Refining Data Flow Information using Infeasible Paths*

Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa

Dept. of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260, USA

Abstract. Experimental evidence indicates that large programs exhibit significant amount of branch correlation amenable to compile-time detection. Branch correlation gives rise to infeasible paths, which in turn make data flow information overly conservative. For example, def-use pairs that always span infeasible paths cannot be tested by any program input, preventing 100% def-use testing coverage. We present an algorithm for identifying infeasible program paths and a data flow analysis technique that improves the precision of traditional def-use pair analysis by incorporating the information about infeasible paths into the analysis. Infeasible paths are computed using branch correlation analysis, which can be performed either intra- or inter-procedurally. The efficiency of our technique is achieved through demand-driven formulation of both the infeasible paths detection and the def-use pair analysis. Our experiments indicate that even when a simple form of intraprocedural branch correlation is considered, more than 2% of def-use pairs in the SPEC95 benchmark programs can be found infeasible.

1 Introduction

Static analysis is an integral component of many software engineering tools. Because static analysis is performed before execution, it is necessarily conservative in its assumptions. One commonly made assumption is that every program path is executable. However, some of the paths may be infeasible in that there is no input for which the paths will be taken. Thus, the static analyzers produce imprecise information.

Imprecision in the analysis information results in undesirable consequences in software engineering applications, particularly in testing and debugging. In path testing, paths may be selected for testing which are, in fact, infeasible. In data flow testing, imprecision may lead to the selection of definition-use (*def-use*) pairs which are impossible to test because they lie on infeasible paths. Considerable effort may be wasted in trying to generate input data, either manually or automatically, that traverses the infeasible paths [9].

Knowledge about infeasible paths can be used to improve the precision of static analyzers because these paths can be excluded from consideration. Although it is impossible to solve the general problem of identifying all infeasible

* Supported in part by Hewlett Packard, the National Science Foundation PYI Award CCR-9157371, and Grant CCR-9402226 to the Univ. of Pittsburgh.

paths, some can be determined by detecting static branch correlation. A conditional branch has *static correlation* along a path if its outcome can be determined along the path from prior statements or branch outcomes at compile time. For example, along a given path, the direction of a branch may be determined from a constant assignment to the variable that is tested in the conditional, or from the outcome of another branch. Experiments show that from 9 to 40 % of conditionals in large programs exhibit correlation that is detectable at compile time [2]. This implies that a significant number of infeasible program paths can be detected prior to program execution.

Although the infeasible path information can be used to sharpen many tools that are based on data flow analysis, it is particularly useful for software engineering applications, including the following:

- The infeasible path information can be directly used by *path testing*. In path testing, the algorithm for selecting paths to be tested can avoid paths found infeasible due to branch correlation and thus reduce the effort to generate test cases. Typically, such algorithms do not consider infeasible paths [7, 16].
- In *def-use testing*, def-use pairs that occur only along infeasible paths can be eliminated from the set of requirements to be covered by test cases. Since 100% test coverage can rarely be achieved on real programs due to presence of infeasible paths, reducing the number of infeasible def-use pairs increases the confidence in regression testing [11] and integration testing [4].
- By avoiding the consideration of infeasible paths during *static slicing* [10, 14, 17], fewer statements are added to the program slice, thus more precisely identifying the potentially erroneous statements.

In this paper we present a static def-use pair analysis technique that avoids identification of infeasible def-use pairs through detection of branch correlation. The technique consists of two algorithms: (1) the detection of branch correlation and identification of infeasible program subpaths, and (2) the def-use pair analysis that excludes def-use pairs spanning the identified infeasible subpaths. (In the remainder of the paper, the terms *infeasible path* and *infeasible def-use pair* refer to paths and pairs, respectively, that are found infeasible by our technique.) Both algorithms are demand-driven, which guarantees good analyzer performance because only nodes that may influence branch correlation or def-use pair computation are visited. Since significantly more correlation can be detected interprocedurally, we have developed both intra- and inter-procedural versions of our analyses.

The algorithm for detection of interprocedural branch correlation was originally developed to support a compiler optimization for the elimination of redundant conditional branches [2]. We extend the correlation detection algorithm in this paper to *identify* shortest infeasible paths and to *label* the control flow graph with these paths. Techniques for static branch correlation detection have also been developed by other researchers [8, 15]. While these techniques can detect correlated branches, they do not identify the shape of infeasible paths, a requirement for eliminating infeasible def-use pairs. Furthermore, only correlation between pairs of branches is detected, which is not sufficient for identifying some infeasible paths that cross multiple conditionals. Finally, only correlation along paths that do not cross procedure boundaries is considered in these techniques.

Improving the precision of data flow analysis by reducing the impact of infeasible paths has also been considered by Holley and Rosen [13]. In their framework, a data flow problem is solved by considering paths feasible under a given set of assertions on variable values. Since this approach tracks the entire program state that *might* determine the outcome of a conditional branch, it necessarily collects assertions not contributing to the correlation. The size of the program state to be maintained by the analysis makes this technique impractical for detection of a meaningful class of static correlation. Our demand-driven approach examines only values that are relevant to the computation of the branch predicate. Thus, in practice, our approach is more efficient and, in particular, more suitable for use in a software engineering environment because, during maintenance, the program change would drive the process of re-analysis.

