \times 100}{S_{ex}}$
bubble	0.02 (0.01)	10.548	22% (35%)	1.5 (2.0)	100.5%
quicksort	0.05 (0.02)	15.072	19% (30%)	0.6 (1.5)	103.4%
hanoi	0.03 (0.02)	12.380	20% (36%)	0.7 (1.0)	101.3%
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 5.4: Accumulated demand-driven analysis time and space with caching (T^{opt}_{cache} and S^{opt}_{cache}), the cache fill, the speedup of the demand-driven analyzer with caching over the exhaustive analyzer and the demand-driven analyzer space utilization as a percentage of the exhaustive analysis space. Parentheses indicate measurements that exclude temporaries.

When considering the complete variable space (i.e., 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 but their definitions may be propagated far beyond their use points. However, Table 5.4 shows that even after excluding temporaries from the analysis, on average more than half of the solution is not needed.

Figure 5.9 (i) displays the speedups $\frac{T_{ex}}{T^{opt}_{cache}}$ of the demand-driven analyzer with caching over the exhaustive analyzer. The demand-driven analyzer computes du-chains faster than the exhaustive analyzer in 11 out of 17 test programs. Importantly, except for one program (gzip), the slowdown of the demand-driven analyzer occurs only for short programs. The demand-driven analysis achieves speedups for the larger programs by factors ranging from 1.1 up to 11.4. Very short programs (less than 100 lines) are less likely to benefit from a demand-driven analysis since the savings of information collection in demand-driven analysis

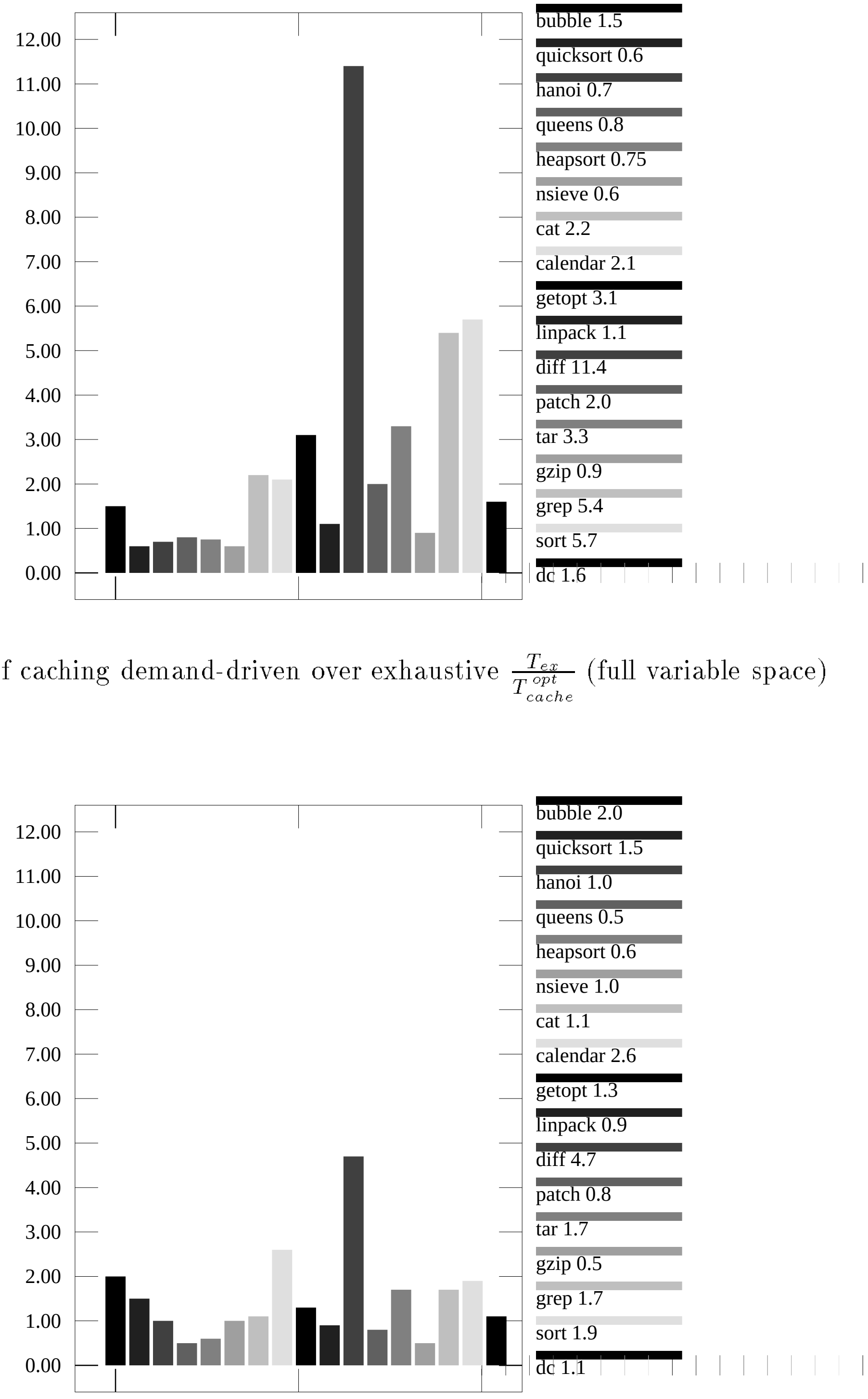
Non-Caching Demand-Driven Analysis (Du-chain)			**Savings**	
program	time (secs) T^{opt}	space (Kbytes) S^{opt}	speedup $\frac{T_{ex}}{T^{opt}}$	space: $\frac{S^{opt}\times 100}{S_{ex}}$
bubble	0.03 (0.02)	9.996	1.0 (1.0)	95.2%
quicksort	0.07 (0.03)	14.256	0.4 (1.0)	97.8%
hanoi	0.03 (0.02)	11.516	0.7 (1.0)	94.2%
queens	0.06 (0.02)	14.612	0.7 (1.0)	95.2%
heapsort	0.17 (0.15)	20.196	0.5 (0.4)	88.7%
nsieve	0.10 (0.05)	19.304	0.3 (0.8)	95.6%
cat	0.11 (0.10)	37.060	1.8 (0.8)	85.3%
calendar	0.07 (0.05)	64.684	2.4 (1.6)	78.7%
getopt	0.64 (0.60)	82.468	1.5 (0.6)	78.3%
linpack	0.86 (0.73)	203.756	0.6 (0.4)	89.7%
diff	1.09 (0.90)	184.856	6.3 (2.5)	59.4%
patch	1.58 (1.43)	159.548	1.3 (0.5)	69.2%
tar	1.87 (1.75)	200.204	2.3 (1.2)	61.3%
gzip	2.84 (2.52)	336.348	0.5 (0.3)	64.0%
grep	1.16 (1.05)	289.552	3.9 (1.2)	66.2%
sort	1.27 (1.08)	352.600	4.6 (1.7)	66.3%
dc	1.04 (0.80)	326.940	1.1 (0.8)	78.5%

Table 5.5: Accumulated demand-driven analysis time and space without caching (T^{opt} and S^{opt}), the speedup of the demand-driven analyzer without caching over the exhaustive analyzer and the demand-driven analyzer space utilization as a percentage of the exhaustive analysis space. Parenthesis indicate measurements that exclude temporaries.

does not outweigh the overhead of starting up repeated query propagations. Moreover, exhaustive analysis of very short programs is usually fast.

The speedup column in Table 5.4 shows in parentheses the speedups that result if temporary variables are excluded and only source code level variables are analyzed. These speedups are graphically displayed in Figure 5.9 (ii). 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 lower speedups. However, demand-driven analysis is still faster than exhaustive analysis in 12 out of the 17 programs.

Table 5.4 also shows the space savings of the demand-driven analyzer as a percentage of the exhaustive space. The demand-driven analysis requires less space to store data flow information in almost all programs. The lower space requirements are to be 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 summary computations. Again the savings of demand-driven analysis are not achieved for the very short programs. In these short programs the savings in summary computations did not outweigh the additional storage requirements for the generated



(i) Speedup of caching demand-driven over exhaustive $\frac{T_{ex}}{T^{opt}_{cache}}$ (full variable space)

(ii) Speedup of caching demand-driven over exhaustive $\frac{T_{ex}}{T^{opt}_{cache}}$ (temporaries excluded)

Figure 5.9: Caching (optimzied) demand-driven analysis vs exhaustive analysis.

queries at each node. Table 5.4 does not show the space consumption and space savings of the demand-driven analyzer if temporaries variables are not considered during the analysis. Excluding temporary variables results in a reduction in the space utilization of both the exhaustive analysis and the demand-driven analyzer. Since the space consumption is reduced in both the exhaustive and the demand-driven analyzer, the proportional space savings of the demand-driven analyzer over the exhaustive analyzer vary only insignificantly from the values shown in Table 5.4.

5.5.2 Experiment 2: Non-Caching Demand-Driven versus Exhaustive

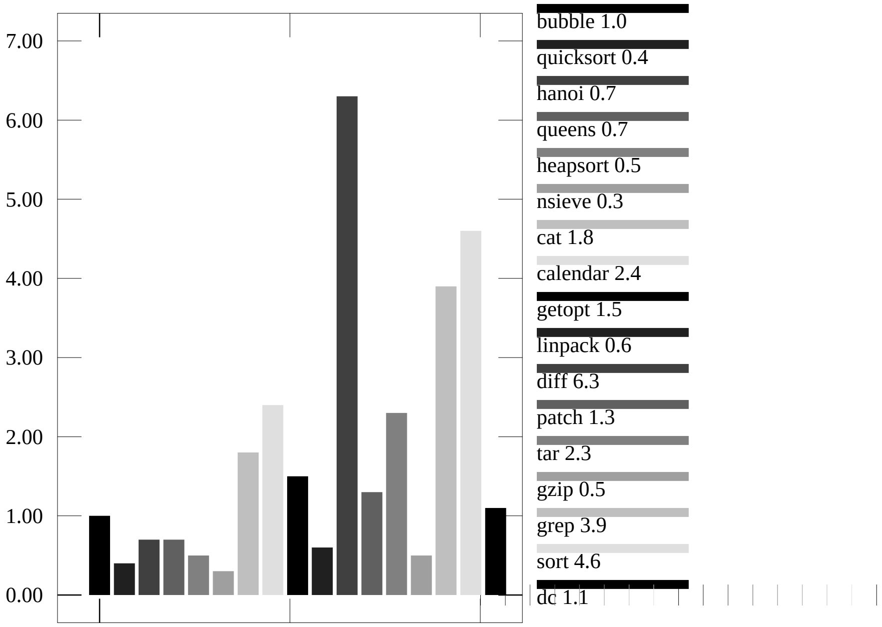
A second experiment was conducted to determine the effect of caching on the performance of the demand-driven analyzer. The non-caching demand-driven analysis (with query advancing) was executed with the same set of queries as in the first experiment. The accumulated analysis times T^{opt} are shown in Table 5.5. Table 5.5 also shows the accumulated space consumption S^{opt}. The speedups $\frac{T_{ex}}{T^{opt}}$ of the demand-driven analyzer over the exhaustive analyzer for both the full variable space and the source variable space are shown in Table 5.5 and are graphically displayed in Figures 5.10 (i) and (ii). As expected, disabling caching resulted in a slight slowdown of the demand-driven analyzer and, at the same time, in a slightly lower space utilization since no cache memory is allocated. A direct evaluation of the caching overhead is shown in Table 5.6. Table 5.6 shows the speedup $\frac{T^{opt}}{T^{opt}_{cache}}$ of the demand-driven analyzer with caching over the demand-driven analyzer without caching. Except for one of the short programs (queens), adding the caching capability resulted in moderate speedup factors of up to 2.6. The analysis of program queens resulted in too few cache hits, causing the savings to be less than the overhead of the cache management.

5.5.3 Experiment 3: Query Advancing

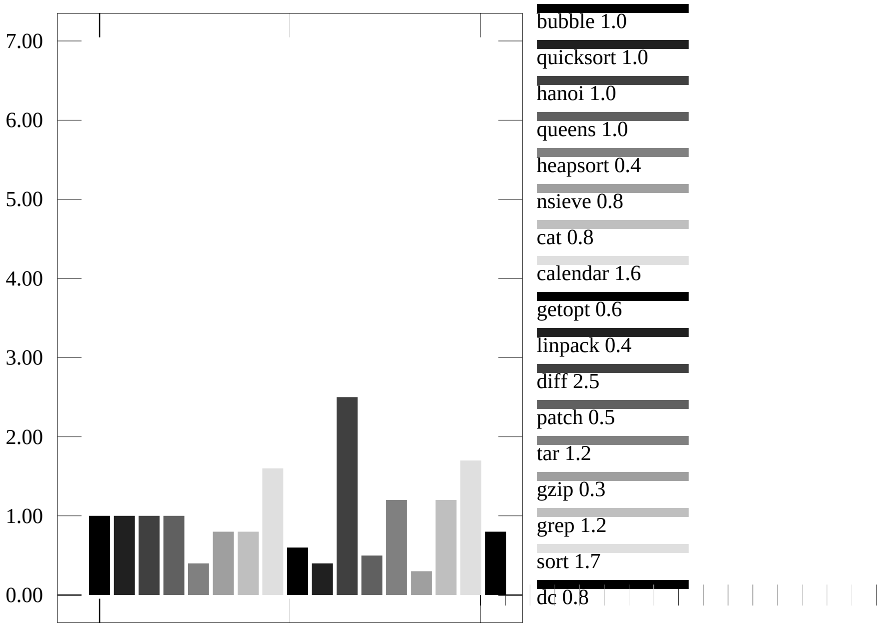
The third experiment evaluates the effects of query advancing. Query advancing was considered for both the caching and the non-caching demand-driven analyzer. The results are shown in Table 5.7 and Table 5.8.

Consider first the results for the caching demand-driven analyzer in Table 5.7. The first two columns show the accumulated demand-driven analysis time and space that result if query advancing is disabled. The analysis time and space measurements are accumulated over the same set of queries that was used in the first two experiments. The third column shows the speedup of the caching demand-driven analyzer with query advancing over the caching demand-driven analyzer without query advancing. The speedup measurements indicate that query advancing is worthwhile, resulting in speedups by factors of up to 1.9 in 14 out of 17 programs while essentially requiring no additional space. Since query advancing primarily applies to global variables, the speedups for the source level variables analysis tend to be higher than for the complete variables space analysis, where temporaries are included.

The evaluation of query advancing without caching yielded similar results as shown



(i) Speedup of non-caching demand-driven over exhaustive $\frac{T_{ex}}{T^{opt}_{cache}}$ (full variable space)



(ii) Speedup of non-caching demand-driven over exhaustive $\frac{T_{ex}}{T^{opt}_{cache}}$ (temporaries excluded)

Figure 5.10: Non-Caching (optimzied) demand-driven analysis vs exhaustive analysis.

Trade-off: Caching vs. Non-Caching (Du-chain)

program	speedup $\frac{T^{opt}}{T^{opt}_{cache}}$	% space overhead $\frac{(S^{opt}_{cache}\times 100)}{S^{opt}}$
bubble	1.5 (2.0)	105.5%
quicksort	1.4 (1.5)	105.7%
hanoi	1.0 (1.0)	107.5%
queens	1.2 (0.5)	106.0%
heapsort	1.4 (1.5)	111.0%
nsieve	2.0 (1.2)	109.6%
cat	1.2 (1.4)	113.1%
calendar	0.9 (1.6)	108.1%
getopt	2.0 (2.1)	121.2%
linpack	1.8 (2.2)	109.3%
diff	1.8 (1.8)	135.1%
patch	1.6 (1.5)	126.5%
tar	1.5 (1.4)	133.2%
gzip	1.5 (1.5)	124.6%
grep	1.4 (1.3)	126.1%
sort	1.2 (1.1)	125.9%
dc	1.5 (1.3)	114.2%

Table 5.6: The accumulated speedup of the demand-driven analyzer with caching over the demand-driven analyzer without caching and the space overhead of the caching demand-driven analysis as a percentage of the space used by the non-caching demand-driven analyzer.

Table 5.8. Query advancing resulted in speedups by factors of up to 2 in 13 out of 17 programs. As for the caching analyzer, the speedups were higher in most programs when the analysis excluded temporary variables.

Overall, including query advancing in the demand-driven analysis is shown to be worthwhile, resulting in speedups in almost all programs for both the caching and the non-caching analyzer versions.

5.5.4 Summary

The experimental results demonstrate that demand-driven analysis performs well in practice. The first experiment showed that demand-driven analysis computes du-chains faster than exhaustive analysis in the majority of cases (in 11 out of 17 programs). In 12 out of 17 programs demand-driven analysis also uses less space for storing data flow information than exhaustive analysis. Importantly, the speedups and space savings of the demand-driven analysis over the exhaustive analysis result even when du-chains are computed over the entire program. Naturally, the benefits of using a demand-driven approach would be

Query Advancing (Du-chain) - Caching

program	time T	space S	speedup $\frac{T}{T^{opt}}$	space: $\frac{(S^{opt} \times 100)}{S}$
bubble	0.02 (0.02)	10.564	1.0 (2.0)	100.1%
quicksort	0.03 (0.03)	15.524	0.6 (1.5)	102.9%
hanoi	0.03 (0.02)	12.700	1.0 (1.0)	102.5%
queens	0.06 (0.03)	16.456	1.2 (0.7)	106.2%
heapsort	0.14 (0.12)	22.396	1.2 (1.2)	99.8%
nsieve	0.04 (0.05)	21.608	0.8 (1.2)	102.1%
cat	0.17 (0.13)	42.980	1.9 (1.8)	102.5%
calendar	0.14 (0.14)	79.692	1.7 (4.6)	114.0%
getopt	0.33 (0.27)	100.580	1.1 (0.96)	100.5%
linpack	0.46 (0.33)	222.432	0.9 (1.0)	99.8%
diff	0.84 (0.73)	256.004	1.4 (1.5)	102.5%
patch	1.17 (1.13)	203.480	1.2 (1.2)	100.7%
tar	1.70 (1.62)	268.280	1.3 (1.35)	100.5%
gzip	2.92 (2.81)	424.380	1.6 (1.7)	98.8%
grep	1.39 (1.25)	387.872	1.7 (1.8)	106.2%
sort	1.35 (1.17)	463.280	1.3 (1.2)	104.3%
dc	1.13 (0.62)	390.800	1.6 (1.1)	104.6%

Table 5.7: The accumulated caching demand-driven analysis time and space without query advancing (T and S), the speedup of the caching demand-driven analyzer with query advancing over the caching demand-driven analyzer without query advancing and the space utilization of the caching demand-driven analyzer with query advancing as a percentage of the space used by the caching demand-driven analyzer without query advancing. Parentheses indicate measurements that exclude temporaries.

even higher if only a fraction of the complete set of du-chains were needed.

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 compute all du-chains. If only a fraction of the du-chains in a program is needed, caching is likely to be less beneficial since there may not be sufficiently many cache hits to compensate for the cache management overhead.

The third experiment evaluated the benefits of query advancing and showed that query advancing is worthwhile in that it enabled speedups in 14 our of 17 programs despite the additional overhead of computing the summary information. However, the speedups that resulted from query advancing were moderate (less than a factor of 2 in all cases). Thus, query advancing does not provide a significant improvement in the demand-driven du-chain analysis.

An additional inspection of the benchmark programs with the highest and lowest speedups was carried out in order to identify the program characteristics that mostly affected the an-

Query Advancing (Du-chain) - Non-Caching						
program	time T		space S	speedup $\frac{T}{T^{opt}}$		space: $\frac{(S^{opt} \times 100)}{S}$
bubble	0.03	(0.03)	9.964	1.0	(1.5)	99.6%
quicksort	0.06	(0.03)	14.660	0.9	(1.0)	102.8%
hanoi	0.04	(0.02)	11.836	1.3	(1.0)	102.7%
queens	0.08	(0.09)	15.640	1.3	(4.5)	107.0%
heapsort	0.14	(0.14)	20.156	0.8	(0.9)	99.8%
nsieve	0.06	(0.06)	19.760	0.6	(1.2)	102.3%
cat	0.17	(0.15)	38.148	1.5	(1.5)	102.9%
calendar	0.14	(0.13)	69.612	2.0	(2.6)	107.6%
getopt	0.69	(0.59)	83.060	1.1	(0.9)	100.7%
linpack	0.85	(0.66)	203.472	0.9	(0.9)	99.8%
diff	1.20	(1.08)	185.956	1.1	(1.2)	100.5%
patch	1.72	(1.70)	160.928	1.1	(1.1)	100.8%
tar	2.23	(2.12)	201.560	1.2	(1.2)	100.6%
gzip	4.23	(3.83)	340.476	1.4	(1.5)	98.7%
grep	1.73	(1.69)	303.536	1.5	(1.6)	104.8%
sort	2.06	(1.84)	371.048	1.6	(1.7)	105.2%
dc	1.45	(0.80)	344.160	1.4	(1.0)	102.2%

Table 5.8: The accumulated non-caching demand-driven analysis time and space without query advancing (T and S), the speedup of the non-caching demand-driven analyzer with query advancing over the demand-driven analyzer without query advancing and the space utilization of the non-caching demand-driven analyzer with query advancing as a percentage of the space used by the non-caching demand-driven analyzer without query advancing. Parentheses indicate measurements that exclude temporaries.

alyzers' 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 in REACH depends primarily on reference locality properties. If variables are defined and used in nearby statements the propagation paths are short. The following program characteristics could be identified as having a direct impact on the analyzers' performances.

- *Program size.* The speedup of demand-driven analysis over exhaustive analysis tends to increase with the length of the program. Reference locality properties usually do not depend on the program size, so that the average length of query propagation paths does not grow at same rate as the program length. Thus, higher speedups are likely for large programs, since the average query propagation time may not change much with program size while the cost of exhaustive analysis does.

- *Nesting depth of control structures:* Programs with deeply nested control structures may generate long query propagation paths and are generally more expensive to analyze than straight-line code. However, the depth of control structures negatively affects the performance of both demand-driven and exhaustive analysis.

- *Number of global variables:* The length of query propagation paths for local variables and temporaries is bounded by the size of the procedure in which the respective queries are raised. In contrast, the query propagation paths for global variables may extend across several procedures. Thus, a large number of global variables may result in more queries with long propagation paths. However, global variables also give opportunities for query advancing across calls and, thus, also contribute to the speedup of demand-driven analysis.

- *Number of procedures:* In contrast to exhaustive analysis, demand-driven analysis with query advancing benefits from large numbers of procedures. Query advancing allows the analysis to skip complete procedures which would not be possible if procedures were, for example, in-lined.

- *Structure of the call graph:* The structure of the call graph determines the maximal length of query propagation paths for global variables. In the best case, the call graph structure is a two-level tree, i.e., every procedure is called only once from the main procedure. In this case, the propagation paths for a global variable can not extend beyond the length of two procedures resulting in fast query evaluations. Cycles in the call graph, i.e., recursion, can create long query propagation paths. However, recursion also negatively impacts on the performance of exhaustive analysis.

- *Density of the call graph:* The density of the call graph, that is, the average number of calls per procedure has a negative effect on both demand-driven and exhaustive analysis. A high number of calls may trigger additional procedure summary computations. In demand-driven analysis, long call chains in very dense call graphs can create long query propagation paths for global variables.

Exceptionally high speedups of demand-driven analysis over exhaustive analysis, as for example in program diff, result in programs that combine several of the speedup supporting program characteristics. However, the presence of these characteristics in a program are not sufficient to guarantee a speedup. Some of the above program characteristics have both negative and positive effects on the demand-driven analyzer's performance (e.g., number of global variables, density of the call graph). Thus, the above listing identifies trends and further studies with larger program sets would be necessary to provide complete insights into the impact of program characteristics on the analyzers' performance.

Chapter 6

A Demand-Driven Analyzer for Copy Constant Propagation

This chapter continues the practical evaluation of the demand-driven approach by presenting experimentation with a demand-driven analyzer for copy constant propagation (CCP). CCP analysis is more complex than the analysis of Gen-Kill problems. Like Gen-Kill problems, CCP is a distributive problem. However, CCP is not partitionable and is therefore more costly to analyze exhaustively than a Gen-Kill problem. The experimentation presented in this chapter shows that the speedups of demand-driven CCP analysis over exhaustive CCP analysis are even higher than in the experiments with the REACH analyzers. These results support the hypothesis that the more expensive an analysis is the higher are the benefits of reducing the analysis effort with a demand-driven approach.

This chapter is organized as follows. Section 6.1 briefly reviews the formal definition of CCP, which was already introduced in Chapter 3. Section 6.2 describes the instance of the demand-driven framework for CCP along with an analysis of its asymptotic cost. The experiments with the CCP analysis are reported in Section 6.3.