The remainder of the paper is organized as follows. In Section 2 we present the demand-driven analysis for identifying infeasible paths. In Section 3, the infeasible path information is used to develop the improved def-use analysis. The interprocedural def-use pair analysis algorithm is in Section 4. The experiments are summarized in Section 5 and the conclusion is in Section 6.

2 Infeasible Paths

We first present the technique to identify infeasible paths and then show in the next section how this information is used to compute more precise def-use pairs. Infeasible paths analysis consists of two steps, each covered in the following two subsections: a) detecting branch correlation and identifying infeasible paths, and b) determining and labeling shortest infeasible paths.

2.1 Detection of branch correlation

A conditional branch exhibits *correlation* if along some paths its outcome is implied by the outcome of other conditionals or by prior program statements, such as assignments to the variable tested in the conditional. The correlation is *static* if this dependence exists along the *correlated path* for any program input and can be determined at compile time. In the presence of correlation, some program paths are not executable because, along the correlated paths leading to the conditional, control will always take either the true or the false direction.

Definition 1. Let b be a conditional branch with predicate expression p and c be a path from the start of the program to the true (false) out-edge of the node b. Path c is **infeasible** if the predicate p always evaluates to false (true) when the control reaches node b along the path c.

The duality between infeasible and correlated paths allows us to determine the shape and extent of infeasible paths using a branch correlation analysis. We have recently developed such analysis to support the elimination of interprocedurally redundant conditional branches [2]. Here we summarize its intraprocedural version; please refer to [2] for details of the interprocedural correlation analysis.

The goal of the analysis is to find paths along which the outcome of branches can be determined from *assertions* generated by other program statements. Our algorithm determines the infeasible paths by detecting correlation of each conditional separately. Given a conditional b with predicate expression p (e.g., `x>0`), we find the infeasible paths in a demand-driven fashion by raising at b a query containing the expression p and propagating it backwards in the flow graph until the query is resolved along all paths. The query is resolved at nodes where the value of the expression p carried within the query can be determined from the assertions generated in the node. We identified four sources of useful assertions:

1. a *constant assignment* to a variable may imply a particular direction of the conditional.
2. a prior *conditional branch* may *subsume* the branch predicate p. The prior conditional generates on its out-edges assertions on the variable tested in its predicate, and these assertions may suffice to evaluate p.
3. *type conversion*. For example, unsigned integers converted to signed integers will always have non-negative values.
4. *pointer dereferencing*. The value of a pointer after it is used to access a memory cell must be non-zero, otherwise an exception would have been raised.

The query can be resolved to one of three answers. If the assertions generated at a node are sufficient to evaluate the expression p, the query is answered to either TRUE or FALSE. The query is resolved to UNDEF at a node that makes the outcome of p unknown at compile time, such as at the procedure entry node or a relevant `read` statement. If the query cannot be answered at a node, it is raised at its predecessors. Propagating the query to predecessors may involve symbolic *query substitution* due to an assignment to a variable from the expression p.

The paths along which the propagation of the query resulted in a TRUE (FALSE) answer correspond to infeasible paths. These paths start at the nodes where the query was resolved and end at the false (true) out-edge of the analyzed branch, respectively. We identify the shapes of these infeasible paths by propagating forward the answers obtained at the resolution nodes. After forward propagation of answers, each query raised at a node can obtain multiple answers, each corresponding to a different set of paths.

In summary, the algorithm has two steps, shown in Fig. 1. The backward query propagation algorithm of the first step is based on the demand-driven analysis framework in [3]. The algorithm finds correlation for a single conditional node b. Line 1 removes from $Q[n]$ all queries raised during the previous invocation of the algorithm. The initial query q_b holds the branch predicate expression and is raised at all predecessors of b at line 2. Starting from the predecessor of the analyzed conditional node b, lines 3–10 process all nodes at which a query was raised. The array $A[n, q]$ stores the set of answers for a query q at a node n.

```
Step 1: Detect correlation of conditional branch b with predicate v < c
1   initialize Q[n] to {} at each node n; set worklist to {}
    raise the initial query q_b = (v < c) at each predecessor of b
2   for each m ∈ Pred(b) do raise_query(m, substitute(b, q_b))
3   while worklist not empty do
4       remove pair (node n, query q) from worklist
        assume unknown outcome of b at procedure entry
5       if n is entry node of a procedure then A[n, q] := UNDEF
        else – – attempt to answer q using assertions generated at n
6           answer := resolve(n, q)
7           if answer ∈ {TRUE, FALSE, UNDEF} then A[n, q] := {answer}
8           else for each m ∈ Pred(n) do raise_query(m, substitute(n, q))
9       end if
10  end while

    Procedure raise_query(node n, query q)
        raise q at n unless previously raised there (terminate analysis of loops)
11      if q ∉ Q[n] then add q to Q[n]; add pair (n, q) to worklist
    end

Step 2: Identify correlated paths of conditional branch b
    start from immed. succ. of nodes where any query was resolved in lines 5 or 7
12  worklist := {Succ(n) : n ∈ R}, where R := {n : a query was resolved at node n}
    raise the initial query at the analyzed branch, to collect final answers
13  add the initial query q_b = (v < c) to Q[b]
14  while worklist not empty do
15      remove a node n from worklist
        determine answers for each query that was not propagated backward
16      for each query q from Q[n] s.t. q was not resolved at node n do
            collect all answers to query q from all predecessors of n
17          for each m ∈ Pred(n) do add A[m, substitute(n, q)] to A[n, q]
18          if value of A[n, q] changed in line 17 then add Succ(n) to worklist
19      end for
20  end while
```