6.1 Copy Constant Propagation

Recall that the lattice in CCP for a program with k variables is a product lattice L^k. 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 component value x_i is either $\top$ (undefined), $\bot$ (any integer) or any of the constant literals c that occur in the program text.[1] A base element in L^k is of the form $[v_i = c]$ denoting a lattice element $(x_1, \ldots, x_k)$ with a single non-bottom component x_i: $x_i = c$ and $x_j = \bot$ for $j \neq i$.

The flow functions in CCP and the reverse flow functions, first shown in Chapter 3 are restated in Table 6.1 for various types of statements.

[1] See Figure 3.1 for a display of the lattice in CCP.

statement at n	flow function $f_n(x)_j$, where $x = (x_1, \ldots, x_k)$	reverse flow function $f_n^r([v_i = c_1])$, where c_i is some constant
$v_i := c$	$f_n(x)_j = \begin{cases} c & \text{if } i = j \\ x_j & \text{otherwise} \end{cases}$	$f_n^r([v_i{=}c_1]) = \begin{cases} - & \textit{if } i{=}j \textit{ and } c_1{=}c_2 \\ \top & \textit{if } i{=}j \textit{ and } c_1{\neq}c_2 \\ [v_i{=}c_1] & \textit{otherwise} \end{cases}$
$v_i := v_l$	$f_n(x)_j = \begin{cases} x_l & \text{if } i = j \\ x_j & \text{otherwise} \end{cases}$	$f_n^r([v_i{=}c_1]) = \begin{cases} [v_l{=}c_1] & \textit{if } i{=}j \\ [v_i{=}c_1] & \textit{otherwise} \end{cases}$
$v_i := expr.$ $read(v_i)$	$f_n(x)_j = \begin{cases} - & \text{if } i = j \\ x_j & \text{otherwise} \end{cases}$	$f_n^r([v_i{=}c_1]) = \begin{cases} \top & \textit{if } i{=}j \\ [v_i{=}c_1] & \textit{otherwise} \end{cases}$

Table 6.1: Flow functions and reverse flow functions for CCP.

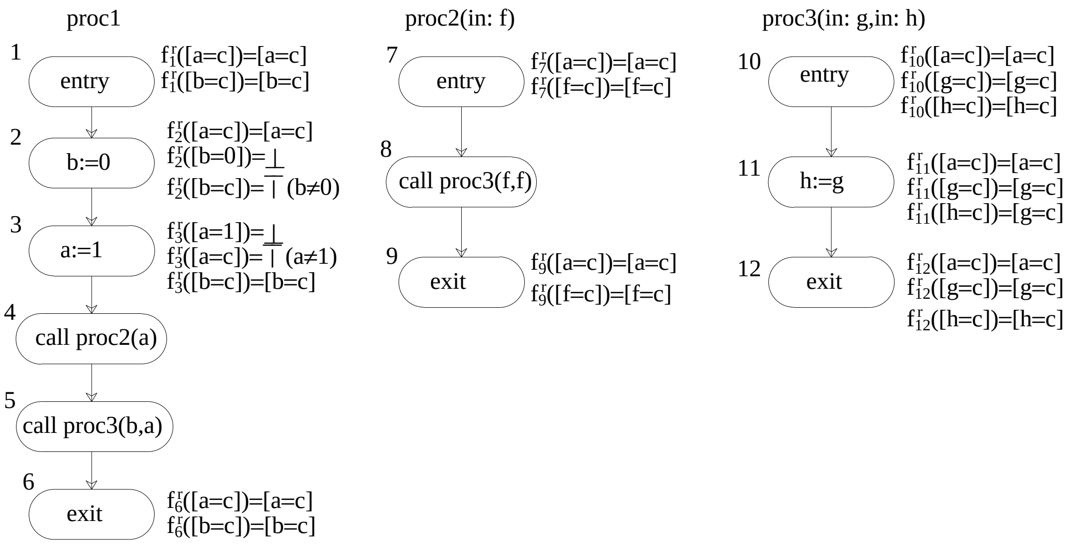
Example: Consider the program example in Figure 6.1. The program consists of three procedures *proc1, proc2* and *proc3*. The address spaces of the three procedures are $Addr(proc1) = \{a, b\}$, $Addr(proc2) = \{a, f\}$ and $Addr(proc3) = \{a, g, h\}$. The lattice elements for procedure *proc1* are tuples (x_a, x_b), where x_a and x_b denote the lattice values for the global a and the local variable b. Similarly, the lattice elements for procedures *proc2* and *proc3* are tuples of the form (x_a, x_f) and (x_a, x_g, x_h), respectively. The CCP solution on entry of each node is shown in Figure 6.1. Figure 6.1 also shows the reverse flow functions next to each non-call site node in the control flow graph.

6.2 A Framework Instance for CCP

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. As in the previous chapter, the analysis for CCP assumes C-style programs with global and local variables and procedures with value parameters.

According to the general framework, a query q in CCP asks for a specific lattice element, i.e., a specific constant value c of a variable v at a node n. For example, the query $q =< [v = 0], n >$ raises the question: "Is variable v a copy constant at node n with value 0?". Using this query format, queries with respect to each constant literal may be nec-

```
declare a; /* global */
procedure proc1              procedure proc2(in: f))      procedure proc3(in: g,in: h)
declare b; /* local */       begin                        begin
begin                            call proc3(f,f);             h:=g;
    b:=0;                    end                          end
    a:=1;
    call proc2(a);
    call proc3(b,a);
end
```



CCP solution (on entry)			
proc1		*proc2*	
n	(x_a, x_b)	n	(x_a, x_f)
1	$(-,-)$	7	$(1,1)$
2	$(-,-)$	8	$(1,1)$
3	$(-,0)$	9	$(1,1)$
4	$(1,0)$	*proc3*	
5	$(1,0)$	n	(x_a, x_g, x_h)
6	$(1,0)$	10	$(1,-,1)$
		11	$(1,-,1)$
		12	$(1,-,-)$

Reverse summary functions			
proc2			
n	$\phi^r_{(n,9)}([a=c])$	$\phi^r_{(n,9)}([f=c])$	
7	$[a=c]$	$[f{=}c]$	
8	$[a=c]$	$[f{=}c]$	
9	$[a=c]$	$[f{=}c]$	
proc3			
n	$\phi^r_{(n,12)}([a=c])$	$\phi^r_{(n,12)}([g=c])$	$\phi^r_{(n,12)}([h=c])$
10	$[a=c]$	$[g{=}c]$	$[g{=}c]$
11	$[a=c]$	$[g{=}c]$	$[g{=}c]$
12	$[a=c]$	$[g{=}c]$	$[h{=}c]$

Figure 6.1: Example for CCP

essary to determine whether a variable has constant value. Generating such a potentially high number of queries is not only costly, it is actually unnecessary. The propagation of multiple queries $< [v = 0], n >, < [v = 1], n >, \ldots$ that only differ in their constant value is identical except for the response upon termination. Thus, these queries can be combined into a single query of the form:"Is variable v a copy constant at node n?". The combined query is written as:

$$q =< [v = c], n >,$$

where c represents an unknown but fixed constant value.

The specialized query propagation rules are shown in Figure 6.2 (i). The definition of the reverse summary function is shown in Figure 6.2 (ii). Reverse summary functions are computed for global variables only since in programs with value parameters side effects of procedure execution can only affect global variables.[2] Since the execution of a procedure p has no effect on variables that are local to the calling procedure, the reverse summary function for local variables is simply the identity function.

6.2.1 Demand-Driven Algorithm for CCP

The specialization of the generic query algorithm *Query* for CCP is shown in Figure 6.3. Procedure *Query_CCP* takes as input a query of the form "Is variable v a copy constant at node n?". Query $< [v = c], n >$ evaluates to *true* at node n if n assigns *any* constant to variable v, in which case the constant value is remembered. If all generated queries evaluate to *true* the join over the remembered constant values is examined. If this join yields a constant, the constant value is returned. Otherwise, the returned response is *false*.

The corresponding instance of the generic procedure $CCP\phi^r$, shown in Figure 6.4, partially evaluates the reverse summary function equation system from Figure 6.2 (ii).

Example: Procedure *Query_CCP* is illustrated using the program example in Figure 6.1 for the query $q =< [h = c], 10 >$ raising the question as to whether the formal h of procedure *proc3* is a copy constant on entry of each invocation of *proc3*. Initially, worklist = $\{10\}$ and $query[10] = [h = c]$. $Query[10]$ is propagated to queries for the corresponding actual parameters at call sites resulting in: $query[5] = [a = c]$ and $query[8] = [f = c]$. Processing $query[8]$ causes the propagation of $[f = c]$ to node 7 and in turn to actual parameters at the call site at node 4, i.e., $query[4] = [a = c]$. Assume $query[5]$ is propagated next across the call to procedure *proc2* at node 4. Since a is global, the reverse summary function value $\phi^r_{(7,9)}([a = c])$ is determined. Since $\phi^r_{(7,9)}([a = c]) = [a = c]$, the query $< [a = c], 5 >$ propagates to node 4: $query[4] = [a = c]$. Applying the reverse function at node 3 yields $f^r_3([a = c]) = \perp$. Thus, when propagating the query through node 4, it evaluates to *true* and 1 is remembered as the actual constant assigned. Since the worklist is exhausted, the

[2] In programs with reference parameters, reverse summary functions are also computed for formal parameters.

$$\text{(i)}\quad \begin{aligned} &< \bot, n > &\Longleftrightarrow\quad & \mathit{true} \\ &< \top, n > &\Longleftrightarrow\quad & \mathit{false} \end{aligned}$$

(ii) For each procedure p:

$$< [v = c], entry_p > \Longleftrightarrow \begin{cases} \mathit{false} & \text{if } p \text{ has no call sites or } v \in Local(p) \\ \displaystyle\bigwedge_{call(m)=p} < [b_m^{-1}(v) = c], m > & \text{otherwise} \end{cases}$$

(iii) For each non-entry node n:

$$< [v = c], n > \Longleftrightarrow \bigwedge_{m \in pred(n)} \begin{cases} < f_m^r([v = c], m > & \text{if } m \text{ is not a call site} \\ < \phi^r_{(entry_q, exit_q)}([v = c]), m > & \text{if } call(m) = q \text{ and } v \text{ is global} \\ < [v = c], m > & \text{otherwise} \end{cases}$$

(i)

For each procedure p and global v:

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

For each node $n \neq exit_p$ in p and global v:

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

(ii)

Figure 6.2: Specialized propagation rules (i) and reverse summary functions (ii) for CCP.

```
Procedure $Query\_CCP(v, n)$
input:   a variable $v$ and a node $n$
output: the constant value $c$ if $v$ is a copy constant at $n$, otherwise $false$
begin
for each $m \in N$ do $query[m] \leftarrow \emptyset$;
$query[n] \leftarrow [v]$ ; $worklist \leftarrow \{(v, n)\}$; $val = -$;
while $worklist \neq \emptyset$ do
    remove a pair $(w, m)$ from $worklist$ and let $p$ be the procedure containing $m$;
    case $m = entry_{main}$:
        return($false$);
    case $m = entry_q$ for some procedure $q$:
        for each call site $m'$ such that $call(m') = q$ do
            if $b_{m'}^{-1}(w) \neq \emptyset$ then /* if $w$ is not local to $q$ */
            $query[m'] \leftarrow query[m'] \cup \tilde{b}_m^{-1}(query[m])$;
            if $query[m']$ changed then add $(b_{m'}^{-1}(w), m')$ to $worklist$;
        endfor;
    otherwise:
        for each $m' \in pred(m)$ do
    $new \leftarrow \begin{cases} f_{m'}^r(query[m]) & \text{if } m' \text{ is not a call site} \\ b_{m'}^{-1}(CCP\phi^r(q, b_{m'}(w), val) & \text{if } call(m') = q \text{ and } w \in Global \\ query[m] & \text{if } m' \text{ is a call site and } w \notin Global \end{cases}$
            let $newvar$ be the variable named in $new$;
            if $new = -$ and $m'$ is not a call site then /* $m'$ must assign a sonstant value $c$ tro $w$ */
                $val \leftarrow val \sqcup c$, where $c$ is the const. assigned at $m'$;
            if( $new = \top$) or ($val = \top$ )then return( $false$ )
            else if $new \sqsupset -$ then /* query still unresolved */
                $query[m'] \leftarrow query[m'] \sqcup new$;
                if $query[m']$ changed then add $(newvar, m')$ to $worklist$ ;
            endif;
        endfor;
endwhile;
if $val < \top$ then return($val$) else return($false$ );
end
```

Figure 6.3: Demand-driven algorithm for CCP.

```
Procedure $CCP\phi^r(p, y, val)$
input:  procedure $p$, variable $y$ and variable $val$ to hold const. value
output: summary value $\phi^r_{(r_p,e_p)}(y)$.
begin
    $worklist \leftarrow \emptyset$; $res \leftarrow -$;
    let $y = [v_1, \ldots, v_k]$, where $v_i \in Addr(p)$;
    for each $v_i$, where $1 \leq i \leq k$ do
      if $M[e_p, v_i] = -$ then add $(e_p, v_i)$ to $worklist$;
        $M[e_p, v_i] = [v_i]$; endif;
    while $worklist \neq \emptyset$ do
      remove a pair $(n, w)$ from $worklist$and let $p'$ be the proc. containing $n$;
      let $[w_1, \ldots, w_j] = M[n, w]$;
      case $n \in N_{call}$ and $call(n) = q$:
        for each $w_i$, where $1 \leq i \leq j$ do
          if $w_i \in Global$ or $w_i$ is an actual param. at $n$ then
            for each $z \in b_n([w_i])$ do
              if $M[e_q, z] = [z]$ then
                for each $m \in pred(n)$ do $Propagate(m, w, b_n^{-1}(M[r_q, z]))$;
                if $M[r_q, z] = -$ then $M[p', w].val \leftarrow M[p', w].val \sqcup M[q, z].val$;
              else $M[e_q, z] \leftarrow [z]$; add $(e_q, z)$ to $wor\ klist$; endif;
            endfor;
          else /* skip call site if u not passed */
            for each $m \in pred(n)$ do $Propagate\ (m, w, [w_i])$;
        endfor;
      case $n = r_q$ for some procedure $q$:
        for each $m \in N_{call}$ such that $call(m) = q$ and $b_m^{-1}([w]) \in M[m, z]$ for some $z$ do
          let $p''$ be the proc. containing $m$;
          for each $m' \in pred(m)$ do $Propagate(m', z, b_m^{-1}(M[n, w])$;
          if $M[r_q, w] = -$ then $M[p'', z].val \leftarrow M[p'', z].val \sqcup M[q, w].val$;
      otherwise:
        for each $m \in pred(n)$ do
            $Propagate(m, w, f_m^r(M[n, w])$;
        if $f_n^r(M[n, w]) = -$ then
            $M[p', w].val \leftarrow M[p', w].val \sqcup c$, where $c$ is the const. assigned at $m$;
    endwhile;
    for each $v_i$, where $1 \leq i \leq k$ do
      $res \leftarrow res \sqcup M[r_p, v_i]$; $val \leftarrow val \sqcup M[p, v_i].val$; endfor;
    return($res$);
end
```

```
Procedure $Propagate(n, v, new)$ /* propagate new to $M[n, v]$ */
input:  node $n$, variable $v$ and set of variables $new$
begin
    $M[n, v] \leftarrow M[n, v] \sqcup new$;
    if $M[n, v]$ changed then add $(n, v)$ to $worklist$; endif
end ;
```

Figure 6.4: Procedure $CCP\phi^r(p, y, val)$ for CCP.

overall response is 1, indicating that the formal h of procedure q always has the value 1 on entry of procedure q.

6.2.2 Asymptotic Cost

Consider the execution time of procedure *Query_CCP*. At each node, at most *MaxAddr* queries can be generated, where *MaxAddr* is the size of the maximal address space in any procedure. Thus, during an invocation of *Query_CCP* a total of $O(\mathit{MaxAddr} \times |N|)$ queries can be generated resulting in $O(\mathit{MaxAddr} \times |N|)$ join and reverse function applications in procedure *Query_CCP*.

Now consider the execution time of procedure $\mathit{CCP}\phi^r$. By the distributivity of the reverse summary functions, it is sufficient to maintain table entries only for base elements resulting in $|\mathit{Global}| \times |N|$ entries.[3] Each entry may contain a set of base elements and is therefore of size *MaxAddr*. To keep track of the actual constant values encountered, the table M includes an extra field $M[p, v].\mathit{val}$ for each procedure p and each variable v. The fixed point computation of table entries requires in the worst case $O(|\mathit{Global}| \times \mathit{MaxAddr} \times |N|)$ table updates. As in the general case, each table update may trigger up to *MaxCall* join and/or reverse function applications, where *MaxCall* is the maximal number of call sites calling a single procedure. Assuming join and reverse function applications are performed pointwise, each join or function application requires $O(|MaxAddr|)$ time resulting in the total time of $O(\mathit{MaxCall} \times |\mathit{Global}| \times \mathit{MaxAddr}^2 \times |N|)$ for procedure $\mathit{GenKill}\phi^r$. Thus, the overall time requirements are $O(\mathit{MaxCall} \times |\mathit{Global}| \times \mathit{MaxAddr}^2 \times |N|)$.[4]

As in the example of Gen-Kill analysis in Chapter 5, the specialization of the framework instance to CCP yields a significantly more efficient algorithm than a straightforward adoption of the generic algorithm. The asymptotic time complexity of the generic algorithm *Query* depends on the size of lattice. The lattice in CCP has $O((l + 2)^{MaxAddr})$ elements, where l is number of constant literals in the program text. Thus, a straightforward adoption of the generic algorithm to CCP would result in an exponential time algorithm.

6.2.3 Query Advancing

The query propagation in procedure *Query_CCP* can be optimized using the same type of query advancing techniques as described for Gen-Kill problems in Chapter 5. Similar to the REACH analyzer, the generation of data flow information in CCP is based on definitions of variables. Thus, the flow-insensitive procedure summary sets *DMOD*(p) that were used to enable query advancing in reaching definitions can also be used to enable query advancing in CCP.

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

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

- **Advancing across calls**
  Propagating a query $q =< [v = c], n >$ for a global variable v across a call site $m \in call(p)$ requires summary information only if $v \in Mod(p)$. Otherwise, q can be directly forwarded across the call:

- **Advancing to entry**
  Consider the propagation of a query $q =< [v = c], entry_p >$ for a variable v into a procedure r that calls p. If $v \notin Mod(r)$ then q can be directly forwarded to $entry_r$

6.3 Experiments

An experimental study was conducted to evaluate the practical benefits of computing copy constant information using demand-driven analysis. The first experiment compares the performance of demand-driven CCP analysis with that of a standard exhaustive algorithm. The second experiment evaluates the benefits of caching and the third experiment examines the benefits of query advancing.

The following three algorithms were implemented:

(CACHE) Caching demand-driven CCP algorithm as described in the previous section with the option of query advancing.

(DD) A non-caching version of the demand-driven CCP algorithm with the option of query advancing.

(EX) An exhaustive CCP algorithm that is based on interprocedural analysis framework by Sharir and Pnueli [SP81].

The three algorithms were implemented as part of the PDGCC compiler project. The experimental evaluation of the three CCP analysis algorithms was carried out under the same conditions as the experimentation with du-chain analysis. For a detailed description of the implementation and experimentation context, including a description of the 17 C benchmarks that served as input to the analysis algorithms, see Section 5.5. As in Chapter 5, the experiments were run in two versions: one version that considers the complete variables space including compiler generated temporaries and one version that considers only source-level variables.

6.3.1 Experiment 1: Caching Demand-Driven versus Exhaustive

The first experiment compares the performance of the caching demand-driven CCP analyzer with that of the exhaustive analyzer. The experiment was carried out under the assumption that copy constant information for a variable is required only at program points that contain a reference of the variable. Thus, the demand-driven analyzer is applied to a set of queries

Exhaustive Analysis (CCP)			Queries (CCP)		
program	time (secs) T_{ex}	space (Kbytes) S_{ex}	program	queries	constant
bubble	0.08 (0.04)	66.828 (28.508)	bubble	127 (51)	11 (11)
quicksort	0.17 (0.10)	190.656 (77.288)	quicksort	170 (76)	9 (9)
hanoi	0.05 (0.03)	75.764 (34.060)	hanoi	120 (63)	7 (7)
queens	0.10 (0.04)	145.324 (41.068)	queens	181 (82)	15 (15)
heapsort	0.71 (0.10)	382.340 (69.220)	heapsort	230 (138)	29 (25)
nsieve	0.33 (0.18)	282.756 (125.78)	nsieve	199 (138)	51 (47)
cat	0.25 (0.10)	334.800 (110.400)	cat	324 (121)	16 (16)
calendar	0.26 (0.09)	463.432 (119.144)	calendar	507 (102)	20 (10)
getopt	2.44 (0.71)	2,003.272 (338.648)	getopt	459 (203)	18 (16)
linpack	2.80 (0.61)	3,556.032 (469.520)	linpack	2014 (909)	112 (109)
diff	3.88 (0.99)	5,344.124 (1,342.140)	diff	1055 (283)	76 (68)
patch	93.53 (51.96)	24,459.728 (7,691.752)	patch	1146 (446)	94 (83)
tar	13.44 (6.91)	9,518.464 (4,813.808)	tar	1046 (335)	42 (32)
gzip	62.16 (31.55)	49,777.040 (21,696.240)	gzip	2101 (714)	135 (91)
grep	3.45 (1.52)	4,277.056 (1,679.680)	grep	1481 (495)	157 (65)
sort	19.93 (7.20)	12,541.632 (2,854.096)	sort	1930 (589)	92 (75)
dc	14.35 (9.69)	10,296.716 (3,673.300)	dc	2325 (784)	49 (46)

Table 6.2: Exhaustive analysis time (T_{ex}) and space (S_{ex}) and the number of queries and the number of constants found for each benchmark. Parentheses indicate measurements that exclude temporaries.

that contains one query for each occurrence of a scalar variable in the program. The order in which these queries were processed was chosen randomly.

Table 6.3 shows the number of queries generated for each program and the number of constants that were discovered. The parentheses indicate the corresponding numbers for the source-level analysis that excludes temporaries. Unlike the problem of du-chain computation, excluding temporary variables from CCP analysis affects the solution since temporaries with constant values may be used to define other variables. As shown in Table 6.3, fewer constants may result. However, local analysis may be sufficient to reveal constant temporaries. The study, therefore, reports analysis times for both the complete variable space and the source-level variable space.

Table 6.2 shows for each program the analysis time T_{ex} and the space consumption S_{ex} of the exhaustive CCP analyzer. The demand-driven analysis time T^{opt}_{cache} accumulated over all queries is shown in Table 6.3. The analysis T^{opt}_{cache} is based on the caching version of the demand-driven analyzer with query advancing. Table 6.3 also shows the accumulated space

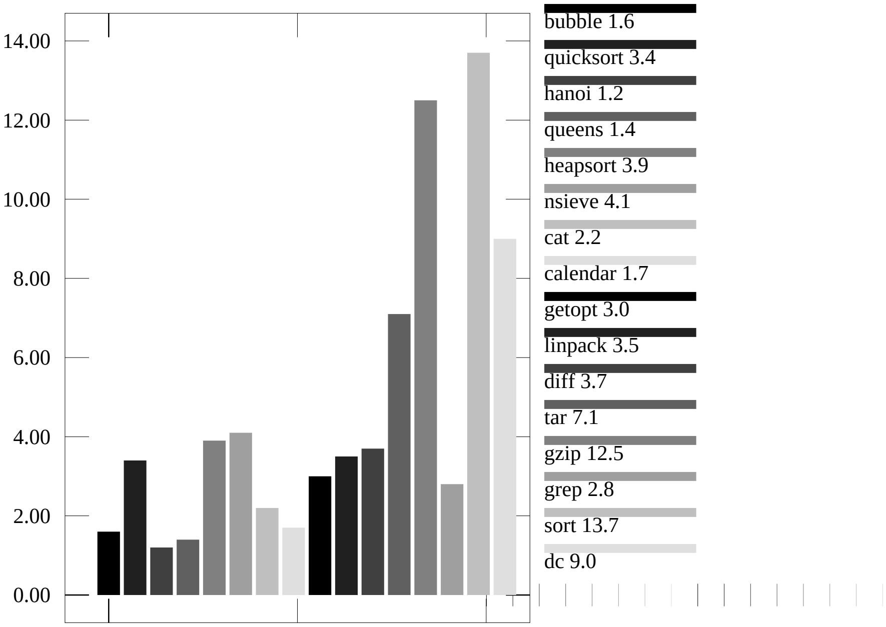
Caching Demand-Driven Analysis (CCP)				**Savings**	
program	time (secs)	space (Kbytes)	cache fill	speedup	space: % of S_{ex}
	T^{opt}_{cache}	S^{opt}_{cache}		$\frac{T_{ex}}{T^{opt}_{cache}}$	$\frac{S^{opt}_{cache} \times 100}{S_{ex}}$
bubble	0.05 (0.02)	66.492	14 (33%)	1.6 (2.0)	99.4%
quicksort	0.05 (0.04)	85.512	15% (33%)	3.4 (2.5)	44.8%
hanoi	0.04 (0.03)	43.736	16% (35%)	1.2 (1.0)	57.7%
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%

Parentheses indicate measurements that exclude temporaries.

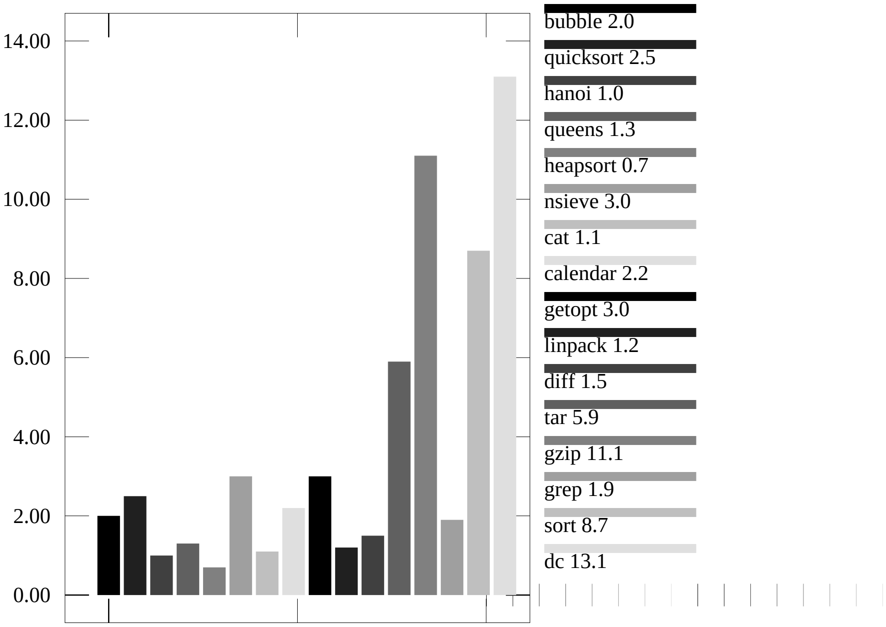
Table 6.3: Accumulated demand-driven analysis time and space with caching (T^{opt}_{cache} and S^{opt}_{cache}), the cache fill, the speedup of the demand-driven analyzer with caching over the exhaustive analyzer and the demand-driven analyzer space utilization as a percentage of the exhaustive analysis space. Parentheses indicate measurements that exclude temporaries.

consumption and the *cache fill*. The cache fill values in Table 6.3 show that the portion of the exhaustive solution that is actually needed to compute copy constant information is even smaller than in du-chain analysis. When considering the complete variable space (i.e., including temporaries), the relevant portion ranges from 4% to only 18%. When excluding temporaries the relevant portion is higher, ranging from 7% to 54%. As for du-chain analysis, the remaining unneeded portion of the solution consists of irrelevant copy constant information of variables that are no longer live in the containing procedure.

Fig. 6.5 (i) displays the speedups $\frac{T_{ex}}{T_{d/c}}$ of the demand-driven analyzer with caching over the exhaustive analyzer. The demand-driven analyzer computes copy constant information faster than the exhaustive analyzer in all programs with speedup factors ranging from 1.2 up to 44.3. The speedup for program patch is exceptionally high (44.3). A closer inspection of program patch revealed that the speedup is primarily due to the suppression of a large number of procedure summary computations that are performed in the exhaustive analysis.



(i) Speedup of caching demand-driven over exhaustive $\frac{T_{ex}}{T^{opt}_{cache}}$ (full variable space)



(i) Speedup of caching demand-driven over exhaustive $\frac{T_{ex}}{T^{opt}_{cache}}$ (temporaries excluded)

Figure 6.5: Caching (optimzied) demand-driven analysis vs exhaustive analysis.

Summary computations can be suppressed in the demand-driven analysis either by means of query advancing or because the query propagation paths are short and do not contain procedure calls.

The speedups and cache fill measurements that result when excluding temporary variables from the analysis are shown in parentheses in Table 6.3. Excluding temporaries resulted in a higher cache fill indicating that a larger fraction of exhaustive solution was computed. Corresponding to the higher cache fill, the speedups of the demand-driven analysis over the exhaustive analysis tend to be lower. However, except for one program, demand-driven analysis is still faster than exhaustive analysis with speedup factors ranging from 1.1 up to 44.7.

As in the experimentation with du-chain analysis, demand-driven CCP analysis requires less space to store data flow information in almost all programs. Table 6.3 shows the space savings of the demand-driven analyzer as the percentage of the exhaustive space requirements. Demand-driven analysis of program patch, which achieved the highest speedup, also resulted in the lowest space usage requiring only 17% of the exhaustive analysis space. The low space usage indicates that demand-driven analysis required less procedure summary computations which is also the primary reason for the observed speedup.

Full Solution on Demand

The performance of the demand-driven CCP analyzer was evaluated based on the assumption that copy constant information about a variable is needed only at the nodes that contain a use of that variable. This assumption is reasonable for applications in compiler optimizations and software tools since copy constant information is primarily used to simplify expressions (constant folding) or portions of code (i.e., procedure cloning).

Conceptually, the demand-driven analyzer may be used to retrieve copy constant information about any variable at any node. Although demand-driven analysis is not a suitable approach for retrieving exhaustive data flow solutions, an additional experiment was carried out to determine the worst case performance of demand-driven analysis if used to compute the complete exhaustive solution. The caching demand-driven analyzer with query advancing was executed on an exhaustive set of queries that contained one query for each variable at each node. The accumulated demand-driven analysis time T^{opt}_{cache} is shown in Table 6.4. Table 6.4 also shows the slowdown of the demand-driven analysis with respect to the exhaustive analysis and the demand-driven analysis' space requirements as a percentage of the exhaustive space requirements. Table 6.4 shows that, in the worst case (program calendar), the slowdown was by a factor of 22.8. Surprisingly, there are two cases (programs patch and gzip) where demand-driven analysis still performed better than exhaustive analysis although the full exhaustive solution was demanded. As previously discussed the primary cause for the high speedup for program patch is the suppression of summary computations that are performed in the exhaustive analysis. The suppression of summary computations

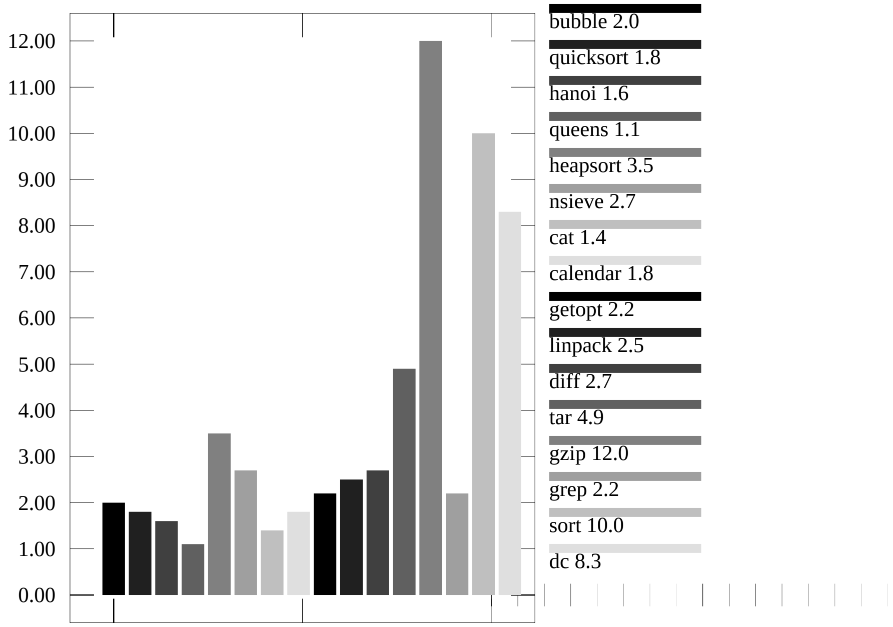
Demand-Driven Analysis: Full Solution (CCP)			**Overhead**	
program	time (secs) T^{opt}_{cache}	space (Kbytes) S^{opt}_{cache}	slowdown $\frac{T^{opt}_{cache}}{T_{ex}}$	space: % of S_{ex} $\frac{(S^{opt}_{cache} \times 100)}{S_{ex}}$
bubble	0.53 (0.15)	119.380	6.6 (3.7)	178.6%
quicksort	0.59 (0.19)	108.496	3.4 (1.9)	56.9%
hanoi	0.60 (0.12)	44.896	12.0 (4.0)	59.2%
queens	0.80 (0.16)	95.976	8.0 (4.0)	66.0%
heapsort	2.12 (0.61)	239.312	2.9 (6.1)	62.5%
nsieve	1.39 (0.37)	156.320	4.2 (2.1)	55.2%
cat	3.10 (0.59)	203.508	12.4 (5.9)	60.7%
calendar	5.93 (0.73)	391.688	22.8 (8.1)	84.5%
getopt	11.68 (1.59)	2,052.656	4.7 (2.2)	102.4%
linpack	23.16 (2.52)	1,609.024	8.2 (4.1)	42.2%
diff	51.97 (10.26)	3,559.236	13.3 (10.3)	66.6%
patch	32.00 (16.80)	4,832.412	0.3 (0.3)	19.7%
tar	43.61 (19.89)	27,280.584	3.2 (2.8)	286.6%
gzip	60.82 (27.80)	14,971.952	0.9 (0.8)	30.0%
grep	33.95 (10.28)	3,142.128	9.8 (6.7)	73.0%
sort	50.51 (8.99)	3,677.584	2.5 (1.2)	29.3%
dc	51.97 (7.75)	3,559.236	3.6 (0.7)	34.5%

Table 6.4: Accumulated demand-driven analysis time and space with caching (T^{opt}_{cache} and S^{opt}_{cache}) when computing the full solution, the cache fill, the slowdown of the demand-driven analyzer with caching with respect to the exhaustive analyzer and the demand-driven analyzer space utilization as a percentage of the exhaustive analysis space. Parentheses indicate measurements that exclude temporaries.

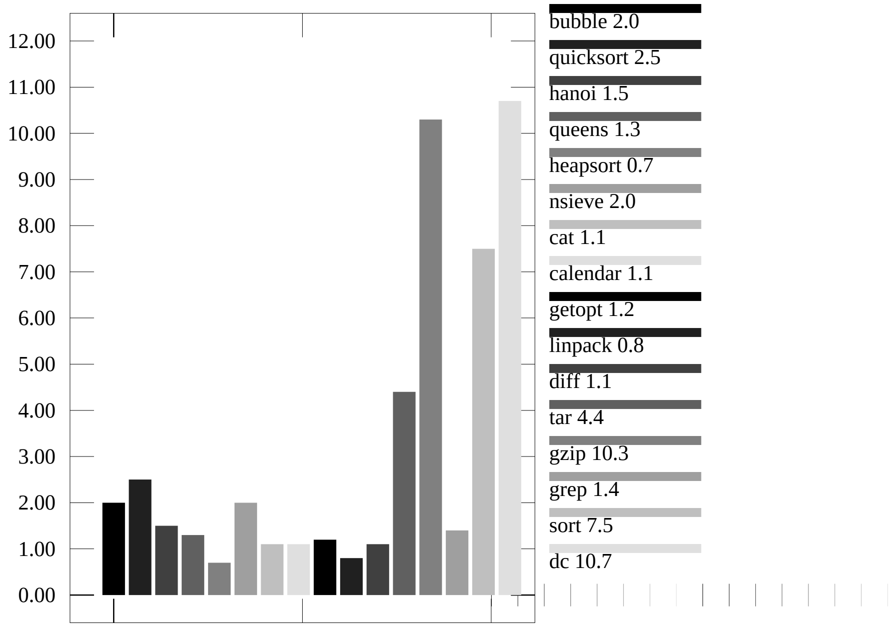
in patch, as well as in gzip, is indicated by the lower space requirements that result although the complete exhaustive solution is collected.

6.3.2 Experiment 2: Non-Caching Demand-Driven versus Exhaustive

The second experiment was carried out to determine the effect of caching on the performance of the demand-driven analyzer. The non-caching demand-driven analyzer with query advancing was executed with the same set of queries as in the first experiment. The accumulated analysis times T^{opt} are shown in Table 6.5. Table 6.5 also shows the accumulated space consumption S^{opt}. The speedups of the demand-driven analyzer over the exhaustive analyzer for both the full variable space and the source variable space are shown in Table 6.5 and are graphically displayed in Figures 6.6 (i) and (ii). Analogous to the experiments with du-chain analysis, disabling caching resulted in a slight slowdown of the demand-driven analyzer and at the same time in slightly lower space utilization since no cache memory



(i) Speedup of caching demand-driven over exhaustive $\frac{T_{ex}}{T^{opt}_{cache}}$ (full variable space)



(ii) Speedup of caching demand-driven over exhaustive $\frac{T_{ex}}{T^{opt}_{cache}}$ (temporaries excluded)

Figure 6.6: Caching (optimzied) demand-driven analysis vs exhaustive analysis.

Non-Caching Demand-Driven Analysis (CCP)			**Savings**	
program	time (secs) T^{opt}	space (Kbytes) S^{opt}	speedup $\frac{T_{ex}}{T^{opt}}$	% space: $\frac{S^{opt}\times 100}{S_{ex}}$
bubble	0.04 (0.02)	39.240	2.0 (2.0)	58.7%
quicksort	0.09 (0.04)	82.808	1.8 (2.5)	43.3%
hanoi	0.03 (0.02)	39.240	1.6 (1.5)	51.7%
queens	0.09 (0.03)	87.448	1.1 (1.3)	60.1%
heapsort	0.20 (0.14)	232.336	3.5 (0.7)	60.7%
nsieve	0.12 (0.09)	88.680	2.7 (2.0)	31.3%
cat	0.17 (0.09)	161.396	1.4 (1.1)	48.2%
calendar	0.14 (0.08)	289.360	1.8 (1.1)	62.4%
getopt	1.10 (0.55)	1,438.120	2.2 (1.2)	71.7%
linpack	1.09 (0.69)	1,418.448	2.5 (0.8)	39.8%
diff	1.39 (0.94)	2,360.244	2.7 (1.1)	44.1%
patch	2.55 (1.55)	3,887.372	36.6 (33.5)	15.8%
tar	2.70 (1.57)	3,734.432	4.9 (4.4)	39.2%
gzip	5.14 (3.04)	8,815.544	12.0 (10.3)	17.7%
grep	1.52 (1.08)	1,899.440	2.2 (1.4)	44.4%
sort	1.99 (0.96)	2,378.416	10.0 (7.5)	18.9%
dc	1.71 (0.90)	2,606.608	8.3 (10.7)	25.3%

Table 6.5: Accumulated demand-driven analysis time and space without caching (T^{opt} and S^{opt}), the speedup of the demand-driven analyzer without caching over the exhaustive analyzer and the demand-driven analyzer space utilization as a percentage of the exhaustive analysis space. Parenthesis indicate measurements that exclude temporaries.

is allocated. A direct evaluation of the caching overhead is shown in Table 6.6. Table 6.6 shows that the speedups of the demand-driven analyzer with caching over the demand-driven analyzer without caching. Except for one of the short programs (hanoi), adding the caching capability resulted in moderate speedup factors of up to 2. The analysis of program hanoi caused too few cache hits to pay off the overhead of the cache management.

6.3.3 Experiment 3: Query Advancing

The third experiment evaluates the effect of query advancing in CCP analysis. The results are shown in Table 6.7 for the caching CCP demand-driven analyzer and in Table 6.8 for the the non-caching version. Consider first the results for query advancing in the caching demand-driven CCP analyzer in Table 6.7. The first two columns show the accumulated demand-driven analysis time and space requirements that result if query advancing is disabled. The third column shows the speedup of the caching demand-driven analyzer with query advancing over the caching demand-driven analyzer without query advancing. The speedup measurements indicate that query advancing is worthwhile resulting in speedup in

Trade-off: Caching vs. Non-Caching (CCP)

program	speedup $\frac{T^{opt}}{T^{opt}_{cache}}$	% space overhead $\frac{(S^{opt}_{cache}\times 100)}{S^{opt}}$
bubble	0.8 (1.0)	169.4%
quicksort	1.8 (1.0)	103.2%
hanoi	0.7 (0.6)	111.4%
queens	1.2 (1.0)	106.2%
heapsort	1.1 (1.1)	102.7%
nsieve	1.5 (1.5)	114.6%
cat	1.5 (1.0)	119.1%
calendar	0.9 (2.0)	112.0%
getopt	1.3 (1.8)	111.8%
linpack	1.3 (1.4)	111.2%
diff	1.3 (1.5)	131.7%
patch	1.2 (1.3)	108.1%
tar	1.4 (1.3)	111.9%
gzip	1.0 (1.1)	105.8%
grep	1.2 (1.3)	130.7%
sort	1.3 (1.1)	136.7%
dc	1.1 (1.2)	112.9%

Table 6.6: The accumulated speedup of the demand-driven analyzer with caching over the demand-driven analyzer without caching and the space overhead of the caching demand-driven analysis as a percentage of the space used by the non-caching demand-driven analyzer.

all programs by factors of up to 6.3 while essentially requiring no additional space. The parentheses indicate the speedups that result if temporaries are excluded from the analysis. Excluding temporaries resulted in slightly lower speedups.

The evaluation of query advancing if caching is not used led to similar results as shown in Table 6.8. Query advancing resulted in speedups by factors of up to 5.9 in 15 out of 17 programs. Again, excluding temporaries affected the speedups only. In the source-level analysis, speedups by factors of up to 7.9 could be achieved in 14 out of 17 programs.

Overall, including query advancing in demand-driven CCP analysis was shown to be worthwhile as it resulted in speedups in all programs for the caching analyzer and in speedups in almost all programs for the non-caching analyzer. In comparison to the experimental results for du-chain analysis, the benefits of query advancing are higher in CCP. Higher speedups are expected because the query propagation paths in CCP are usually longer than in du-chain analysis. Longer propagation paths are more likely to contain procedure calls which represent the opportunities for query advancing.

Query Advancing (CCP) - Caching				
program	time T	space S	speedup $\frac{T}{T^{opt}}$	space: $\frac{(S^{opt} \times 100)}{S}$
bubble	0.05 (0.02)	76.012	1.0 (1.0)	87.4%
quicksort	0.10 (0.05)	140.652	2.0 (1.2)	60.7%
hanoi	0.05 (0.05)	63.648	1.2 (1.6)	68.7%
queens	0.11 (0.04)	155.900	1.5 (1.3)	59.5%
heapsort	0.18 (0.13)	238.632	1.0 (1.0)	100.0%
nsieve	0.12 (0.09)	181.336	1.5 (1.5)	56.0%
cat	0.41 (0.20)	695.788	3.7 (2.2)	27.6%
calendar	0.47 (0.17)	981.432	3.1 (4.2)	33.0%
getopt	0.86 (0.31)	1,635.616	1.1 (1.1)	98.3%
linpack	0.80 (0.49)	1,577.404	1.0 (1.0)	100.0%
diff	1.47 (1.08)	3,818.512	1.4 (1.7)	81.4%
patch	5.79 (2.69)	11,917.592	2.7 (2.3)	35.2%
tar	6.54 (3.89)	13,594.516	3.4 (3.3)	30.7%
gzip	29.91 (15.82)	54,745.756	6.0 (5.5)	17.0%
grep	7.96 (4.12)	15,493.968	6.3 (5.2)	13.6%
sort	6.21 (2.79)	12,060.096	4.2 (3.4)	26.9%
dc	5.56 (2.89)	10,111.556	3.5 (3.9)	29.1%

Table 6.7: The accumulated caching demand-driven analysis time and space without query advancing (T and S), the speedup of the caching demand-driven analyzer with query advancing over the caching demand-driven analyzer without query advancing and the space utilization of the caching demand-driven analyzer with query advancing as a percentage of the space used by the caching demand-driven analyzer without query advancing. Parentheses indicate measurements that exclude temporaries.

6.3.4 Summary

The experimental results demonstrated that the demand-driven CCP analyzer performs well in practice. The first experiment showed that demand-driven CCP analysis is faster than exhaustive analysis even if copy constant information is demanded at all uses in the program. Except for one program, demand-driven analysis also used less space for storing data flow information than exhaustive analysis. Based on these results, applications of constant propagation information, in particular if utilized only selectively, clearly benefit from a demand-driven analysis approach. Even if the demand-driven analyzer is used to compute copy constant information exhaustively over the program, the resulting slowdown is moderate, on average by a factor of 6.

The second experiment showed that, except for three shorter programs, demand-driven analysis benefits from caching. However, if only very few queries are raised caching is less likely to be beneficial since there may not be sufficiently many cache hits to compensate for the cache management overhead.

Query Advancing (CCP) - Non-Caching				
program	time T	space S	speedup $\frac{T}{T^{opt}}$	space: $\frac{(S^{opt} \times 100)}{S}$
bubble	0.07 (0.02)	73.940	1.7 (1.0)	53.0%
quicksort	0.13 (0.06)	137.764	1.4 (1.5)	60.1%
hanoi	0.06 (0.03)	58.992	2.0 (1.5)	66.5%
queens	0.14 (0.09)	151.580	1.5 (3.0)	57.6%
heapsort	0.20 (0.13)	232.296	1.0 (0.9)	100.0%
nsieve	0.16 (0.11)	168.344	1.3 (1.2)	52.6%
cat	0.42 (0.21)	669.508	2.4 (2.3)	24.1%
calendar	0.51 (0.18)	935.064	3.6 (2.2)	30.9%
getopt	1.02 (0.51)	1,464.816	0.9 (0.9)	98.1%
linpack	0.97 (0.64)	1,418.164	0.8 (0.9)	100.0%
diff	1.96 (1.32)	3,068.096	1.4 (1.4)	76.9%
patch	6.02 (2.96)	11,599.008	2.3 (1.9)	33.5%
tar	7.32 (4.21)	13,139.940	2.7 (2.6)	28.4%
gzip	30.81 (16.14)	54,247.364	5.9 (5.3)	16.2%
grep	8.54 (4.62)	14,938.288	5.6 (7.9)	12.7%
sort	7.32 (3.69)	11,225.008	3.6 (3.8)	21.1%
dc	5.73 (3.08)	9,781.204	3.3 (3.4)	26.6%

Table 6.8: The accumulated non-caching demand-driven analysis time and space without query advancing (T and S), the speedup of the non-caching demand-driven analyzer with query advancing over the demand-driven analyzer without query advancing and the space utilization of the non-caching demand-driven analyzer with query advancing as a percentage of the space used by the non-caching demand-driven analyzer without query advancing. Parentheses indicate measurements that exclude temporaries.

The third experiment examined the benefits of query advancing and showed that query advancing is worthwhile. In spite of the additional overhead of computing the flow-insensitive summary information, query advancing lead to speedups by factors of up to 6.3.

The previous chapter identified several program characteristics that either positively or negatively affected the performance of demand-driven du-chain analysis.[5] The identified characteristics affect demand-driven CCP in the same way.

In comparison with the experimental results for du-chain analysis, the speedups and space savings of the demand-driven CCP analyzers were generally higher. The primary cause for this effect is that exhaustive CCP analysis has a more costly implementation than exhaustive REACH analysis. Unlike the exhaustive REACH analyzer, the exhaustive analyzer for CCP does not use bit vector implementations and processes each variable at each node separately. However, the implementation of the demand-driven analyzers for both problems do not use bit vectors, making higher speedups for CCP more likely.

[5] See section 5.5.4.

Chapter 7

Application in Software Testing

This chapter demonstrates the utility of demand-driven analysis in a software development application. The application considered is data flow testing at the integration level. Data flow testing relies heavily on the support of data flow analysis for computing the du-chains which serve as the test case requirements for a program. During intergration testing, data flow analysis is performed repeatedly since new test requirements must be identified at each integration step. As a result, the accumulated cost of performing data flow analysis can considerably contribute to the overhead of testing. Previous approaches to data flow testing are either based on costly exhaustive computations or based on incremental updates. This chapter presents a new approach to data flow integration testing that is based on the demand-driven du-chain analyzer developed in Chapter 5. Results of a set of experiments demonstrate the practical benefits of the new demand-driven approach by showing that demand-driven analysis can outperform both exhaustive and incremental analysis if used in the context of integration testing.

This chapter is organized as follows. Section 7.1 discusses the analysis problems in integration testing and outlines how demand-driven analysis can be used to solve them. The pertinent background in data flow testing is presented in Section 7.2. Section 7.3 describes the demand-driven analyzer for use in integration testing in detail. Experimental results are reported in Section 7.4 and Section 7.5 summarizes the contributions of this chapter.