Fig. 1. Intraprocedural static correlation analysis.

A query raised at procedure entry node resolves to UNDEF because nothing can be concluded about the outcome of the analyzed branch (lines 5). At any other node n, the function **resolve** determines if assertions on n exist that evaluate the predicate expression. If no answer can be concluded, the query is propagated to predecessors, possibly modified by the call to **substitute** due to symbolic substitution in the predicate expression. The algorithm terminates when all queries are resolved. In Step 2, query answers are propagated forward in lines 12–20.

2.2 Marking shortest infeasible paths

To make the def-use pairs analysis aware of the detected infeasible paths, we mark the paths on the flow graph. The marking can be compared to placing finite-length threads on the graph, with the meaning that any program path which fully includes any thread is infeasible. While we are not enumerating all infeasible paths, all of them can be identified from the infeasible-path markings.

Labeling of the flow graph with an infeasible path is achieved through placing of *start*, *end*, and *present* marks on flow graph edges. The three marks identify the edges where a path begins, the edge where it ends, and all the edges the path follows, respectively. The marks are implemented as unique integers; they are stored on each control flow edge in three corresponding sets that identify the start, end, and present marks of the infeasible paths that cross the edge.

The placement of marks is derived from the answers to queries collected during identification of infeasible paths (Step 2 in Fig. 1). Along each infeasible path in the graph there will be a sequence of queries with answers TRUE or FALSE such that the queries were raised in response to one another starting from the conditional. A pair (*query*, *answer*) can thus serve to uniquely identify an infeasible path from its start to the end. Each query-answer pair is assigned a unique small integer to facilitate efficient bit-vector operations. This integer id identifies the path using the start, end, and present sets maintained on graph edges. Due to predicate expression symbolic substitution carried out by the correlation analysis, while tracing the path it may be necessary to switch from $(q, answer)$ to $(q', answer)$ at a node where the query q was changed into q'.

Because the infeasible paths identified by the correlation detection extend from the correlated branch all the way to the source of the correlation, they are not the *shortest infeasible paths*. The start of each infeasible path can be delayed in the forward direction, as specified by the definition below.

Definition 2. An infeasible path $p = e_{i_1}.e_{i_2} \ldots e_{i_n}$ is a **shortest infeasible** path if the subpath $e_{i_2} \ldots e_{i_n}$ is not infeasible.

Determining shortest infeasible paths maximizes the number of def-use pairs that can be excluded during def-use analysis because more def-use pairs can span shorter infeasible paths. The central idea behind placing the start mark is that, if a query at a node n has a single answer, then the start of the corresponding path can be delayed past n because only infeasible paths enter n. However, if a query has multiple answers at n, the start must precede n because some feasible paths may pass through n. Thus the start mark is placed at the edge where the query has a single answer but the destination node of the edge has multiple answers to the query. The marks for the shortest infeasible paths leading to branch b are placed in Step 3 (Fig. 2). The end marks are always placed in the out-edges of the conditional branch (lines 21–23). The present marks are placed at all nodes where a query was raised (lines 28–29). Throughout the remainder of the paper, the term infeasible path refers to the shortest infeasible path.

The algorithm we have presented can be used to exhaustively compute infeasible paths or to incrementally update infeasible path information. Following

Step 3: Label CFG with shortest infeasible paths that end at branch b

let e_t, e_f be the true and false out-edges of b
if TRUE $\in A[b, q_b]$ **then** $end[e_t] := (q_b, t)$
if FALSE $\in A[b, q_b]$ **then** $end[e_f] := (q_b, f)$
for each node n visited during correlation analysis (**Step 1**) **do**
    for each query $q \in Q[n]$ **do**
        $q' :=$ **substitute**(n, q)
        for each edge $e = (m, n)$ **do**
            if TRUE $\in A[m, q']$ **then** add (q', t) to $present[e]$
            if FALSE $\in A[m, q']$ **then** add (q', f) to $present[e]$
            shortest paths start where two different answers meet
            if $(A[m, q'] = \{\text{TRUE}\}$ or $A[m, q'] = \{\text{FALSE}\})$
            and $|A[n, q']| > 1$ **then** add $(q', A[m, q'])$ to $start[e]$
        end for
    end for
end for

Fig. 2. Marking infeasible paths on the control flow graph.

a program change, the infeasible paths for only those branch exits that may be affected by program changes need to be recomputed. Consider a statement n at which the set of path marks *present* is not empty. Any modification to n or insertion of new statements in n will require the infeasible paths included in the *present* set to be recomputed.