7.1 Motivation

Data flow testing uses coverage criteria to select sets of du-chains in a program that serve as the test case requirements [Nta84, CPRS85, FW88]. While short programs may be tested all at once, the testing of larger programs usually takes place in several phases and at different levels of program abstraction. The individual program units are tested first in isolation during *unit testing*. Then, their interfaces are tested separately during one or more *integration steps* [HS89b].

The data flow analysis requirements that arise during testing vary with different testing phases. Unit testing requires the determination of the *intraprocedural* du-chains within each unit. Since the complete set of intraprocedural du-chains is needed, using standard exhaustive intraprocedural data flow analysis appears appropriate. The situation is different when considering the analysis needs during the procedure integration. Each integration step requires the determination of only those du-chains that cross the most recently integrated procedure interfaces to establish the new test requirements. Exhaustively re-computing du-chains at the beginning of each integration step is inefficient and may easily result in overly high analysis times. This chapter shows that the analysis needs that arise during integration testing can be handled in a far more efficient way using demand-driven analysis.

The problem of avoiding costly re-computations of information in response to a program change 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 [Ros81, Zad84, RP88, PS89]. Incremental analysis avoids exhaustive re-computations by performing the appropriate updates of a previously computed exhaustive solution. Incremental analysis techniques can also be used in integration testing to extend the solution after each integration step with the newly established reaching definitions. However, incremental analysis still requires the exhaustive reaching definition solution to be computed initially and to be maintained between integration steps in addition to the du-chains. Moreover, the incremental update of the solution at each integration step 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.

This chapter presents a new approach to integration testing that uses demand-driven analysis to efficiently provide the newly established data flow information during each integration step. A set of experiments was conducted to experimentally evaluate the benefits of demand-driven analysis in the context of integration testing. The performance of the demand-driven analyzer during the integration process is experimentally compared with the performance of (i) an exhaustive analyzer, and (ii) an analyzer based on incremental updates. The experiments show that demand-driven analysis is faster than exhaustive analysis by factors ranging from 2.6 up to 25. If caching is used the speedups increase even further up to a factor of 30. The demand-driven analyzer also outperforms the incremental analyzer in 8 out of 12 programs by factors up to 5. Again, if caching was used, the speedups increase and the demand-driven analyzer outperforms the incremental analyzer in all but one program.

7.2 Data Flow Testing

Data flow testing uses *coverage criteria* [RW85] to select subpaths in the program for testing based on sets of du-chains. After the du-chains in a program have been computed, test cases are generated, manually or automatically, to exercise du-chains according to a selected coverage criterion. 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 *all-uses* criterion requires that for each definition, paths to all reachable uses are exercised.

Recall that du-chains are determined based on the sets $RD(v, n)$ and $RU(v, n)$ of reaching definitions and reachable uses. Given a definition d of a variable v at node n and a use u of variable w at node m, the pair (d, u) is a du-chain if $d \in RD(w, m)$ or, equivalently if $u \in RU(v, n)$.[1]

In integration testing, the program under analysis changes during each integration step since more procedures are integrated at each step. To distinguish the sets of reaching definitions and reachable uses that result at different integration stages of a program, the current version of the program P is added as a parameter:

$$\begin{aligned} RD(v, n, P) &= RD(v, n) \text{ for program } P \text{, and} \\ RU(v, n, P) &= RU(v, n) \text{ for program } P. \end{aligned}$$

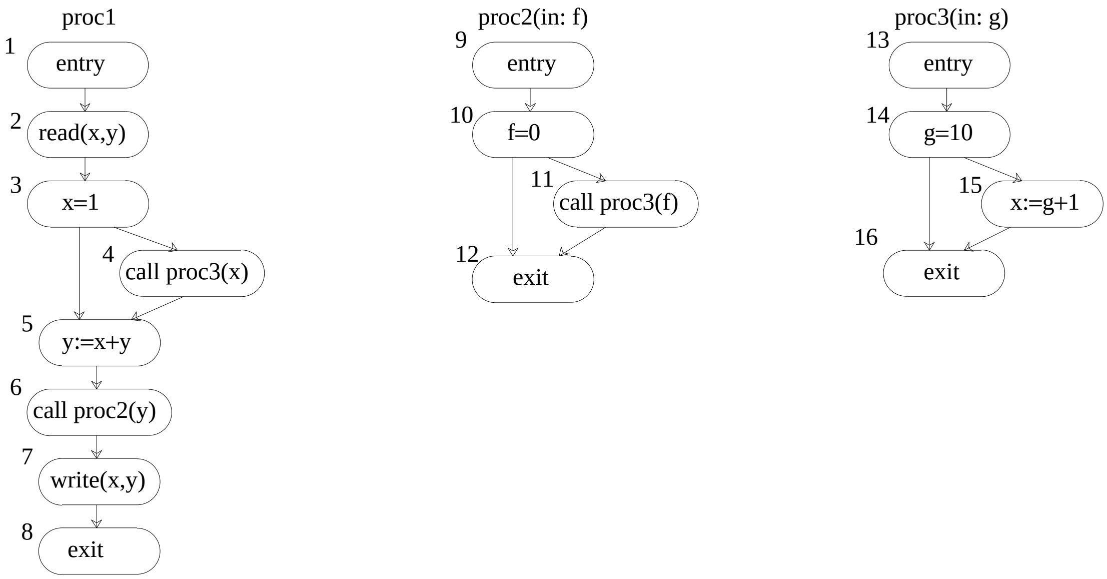
Figure 7.1 re-displays the program example from Chapter 5 that will be used to illustrate the integration testing process. The table in Figure 7.1 shows again the complete set of intra- and interprocedural du-chains for the program. Note that pair (x_2, x_5) is both an inter- and an intraprocedural du-chain.

7.3 Integration Testing

The objective of data flow integration testing is to structure the overall testing process into several phases by explicitly separating the testing of intra- and interprocedural du-chains. During *unit testing* each procedure is tested in isolation based on only the *intra*procedural du-chains within the procedure. After the individual units have been processed the interactions among procedures are tested separately during procedure integration. Integration testing takes place in several integration steps. During each step, one or more procedures are selected according to an integration strategy, such as bottom-up or top-down integration [Mye76]. The testing at each integration step is only concerned with the interprocedural du-chains that cross an interface of the procedures that are currently being integrated.

Prior to the integration of a procedure p, certain assumptions must be made about the interfaces to both the procedures that call p and the procedures that are called from p. 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 procedures that are called but that are not yet integrated. Thus, it is assumed that no def-clear paths exist

[1] See Section 5.3.1 for further details.



definition	du-chains	
	intraprocedural	interprocedural
x_2	$(x_2, x_3), (x_2, x_4), (x_2, x_5)$	$(x_2, x_5), (x_2, x_7), (x_2, g_{14}), (x_2, g_{15})$
x_{15}		$(x_{15}, x_5), (x_{15}, x_7)$
y_2	(y_2, y_5)	
y_5	$(y_5, y_6), (y_5, y_7)$	$(y_5, f_{10}), (y_5, f_{11}), (y_5, g_{14}), (y_5, g_{15})$

Figure 7.1: Example program with interprocedural du-chains.

through a non-integrated procedure. As the integration proceeds, temporary definitions are removed and actual def-clear paths through called procedure are identified and considered.

Example: Consider the example in Figure 7.1. During unit testing the two temporary definitions f_{in} and g_{in} are added for the formal parameters f in procedure *proc2* and for the formal g in procedure *proc3*, respectively. Each call site is assumed to kill the value of the global value x. Figure 7.1 shows the intraprocedural du-chains that result for each procedure. In addition, the temporary du-chains (f_{in}, f_{10}),(f_{in}, f_{11}), (g_{in}, g_{14}) and (g_{in}, g_{15}) are considered during unit testing.

Next consider the testing performed at each integration step. Assume for simplicity, that

```
    procedure p                   procedure q
    begin                         begin
        ...                         ...
      $v_1$ := 0;                   $w_1$ := $v_2$;
 s:   call q;                       ...
      ...:=$w_2$;                 end
        ...
    end
```

Figure 7.2: Cross-on-entry and cross-on-exit du-chains.

during each step a single procedure q is integrated with one of its calling procedures p. To integrate procedure q with procedure 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 testing of the current integration step concerns only the interprocedural du-chains that are established by the integration of q with p. These newly established du-chains are captured in the set $Cross(p, q)$ defined as follows:

Definition 7.1 (Cross Chains) *Let p be a procedure that calls a procedure q. The set of* **cross chains** *$Cross(p, q)$ that are established by integrating q with p is the set of interprocedural du-chains that have a def-clear path that contains the entry and/or exit node of procedure q.*

Example: Consider the integration of procedure *proc2* with procedure *proc1* in Figure 7.1 and assume that procedure *proc3* has not yet been integrated. The du-chains that cross the entry node or exit node of *proc2* are $Cross(q, p) = \{(x_2, x_7), (y_5, f_{10}), (y_5, f_{11})\}$. Note, that du-chains that cross both, entry/exit nodes of procedure *proc2* and entry/exit nodes of procedure *proc3* are not included since *proc3* has not yet been integrated.

A du-chain in $Cross(p, q)$ may cross several interfaces. However, a du-chain will not be considered for testing unless there exists a def-clear path that only crosses interfaces of procedures that have already been integrated. A du-chain with multiple def-clear paths that cross different procedure interfaces, is correspondingly also relevant for testing during multiple integration steps and may, thus, be tested repeatedly.

To define the set $Cross(p, q)$ in terms of data flow sets requires a closer look at the way a du-chain crosses a procedure interface. Consider Figure 7.2 and the integration of procedure q with procedure p at the call site s in p. Let P and P' be the programs prior to and after the integration. Thus, P' is obtained from P by removing temporary definitions from q and by including the call site s in p that calls q. The du-chains (v_1, v_2) and (w_1, w_2)

```
Procedure $ComputeCross(p, q)$
input: $p, q$: procedures in a program $P$;
output: the set $Cross(p, q)$
begin
 $Cross := \emptyset$;
 let $P$ and $P'$ be the programs prior to and after integrating $q$, respectively
 for each call site $s$ in $p$ where $s \in call(q)$ do
     for each variable $v$ such that $b_s(v) \neq \emptyset$ do
         compute $Def = RD(v, s, P)$;
         compute $Use = \bigcup_{w \in b_s(v)} RU(w, entry_q, P')$;
         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;
end
```

Figure 7.3: Procedure *ComputeCross*.

in Figure 7.2 are both contained in $Cross(p, q)$. However, the chain (v_1, v_2) is in $Cross(p, q)$ because it crosses the entry of q and (w_1, w_2) is in $Cross(p, q)$ because it crosses the exit of q.

Let (d, u) be a cross chain that crosses a call site s and assume for simplicity that d and u are a definition and a use of a global variable v. The chain crosses the boundaries to the procedure called at s in one of two ways:

Cross-on-entry: The definition d reaches the call site s in P and the use u is reachable in P' from the entry of the called procedure:

$$d \in RD(v, s, P) \text{ and } u \in RU(v, entry_q, P').$$

Cross-on-exit: The definition d reaches the q's exit in P and u is a reachable use at the call in P':

$$d \in RD(v, exit_q, P) \text{ and } u \in RU(v, s, P').$$

$Cross(p, q)$ results as the set of cross-on-entry and cross-on-exit chains. Note that a du-chain (d, u) that crosses both the entry and the exit of the currently integrated procedure

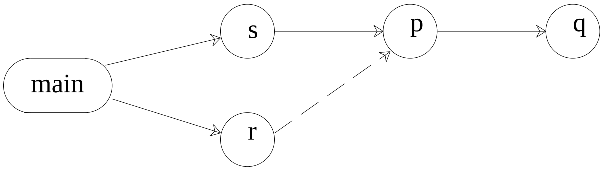


Figure 7.4: Call graph with non-integrated call sites shown in dashed lines.

classifies only as a cross-on-entry chain since $d \notin RD(v, exit_q, P)$ prior to the integration.

Procedure *ComputeCross* in Figure 7.3 summarizes the computation of the cross-on-entry and cross-on-exit chains for global and local variables taking possible parameter bindings into account.

Example: Consider again the integration of procedure *proc2* with procedure *proc1* in Figure 7.1 assuming that procedure r has not yet been integrated. The set $Cross(proc1, proc2)$ is computed as: (cross-on-entry) $RD(x, 6, P) = \{x_2\}$ and $RU(x, 9, P') = \{x_7\}$ and $RD(y, 6, P)$ $= \{y_5\}$ and $RU(f, 9, P') = \{f_{10}, f_{11}\}$ resulting in the set of chains $\{(x_2, x_7), (y_5, f_{10}), (y_5, f_{11})\}$. There are no cross-on-exit chains prior to the integration of procedure *proc3*.

7.3.1 Computing Cross Chains

The efficiency of procedure *ComputeCross* from Figure 7.3 depends primarily on the algorithm that is used to compute the data flow sets $RD(v, n, P)$ and $RU(v, n, P)$ at each integration step. There are various analysis approaches that may be pursued.

Exhaustive Analysis

Previously, Harrold and Soffa discussed data flow testing in the presence of procedures and presented an exhaustive analysis approach to compute du-chains over the complete program [HS90]. An exhaustive approach requires re-analysis of the program at the beginning of each integration step to account for new procedure interfaces. However, the exhaustive computation at each integration step can be slightly optimized by performing exhaustive analysis only over the procedures that are *affected* by the current integration step. Assume a procedure q is currently integrated with a procedure p. Let r be another procedure, such that neither p nor q calls procedure r directly or indirectly through some call chain of currently integrated procedure. Furthermore, assume procedure r calls neither p nor q directly or indirectly. This situation is depicted in the call graph in Figure 7.4. There are no interprocedural execution paths that connect procedure r and an interface between

p and q. Consequently, no data flow information can be propagated from procedure r to procedures p and q or vice versa. Hence, procedure r is not affected by the integration of p and q, and the data flow of r remains unchanged. It follows that the exhaustive re-analysis after integrating procedure q with procedure p can be limited to the procedures that are connected to q or p through a call chain of currently integrated procedures. For example in Figure 7.4, exhaustive re-analysis only has to consider procedures p and q. While this optimization strategy may result in the exclusion of some procedures from the analysis at each integration step, the set of affected procedures is still analyzed exhaustively.

Incremental Analysis

Exhaustive re-computations at the beginning of each integration step can be avoided by using an incremental analysis approach [Ros81, Zad84, RP88, PS89]. If incremental analysis is used, the complete exhaustive reaching definition solution must be maintained between subsequent integration steps. The number of established reaching definitions and resulting du-chains can only increase 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 only additions to the solution and no deletions. Assuming the exhaustive solution computation is based on fixed point iteration (e.g., [SP81]), an incremental version of the exhaustive solution computation is obtained in a straightforward way. The solution from the previous integration step is incrementally updated by simply using it as the initial value to re-start the fixed point iteration for the current integration step.

Demand-Driven Analysis

The final approach uses demand-driven analysis. Two demand-driven analyzers are required to implement procedure *ComputeCross*: a demand-driven analyzer for reaching definitions and a demand-driven analyzer for reachable uses. Chapter 5 described the demand-driven analyzer for computing reaching definition sets $RD(v, n, P)$. The analogous demand-driven analyzer for computing the symmetric sets $RU(v, n, P)$ of reachable uses can be similarly developed. Based on the demand-driven analyzers, each access of a data flow set $RD(v, n, P)$ or $RU(v, n, P)$ in procedure *ComputeCross* is simply replaced by a call to the appropriate demand-driven analysis routine.

7.4 Experiments

A set of experiments was conducted to evaluate the performance of the various analysis approaches if used to provide the relevant data flow information during integration testing. The analyzers were evaluated in the context of bottom-up integration testing. Bottom-up

Benchmarks							
No.	program	#code lines	#nodes	#procedures	#calls	#du-chains	#integr. steps
1	queens	89	150	4	4	119	4
2	cat	240	377	5	4	165	4
3	calendar	352	731	10	14	236	9
4	getopt	395	739	5	6	268	4
5	linpack	564	686	12	30	1160	14
6	diff	899	1561	12	33	685	11
7	patch	753	1316	14	13	599	12
8	gzip	1387	3024	38	123	1461	68
9	tar	1451	1756	27	68	847	37
10	grep	1488	2906	32	72	1048	47
11	sort	1528	3554	35	145	1570	80
12	dc	1576	3298	67	230	1958	153

Table 7.1: Benchmark programs.

integration testing processes the procedures in a program in depth-first (bottom-up) order of the program's call graph. During each integration step one edge (q, p) in the call graph is processed and the new du-chains are determined as described by procedure *ComputeCross* from Figure 7.3.

The following four analysis algorithms were implemented:

(CACHE) Demand-driven du-chain analyzer based on the caching versions of demand-driven analyzers for reaching definitions and reachable uses.

(DD) A non-caching version of the demand-driven du-chain algorithm (CACHE).

(EX) The exhaustive du-chain algorithm from Chapter 5 based on bit vector implementations.

(INCR) An incremental version of the exhaustive algorithm as outlined in the previous section. As the exhaustive analysis, the incremental version uses bit vector implementations.

The four analysis algorithms were also implemented as part of the PDGCC compiler project. For details of the implementation and experimentation context see Section 5.5. The experimentation in this chapter uses twelve of the larger programs form the C benchmarks used in Chapters 4 and 5. Table 7.1 lists the twelve benchmarks along with size parameters and the number of bottom-up integration steps for each program.

y

Accumulated Analysis Times

program	demand-driven analysis		exhaustive analysis	incremental analysis
	cache	n/cache		
	T_{cache}	T_{dd}	T_{ex}	T_{incr}
queens	0.09	0.06	0.17	0.08
cat	0.22	0.20	0.52	0.29
calendar	0.21	0.20	0.78	0.32
getopt	0.99	0.98	3.80	1.43
linpack	0.57	0.49	3.95	1.25
diff	15.74	16.26	67.99	8.60
patch	5.27	5.76	17.01	3.51
gzip	23.88	15.53	96.87	14.85
tar	10.34	11.45	37.59	6.91
grep	4.69	5.50	57.86	6.44
sort	7.58	9.22	193.76	15.00
dc	2.17	2.58	66.48	13.38

Table 7.2: The accumulated analysis times for the demand-driven analyzer with and without caching (T_{cache} and T_{dd}), for the exhaustive analyzer (T_{ex}), and for the incremental analyzer (T_{incr}).

7.4.1 Experiment 1: Demand-Driven versus Exhaustive Analysis

The first set of experiments compares the performance of the demand-driven analyzer with the performance of the exhaustive analyzer. The analysis times during the integration were measured to determine for each test program the accumulated analysis times shown in Table 7.2, where:

$$\begin{aligned} T_{ex} &= \text{ accumulated analysis time of exhaustive analysis,} \\ T_{cache} &= \text{ accumulated analysis time of demand-driven analysis with caching,} \\ T_{dd} &= \text{ accumulated analysis time of demand-driven analysis without caching.} \end{aligned}$$

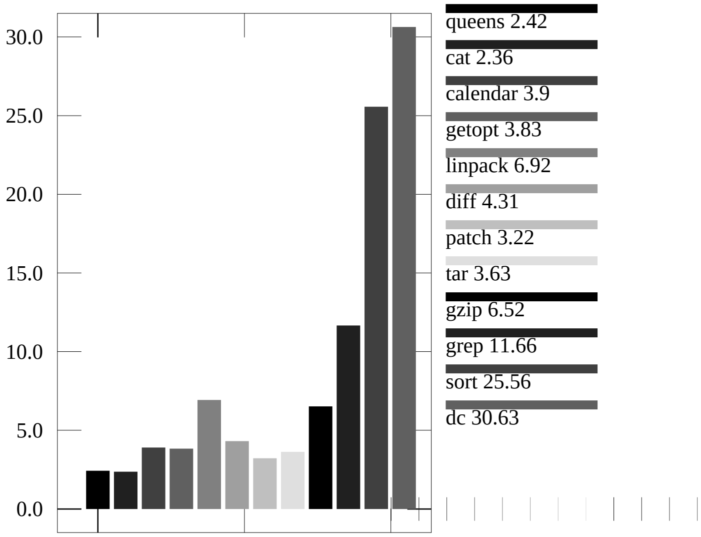
Based on the measured analysis times, the accumulated speedup of the demand-driven analyzer with caching over the exhaustive analyzer is given by $\frac{T_{ex}}{T_{cache}}$. Analogously, the accumulated speedup of the demand-driven analyzer without caching over the exhaustive analyzer is given by $\frac{T_{ex}}{T_{dd}}$. These speedups are shown in Table 7.3 and graphically displayed in Figures 7.5 (i) and (ii). The measurements show that the demand-driven analyzer with caching is significantly faster than the exhaustive analyzer by factors ranging from 2.3 up

Speedups				
program	**Experiment 1**		**Experiment 2**	
	demand-driven vs. exhaustive		**demand-driven vs. incremental**	
	cache	n/cache	cache	n/cache
	speedup $\frac{T_{ex}}{T_{cache}}$	speedup $\frac{T_{ex}}{T_{dd}}$	speedup $\frac{T_{incr}}{T_{cache}}$	speedup $\frac{T_{incr}}{T_{dd}}$
queens	2.42	2.83	1.14	1.33
cat	2.36	2.6	1.31	1.45
calendar	3.9	3.9	1.52	1.6
getopt	3.83	3.87	1.44	1.45
linpack	6.92	8.06	2.19	2.55
diff	4.31	4.18	4.31	0.52
patch	3.22	2.95	0.66	0.61
gzip	6.52	4.05	1.04	0.65
tar	3.63	3.28	3.63	0.66
grep	11.66	10.52	1.37	1.17
sort	25.56	21.01	1.97	1.62
dc	30.63	25.76	6.16	5.18

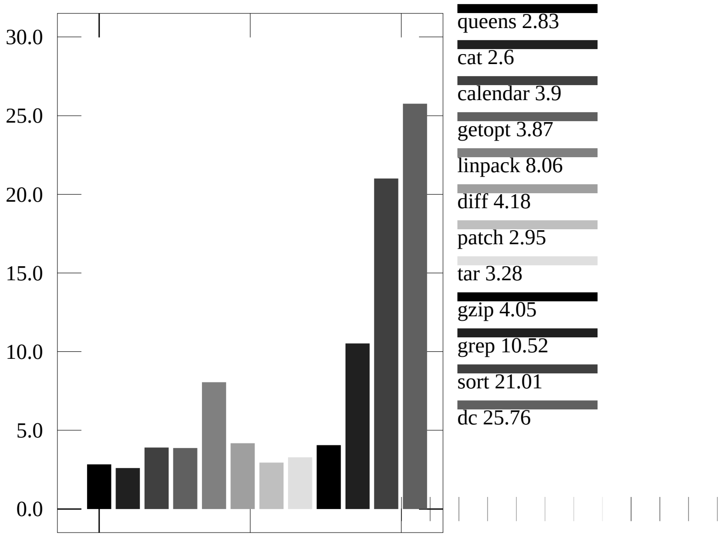
Table 7.3: The accumulated speedups of the demand-driven analyzer with and without caching over the exhaustive analyzer ($\frac{T_{ex}}{T_{cache}}$ and $\frac{T_{ex}}{T_{dd}}$) and the accumulated speedups of the demand-driven analyzer with and without caching over the incremental analyzer ($\frac{T_{incr}}{T_{cache}}$ and $\frac{T_{incr}}{T_{dd}}$).

to 30.0. As shown in Figure 7.5 (ii), disabling caching resulted in similar speedups ranging from 2.6 up to 25.7. Compared to the non-caching version, caching increased the speedups for the seven larger programs but did not pay off for the five shorter programs. For the shorter programs the number of cache hits was too small to compensate for the overhead of allocating and maintaining the cache. In larger programs that generate a higher number of queries, the savings from cache hits quickly outweigh the cache overhead. Figures 7.5 (i) and 7.5 (ii) indicate that the speedups of the demand-driven analyzer tend to grow with increasing program size (in terms of code lines).