The example in Fig. 3 shows the infeasible and shortest infeasible paths that end at the true and false exits of the conditional node 7, and at the false exit of the node 10. We use regular expression notation to denote the paths. A subpath of the form $[p]$ indicates that the subpath p is optionally included in the path, while a subpath of the form $(p)*$ indicates that p may be repeated zero or more times. The variable names below each node number denote the predicate expression from the query raised at that node. For example, w for the node 7 means that the query has the form $(w = 5)$. Consider the computation of the infeasible paths for node 7. Using the algorithm in Fig. 1 (Step 1), node 3 is identified as a constant definition that makes the false exit of node 7 impossible. The infeasible paths that start at node 3 are 3 4 5 6 (14 10 11 13 6)* 7. After path marking (Fig. 2, Step 3), the paths 4 5 6 (14 10 11 13 6)*7 are identified as the shortest infeasible paths. Thus no path going through the out-edge of node 4 can take the false exit of node 7. Also during the analysis of node 7, the copy assignment in node 9 changes the query from $(v = 5)$ to $(w = 5)$, with the result that node 1 is identified as a node that makes the true exit of node 7 impossible. The resulting infeasible paths from node 1 are 1 2 [3 4] 5 6 7 8 9 (10 11 13 6 14)* 10 11 13 6 7. The shortest infeasible paths exclude 1 2 [3 4] because it is guaranteed that at node 5, the value of w is still the constant 1. The infeasible paths for the false exit of node 10 are also shown in the figure.

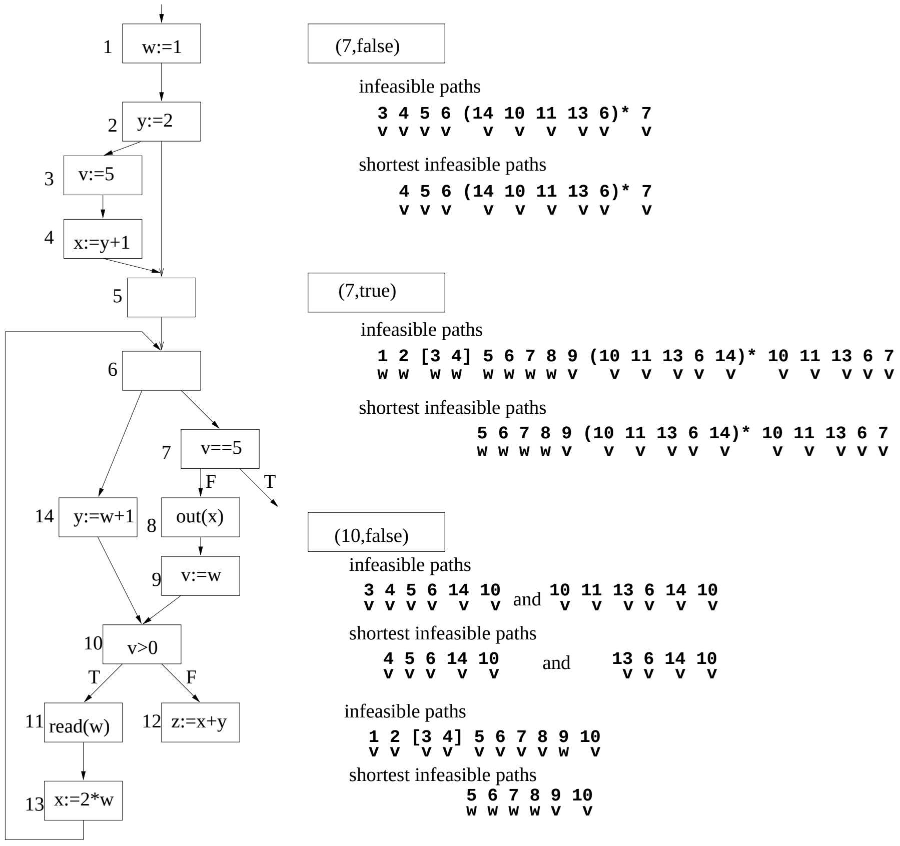

Fig. 3. An example of intraprocedural infeasible paths.

3 Def-Use Analysis

The previous section described how infeasible paths are identified and marked on the control flow graph. In this section we present a data flow analysis method that provides refined data flow information by tracing the infeasible paths and excluding def-use pairs that are formed exclusively along infeasible paths.

Def-use pairs are determined by solving the data flow problem of reaching definitions. Given the set of definitions that reach a node, we can determine def-use pairs for all of the uses in the node. Traditional data flow analysis conservatively assumes that all program paths are executable and computes the data flow information as a meet operation along all paths. To refine the result of the analysis, we strengthen the definition of a def-use pair: if all paths between a definition and a use contain an infeasible path, then this def-use pair is infeasible and is excluded from the set of def-use pairs found by the analysis.

Definition 3. Given a definition d and a use u of a variable v, (d, u) is a **def-use pair** iff a path p_{du} from d to u exists such that v is not redefined along p_{du} and p_{du} does not contain any infeasible path.

To exclude infeasible reaching definitions, we associate with the data flow information of each reaching definition (traditionally, a single bit in a data flow vector) information about infeasible paths that have been encountered in the propagation of the reaching definition. We call these paths *infeasible paths in progress.* When the propagation of a reaching definition d encounters the start mark of an infeasible path, we remember the path in the propagated data flow information in order to trace the encountered path. When d reaches the end mark of the path without previously leaving the path, we can remove d from further consideration, as d has traversed an infeasible path. However, when d leaves the infeasible path before its end mark is reached, the tracing information about the path is removed from the data flow information because the path is no longer in progress, and d is propagated further.

Reaching definitions can be computed either using an exhaustive data flow algorithm (such as iterative) [1] or using a demand-driven algorithm [3]. In an exhaustive algorithm, the reaching definitions are computed for all variables at all nodes. In the demand-driven algorithm, the reaching definitions for each variable used in a node are computed. Recent studies have demonstrated that demand-driven algorithms take less time and space to compute reaching definitions than exhaustive ones, even when the computation of all def-use pairs is required [4, 12]. We present in this paper the demand-driven version of def-use analysis.

The demand-driven algorithm is presented in Fig. 4. It computes def-use pairs for a variable v used at a node u. Similar to the demand-driven algorithm in Fig. 1, the algorithm raises a query at the use node u and propagates it backward through the graph until a reaching definition of v is encountered or until the query is discarded due to having followed an infeasible path. Removing the query ensures that when a definition of v is encountered by a query, only a feasible def-use pair is recorded. To determine when a query can be safely discarded, a set of paths in progress are carried with each query and are updated by the algorithm as the query traverses paths marks.

The initial query is formed and raised at lines 2–3. The query has a single component, *ipp*, the set of infeasible paths in progress. Initially, this set is empty. Note that, since the propagation proceeds in the opposite direction as in the exhaustive analysis, the *end* mark is considered to be the start of the path. Line 6 considers each propagated query. Function **resolve** updates the *ipp* information for the query and determines if a reaching definition has been encountered. Line 8 discards the query if an infeasible path in progress ends at the edge e. Lines 9 to 14 update the tracing information. Lines 15–19 record a new def-use pair and terminate propagation of the query if a definition has been reached. Procedure **raise_query** performs the *meet* data flow function for the *ipp* information. While the problem of reaching definitions is a *may*-problem, the set *ipp* computes a *must*-problem; a path in progress is preserved at a control flow meet point only if it was in progress in each query that reached the meet point. As in every

Procedure Demand_Driven_Def-Use_Analysis (var v, node u)

$Q[n] = nil$ means no query for var v and use u was raised at n,
$Q[n] \neq nil$ stores infeasible paths that are in progress for the query
1 initialize $Q[n]$ to nil at each node n; set $worklist$ to $\{\}$
Initial query carries an empty set of paths in progress:
2 form the initial query $q_{v,u} = (\{\})$
3 **for** each edge $e = (m, u)$ **do raise_query**$(e, q_{v,u})$
4 **while** $worklist$ not empty **do**
5 remove pair (node n, query q) from $worklist$
6 **for** each edge $e = (m, n)$ **do raise_query**(e, q)
7 **end while**
end

Function resolve(edge $e = (m, n)$, query (ipp))

Terminate query propagation if query followed an infeasible path:
8 **if** $ipp \cap start[e] \neq \{\}$ **then return** nil
Remove paths in progress that are no longer followed:
9 $ipp := ipp \cap present[e]$
Add paths in progress that are started at edge e:
10 $ipp := ipp \cup end[e]$
Rename paths in progress due to query substitution at node m:
11 **for** each $q \in ipp$ **do**
12 remove q from ipp
13 add **substitute**(m, q) to ipp
14 **end for**
Terminate propagation if m defines v:
15 **if** node m defines v **then**
16 add def-use pair (m, u) to $DEF\text{-}USE$
17 **return** nil
18 **end if**
19 **return** (ipp)
end

Procedure raise_query(edge $e = (m, n)$, query $q = (ipp)$)

20 $q' :=$ **resolve**(e, q)
nil is returned when q not to be propagated across e
21 **if** $q' \neq nil$ **then**
22 **if** $Q[m] = nil$ **then** $Q[m] := ipp$
Preserve only paths that are in progress along all merging paths:
23 **else** $Q[m] := Q[m] \cap ipp$
If merge in line 23 removed a path in progress, re-raise query:
24 **if** $Q[m]$ changed **then** add pair $(m, (Q[m]))$ to $worklist$
25 **end if**
end

Fig. 4. Intraprocedural demand-driven def-use analysis.

distributive data flow problem, this conservative merge of data flow information provides efficiency of the analysis but may prevent detection of some infeasible def-use pairs when multiple infeasible paths contribute to the infeasibility of the def-use pair. In the algorithm, the query is propagated further (in line 24) when it is raised at the node for the first time (line 22) or when a path previously in progress at node m has been removed at line 23.