To illustrate how the speedups evolve throughout the integration, Figure 7.6 shows the individual speedup curves as a function of the integration degree. The range of integration steps is normalized for all programs to the range [0..1]. Figure 7.6 (i) shows the speedup curves for the demand-driven analyzer with caching and Figure 7.6 (ii) shows the same speedup curves for the demand-driven analysis without caching. The curves indicate that the speedup evolves gradually in most programs. Thus, the accumulated speedup is already noticeable after the first integration steps and continuously grows as the integration proceeds. Note, however, that the speedup curves for the five larger programs (i.e., tar, gzip,

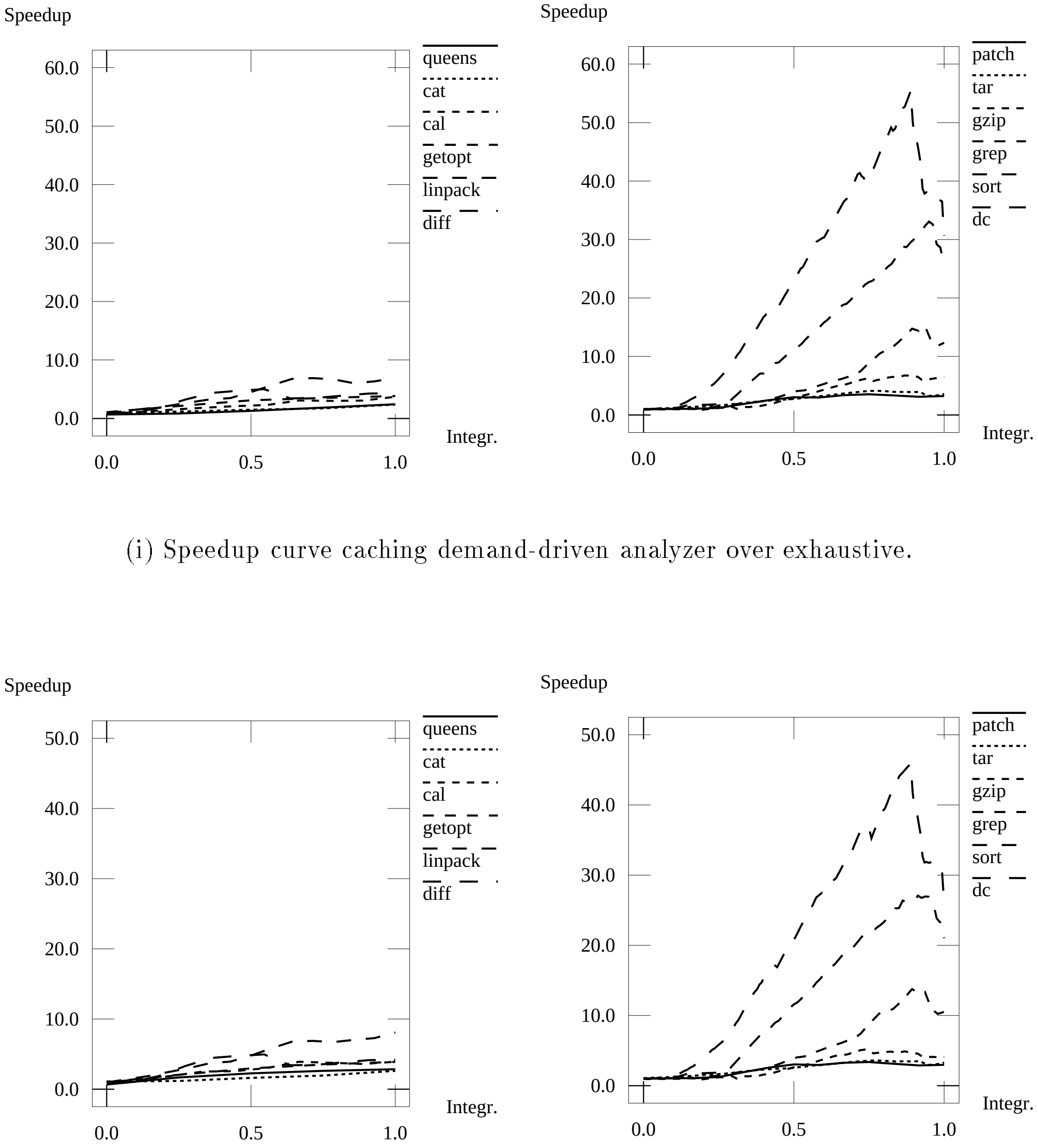


(i) Speedup of caching demand-driven over exhaustive: $\frac{T_{ex}}{T_{cache}}$



(ii) Speedup of non-caching demand-driven over exhaustive: $\frac{T_{ex}}{T_{dd}}$

Figure 7.5: Measured speedups of demand-driven over exhaustive analysis.



(i) Speedup curve caching demand-driven analyzer over exhaustive.

(ii) Speedup curve of non-caching demand-driven analyzer over exhaustive.

Figure 7.6: Measured speedup curves of demand-driven over exhaustive analysis.

grep, sort and dc) reach their peak shortly before the integration is complete. This effect is primarily due to the bottom-up integration strategy. Shortly before the integration is complete, the top-level calls in the main program are integrated. The top-level integration steps are the most expensive ones for both exhaustive and demand-driven analysis. Since nearly all procedures are affected by integration at the top level, the exhaustive analysis is performed over all procedures during the late integration steps. Thus, the exhaustive analysis times are almost constantly at their peak during the late integration steps. The performance of the demand-driven analyzer evolves differently during the late integration steps. The integration of top-level calls creates the longest propagation paths of data flow information in the program. Moreover, the length of propagation paths is likely to keep growing even during the late integration steps. Thus, while the cost of exhaustive analysis is nearly constant during the late integration steps, the cost of demand-driven analysis tends to still increase. Consequently, the peak of the accumulated speedup is reached just before the top-level calls are integrated. This phenomenon does not occur in smaller programs since even after the integration of top-level calls the propagation paths are not sufficiently long.

7.4.2 Experiment 2: Demand-Driven versus Incremental Analysis

The second set of experiments compares the performance of the demand-driven analyzer with the performance of the incremental analyzer. The integration system using the incremental analyzer was run to measure the accumulated analysis time T_{incr}, where:

$$T_{incr} = \text{ accumulated analysis time of incremental analysis.}$$

The results are shown in Table 7.2. Figure 7.7 (i) displays the accumulated speedup $\frac{T_{incr}}{T_{cache}}$ of the demand-driven analyzer with caching over the incremental analysis. The corresponding accumulated speedups $\frac{T_{incr}}{T_{dd}}$ for the demand-driven analyzer without caching are shown in Figure 7.7 (ii).

As shown in Figure 7.7 (ii), the caching demand-driven analyzer achieves speedups over the incremental analyzer in all but one program (patch) by factors of up to 7.16. As in the first experiment, disabling caching affects the speedups only slightly. Except for two programs (patch and gzip) the demand-driven analyzer without caching is faster than the incremental analyzer by factors of up to 5.18. Disabling the cache results in a slight slowdown for the five larger programs but improves the speedups for the five shorter programs. The demand-driven analyzer without caching has an important advantage over the incremental analyzer in that no storage of information other than the du-chains is required between integration steps. In contrast, the incremental analyzer maintains the complete reaching definition solution in addition to the du-chains throughout the integration process.

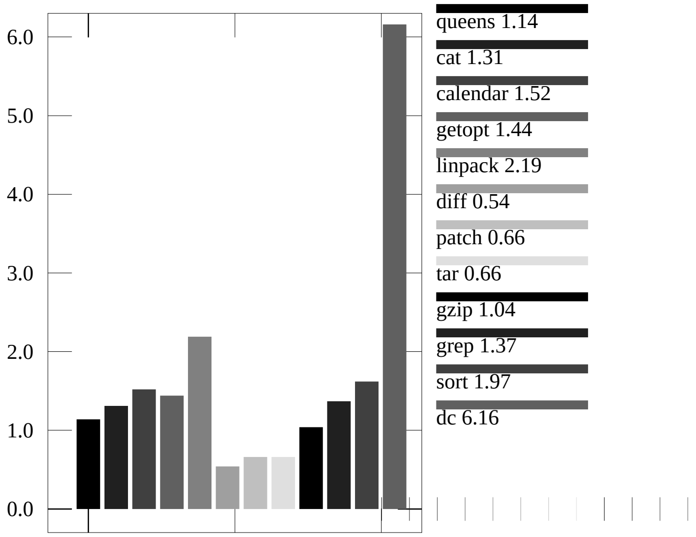
An examination of the programs patch and gzip revealed that they have a high percentage of global variables. Queries for global variables may require much longer propagation paths than queries for locals, which explains why demand-driven analysis does not perform as well.

Figure 7.8 shows the individual speedup curves for both the demand-driven analyzer with caching (i) and the demand-driven analyzer without caching (ii). As in the first experiment, the curves indicate that the speedup evolves gradually in most programs. Furthermore the speedup curves for larger programs reach their peak again shortly before the integration is complete. However, the degradation of the speedup after the peak has been reached is less severe than in the first experiment. A comparison of analysis times during the individual integration steps of the demand-driven and the incremental analysis reveals that the speedups of the demand-driven analyzer over the incremental analyzer are far higher during the early integration steps. During the late integration steps that are more expensive for the demand-driven analyzer, the benefits of using bit vectors in the incremental analyzer show a higher pay-off. This effect is not noticeable in smaller programs, since the information propagation paths are not sufficiently long even in the late stages of the program integration.

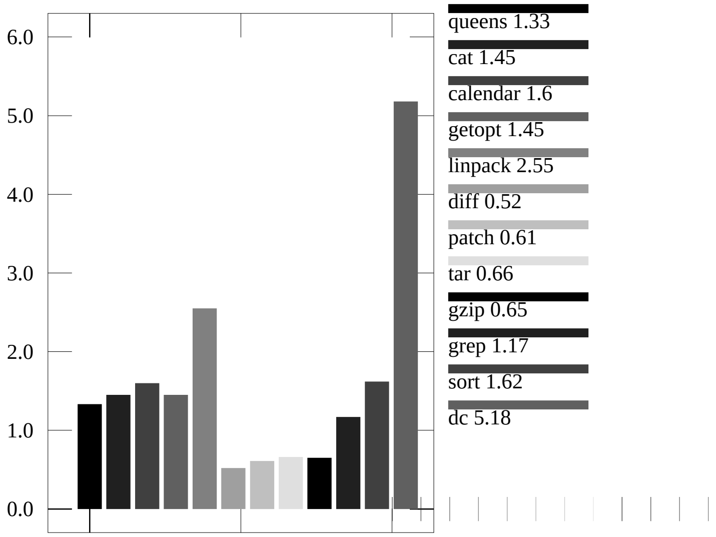
7.5 Summary

In summary, demand-driven analysis has been shown to provide an efficient analysis approach for integration testing in practice. The experiments demonstrate that using demand-driven analysis in integration testing is significantly faster than using exhaustive analysis. Even compared to an incremental analysis approach, demand-driven analysis has shown to be the more efficient approach. An important advantage of demand-driven analysis over incremental analysis is that, no storage and maintenance of data flow solutions in addition to the du-chains themselves is necessary between integration steps.

The encouraging results of the experimental comparison of demand-driven analysis with incremental analysis suggest further potential benefits of demand-driven analysis in another related field of data flow testing, namely regression testing. The analysis task in regression testing is to determine the test requirements for a modified program to ensure that no new errors are introduced into previously tested code. Selective regression testing [OW88, TTL89, GHS92, AHKL93, BH93, RM93, RM94] attempts to re-test only those du-chains that are affected by the modification. To identify the affected du-chains that require re-testing, techniques based on incremental data flow analysis have been used. It may be possible to avoid the use of incremental analysis technique and instead compute the affected du-chains from scratch after each program change using a demand-driven analysis approach. Exploring the benefits and drawbacks of using demand-driven analysis in regression testing presents an interesting future extension of the current work on demand-driven analysis in data flow testing.

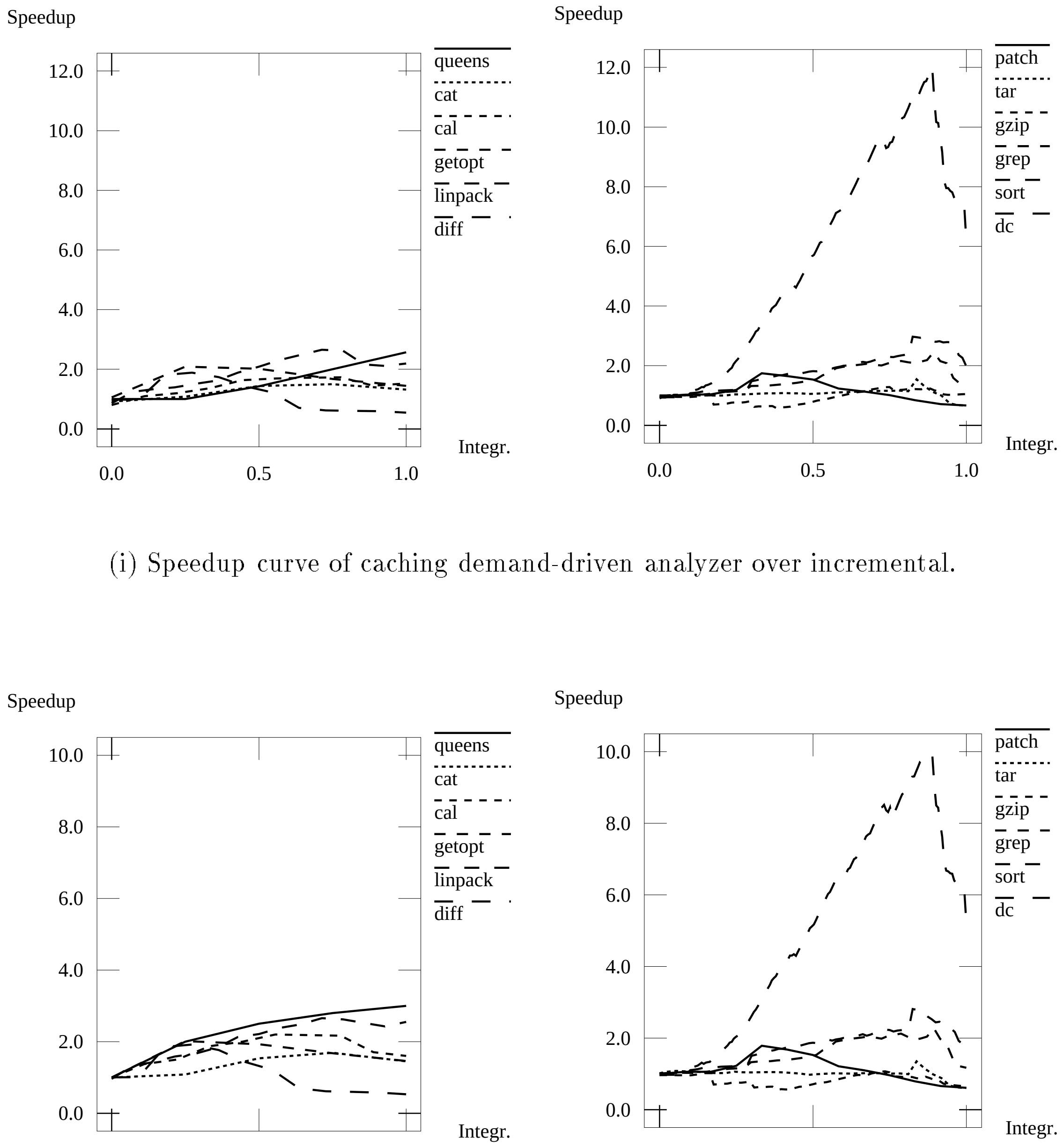


(i) Speedup of caching demand-driven over incremental: $\frac{T_{incr}}{T_{cache}}$



(ii) Speedup of non-caching demand-driven over incremental: $\frac{T_{incr}}{T_{dd}}$

Figure 7.7: Measured speedups of demand-driven over incremental analysis.



(i) Speedup curve of caching demand-driven analyzer over incremental.

(ii) Speedup curve of non-caching demand-driven analyzer over incremental.

Figure 7.8: Measured speedup curves of demand-driven over incremental analysis.

Chapter 8

Congruence Partitioning

The previous chapters developed a demand-driven analysis approach to reduce the analysis cost by avoiding the computation of information that is not needed in the current application. While experimentation demonstrates that demand-driven analysis is an effective approach to reduce the exhaustive analysis overhead, the analysis may still not be computationally minimal. Demand-driven analysis may still perform redundant computations that occur on the lower *application-independent* level of intermediate computations. Application-independent optimization of intermediate data flow computations is the subject of *forwarding techniques* [1]. In comparison to demand-driven analysis, forwarding techniques follow an orthogonal and complimentary approach to the problem of reducing the analysis cost. The application-dependent improvements of demand-driven analysis are not achievable by the application-independent forwarding techniques and vice versa. Thus, in order to achieve maximal cost reductions both approaches, demand-driven analysis and forwarding techniques should be considered.

This chapter presents *congruence partitioning* as a new forwarding technique that can be used in combination with demand-driven analysis or as a stand-alone optimization technique of exhaustive analyses. The organization of this chapter is as follows. Section 8.1 provides an overview of congruence partitioning. The formal framework for modeling congruence partitioning is presented in Section 8.2. Section 8.3 shows that for a particular class of data flow problems, congruence partitioning is sufficiently powerful to completely replace the traditional fixed point computation and directly solve the data flow problem. Section 8.4 compares congruence partitioning with sparse evaluation graphs and other related work is discussed in Section 8.5. This chapter concludes in Section 8.6 with a discussion of combining congruence partitioning and demand-driven data flow.

[1] See discussion in Chapter 1, Section 1.2.

8.1 Overview

Previous forwarding techniques suppress unnecessary data flow computations by using a specialized graphical representation of the program [WZ85, AWZ88, RWZ88, CLZ86, CCF90, JP93]. The need to construct a specialized graph makes it difficult to combine these forwarding techniques with the demand-driven analysis concepts that are based on a standard control flow graph representation of the program. Furthermore, except for the sparse evaluation graphs [CCF90], previous forwarding techniques are not general and only applicable to certain data flow problems.

Congruence partitioning is a new forwarding technique that avoids the limitations of previous approaches. Unlike previous forwarding techniques, congruence partitioning is an algebraic approach that directly manipulates the data flow equation system. It will be shown that viewing the problem as an algebraic problem of congruence relations, leads to conceptually simple algorithms that are both more general and powerful than previous graph-oriented methods. Congruence partitioning applies to all monotone data flow problems and can be used to optimize both traditional exhaustive analysis or demand-driven analysis.

Congruence partitioning is applied in order to restructure and optimize data flow equation systems prior to the actual solution computation. Recall that the solution of a data flow problem is the greatest fixed point of a system of monotone equations. Each equation expresses the solution at one program point in terms of the solutions at immediately preceding (or succeeding) points. A closer inspection of these equation systems reveals that their sizes are unnecessarily enlarged due to the inherent inclusion of redundant equations. The structure of data flow equation systems requires the propagation of intermediate results throughout the program, including the propagation to program points where these results are of no relevance. As a consequence, multiple equations in the system carry identical information. Equations that duplicate information already expressed by other equations are redundant and their repeated evaluation during the fixed point iteration is clearly undesirable. If equivalent but smaller equation systems without redundancies were constructed, the fixed point computation would be faster, independently of the evaluation algorithm used.

Congruence partitioning presents a systematic approach to minimize the size of data flow equation systems by discovering congruence relationships among equations. Two equations are *congruent* if their greatest fixed points are equal. Thus, at least one of two congruent equations is redundant and can therefore be eliminated. By repeatedly applying this elimination process an equivalent but reduced equation system can be constructed that includes only a single equation from each class of congruent equations.

A systematic framework for congruence partitioning is developed to model congruence relations by exploiting known algebraic properties of the equation system. The framework is used to establish a congruence relation based on the idempotence property of the meet operator in the system. A fast partitioning algorithm is presented to compute the idempotence congruence relation in $O(n \log n)$ time and $O(n)$ space, where n is the size of the program. Using the computed congruence relation, a reduced equation system is constructed that only contains a single equation from each congruence class. By the definition of congruence, it is sufficient to compute the fixed point over only the reduced system using any of the standard evaluation strategies.

The approach of reducing equation systems by computing congruence relations can easily be extended to include other notions of congruence. The congruence relations discussed in [DST80, NO80] are based on common subexpressions. Alpern et al. [AWZ88] used a fast $O(n \log n)$ algorithm due to Hopcroft for minimizing finite automata to compute CSE congruences for program optimization. Section 6.3.2 shows that Hopcroft's algorithm can equally well be applied to discover common subexpressions in data flow equation systems in order to enable further reductions.

The asymptotic performance of congruence partitioning depends only on the size of the equation system. The complexity of the data flow problem, i.e., the cost of actually evaluating the equations, does not impact the performance of the partitioning algorithm. The complexity of data flow problems varies dramatically, ranging from simple problems, such as live variable analysis, that can be implemented efficiently using bit vectors, to sophisticated time- and space-intensive analyses, such as alias analysis. Naturally, the benefits of congruence partitioning increase with the complexity of the data flow problem.

8.2 A Framework for Congruence Partitioning

A congruence partitioning is defined over the equation system of a data flow problem. Recall that a data flow equation $X(n)$ is defined at each node n in an ICFG of a program such that:

$$X(n) = f_n(\bigsqcap_{m \in dep(n)} X(m)),$$

where $dep(n)$ is a set of *dependent* nodes of n, usually the set of predecessors $pred(n)$ or a set of call sites if n is the entry node of a procedure.

An equation system can be viewed as a labeled directed graph $G = (V, E)$. The vertices in V represent equation variables and the operations on the right hand side of the equations. An edge (v, w) in E expresses that the expression represented by vertex v depends on the input represented by vertex w. A labeling function assigns a label $label(v)$ to each vertex $v \in V$. This graph is called an *equation graph.*

The equation graph represents an equation $X(n) = f_n(\bigsqcap_{m \in dep(n)} X(m))$ by the subgraph shown in Figure 8.1 (i). Corresponding to the function symbol f_n is a vertex $v(X(n))$ with $label(v(X(n)) = f_n$ that has a single successor vertex with label $\sqcap$. The vertex labeled $\sqcap$ has successors $v(X(m))$ for each predecessor m of node n. If the function f_n is the identity

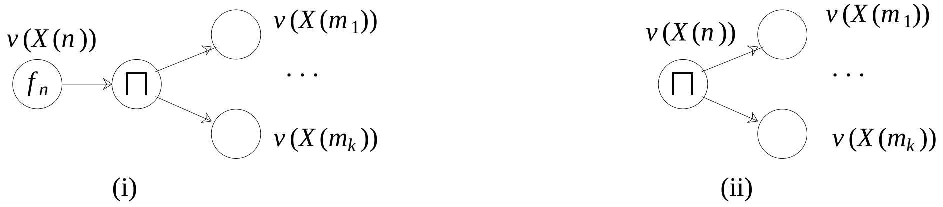


Figure 8.1: The translation of equations into graphs.

function, the equation reduces to $X(n) = \mathop{\sqcap}\limits_{m \in dep(n)} X(m)$. In this case no vertex for the function symbol is created, and the vertex $v(X(n))$ is the vertex labeled $\sqcap$ as shown in Figure 8.1 (ii). The vertex set V is partitioned into a set $V_\sqcap$ of vertices labeled $\sqcap$ (meet vertices) and a set V_f of vertices with a label denoting any other function symbol (function vertices).

When discussing equation systems it is assumed that their graphs are transformed into graphs whose vertices have an indegree and outdegree of at most 2. This transformation is analogous to transforming the textual representation of the equation system into some form of three-address-code. The associativity of the meet operator ensures that a graph can always be transformed into this form by adding some additional vertices for each vertex whose indegree or outdegree is greater than 2. At most a constant number of vertices is added per edge in this process and the number of vertices remains $O(n)$ [DST80], where $n = |N|$ is the number of nodes in the control flow graph.

8.2.1 Example

As a running example, consider alias analysis performed over the procedure *Insert* shown in Figure 8.2, where the operator $*$ denotes pointer dereferencing. Alias analysis computes pairs of aliased variables. To simplify the representation, a simple alias analysis is considered based on the assumption that whenever a variable q is aliased to a variable p, any variable that q points to is aliased to any variable that p points to. The lattice elements are collections of alias relations. A collection could simply be a set of alias pairs or, alternatively, a partition of the variables into sets of aliased variables. The entry and exit points of the nodes at which data flow information is computed are marked by numbers.

The equation system that expresses the analysis over procedure *Insert* is shown in Figure 8.3 along with its equation graph. Each equation $X(n)$ refers to the alias information that holds at the program point n marked in the control flow graph in Figure 8.2. The meet operator $\sqcap$ represents the union of two collections of alias relations into a single collection.

```
/* insert a value  val in a binary tree  x */
 procedure Insert(in: x,in: val)
 begin
     val:=h(val);
     repeat
         p:=x;
         if (val ≤ (∗x.key))
             then x:= (∗x.left);
             else x:=xright;
    until (x = NULL);
    new(x);
    (∗x.key):=val;
    (∗x.left):=NULL;
    (∗x.right):=NULL;
    if (val ≤ (∗p.key))
        then (∗p.left):=x
        else (∗p.right):=x;
end
```

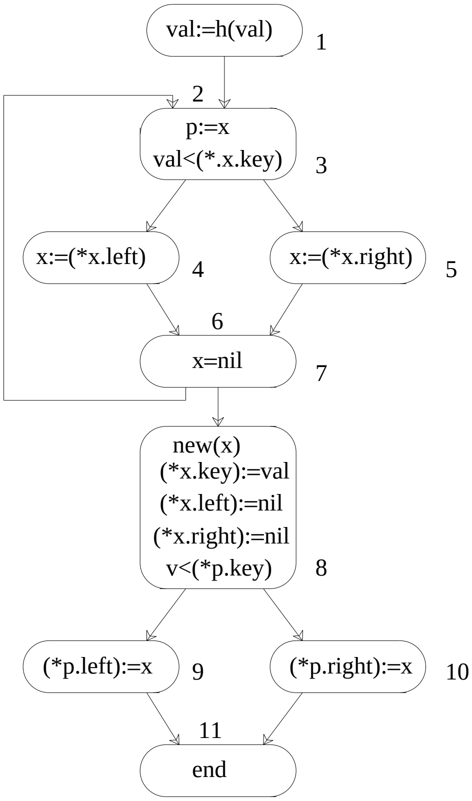


Figure 8.2: A sample program and its control flow graph.

$$X(1) = init$$
$$X(2) = X(1) \sqcap X(7)$$
$$X(3) = kill[p](X(2)) \sqcap (p, x)$$
$$X(4) = X(3)$$
$$X(5) = X(3)$$
$$X(6) = X(4) \sqcap X(5)$$
$$X(7) = X(6)$$
$$X(8) = kill[x](X(7))$$
$$X(9) = (p, x) \sqcap X(8)$$
$$X(10) = X(8) \sqcap (p, x)$$
$$X(11) = X(9) \sqcap X(10)$$

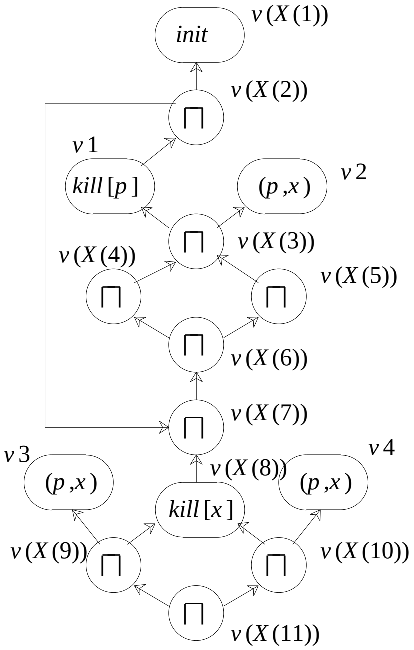


Figure 8.3: Data flow equations and their graphical representation.

The data flow equations refer to a function $kill[y]$ that takes as an argument a collection of alias relations C and eliminates all alias relations for variable y from C. For more details of the analysis, the reader is referred to [CC77b]. With respect to congruence partitioning, the meet $\sqcap$ and other functions like $kill[y]$ are merely uninterpreted symbols.

8.2.2 Congruence Relations

Given an equation system, the goal is to minimize the size of the system without actually evaluating the equation. Unfortunately, even the following restricted version of this minimization problem is NP-complete [GJ79]:

> Given a set of expressions constructed from uninterpreted constants and a single commutative and associative operator, determine the minimal number of operations needed to evaluate all expressions.

Thus, one cannot expect to find an efficient algorithm for eliminating all redundancies. However, as will be shown, it is possible to minimize an equation system with respect to certain well-defined classes of redundancies using fast algorithms.

Redundancies are eliminated by discovering congruence relationships among equations. Congruence relationships are established among the final fixed point values of equations as defined below.

Definition 8.1 (Congruence) *Let $gfp(n)$ denote the greatest fixed point value of equation $X(n)$. Two equations $X(n)$ and $X(m)$ in a system X are called* **congruent** *only if*

$$gfp(n) = gfp(m)$$

Note that no assumptions are made on the sequence of intermediate values an equation may take during the fixed point iteration. These sequences of values are highly dependent on the particular iteration strategy that is used to compute the fixed point, but the notion of congruence is a valid relation for any such strategy.

Congruence is an equivalence relation, that is, a symmetric, reflexive and transitive relation. Hence, a congruence relation induces a partition π of the equations into *congruence classes*. All equations that are contained in the same congruence class in π have an identical fixed point. Given π we can reduce the original equation system by eliminating all but one equation from each congruence class. By the definition of congruence, the resulting reduced system is guaranteed to provide the same fixed point solution as the original system, independent of the particular evaluation strategy used. If needed, the solution of the reduced system can be later expanded to the solution of all original equations using the computed partition π.

By definition 8.1 there is no unique congruence relation. The most aggressive reductions of an equation system are achieved by a congruence relation that induces a minimal number

$$\begin{array}{ccc}
\begin{aligned} X(1) &= f(X(0)) \\ X(2) &= X(1) \sqcap X(3) \\ X(3) &= X(2) \end{aligned} &
\begin{aligned} X(1) &= f(X(0)) \\ X(2) &= X(1) \sqcap X(2) \\ & \end{aligned} &
\begin{aligned} X(1) &= f(X(0)) \\ & \\ & \end{aligned} \\
\\
\text{(i)} & \text{(ii)} & \text{(iii)}
\end{array}$$

Figure 8.4: Idempotence congruences in equation systems

of congruence classes. Such an ideal congruence relation is called a *coarsest* congruence relation.

Definition 8.2 (Coarsest Congruence Relation) *Let C be a congruence relation on a vertex set V and let π be partition of V into congruence classes according to C. C is a* **coarsest** *congruence relation on V if any other congruence relation C' on V induces a partition with at least as many congruence classes as π.*

If C is a coarsest congruence relation then the induced congruence class partition π is called a coarsest partition. Note that the *finest* congruence partition results if every congruence class contains only a single vertex.

8.2.3 Congruence by Idempotence

This section describes the detection of congruences among data flow equations that result from the idempotence of the meet operator $\sqcap$. Consider a data flow equation of the form:

$$X(n) = f_n(\bigsqcap_{m \in pred(n)} X(m)).$$

Trivial congruences result from a special case, where the function f_n is the identity function and node n has only a single predecessor m. In this case the equation reduces to a simple copy equation $X(n) = X(m)$. Clearly, the fixed points of $X(n)$ and $X(m)$ are identical and $X(n)$ and $X(m)$ are congruent.

The congruence relation based on copies can be easily computed in a single pass over the equation system. Initially, each equation $X(n)$ is in a separate congruence class. For each copy equation $X(n) = X(m)$ that is encountered, the congruence class of $X(m)$ is merged with the class of $X(n)$ creating a single class. A reduced equation system without copies is constructed by including from each congruence class only a single representative equation. Each operand that occurs in an included equation is replaced by the representative of its congruence class.

Idempotence congruence extends this trivial notion of copy congruences by also covering *hidden copies*. A hidden copy is an equation of the form $x = y \sqcap z$ with y and z being congruent. By the idempotence of the meet operator, the congruence of y and z implies that $\mathit{gfp}(y) \sqcap \mathit{gfp}(z)$ reduces to $\mathit{gfp}(y)$ and equation x is essentially a copy. Thus, it can be determined that all three variables x, y, and z are congruent.

Definition 8.3 (Congruence by idempotence (IP)) *Let $G = (V, E)$ be an equation graph. A relation C on V is called an* **idempotence congruence (IP) relation***, if $(v, w) \in C$ implies one of the following conditions:*

(1) *$v = w$ (the vertices v and w are identical), or*

(2) *one of the vertices, say v, is labeled $\sqcap$ and $(v, u) \in E$ implies $(u, w) \in C$.*

To verify that C is indeed a congruence relation we have to ensure that the base case of the recursive rule (2), as well as the application of rule (2), can only yield congruent pairs of vertices. The base case of rule (2) declares $(v, w) \in C$ if w is the sole destination of edges leaving v. In this case v represents a copy equation and thus v and w are congruent. If all destinations of edges leaving v are congruent to a vertex w then v reduces to w by idempotence and v and w are congruent (application of rule 2).

By its recursive definition, the IP relation is not unique if G contains cycles. Consider the equations in Figure 8.4 (i). The partition $\pi_1 = \{c_1 = \{X(1)\}, c_2 = \{X(2), X(3)\}\}$ with the corresponding system shown in Figure 8.4 (ii) describes an IP relation. However, the partition $\pi_2 = \{c_1 = \{X(1), X(2), X(3)\}\}$ also describes an IP relation that provides the reduced system shown in Figure 8.4 (iii). Note that the congruence between $X(1)$ and $X(2)$ only holds with respect to the *greatest* fixed point defined with the initial value $\top$ at each equation.

The goal is to find the maximal IP relation (fewest number of congruence classes) for an equation graph. The symbol $C^\star$ is used to refer to the maximal IP relation according to Definition 8.3. The relation $C^\star$ provides the *coarsest* partition $\pi^\star$ of the vertices in an equation graph such that two vertices are in the same partition only if they are congruent according to Definition 8.3.

Figure 8.5 illustrates equation system reductions that can be achieved through congruence partitioning. Figure 8.5 (i) restates the original equation system from the example of Figure 8.3. The reduced equation system that results from IP partitioning is shown in Figure 8.5 (ii).

The next section presents a fast partitioning algorithm to compute $\pi^\star$ from an initial partition π that places all possibly congruent pairs of equations in the same class. The partition π is iteratively refined until a stable partition $\pi^\star$ is reached that is consistent with the definition of $C^\star$. Given partition $\pi^\star$, the equation system that is minimized with respect to IP is constructed in the same way as previously described. That is, from each

$$\begin{aligned}
X(1) &= \mathit{init}\\
X(2) &= X(1) \sqcap X(7)\\
X(3) &= \mathit{kill}[p](X(2) \sqcap (p, x)\\
X(4) &= X(3)\\
X(5) &= X(3)\\
X(6) &= X(4) \sqcap X(5)\\
X(7) &= X(6)\\
X(8) &= \mathit{kill}[x](X(7))\\
X(9) &= X(8) \sqcap (p, x)\\
X(10) &= X(8) \sqcap (p, x)\\
X(11) &= X(9) \sqcap X(10)
\end{aligned}$$

(i)

$$\begin{aligned}
X(1) &= \mathit{init}\\
X(2) &= X(1) \sqcap X(3)\\
X(3) &= \mathit{kill}[p](X(2)) \sqcap (p, x)\\
X(8) &= \mathit{kill}[x](X(3))\\
X(9) &= X(8) \sqcap (p, x)\\
X(10) &= X(8) \sqcap (p, x)\\
X(11) &= X(9) \sqcap X(10)
\end{aligned}$$

(ii)

$$\begin{aligned}
X(1) &= \mathit{init}\\
X(2) &= X(1) \sqcap X(3)\\
X(3) &= \mathit{kill}[p](X(2)) \sqcap (p, x)\\
X(8) &= \mathit{kill}[x](X(3))\\
X(9) &= X(8) \sqcap (p, x)
\end{aligned}$$

(iii)

Figure 8.5: Original equation system (i), reduced system by IP (ii), and the reduced system after combined partitioning by CSE and IP (iii).

congruence class in $\pi^\star$ only one representative equation is included. The resulting equation system contains no copy equations and no hidden copies due to idempotence.

8.2.4 Partitioning Algorithm

Computing the partition $\pi^\star$ by iterative refinement requires first determining an appropriate initial partition. If two vertices are initially placed in different congruence classes they can never be discovered to be congruent. Thus, the initial partition must overestimate the congruence relation $C^\star$. A partition π overestimates $C^\star$ if $(v, w) \in C^\star$ implies that the vertices v and w are in the same congruence class in π. In order to enable the partitioning algorithm to converge quickly to $\pi^\star$, the initial partition should be the *finest* partition that overestimates $C^\star$.

Standard graph partitioning algorithms [AHU74] are based on an initial partition of the vertices by their label. Unfortunately, the same approach cannot be pursued for computing $C^\star$. Although function vertices with a different label cannot be IP congruent, meet vertices may be congruent to any function vertex. Therefore, a new partitioning algorithm is presented that is based on an overestimate of the initial partition of the vertices that can be constructed in a canonical way.

Congruence classes in a partition are represented as *reverse trees* of vertices in an equation graph G. A reverse tree is a tree in which edges are directed from children to parent

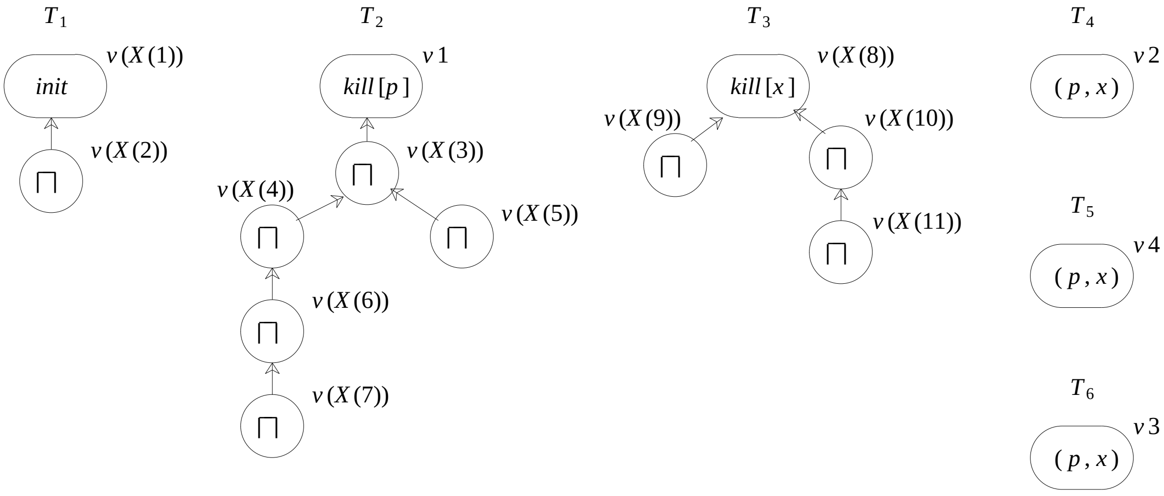


Figure 8.6: Reverse DFST partition of the equation graph from Figure 8.3.

vertices. Thus, $\pi = T_1, ..., T_k$ is a collection of disjoint reverse trees and each tree T_i is a subgraph of the equation graph G. The trees in a partition will be simply referred to as trees and the following notation is used for a given partition forest π. The root vertex of a tree T in π is denoted *root(T)*. For a given vertex v in a tree T, *parent*(v) is the unique predecessor of v in G that is contained in T.

An initial partition of the vertices in an equation graph G is constructed during a single reverse depth-first traversal of G, i.e., a depth-first traversal of the transposed graph of G. The resulting partition contains one tree (congruence class) for each function vertex in G. The tree T_v for a function vertex v is constructed by traversing each reachable edge in reverse direction, such that T_v is a reverse depth-first spanning tree (DFST) that is rooted at v and that does not include any other function vertex. The resulting forest of reverse DFSTs is called a *reverse DFST partition.* A reverse DFST partition for the equation graph from Figure 8.3 is shown in Figure 8.6. Figure 8.7 shows the procedure to construct an initial reverse DFST partition.

A reverse DFST partition for an equation graph is not unique since selections among multiple candidates to visit next are made arbitrarily. The following lemma shows that any reverse DFST partition π safely overestimates $C^\star$.

Lemma 8.1 *Let π be a reverse DFST partition for a graph G and let v and w be vertices in G. If $(v, w) \in C^\star$ then v and w are in the same tree in π.*

Proof: For a vertex v in a tree T in π the notation *level*(v) is used to denote the length of the path from v to *root*(T). Given two distinct trees T_1 and T_2 in π, it is first shown

```
**Procedure** *Reverse DFST Partitioning*
**input:**  equation graph $G = (V_f \cup V_\sqcap, E)$
**output:** partition $\pi = T_1, \ldots, T_k$, where $k = |V_f|$ and each $T_i$ is a reverse tree contained in $G$
**begin**
**for each** $v_i \in V_f$ **do**
    create a new tree $T_i$ in $\pi$;
    *dfst(* $T_i, v_i$*)*;
**endfor**
**end**
```

```
**Procedure** *dfst(t,v) /* construct a depth-first spanning tree */*
**input:**  a tree $t$ and a vertex $v$
**begin**
    **for each** unvisited meet predecessor $w$ of $v$ **do**
        add the edge $(w, v)$ to $t$;
        *dfst(t,w)*;
    **endfor**;
**end**;
```

Figure 8.7: Algorithm to construct a reverse DFST partition.

that if v is a vertex in T_1 then $(v, \mathit{root}(T_2)) \notin C^\star$ by induction on $l = \mathit{level}(v)$. (Basis $l = 0$) Clearly, $(\mathit{root}(T_1), \mathit{root}(T_2)) \notin C^\star$ since two distinct function vertices cannot be congruent by idempotence. (Ind. $l > 0$) By hypothesis $(w, \mathit{root}(T_2)) \notin C^\star$ if $\mathit{level}(w) < l$. Assume $(v, \mathit{root}(T_2)) \in C^\star$ and $\mathit{level}(v) = l$. Then by rule (2) of Definition 8.3 also $(\mathit{parent}(v), \mathit{root}(T_1)) \in C^\star$ which contradicts the hypothesis since $\mathit{level}(\mathit{parent}(v)) < l$.

Consider now two vertices v and w that are in distinct trees T_1 and T_2 and neither v nor w are the root vertex in their tree. If $(v, w) \in C^\star$ then it follows by rule (2) of Definition 8.3 that for the parent of at least one of the vertices, say v, $(\mathit{parent}(v), w) \in C^\star$. Repeatedly applying this argument will eventually show that the root vertex of one of the trees must be congruent under $C^\star$ to a vertex in the other tree, which was however shown not to be possible. Hence, $(v, w) \notin C^\star$. □

Figure 8.8 displays procedure *Partition* that operates on an initial reverse DFST partition π by subsequently refining π until the current partition is consistent with the definition of $C^\star$. In the resulting partition $\pi^\star$ two vertices v and w are left in the same tree only if $(v, w) \in C^\star$.

```
Procedure Partition /* Partitioning by idempotence */
input:  Equation graph $G = (V = V_f \cup V_\sqcap, E)$
output: Partition $\pi^\star = T_1, \ldots, T_k$ of $V$ according to $C^\star$
begin
1.   create an initial reverse DFST partition $\pi = T_1, \ldots, T_l$ of the vertices in $V$;
2.   worklist $\leftarrow \{T_1, \ldots, T_l\}$;
3.   while worklist $\neq \emptyset$ do
4.       select and remove a tree $T$ from worklist;
5.       splitlist $\leftarrow \{v \in V_\sqcap \mid v$ has one successor in $T$ and one successor not in $T\}$ ;
6.       for each $u \in$ splitlist such that $u$ is not a root vertex in $\pi$ do
7.           let $T_1$ be the tree containing vertex $u$;
8.           add $T_2 \leftarrow split(u)$ as a new tree to $\pi$;
9.           if $T_1 \in$ worklist then add $T_2$ to worklist
10.          else add the smaller of $T_1$ and $T_2$ to worklist;
11.      endfor;
12.  endwhile;
end
```

Figure 8.8: Idempotence partitioning algorithm.

The operation *split*(v) disconnects and returns the subtree rooted at v. Procedure *Partition* maintains two lists of vertices, *worklist* and *splitlist*. *Worklist* is a list of current partition trees to be examined. Each tree T in *worklist* is examined in line (5) to determine whether it contains an interior vertex v that has a successor not in T. In this case, the vertices v and *parent*(v) in T cannot be IP congruent. To ensure that the two vertices do not remain in the same tree, vertex v is placed in *splitlist*. During the inner loop the tree of each vertex u in *splitlist* is split by disconnecting the subtree rooted at u. After the split one of the two resulting subtrees is placed in *worklist* to ensure that vertices that may trigger a subsequent split will be examined. *Partition* terminates when *worklist* is exhausted with the final partition $\pi^\star$.

Example: Consider the application of procedure *Partition* to the initial reverse DFST partition from Figure 8.6. The initial reverse DFST partition π and the final partition $\pi^\star$, after procedure *Partition* terminates, are shown in Figure 8.9. In Figure 8.9 the congruence classes of each partition are displayed in columns. The final partition $\pi^\star$ describes the congruences in that system that result from the copy equations $X(4), X(5), X(7)$ and from the hidden copy equation $X(6)$. Specifically, all equations in the column for $X(3)$ in $\pi^\star$ are found to have the same fixed point as equation $X(3)$. The reduced equation systems in which the four redundant (hidden) copy equations are eliminated was shown in Figure 8.5

π

$X(1)$	$X(3)$	$X(8)$
$X(2)$	$X(4)$	$X(9)$
	$X(5)$	$X(10)$
	$X(6)$	$X(11)$
	$X(7)$	

(i)

$\pi^\star$

$X(1)$	$X(2)$	$X(3)$	$X(8)$	$X(9)$	$X(10)$	$X(11)$
		$X(4)$				
		$X(5)$				
		$X(6)$				
		$X(7)$				

(ii)

Figure 8.9: Initial (i) and final (ii) partition of the equation system from Figure 8.3.

(ii) next to original equation system in (i).

Analysis

It will be shown that procedure *Partition* computes the congruence relation $C^\star$, that is, the output partition $\pi^\star$ is the coarsest partition, such two vertices v and w are contained in the same tree in $\pi^\star$ only if $(v, w) \in C^\star$. The proof proceeds by first showing in Lemma 2 that $\pi^\star$ is consistent with the definition of $C^\star$, that is, $\pi^\star$ is not too coarse. Then Lemma 3 shows that procedure *Partition* is optimal in that $\pi^\star$ is the coarsest consistent partition.

Lemma 8.2 (Consistency) *Partition $\pi^\star$ is consistent with the definition of $C^\star$, for if v is a vertex in a tree T in $\pi^\star$ and v is not the root vertex of T then all successors of v are also in T.*

Proof: Assume v is a vertex in a tree T in $\pi^\star$ that is not the root vertex of T. Then v has one successor *parent*(v) in T. Assume that contrary to the claim v has another successor w not in T. In the initial partition π, vertex v is in some tree $T_1 \supseteq T$ and all trees are initially placed in *worklist*. The construction of *splitlist* in line (5) implies that w must also be in T_1 since otherwise a split during the first iteration would have separated vertex v from *parent*(v) contradicting the assumption. Now, consider the point during the algorithm at which vertex w is separated from the vertices v and *parent*(v) and the vertices are placed in two different trees $T_2 \subseteq T_1$ containing w and $T_2' \subseteq T_1$ containing v and *parent*(v). After this separation at least one of T_2 and T_2' will be in *worklist*, which implies that vertex v will be separated from *parent*(v) after the new contents of *worklist* are exhausted, which again contradicts the assumptions. Hence, all successors of v must be in T. □

Lemma 8.3 (Optimality) *Partition $\pi^\star$ is as coarse as possible; that is, if $(v, w) \in C^\star$ then v and w are in the same tree in partition $\pi^\star$.*

Proof: It will be shown by induction on the number i of *split* operations performed in procedure *Partition* that two vertices v and w are in two distinct trees only if $(v, w) \notin C^{\star}$. (Basis $i = 0$) The claims holds for the initial partition by Lemma 1. (Ind. $i > 0$) Let π be the partition resulting after $i \perp 1$ split operations. The i-th split operation splits an edge (v, w) in some tree T only if v has another successor u in a different tree and by induction hypothesis: $(u, w) \notin C^{\star}$ and $(u, root(T)) \notin C^{\star}$. Hence, by rule (2) of Definition 8.3: $(v, w) \notin C^{\star}$ and also $(v, root(T)) \notin C^{\star}$. Let T_1 be the subtree of T rooted at v and let T_2 be the remaining portion of T after disconnecting T_1. Since the root vertices of the two trees, v and $root(T)$, are not congruent under $C^{\star}$, an analogous induction argument to the one in the proof of Lemma 1 shows that no vertex in T_1 can be congruent to a vertex in T_2 under $C^{\star}$. Thus, two vertices are in different trees in the new partition only if they are not congruent under $C^{\star}$. □

Corollary 8.1 *Procedure Partition correctly computes the IP relation $C^{\star}$ (by Lemma 7.2 and Lemma 7.3).* □

Consider the complexity of procedure *Partition* and show that procedure *Partition* can be implemented in $O(n \log n)$ time and $O(n)$ space, where n is the number of vertices in the equation graph G. Constructing the initial partition takes $O(n)$ time. To calculate the total time spent in the while loop, consider the number of times the tree of each vertex can be placed in *worklist*. Each time the current tree of a vertex w is added to *worklist* the tree's size is at most half the size of the previous tree containing w. Hence, a vertex' tree can be added at most $\log n + 1$ times to *worklist*. *Splitlist* is constructed by a scan of the vertices whose tree was removed from *worklist* and the total number of vertices scanned is $O(n \log n)$. Operation *split* is executed at most n times, since there can be at most n partitions. Each call to *split* is implemented in $O(1)$ time by maintaining for each vertex a pointer to its position in the partition forest. To find the smaller of the two subtrees after a split in time proportional to the smaller tree (i.e., in total time $O(n \log n)$), the vertices in the two trees are counted by alternating between the trees after each vertex. The algorithm also requires a pointer for each vertex to its current partition tree, which is updated after each split only for the vertices of the smaller resulting tree. In summary, the total time spent in executing procedure *Partition* is $O(n \log n)$. The size of no auxiliary data structure is more than $O(n)$ and $O(n)$ space is used to store the partition.