For the example in Fig. 3, our analysis detects three less def-use pairs than the traditional def-use analysis. In response to a query raised at the use of x in node 8, our def-use analysis excludes the def-use pair $(4, 8)$ because the propagated query is removed at the edge $(4, 5)$. This edge is the start of the infeasible path that ends at the false exit of node 7. The def-use pair $(4, 12)$ on variable x is excluded due to the first infeasible path leading to the false exit of node 10. Finally, the def-use pair $(2, 12)$ on y is excluded due to the infeasible path that leads to the true exit of node 7.

The demand-driven algorithm which finds the definitions reaching a given use is also useful for determining more precise *program slices* [17]. By repeated application of this algorithm, the data slice corresponding to a given statement node can be easily computed. Due to the refined def-use analysis, this algorithm computes smaller slices than traditional slicing algorithms.

Time complexity. The cost of our technique can be divided between the infeasible paths analysis and the def-use analysis. The cost of the former is dominated by Step 1 in Fig. 1. In our experiments, the pattern of analyzed conditionals was restricted to branches that compare a variable with a constant (e.g., `x<10`) and the only statements on which we performed symbolic substitutions of the propagated predicate expression were copy assignments (e.g., `x:=y`). Under these restrictions, the cost to find infeasible paths leading to a single branch is $O(NV)$, where N is the number of nodes in the program CFG, and V is the number of program variables. All infeasible paths can be found in $O(N^2V)$ steps.

The cost of finding all def-use pairs for a single use is bounded by $O(NI)$, where I is the maximum number of infeasible paths that cross a node. The value of I bounds the number of times a query can be re-raised on a single node (line 24 in Fig. 4) because the value of $Q[n]$ can be monotonically decreased at most I times (line 23). All def-use pairs can thus be found in $O(N^2I)$ steps. While in the worst case $I = O(N^2V)$, we observed in our experiments, that I was never higher than 75 and averaged below 2.01 (see Table 2, columns *present*).

4 Interprocedural Analysis

Both infeasible path analysis and def-use analysis can be extended to operate across procedure boundaries. When implementing a practical def-use analyzer, it is however not required to develop both techniques interprocedurally. Obviously, interprocedural def-use analysis can benefit from purely intraprocedural infeasible paths. Combining interprocedural infeasible paths analysis with intraprocedural def-use pair analysis appears even more attractive: by examining the calling context of each procedure, interprocedural correlation detection may discover strictly intraprocedural infeasible paths, which will benefit intraprocedural def-use analysis, typically employed in def-use testing. In the remainder of this section we describe the interprocedural versions of both infeasible path analysis and the def-use analysis.

Detecting infeasible paths. The interprocedural version of infeasible path analysis from Fig. 1 is based on the algorithm in [2]. The extension is based on propagating queries between callers and callees and maintaining appropriate procedure summary nodes. The infeasible path marking algorithm in Fig. 2 does not require changes.

Interprocedural Reaching Definitions. The extension of the def-use analysis requires the elimination of reaching definitions that occur exclusively along infeasible interprocedural paths. This may affect not only interprocedural but also intraprocedural def-use pairs. For example, at a call site node, a reaching definition may be killed along one path through the called procedure but may reach the procedure exit along the other possible path through the callee. If the latter path is found infeasible by interprocedural analysis, this interprocedural exclusion of the reaching definition may eliminate an intraprocedural def-use pair in the calling procedure.

The extension involves the introduction of procedure summary nodes, which are computed independently of the calling context. The summary node of a procedure maps a variable and a specific set of infeasible paths in progress (*ipp*) to the set of reaching definitions generated within the procedure. Due to the many possible subsets of the *ipp* set, the amount of information to be stored in the summary node is significant. However, since only a fraction of a summary node will likely be referenced, we present in this section the demand-driven def-use analysis based on [3], which achieves efficiency by computing only the needed subset of each summary node.

In the context of query-based analysis, the purpose of summary node is to cache for each query raised on the exit of a procedure the answers to the query that were found within the procedure and its callees. In our reaching definitions analysis, each query is associated with the analyzed variable v and the set of infeasible paths *ipp* encountered during propagation of the query. Therefore, the summary node maps a procedure exit node x, a variable v, and the set of paths *ipp* to: (1) *RD*, the set of reaching definitions from the procedure that are feasible *given* the paths in *ipp*, (2) *transp*, the boolean variable indicating whether the query propagated to the procedure entry (*transp* is true if there is a feasible path through the procedure with no definition of v), and (3) ipp', the set of paths that are in progress for the query at the procedure entry, if $transp = true$. The summary node entry $SN[x, v, ipp]$ is thus a triple $(RD, transp, ipp)$.