If the equation graph is constructed as described in Section 8.2, the size n of the graph is linear in the size of the program. In data flow problems that are based on a product lattice L^V, such as constant propagation, the equation at each program point is a vector $x = (x_1, \ldots, x_V)$. In constant propagation there is a component x_i for each of the V program variables. In general, it will be beneficial to break the vector equation x into a set of V components equations $x_1, \ldots, x_V$ in order to expose additional congruences. In this granularity, the size of the equation graph increases to $V \times n$.

8.2.5 Congruence by Common Subexpression

Additional reductions in an equation system can be achieved by extending the definition of congruence to capture redundancies that result from sources other than idempotence. In [DST80, NO80] congruence relations are defined based on common subexpressions. For example, consider the equation system after IP partitioning in Figure 8.5 (ii). The term $X(8) \sqcap (p, x)$ is a common subexpression in equations $X(9)$ and $X(10)$. The congruence relation by common subexpressions is defined below by observing the commutativity of the meet operator.

Definition 8.4 (Congruence by common subexpression (CSE)) *Let $G = (V, E)$ be an equation graph. A relation S on V is called* **common subexpression congruence (CSE) relation** *if for vertices v and w with successors $v_1, \ldots, v_k$ and $w_1, \ldots, w_k$, $(v, w) \in S$ implies label(v) = label(w) and $\forall\ 1 \leq i \leq k$:*

$$\begin{cases} (v_i, w_{p(i)}) \in S \text{ for some permutation } p \text{ on } \{1, \ldots k\} & \text{if } label(v) = \sqcap \\ (v_i, w_i) \in S & \text{otherwise} \end{cases}$$

Partitioning a graph by CSE is a well known problem and a fast $O(n \log n)$ algorithm follows from Hopcroft's algorithm for minimizing finite automata [Hop71]. Among other applications, Hopcroft's algorithm was used to eliminate common subexpression in program optimization [AWZ88]. This chapter presents a different application by employing the algorithm to reduce data flow equation systems.

Hopcroft's algorithm starts with an initial partition π in which all vertices with an identical label are placed in the same congruence class in π. The algorithm iterates over the congruence classes to subsequently refine the current partition until it is consistent with Definition 8.4. The algorithm terminates with the coarsest partition in which two equations are in the same class only if they are congruent under S. An adaptation of Hopcroft's partitioning algorithm to partition equation graphs is shown in Figure 8.10.

Consider the application of Hopcroft's algorithm over the equation graph for the alias analysis of procedure *Insert* shown in Figure 8.3. The two equations $X(9) = (p, x) \sqcap X(8)$ and $X(10) = X(8) \sqcap (p, x)$ are discovered to be CSE congruent. The discovery of CSE congruences may enable the detection of additional congruences by idempotence. For example, once it is known that the two equations $X(9)$ and $X(10)$ are congruent, it can in turn be determined that equation $X(11) = X(9) \sqcap X(10)$ is actually a hidden copy and in fact all three equations $X(9)$, $X(10)$ and $X(11)$ are congruent. To enable these second order effects, the results of CSE partitioning can be incorporated into the initial partition for IP partitioning. This is achieved by applying procedure *Partition* to the equation graph that results if all vertices that were already found to be congruent are merged into a single

```
Procedure An adaptation of Hopcroft's partitioning algorithm
input:  Equation graph $G = (V = V_f \cup V_\sqcap, E)$
output: Partition $\pi^\star = C_1, \ldots, C_k$, where $C_i$ is a collection of vertices in $G$
begin
  create an initial partition $\pi = C_1, \ldots, C_l$ of the vertices in $V$ by their label;
  worklist $\leftarrow \{C_1, \ldots, C_l\}$;
  while worklist $\neq \emptyset$ do
      select and remove $C_i$ from worklist;
      for $n \leftarrow 1$ to 2 do
          splitlist $\leftarrow \{v \in V_f \mid$ the $n$-th succ. of $v$ is in $C_i\}$
                  $\cup\ \{v \in V_\sqcap \mid v$ has exactly $n$ succ. in $C_i\}$; /* commut. of $\sqcap$ */
          for each $C_j$ such that (splitlist $\cap\ C_j) \neq \emptyset$ and ($C_j \not\subseteq$ splitlist) do
              create a new tree collection $C$ in $\pi$;
              move each $u \in$ (splitlist $\cap\ C_j$) to $C$;
              if $C_j \in$ worklist then add $C$ to worklist
                                  else add the smaller of $C_j$ and $C$ to worklist;
          endfor; endfor;
  endwhile
end
```

Figure 8.10: An adaption of Hopcroft's algorithm for minimizing finite automata.

vertex.

Example: Consider again the equation system in Figure 8.5 (i). The reduced equation system that results if CSE partitioning is applied prior to IP partitioning is shown in Figure 8.5 (iii). The additional improvements over the equation system that result from IP partitioning (Figure 8.5 (ii)) are due to the discovery of the congruence among equations $X(9), X(10)$ and $X(11)$.

Unfortunately, applying each partitioning algorithm once may not provide optimal results. In general, congruences that are found based on IP may enable the discovery of additional common subexpressions and vice versa. Thus, to find the maximal number of congruences requires computing the transitive closure of the union of the two congruence relations. This closure can be computed by iterating over the two partitioning algorithms until no more congruence can be discovered. Each time a new iteration is started the size of the equation graph is reduced resulting in a worst case bound of $O(n^2 \log n)$.

8.2.6 Minimality

The previous sections presented algorithms to discover congruences by exploiting the idempotence property of the meet operator and commutativity of the meet in common subexpressions. By computing the transitive closure of congruence partitionings based on idempotence and based on common subexpressions, reduced equation systems are constructed that contain no redundancies by idempotence and no common subexpressions. A still unresolved question concerns the optimality of the approach, that is, the question as to whether the resulting equation systems are minimized in terms of the number of equations. Unfortunately, the problem of minimizing data flow equation systems, without evaluating any equations in the system, is NP-complete [GJ79]. The NP-completeness of this minimization problem is due to the existence of associative operations in the systems, such as the meet operation. The difficulty of discovering congruences by associativity results from the fact that an exponential number of different sequences of meet operations can yield congruent values by associativity. By associativity of the meet two equations may be congruent even if they are based on different sequences of operations. For example, two equations $x = x_1 \sqcap x_2$ and $y = y_1 \sqcap y_2$ are congruent by associativity of the meet if x_1 is congruent to $y_1 \sqcap z$ and y_2 is congruent to $x_2 \sqcap z$. That is, by substitution it follows that: $x \equiv (y_1 \sqcap z) \sqcap x_2)$ and $y \equiv y_1 \sqcap (x_2 \sqcap z)$.

The problem with associative operators also arises in program optimizations that are based on discovering common subexpressions. Program optimizations distinguish equivalence among expressions that are based on identical sequences of operations up to commutativity. The equivalences are termed *transparent* equivalences [RWZ88]. Non-transparent equivalences include, in addition, equivalences that result from associativity. Program optimization techniques are usually limited to the discovery of *transparent* equivalence. Heuristics have been used to discover some of the equivalences that result from associativity by using *reassociation techniques* [CM80].

8.3 Data Flow Solutions by Congruence Partitioning

This section returns to a closer inspection of the properties of reverse DFST partitions as constructed by the algorithm from Figure 8.7. It was shown in Lemma 8.1 that every reverse DFST partition is a safe overestimate of the IP partition of an equation system. This result is obtained without interpreting any function symbol in the equation system other than the meet operator. This section shows that stronger properties of reverse DFST partitions can be proven if the partial order $\sqsubseteq$ among lattice elements is taken into account when partitioning the equations. If the data flow equation system meets certain conditions, the reverse DFST partition is sufficiently powerful to completely replace the fixed point iteration and directly solve the data flow problem.

Consider the data flow problem of live variables (LIVE). With respect to a single variable the lattice L for LIVE consists of two values:

$$L = \{\bot = \mathit{live}, \top = \mathit{dead}\}.$$

Thus, the flow functions f for the analysis of a single variable are of the form:

$$f(x) = x \text{ or } f(x) = c, \text{ where } c \in L,$$

and the set of labels in any equation graph G is limited to:

$$\mathit{Label}(G) \subseteq \{\bot, \top, \sqcap\}.$$

Consider now a reverse DFST partition π constructed for an equation graph with such a limited label set and assume that the partition trees are constructed in increasing order of their labels. Thus, first the set of function vertices is ordered in increasing order of the function label according to the partial lattice order $\sqsubseteq$. During each step, a tree is constructed for the currently lowest function vertex that was not already previously included in a tree. For the above example, where function vertices are either labeled $\bot$ or $\top$, first the partition trees with label $\bot$ are constructed and then the remaining trees with label $\top$ are constructed. The modified algorithm to construct such an *ordered reverse DFST partition* is shown in Figure 8.11.

By constructing partition trees in increasing order of their root vertex label it is guaranteed that whenever a meet vertex v is included in a tree with a root label l, then l must be the lowest lattice value that reaches the meet at v. Otherwise, v would have been included in a previous tree. Thus, the value of the equation represented by v can be at most l since l contributes to the meet represented by v and the value cannot be lower than l since no lower lattice reaches the meet at v. It follows that the fixed point of the equation represented by vertex v can be already determined during the partition construction, since it must be that $\mathit{gfp}(v) = l$. Thus, the following property of a reverse DFST partition results:

$$\forall\, T \in \pi : \quad v \in T \implies (\mathit{gfp}(v) = \mathit{label}(\mathit{root}(T))).$$

The following theorem generalizes this property for a certain class of data flow problems.

Theorem 8.1 *Let $G = (V, E)$ be an equation graph over a chain lattice L such that $\mathit{Label}(G) \subseteq L \cup \{\sqcap\}$ and let π be an ordered reverse DFST partition of V:*

$$(\,\forall\, T \in \pi\,) : \quad v \in T \implies \mathit{gfp}(v) = \ \mathit{label}(\mathit{root}(T)).$$

Proof: The proof proceeds by induction on the number k of vertex inclusions during the construction of π. ($k = 1$) Let l be the root vertex label of the tree T that includes the first vertex v_1. Then v_1 is an immediate predecessor of the root of T and therefore $\mathit{gfp}(v_1) \leq l$. Moreover, l must be the lowest element that enters the meet at vertex v_1. If there would be a vertex w with a label lower than l and w would be reachable from v_1 then v_1 would have

```
Procedure Ordered Reverse DFST Partitioning
input:   equation graph $G = (V_f \cup V_\sqcap, E)$
output: partition $\pi = T_1, \ldots, T_k$, where $k = |V_f|$ such that if $v \in T_i$ then $gfp(v) = label(root(T_i))$
begin
let $S$ be the sorted sequence of vertices in $V_f$ in increasing order of their labels in $L$;
for each $v \in S$ in increasing order do
    create a new tree $T$ in $\pi$;
    dfst( $T, v$);
endfor
end
```

Figure 8.11: Algorithm to construct an ordered reverse DFST partition.

been included in the tree rooted at w, contradicting the assumption. Hence, $gfp(v_1) \geq l$. ($k > 1$) Let v_k be the kth-vertex included and let l be the label of the root vertex of the tree T that includes v_k. Let w be the predecessor of v_k in T. By induction hypothesis $gfp(w) = l$ and therefore $gfp(v_k) \leq l$. As in the case for $k = 1$, l must be the lowest value entering the meet at vertex v_k since otherwise vertex v_k would have been included earlier in a tree whose root label is lower than l. Hence also $gfp(v_k) \geq l$. □

A data flow problem with a chain lattice satisfies the label set requirement $Label(G) \subseteq L \cup \{\sqcap\}$ if all flow functions f are *meet linear functions* [Zad84], that is:

$$f(x) = c \quad \text{or} \quad f(x) = x \sqcap c \ \text{, where } c \text{ is a constant in } L.$$

For example, the class of partitionable problems can be decomposed (i.e, partitioned) into k disjoint problems, one for each variable or program expression, respectively, such that each disjoint problem has only meet linear functions. Examples of partitionable problems are the four classical bit vectors problems REACH, LIVE, AVAIL and BUSY.

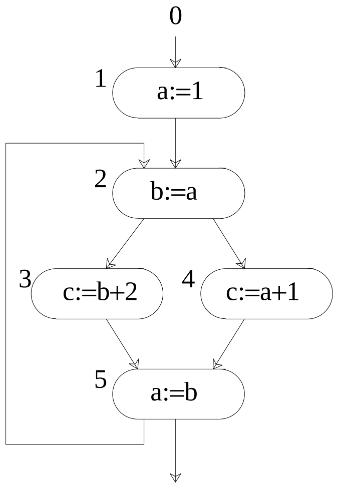
Constructing ordered reverse DFST partitions provides an efficient way of solving simple data flow problems, namely the partitionable problems. The classical bit vectors problems have a lattice of height two. Thus, the worst case partitioning time per variable is linear, resulting in $O(k \times n)$ total time to solve the problem for all k variables (or program expressions, respectively). The solution is computed simply by a series of reverse DFST partitions, requiring no bit vector operations or equation evaluations at any point.

8.4 Comparison with Sparse Evaluation Graphs

This section compares congruence partitioning with the related approach of sparse evaluation graphs (SEG) [CCF90]. Instead of directly reducing a data flow equation system, the SEG approach specializes a program's control flow graph G with respect to each analysis problem such that an equation system will be generated that is smaller than it would be using G. A SEG is obtained from a control flow graph G by eliminating some of the nodes in G that have an identity flow function. The equation system that results from a SEG consists of (1) the equations that are based on non-identity functions and (2) meet equations that are needed to combine new information. The construction of a SEG requires $O(e + n^2)$ time using dominance frontiers [CCF90] and $O(e \times \alpha(e))$ time using a more recent algorithm [CF93], where e is the number of edges in a program's control flow graph G and n is the number of nodes in G. If possible, a data flow problem that is based on a product lattice L^V is broken into V separate problems in order to increase the likehood of nodes with an identity flow functions. In this case, a series of V separate SEGs are constructed requiring $O(e \times V + n^2)$ time or $O(V \times e \times \alpha(e))$ time using the fast algorithm.

SEGs target the elimination of the same kind of redundancies as IP congruence partitioning, namely (1) identity flow functions that result in copy equations and (2) redundant meet equations that combine identical information. However, there are important data flow problems for which IP congruence partitioning results in strictly smaller equation systems than the SEG approach. The SEG approach is not capable of eliminating congruent copy operations if the congruence is not a result of an identity flow function at the respective node. Consider a data flow problem with a product lattice L^V that cannot be divided into V disjoint problems. Constant propagation is an example of such a problem. Figure 8.12 shows the constant propagation equation system for a control flow graph fragment. The equation system in Figure 8.12 (ii) shows each solution vector expanded into a set of equations $X(n)_v$ for each variable v. Thus, each equation $X_v(n)$ expresses whether variable v has constant value on exit of flow graph node n. Since each node in the graph modifies the value of at least one variable, there are no nodes with a complete identity flow function in this example. It follows that the equation system cannot be improved by the SEG approach since no node can be eliminated. Thus, the SEG for this example would be identical to the original graph. However, the equation system can be reduced using congruence partitioning. After applying the IP partitioning algorithm from Figure 8.8, the partition $\pi^\star$ of the equation system shown in Figure 8.13 (i) is obtained. The corresponding reduced equation system based on partition $\pi^\star$ is shown in Figure 8.13 (ii).

Furthermore, congruence partitioning is extensible to additional types of congruence relations, such as congruence by common subexpressions. In contrast, redundancies that result from common subexpressions cannot be eliminated using SEGs.



(i)

$$
\begin{aligned}
X_a(1) &= \{1\} \\
X_b(1) &= \mathit{init} \\
X_c(1) &= \mathit{init} \\
X_a(2) &= X_a(1) \wedge X_a(5) \\
X_b(2) &= X_a(1) \wedge X_a(5) \\
X_c(2) &= X_c(1) \wedge X_c(5) \\
X_a(3) &= X_a(2) \\
X_b(3) &= X_b(2) \\
X_c(3) &= X_b(2) \oplus \{2\} \\
X_a(4) &= X_a(2) \\
X_b(4) &= X_b(2) \\
X_c(4) &= X_a(2) \oplus \{1\} \\
X_a(5) &= X_b(3) \wedge X_b(4) \\
X_b(5) &= X_b(3) \wedge X_b(4) \\
X_c(5) &= X_c(3) \wedge X_c(4)
\end{aligned}
$$

(ii)

Figure 8.12: A flow graph fragment (i) and the induced equation system for constant propagation (ii).

$$\begin{aligned}\pi^\star = \{ \quad & T_1 = \{X_b(1)\},\ T_2 = \{X_c(1)\},\ T_3 = \{X_c(2)\},\ T_4 = \{X_c(3)\}\ ,T_5 = \{X_c(4)\}\ , \\ & T_6 = \{X_c(5)\},\ T_7 = \{X_a(1), X_a(2), X_b(2), X_a(3), X_b(3), X_a(4), X_b(4), X_a(5), X_b(5)\}\}\end{aligned}$$

(i)

$$\begin{aligned} X_a(1) &= \{1\} \\ X_b(1) &= \mathit{init} \\ X_c(1) &= \mathit{init} \\ X_c(2) &= X_c(1) \wedge X_c(5) \\ X_c(3) &= X_a(1) \oplus \{2\} \\ X_c(4) &= X_a(1) \oplus \{1\} \\ X_c(5) &= X_c(3) \wedge X_c(4) \end{aligned}$$

(ii)

Figure 8.13: Idempotence congruence partiton $\pi^\star$ (i) and the reduced equation system (ii).

8.5 Related Work

Several forwarding techniques use a derived graph representation that embodies some form of du-chain information. The *global value graph* [RT82, RL77], the *program dependence graph* (PDG) [FOW87] and *static single assignment form* (SSA) [CFR$^+$91] are examples of these derived graphs. The primary aim of the PDG is to facilitate the application of optimizing and parallelizing code transformations. In data flow analysis, the PDG has been used for program slicing [OO84]. Using SSA form, efficient data flow algorithms have been developed for constant propagation [WZ85], redundancy detection [RWZ88], global value numbering [AWZ88], code motion [CLZ86] and induction variable detection [Wol92]. However, these approaches are not general in that the derived graphs can only facilitate the analysis of problems that take advantage of definitions-use connections. A common problem that does not benefit from definition-use connections is the computation of available expressions.

Another approach to exploit direct forwarding opportunities is based on constructing a specialized graph for each data flow problem to be solved. The *partitioned variable technique (PVT)* [Zad84] is an approach, applicable to only partitionable problems, that allows the partitioning of the original problem into a series of independent and simpler problems, one for each variable (see also discussion in Section 2.2). PVT requires the construction of a derived graph for each program variable. Once the graph is available, the fixed point solution is found during a topological graph traversal. An approach, similar to PVT, that is also limited to partitionable problems is the *slotwise analysis* described in [DRZ92]. Like PVT, slotwise analysis breaks a data flow problem into a series of single-bit problems. However, slotwise analysis does not require the explicit construction of a derived graph and uses a worklist algorithm to enable the information forwarding on the control flow graph.

The most general of the previous approaches to forwarding are the *sparse evaluation graphs* (SEG) [CCF90]. The previous section provided a detailed comparison of the SEG approach with congruence partitioning.

Computing congruence relations based on common subexpressions is a well known problem and efficient algorithms have been developed [NO80, DST80, Hop71]. Hopcroft's partitioning algorithm for minimizing finite automata was used in program optimization to detect equalities among variables based on common subexpressions over an extended SSA form of the program [AWZ88]. The authors describe a strategy to manipulate the SSA representation in order to combine congruent (i.e., equal) variable values from different branches of a structured if-statement. This treatment can be viewed as handling a special case of detecting IP congruences. Other methods to eliminate redundant program computations include value numbering [CS70], global value numbering based on SSA form [RWZ88] and methods based on the global value graph [RT82].

8.6 Summary

This chapter presented a new and efficient approach to improve the performance of data flow analysis by reducing the size of data flow equation systems through congruence partitioning. The partitioning algorithms discover congruences among data flow equations by exploiting the algebraic properties of idempotence and commutativity of the meet operator. Congruence partitioning is a general approach that is applicable to any monotone data flow problem. Moreover, congruence partitioning can be applied to optimize the performance of either standard exhaustive analysis or of the demand-driven analysis as it is developed in this thesis. The combination of congruence partitioning with exhaustive or demand-driven analysis is discussed further in the following two sections.

8.6.1 Congruence Partitioning and Exhaustive Analysis

When combining congruence partitioning with an exhaustive analysis approach, congruence partitioning serves as a preparatory phase to the actual exhaustive analysis. Prior to starting the exhaustive fixed point computation, the congruence partitioning algorithm is applied to the exhaustive data flow equation system in order to construct a new and reduced system. The optimized data flow equation system is then passed to the exhaustive fixed point computation routine. To compute the fixed point over the reduced equation system does not require any modifications in the fixed point finding strategy. Algorithms for computing the fixed point of an exhaustive equation system can equally well be applied to a reduced equation system by simple traversing the new forwarding edges in the reduced system instead of the regular control flow edges that connect equations in the original exhaustive system.

8.6.2 Congruence Partitioning and Demand-Driven Analysis

As in the combination with exhaustive analysis, congruence partitioning serves as a preparatory phase to demand-driven analysis. Note that congruence partitioning only has to be computed once for each analysis problem. The resulting reduced system is then used for a faster query propagation for all queries that are issued for the respective analysis problem. Thus, the benefits of computing a congruence partitioning increase with the number of queries that are generated for the problem. The query propagation rules are applied after congruence partitioning by simply traversing the new forwarding edges instead of the regular control flow edges.

Note that congruence partitioning represents an additional phase in the analysis process. Although reductions in the equation system are expected, they cannot be guaranteed in general. The actual reductions that can be achieved depend highly on the program text and the specific data flow problem under consideration. Thus, as with any forwarding technique, a determination of the effective benefits of congruence partitioning requires an experimental evaluation.

Chapter 9

Concluding Remarks

9.1 Summary

This dissertation has explored new approaches for improving the efficiency of interprocedural data flow analysis with respect to both the space and time requirements of the traditional exhaustive approach. The major contribution of this work is the development of a new demand-driven approach to interprocedural data flow analysis whose practical benefits have been demonstrated through experimentation.

The demand-driven approach has been developed through a general framework. The framework is used to derive demand-driven interprocedural analysis algorithms from standard algebraic descriptions of the data flow problems. Conceptually, demand-driven algorithms result through the functional reversal of a standard exhaustive analysis. The framework is applicable to the class of distributive and finite interprocedural data flow problems. In the case of intraprocedural analysis, the framework also applies to distributive problems with infinite lattices. To handle the non-distributive case, this work also includes a two-phase framework variation. As a two-phase approach, the framework variation is generally less efficient than the single-phase analysis reversal. However, the framework extension is more general in that it is applicable to any monotone data flow problem.

Numerous experiments were carried out to evaluate the performance of demand-driven analysis in practice. Demand-driven analyzers have been implemented for two data flow problems: reaching definitions and copy constant propagation. Experiments were conducted to compare the performance of computing reaching definitions and copy constant propagation information at each use/reference of a variable using either the demand-driven analyzers or using their exhaustive counterparts. The experimental results show that the demand-driven reaching definition analyzer is faster and uses less space than exhaustive reaching definition analysis in 11 out of 17 programs. In copy constant propagation, the demand-driven analyzer outperforms the exhaustive analyzer in all 17 programs. Additional experimentation was conducted to evaluate the performance of demand-driven analysis in a specific software engineering application. The example chosen was data flow integration testing. An experimental study was performed to compare the performance of the demand-driven analyzer if used during procedure integration with that of (i) an exhaustive analyzer and (ii) an improved analyzer that is based on incremental updates. The experimental results show that demand-driven analysis is significantly faster than exhaustive analysis for all programs and even outperforms the incremental analyzer in 11 out of 12 programs. As an additional result, the experimental study has shown that the demand-driven algorithms can be easily integrated into data flow applications.

The analysis improvements achievable by a demand-driven approach are orthogonal and complimentary to the improvements of previous preparatory techniques that optimize data flow analysis performance by direct information forwarding. As an additional contribution and to further improve the analyzer's performance, this dissertation also developed a new forwarding technique, congruence partitioning, that is more general and more powerful than previous techniques for forwarding. Congruence partitioning provides a preparatory optimization technique that may be used to improve the performance of either the demand-driven algorithms developed in this work or of conventional exhaustive analysis algorithms.

9.2 Merit of the Work

A demand-driven approach has important advantages over other techniques to improve the efficiency of data flow analysis. Previous approaches to improve data flow analysis typically act as an additional preparatory phase that is performed prior to the actual data flow analysis. For example, forwarding techniques are preparatory approaches that are applied before the analysis in order to shorten the information propagation paths in the control flow graph. In contrast, the demand-driven approach developed in this thesis constitutes the actual analysis phase itself. As a consequence, using a demand-driven approach to data flow analysis in compilers and software tools significantly changes their overall design. Unlike preparatory techniques, demand-driven analysis does not obey the classical strict phased design of a compiler. In this phased design, data flow analysis is performed in isolation and independent of its context; and in particular, independent 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, it also limits the information available to each individual phase and may thereby render the phases unnecessarily inefficient. For example, performing data flow analysis in a separate phase from the actual analysis application (e.g., optimization) is likely to result in an over-analysis of the program. 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, if a demand-driven analysis is used, the actual information needs can be taken into account and the over-analysis of a program can be avoided. Demand-driven analysis is directly interleaved with the application such that analysis is performed only if triggered by a demand for information from the application. Ideally, 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.

The major contributions of the developed demand-driven approach to interprocedural data flow analysis are summarized as follows.

- Integration of analysis with applications that require data flow information.
  As discussed above, using a demand-driven analysis approach leads to a new design of compilers and data flow based software engineering tools that interleaves analysis and application.

- General approaches for distributive and non-distributive data flow problems.
  The demand-driven approach is developed as a framework and as such avoids the need for redeveloping demand-driven algorithms for each analysis problems that must be solved.

- Time and space efficient demand-driven analysis algorithms.
  Analytical analysis of the asymptotic worst case cost of the developed demand-driven algorithms shows that demand-driven analysis is never more costly than standard exhaustive analysis.

- Practical benefits experimentally demonstrated.
  The practical performance benefits of demand-driven analysis has been demonstrated through numerous experiments.

- Simple and easy to implement design of the demand-driven analysis algorithms.
  The developed demand-driven analyzers have a simple design and are easy to implement, thus providing the compiler writer with an attractive alternative to standard exhaustive algorithms,

9.3 Future Directions

The theoretical and practical results presented in this thesis encourage continuing research with respect to both the conceptual and the experimental aspects of this work.

• Additional Experimental Evaluation

To provide further insights into the performance benefits of the demand-driven approach, the experimental evaluation of demand-driven analysis can be continued. One direction of future experimental work would include the consideration of additional cost measures. The experimental results presented in this thesis are based on execution timings. Alternatively, performance can be measured through operation counts. Unlike execution timings, counting the number operations provides a performance measure that is implementation independent and uneffected by implementation related optimizations (e.g., bit vector operations).

Future experimentation will also consider additional data flow problems. The experimental results in Chapters 5 and 6 show that the benefits of demand-driven analysis are higher in copy constant propagation, which is a more complex problem than the problem of computing reaching definitions. This result suggests that the benefits of demand-driven analysis grow, not only with certain program characteristics such as program size, but also based on characteristics of the data flow problem such as its complexity. Future research could investigate how characteristics of the data flow problem, such as the complexity of the lattice and meet operator implementations, affect the performance of demand-driven analysis.

Another important extension of the experimental work would be to evaluate the performance of complete compiler optimizations based on demand-driven analysis. The experimental evaluation of demand-driven analysis in integration testing provided the first encouraging results for using demand-driven analysis in a software engineering application. Evaluating the performance of an optimizer that is based on demand-driven analysis would complement the work on integration testing and establish further insights into the usability of demand-driven analysis in an optimizing compiler. Such an experimental evaluation would serve two purposes. First, it would provide a comparison of the optimizer's performance if based on demand-driven analysis with that of a standard implementation of the optimizer. In addition, the experimental study would also reveal the ease or difficulty of designing and implementing an optimizer using a demand-driven analysis approach.

- **Experimental Evaluation of Congruence Partitioning**

Other future experimental work would include the practical evaluation of congruence partitioning. In particular, an evaluation of the benefits of combining congruence partitioning with demand-driven analysis would be interesting. The existing demand-driven analyzer implementations could be extended to include an additional preparatory phase for congruence partitioning. The results of this preparatory phase would provide forwarding information of the reduced equation system that can be used during the demand-driven query analysis to shorten the query propagation paths. Congruence partitioning may also be applied prior to standard exhaustive analysis. Thus, additional experimental evaluation would cover the benefits of congruence partitioning in a standard exhaustive compiler or software tool.

- **Parallelizing Demand-Driven Analysis**

There are a number of potential benefits of demand-driven analysis that deserve further investigation. One domain in which the demand-driven algorithms appear to be a promising approach is the parallelization of the data flow analysis. The discussion in Chapter

4 pointed out that the demand-driven algorithms are parallelizable in a straight-forward way and several modes of parallel execution of the analysis were outlined. A future experimental evaluation would determine the practical benefits of the parallel implementations of the demand-driven analysis, in particular, in comparison with other data flow analysis parallelization strategies.

- **Analysis of Non-Distributive Problems**

Other future directions target the theoretical aspects of this research. In particular, further investigations of the handling of non-distributive problems would be of interest. The concepts of precise analysis reversal that enabled the development of the demand-driven framework are not applicable to non-distributive problems. Applying analysis reversal to a non-distributive problem results in loss of precision. To precisely handle non-distributive problems, this work includes a two-phase framework extension that, although less efficient, is applicable to any monotone data flow problem. An interesting problem is the question as to whether it is possible to improve the efficiency of the two-phase framework variation by developing a hybrid approach for handling non-distributivity. A hybrid approach could be based on the observation that it is always possible to detect the first time a loss of precision occurs when analysis reversal is applied to a non-distributive problem. Thus, is may be possible to start the demand-driven analysis for a non-distributive problem optimistically using the more efficient analysis reversal approach. Only if information loss actually occurs would the analysis switch to the two-phase algorithm. Moreover, it may be possible to isolate the portions of the program that result in information loss during the query propagation. In this case, a hybrid approach would attempt to apply analysis reversal whenever possible and only switch to the less efficient two-phase approach to cover the portions of the program than cannot be analyzed precisely using the reversal based query propagation rules.

- **Combining Demand-Driven Analysis with Incremental Analysis**

Finally, the utility of using demand-driven analysis for bypassing the incremental update problem should be investigated in the future. The experimental comparison of demand-driven algorithms with incremental algorithms in integration testing has provided some promising results showing that demand-driven analysis can outperform incremental analysis. Another context where the demand-driven approach may be useful to eliminate the need for incremental analysis are optimizing compilers that perform aggressive code re-ordering. The incremental update problem of keeping the data flow solution consistent while the code is being transformed can become considerably complex if the transformations involve more than simple local code changes. If demand-driven analysis performs sufficiently well to allow its use throughout the optimization phase, such an incremental update problem would not even arise. Each time data flow information is requested, it is computed based on the latest valid version of the program.

A hybrid approach that combines the advantages of both demand-driven and incremental analysis results if the demand-driven analysis uses a cache. If the program changes while the cache is in use, the cache can be updated using the same incremental techniques that are used to update the exhaustive solution. Thus, for transformations that require only small changes, incremental techniques could be used to update the cache. After the application of more aggressive transformations that require complex updates, the cache can simply be flushed. Unlike an exhaustive approach that completely re-computes the exhaustive solution after a previous solution has been flushed, using demand-driven analysis guarantees that after flushing, only the information actually needed will be re-computed. Experimental evaluation will be necessary to determine the actual benefits of using demand-driven analysis in combination with an incremental approach.

Bibliography

[AC77] F.E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137–147, March 1977.

[AHKL93] H. Agrawal, J. Horgan, E. Krauser, and S. London. Incremental regression testing. In *Conf. on Software Maintenance*, pages 348–357, Sept. '93.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, '74.

[AK87] F.E. Allen and K. Kennedy. Automatic translation of fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, principles, techniques, and tools*. Addison-Wesley Publishing Company, Massachusetts, '86.

[AWZ88] B. Alpern, M. Wegman, and F.K. Zadeck. Detecting equality of values in programs. In *15th ACM Symp. on Principles of Programming Languages*, pages 1–11, San Diego, CA, Jan. '88.

[BE95] W. Blume and R. Eigenmann. Demand-driven symbolic range propagation. In *Workshop on Languages and Compilers for Parallelism*, Columbus, OH, Aug. '95.

[BH93] S. Bates and S. Horwitz. Incremental program testing. In *20th Annual ACM Symp. on Principles on Programming Languages*, Jan. 1993.

[Bir84] G. Birkhoff. *Lattice theory*, volume 25. American Mathematical Society, Colloquium Publication, Washington, DC, 3rd edition, '84.

[BJ78] W.A. Babich and M. Jazayeri. The method of attributes for data flow analysis: Part II . Demand analysis. *Acta Informatica*, 10(3), Oct. '78.

[BMO90] R. Ballance, A.B. Maccabe, and K.J. Ottenstein. The program dependence web: a representation supporting control-,data-, and demand-driven interpretation of imperative languages. In *SIGPLAN '90 Conf. on Programming Language Design and Implementation*, pages 257–271, Jun. '90.

[Bou93] F. Bourdoncle. Abstract debugging of high-order imperative languages. In *SIGPLAN '93 Conf. on Programming Language Design and Implementation*, pages 36–45, Albuquerque, NM, Jun. '93.

[Bur87] M. Burke. An interval analysis approach toward exhaustive and incremental data flow analysis. Technical Report RC 12702, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, '87.

[CBC93] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proc. of the 20th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 232–245, Jan. '93.

[CC77a] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *6th ACM Symp. on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, Jan. '77.

[CC77b] P. Cousot and R. Cousot. Static determination of dynamic properties of generalized type unions. In *ACM Conf. on Language Design for Reliable Software*, pages 77–93, Raleigh, NC, Mar. '77.

[CC77c] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E.J. Neuhold, editor, *IFIP Conf. on Formal Description of Programming Concepts*, pages 237–277. North-Hollan Pub. Co., '77.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6th ACM Symp. on Principles of Programming Languages*, pages 269–282, San Antonio, TX, Jan. '79.

[CCF90] J.D. Choi, R.K. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *18th ACM Symp. on Principles of Programming Languages*, pages 55–66, Orlando, FL, Jan. '90.

[CCF92] J.D. Choi, R. Cytron, and J. Ferrante. On the efficient engineering of ambitious program analysis. *IEEE Trans. on Software Engineering*, 20(2):105–114, Feb. '92.

[CF93] R.K. Cytron and J. Ferrante. Efficiently computing ϕ-nodes on-the-fly. *'93 Workshop on Languages and Compilers for Parallelism*, 1993.

[CFR+91] R.K. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F.K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, 13(4):451–490, Oct. '91.

[CG93] R. Cytron and R. Gershbein. Efficient accommodation of may-alias information in SSA form. In *SIGPLAN '93 Conf. on Programming Language Design and Implementation*, pages 36–45, Albuquerque, NM, Jun. '93.

[CHK92] K. Cooper, M. Hall, and K. Kennedy. Procedure cloning. In *IEEE 1992 Int. Conf. on Computer Languages*, pages 96–105, San Francisco, CA, April 1992.

[CK88] K. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. *SIGPLAN '88 Symp. on Compiler Construction, published in SIGPLAN Notices*, 23(7):57–66, Jun. '88.

[CLZ86] R.K. Cytron, A. Lowry, and F.K. Zadeck. Code motion of control structures in high-level languages. In *13th ACM Symp. on Principles of Programming Languages*, pages 70–85, St. Petersburg Beach, FL, Jan. '86.

[CM80] J. Cocke and P.W. Markstein. Measurement of program improvement algorithms. In *Information Processing 80.* North Holland Publishing Company, '80.

[Coc70] J. Cocke. Global common subexpression elimination. *SIGPLAN Notices*, 5(7):20–24, July 1970.

[Coo85] K. Cooper. Analyzing aliases of reference formal parameters. In *12th ACM Symp. on Principles of Programming Languages*, pages 281–290, '85.

[Cou81] P. Cousot. Semantic foundations of program analysis. In S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 303–342. Prentice-Hall, '81.

[CPRS85] L.A. Clarke, A. Podgurski, D. Richardson, and S.Zeil. A comparison of data flow path selection criteria. In *8th Int. Conf. on Software Engineering*, pages 244–251, Aug. '85.

[CS70] J. Cocke and J.T. Schwartz. Programming languages and their compilers; preliminary notes. Courant Institute of Mathematical Sciences, New York University, Apr. '70.

[CS89] D. Callahan and J. Subhlok. Static analysis of low-level synchronization. *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):100–111, Jan. '89.

[CWZ90] D.R. Chase, M. Wegman, and F.K. Zadeck. Analysis of pointers and structures. In *Proc. of the SIGPLAN '90 Conf. on Programming Language Design and Implementation*, pages 296–310, White Plains, NY, Jun. '90.

[Deu94] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *Proc. of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 230–241, Orlando, FL, Jun. '94.

[DGS92a] E. Duesterwald, R. Gupta, and M.L. Soffa. Distributed slicing and partial re-execution of distributed programs. In *Fifth Workshop on Languages and Compilers for Parallelism*, pages 497–511, New Haven, Connecticut, Aug. '92. Springer Verlag, LNCS 757.

[DGS92b] E. Duesterwald, R. Gupta, and M.L. Soffa. Rigorous data flow testing through output influences. In *2nd Irvine Software Symposium*, pages 131–145, Irvine, CA, Mar. '92.

[DRZ92] D.M. Dhamdhere, B.K. Rosen, and F.K. Zadeck. How to analyze large programs efficiently and informatively. In *SIGPLAN '92 Conf. on Programming Language Design and Implementation*, pages 212–223, San Francisco, CA, Jun. '92.

[DS91] E. Duesterwald and M.L. Soffa. Static concurrency analysis in the presence of procedures using a data-flow framework. In *ACM Symp. on Testing, Analysis, and Verification*, pages 36–48, Victoria, B.C., Oct. '91.

[DST80] P.J. Downey, R. Sethi, and R.E. Tarjan. Variations on the common subexpression problem. *Journal of the ACM*, 27(4):758–771, Oct. '80.

[EGH94] M. Emami, R. Ghiya, and L.J. Hendren. Context-sensitive interprocedural points-to analysis on the presence of function pointers. In *Proc. of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256, Orlando, FL, Jun. '94.

[EP89] P. Emrath and D. Padua. Automatic detection of nondeterminancy in parallel programs. *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):89–99, Jan. '89.

[FOW87] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Programming Languages*, 9(3):319–349, July '87.

[FW88] 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.

[GHS92] 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.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability.* Freeman and Company, New York, '79.

[GN93] W.G. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology*, 2(3):228–269, July '93.

[GPS90] R. Gupta, L.L. Pollock, and M.L. Soffa. Parallelizing data flow analysis. In *Workshop on Parallel Compilation*, Kingston, ON, Canada, '90.

[GS92] R. Gupta and M.L. Soffa. Automatic generation of compact test suites. In *12th IFIP World Computer Congress*, Madrid, Spain, Sept. '92.

[GW76] S. Graham and M. Wegman. A fast and usually linear algorithm for global flow analysis. *Journal of the ACM*, 23(1):172–202, Jan. '76.

[GZZ89] T. Gross, A. Zobel, and M. Zolg. Parallel compilation for a parallel machine. In *SIGPLAN '89 Conf. on Programming Language Design and Implementation*, pages 91–100, Jun. '89.

[HDT87] S. Horwitz, A. Demers, and T. Teitelbaum. An efficient general iterative algorithm for data-flow analysis. *Acta Informatica*, 24(6):679–694, Nov. '87.

[HL92] J. Hughes and J. Launchbury. Reversing abstract interpretations. In *4th European Symp. on Programming*, pages 269–286, Rennes, France, Feb. '92. Springer Verlag, LNCS 582.

[Hop71] J.E. Hopcroft. An n log n algorithm for minimizing states in finite automata. In *Theory of Machines and Computations.* Academic Press, New York, '71.

[HPR89] S. Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July '89.

[HS89a] M.J. Harrold and M.L. Soffa. Interprocedural data flow testing. In *ACM Symp. on Software Testing, Analysis, and Verification*, pages 158–167, Key West, FL, Dec. '89.

[HS89b] M.J. Harrold and M.L. Soffa. Interprocedural data flow testing. In *3rd Testing, Analysis and Verification Symp.*, pages 158–167, Dec. '89.

[HS90] M.J. Harrold and M.L. Soffa. Computation of interprocedural definition and uses dependencies. In *Int. Conf. on Computer Languages*, pages 297–306, Los Alamitos, CA, '90. CS Press.

[HU73] M.S. Hecht and J.D. Ulman. Analysis of a simple algorithm for global flow problems. In *First ACM Symposium on Principles of Programming Languages*, pages 207–217, Boston, MA, Oct. '73.

[JM73] N.D. Jones and A. Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *13th Symp. on Principles of Programming Languages*, pages 194–206, Florida, 1973.

[JP93] R. Johnson and K. Pingali. Dependence-based program analysis. In *SIGPLAN '93 Conf. on Programming Language Design and Implementation*, pages 78–89, Albuquerque, NM, Jun. '93.

[Ken75] K. Kennedy. Node listing applied to data flow analysis. In *2nd ACM Symp. on Principles of Programming Languages*, pages 10–21, Jan. '75.

[KGS94] R. Kramer, R. Gupta, and M.L. Soffa. The combining dag: a technique for parallel data flow analysis. *IEEE Transactions on Parallel and Distributed Systems*, 5(8), Aug. '94.

[Kil73] G. Kildall. A unified approach to global program optimization. In *1st ACM Symp. on Principles of Programming Languages*, pages 194–206, Boston, Massachusetts, Jan. '73.

[Knu71] D.E. Knuth. An empirical study of fortran programs. *Software Practice and Experience*, 1(2):105–13, Apr. '71.

[KRS92] J. Knoop, O. Ruething, and B. Steffen. Lazy code motion. In *SIGPLAN '92 Conf. on Programming Language Design and Implementation*, pages 224–234, Jun. '92.

[KS92] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *4th Int. Conf. on Compiler Construction*, pages 125–140, Paderborn, Germany, Oct. '92. Springer Verlag, LNCS 641.

[KU76] J.B. Kam and J.D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, Jan. '76.

[KU77] J.B. Kam and J.D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, Jul. '77.

[Lan92] W.A. Landi. *Interprocedural aliasing in the presence of pointers.* PhD thesis, Rutgers University, New Brunswick, NJ, '92.

[LH88] J.R. Larus and P.N. Hilfinger. Detecting conflicts between structure accesses. In *Proc. of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 21–34, '88.

[LMR91] Y.F. Lee, T.J. Marlowe, and B.G. Ryder. Experiences with a parallel algorithm for data flow analysis. *Journal Supercomputing*, 5:163–188, Oct. '91.

[LR92] W. Landi and B. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proc. of the SIGPLAN '92 Conf. on Programming Language Design and Implementation*, pages 56–67, San Francisco, CA, Jun. '92.

[MR79] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):97–103, Feb. '79.

[MR90] T.J. Marlowe and B.G. Ryder. Properties of data flow frameworks, a unified model. *Acta Informatica*, 28(2):121–163, Dec. '90.

[Mye76] G.J. Myers. *Software reliability: principles and practices.* Wiley-Interscience, New York, '76.

[Mye81] E.W. Myers. A precise inter-procedural data flow algorithm. In *8th ACM Symp. on Principles of Programming Languages*, pages 219–230, Williamsburg, Virginia, Jan. '81.

[NO80] G. Nelson and D.C. Oppen. Fast decision procedures based on congruence closures. *Journal of the ACM*, 27(2), Apr. '80.

[Nta84] S.C. Ntafos. An evaluation of required element testing strategies. In *7th Int. Conf. on Software Engineering*, pages 250–256, Mar. '84.

[OO84] K Ottenstein and L Ottenstein. The program dependence graph in a software development environment. *ACM SIGSOFT/SIGPLAN Symp. on practical SDEs, SIGPLAN Notices*, 19(5):177–184, May '84.

[OW88] 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.

[PS89] 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.

[PW86] D. Padua and M.J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 22(12):1184–1201, Dec. '86.

[RC87] B.G. Ryder and M. Carroll. An incremental algorithm for software. *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices*, 21(1):171–179, Jan. '87.

[Rep94] 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.

[RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *22nd ACM Symp. on Principles on Programming Languages*, pages 49–61, San Francisco, CA, Jan. '95.

[RL77] J. Reif and J. Lewis. Symbolic evaluation and the global value graph. In *4th ACM Symp. on Principles of Programming Languages*, pages 104–118, Jan. '77.

[RM93] G. Rothermel and M.J.Harrold. A safe, efficient algorithm for regression test selection. In *Conf. on Software Maintenance*, pages 358–367, Sept, '93.

[RM94] G. Rothermel and M.J.Harrold. Selecting tests and identifying test coverage requirements for modified software. In *1994 Int. Symp. on Software Testing and Analysis*, pages 169–184, Aug. '94.

[Ros81] B. Rosen. Linear cost is sometimes quadratic. In *8th ACM Symp. on Principles of Programming Languages*, pages 117–124, Jun. '81.

[RP86] B.G. Ryder and M.C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–316, '86.

[RP88] B.G. Ryder and M.C. Paull. Incremental data flow analysis algorithms. *ACM Trans. Programming Languages and Systems*, 10(1):1–50, '88.

[RR91] G. Ramalingam and T. Reps. On the computational complexity of incremental algorithms. Technical Report TR-1033, Computer Science Department, University of Wisconsin, Madison, WI, Aug. '91.

[RSH94] T. Reps, M. Sagiv, and S. Horwitz. Interprocedural dataflow analysis via graph reachability. Technical Report 94-14, Datalogisk Institut, University of Copenhagen, Copenhagen, Denmark, '94.

[RT82] J. Reif and R.E. Tarjan. Symbolic program analysis in almost linear time. *SIAM Journal of Computing*, 11(1):81–93, Feb. '82.

[RTD83] T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *ACM Trans. on Programming Language and Systems*, 5(3):449–477, Jul. '83.

[RW85] S. Rapps and E. Weyuker. Selecting software test data using data flow information. *IEEE Trans. on Software Engineering*, 11(4):367–375, Apr. '85.

[RWZ88] B. Rosen, M. Wegman, and F.K. Zadeck. Global value numbers and redundant computations. In *15th ACM Symp. on Principles of Programming Languages*, pages 12–27, San Diego, CA, Jan. '88.

[Ryd83] B.G. Ryder. Incremental data flow analysis. In *9th ACM Symp. on Principles of Programming Languages*, pages 167–176, Jan. '83.

[Set76] R. Sethi. *Algorithms for minimal-length schedules.* Wiley Publishing Company, New York, NY, '76.

[SH86] V. Sarkar and J. Hennessy. Compile-time partitioning and scheduling of parallel programs. In *Symp. on Compiler Construction*, pages 17–26, '86.

[SMHY93] A.D. Stoyenko, T.J. Marlowe, W.A. Halang, and M. Younis. Enabling efficient schedulability analysis through conditional linking and program transformations. *Control Engineering Practice*, 1(1):85–105, '93.

[SP81] 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.

[SRH95a] 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.

[SRH95b] 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.

[SS88] D. Shasah and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, Apr. '88.

[SY93] R.E. Strom and D.M. Yellin. Extending typestate checking using conditional liveness analysis. *IEEE Trans. on Software Engineering*, 19(5):478–485, May '93.

[Tar81a] R.E. Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, Jul. '81.

[Tar81b] R.E. Tarjan. A unified approach to path problems. *Journal of the ACM*, 28(3):576–593, Jul. '81.

[TTL89] 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.

[Wei84] M. Weiser. Program slicing. *IEEE Trans. on Software Engineering*, SE-10(4):352–357, Jul. '84.

[WL95] R.P. Wilson and M.S. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proc. of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–12, Jun. '95.

[Wol92] M. Wolfe. Beyond induction variables. In *SIGPLAN '92 Conf. on Programming Design and Implementation*, pages 162–174, San Francisco, CA, June '92.

[WZ85] M. Wegman and F.K. Zadeck. Constant propagation with conditional branches. In *12th ACM Symp. on Principles of Programming Languages*, pages 291–299, New Orleans, Jan. '85.

[Zad84] F.K. Zadeck. Incremental data flow analysis in a structured program editor. In *SIGPLAN Symp. on Compiler Construction*, pages 132–143, Jun. '84.