The algorithm in Fig. 5 extends its intraprocedural counterpart with the functionality to compute on demand the summary nodes. Whenever a query is about to enter a procedure exit node, the summary node is looked up for a previously cached result. If the lookup fails, the summary node entry is computed by raising at the procedure exit an identical query, which is however marked as a special, *summary node query*. This query never leaves the scope of its procedure;

```
    Procedure Demand_Driven_Def-Use_Analysis (var v, node u)
1       initialize Q[n] to nil at each node n; set worklist to {}
2       form the initial query q_{v,u} = ({}, nil)
3       for each edge e = (m, u) do raise_query(e, q_{v,u})
4       while worklist not empty do
5           remove pair (node n, query q = (ipp, sn)) from worklist
            case n is call site node:
                let x be callee's exit node and s be summary node entry SN[x, v, q.ipp]
6               if s = nil then
                    summary node (SN) lookup failed, create a new SN entry
7                   s := ({}, false, {})
                    raise SN query to compute the new entry
8                   raise_query((x, n), (ipp, s))
9               end if
                use information cached in the SN entry
10              add s.RD to DEF-USE
11              if s.transp then for each e = (m, n) raise_query(e, (s.ipp, q.sn))
            case n is procedure entry node:
                record information into SN entry
12              if q is summary node query (ipp, s) then s.transp:=true; s.ipp:=ipp
13              for each call site m of the procedure do
14                  if q is standard query or q ∈ Q[m] then raise_query((m, n), q)
15              end for
            otherwise :
16              for each edge e = (m, n) do raise_query(e, q)
            end case
17      end while
    end
```

Fig. 5. Interprocedural demand-driven def-use analysis.

its only task is to compute the cached summary node entry. To support both standard and summary node queries, the algorithm represents a query with a pair (ipp, sn), where *sn* is a pointer to the summary node entry that is being computed by the query. The initial query is created at line 2, where the *nil* value of the *sn* pointer signifies that this is a standard query (i.e., one that does not compute a summary node). If the summary node lookup fails in line 6, a new summary node entry $SN[x, v, ipp]$ is created in line 7 and the computation of the entry is initiated by raising the summary node query in line 8. The *transp* and *ipp* fields of the entry are recorded in line 12. Line 14 raises the query in all callers, restricting the scope of summary node queries to the procedure. The procedures **resolve** and **raise_query** from Fig. 4 are unchanged, except that line 16 may now add the reaching definitions to the *RD* set of the summary node entry. An example of interprocedural infeasible paths is shown in Fig. 6.

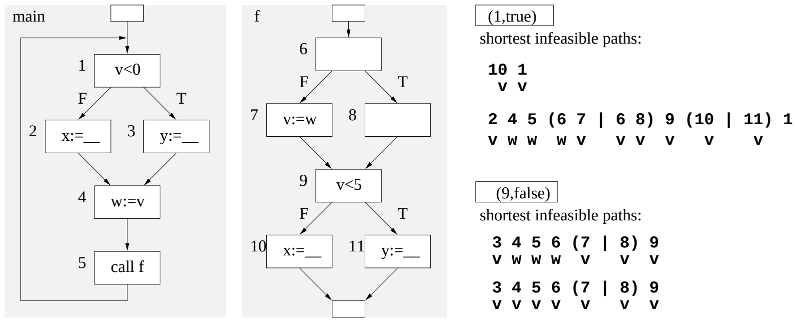

Fig. 6. An example of interprocedural infeasible paths.

5 Experimental Results

To measure the cost and benefit of our analysis technique, we implemented the algorithms in our interprocedural compiler which is based on the `lcc` compiler [6]. In this section, we compare the traditional *intra*procedural def-use analysis with *two* configurations of our technique: the *intra*procedural def-use analysis utilizing a) *intra*procedural and b) *inter*procedural infeasible paths analysis.

The experiments were performed on the integer benchmarks from the SPEC95 suite. All benchmarks are real application programs and, as Table 1 shows, are of considerable size. The first three columns list the number of source lines, the number of procedures defined in the program, and the number of external library procedures. Columns 4 and 5 list the total number of nodes in the interprocedural control flow graph of the program and the number of conditional nodes in the graph.

In both the intra- and inter-procedural versions of our infeasible paths analysis, we considered only conditionals whose predicate expressions p was of the form (v relop c), where v is a scalar variable and c is a constant. About 45% of program conditionals were analyzable under this restricted pattern. Given a predicate p of this form, we found those infeasible paths to p along which the outcome of p can be determined from a prior constant assignment or a prior conditional branch, which are the first two types of the static assertions described in Section 2. Another implementation restriction was that the function **substitute** used in Fig. 1 performed query substitution only on copy assignments of the form $v := w$. Our infeasible path detection technique, however, supports the analysis of arbitrary predicate expressions and is limited only by the capabilities of the symbolic evaluation routines in the compiler.

The effectiveness of our branch correlation analysis is described in the last group of columns in Table 1. The column **analyzable** describes what percentage of all conditionals in the graph were analyzed by the analysis, given the restrictions mentioned above. The columns **corr-P** and **corr-I** give the number of conditionals that have some correlation, detected using intra- and inter-procedural infeasible paths analyses, respectively. For each conditional node with correla-

benchmark program	source lines	procedures defined	procedures library	nodes all	nodes cond	correlation [% of cond] analyzable	corr-P	corr-I
099.go	29 246	372	11	36 283	5 304	31.1	11.1	14.6
124.m88ksim	19 915	252	35	20 616	2 431	57.9	13.1	24.0
129.compress	1 934	24	6	942	92	61.8	7.9	27.0
130.li	7 597	357	26	9 600	878	30.2	6.4	19.2
132.ijpeg	31 211	467	30	25 420	2 355	46.1	6.2	9.2
134.perl	26 871	276	66	46 891	5 628	28.6	6.7	9.4
147.vortex	67 202	923	63	96 380	9 646	59.2	8.5	40.5

Table 1. The benchmarks: program size and amount of correlated branches.

benchmark program	*present* max-P	avg-P	max-I	avg-I	analysis steps [k] tradit	inf-P	inf-I	def-use pairs tradit	elim-P	elim-I
099.go	27	0.29	75	0.86	1 465	1 758	1 781	43 737	910	921
124.m88ksim	17	0.06	74	1.23	322	349	410	12 415	400	506
129.compress	5	0.00	19	0.02	6.6	6.9	7.3	651	0	0
130.li	26	0.02	41	0.63	65	70	72	5 061	198	209
132.ijpeg	12	0.04	19	0.10	608	642	647	17 930	110	115
134.perl	30	0.47	60	0.77	3 467	4 769	4 980	123 295	2 460	2 476
147.vortex	28	0.51	46	2.01	3 996	5 252	5 836	93 282	3 135	3 389

Table 2. The costs and results of our def-use analysis.

tion, there is at least one infeasible path. The number of correlated conditionals is given as a percentage of all conditional nodes.

The comparison of the traditional def-use analysis and our def-use analysis is given in the last three columns of Table 2. Both analyses were restricted to intraprocedural def-use pairs and each call site node was assumed to be a definition of each global variable. Column **tradit** gives the number of def-use pairs found by the traditional def-use analysis. The columns **elim-P** and **elim-I** give the number of pairs that were eliminated by our def-use analysis, using the intra- and inter-procedural infeasible paths analyses, respectively. With intraprocedural infeasible paths, we are able to eliminate 2.2% of def-use pairs, on an average. Some additional pairs can be removed when interprocedural infeasible paths analysis is used. While this may appear to be a small amount, knowing these infeasible def-use pairs will significantly strengthen the confidence in the testing level of a program. Assume that a def-use testing of a program achieved 97% testing coverage. After the infeasible def-use pairs are removed from consideration, the testing coverage of the program will be over 99%, without expending additional testing effort. We should also point out that, while using interprocedural infeasible paths enables elimination of only a small additional amount of def-use pairs, these def-use pairs are extremely difficult to confirm as infeasible manually because the calling context of procedures must be carefully examined, as reported in [5].

Table 2 also presents the cost of our analysis. Since the set of infeasible paths *ipp* is best implemented by a bit vector, we are interested in the maximum and average size of the *present* marking sets, across all edges in the control flow graph. These values are reported in the first four columns, separately for the intra- and inter-procedural infeasible paths. The low average values suggest that many edges in the graph contain no infeasible paths markings. Note that when the *ipp* set and all three mark sets are empty at lines 8–14 in Fig. 4, these statements can be bypassed, resulting in query processing time that is equivalent to that of the traditional def-use analysis. Next, we report the number of steps performed by the considered demand-driven def-use analyses, measured in the number of times a query was removed from the worklist at line 5 in Fig. 4. We report the amount of work for the traditional def-use analysis (which was also implemented as a demand-driven analyzer) and for the two versions of our def-use analysis (columns 5 to 7). We can observe that infeasibility-aware analysis does not require significantly more steps to terminate than the traditional def-use analysis.

6 Conclusion and Future Work

We have presented a method for improving the accuracy of def-use pair analysis. Our technique consists of two parts. First, program paths that are infeasible due to branch correlation are detected and marked on the flow graph. Second, the def-use analysis is modified to be aware of the infeasible paths, with the goal of excluding def-use pairs that occur exclusively along such infeasible paths. The infeasible path analysis uses assertions known at compile time from the program text alone, e.g, from constant assignments.

To provide more precise def-use pairs and, consequently, slices during run-time analysis and debugging, our technique can take advantage of available dynamic information and identify infeasible paths that are specific to a given program execution. These *dynamic* infeasible paths can be detected if our analysis is provided with a small amount of run-time information collected inexpensively at user-defined breakpoints. It is sufficient to record at each breakpoint the value of those variables that contribute to evaluation of branch predicates. This subset of variables can be identified for each node by our analysis prior to program execution.

In the future, we will extend the def-use data flow analysis to define a general data flow framework in which other data flow problems can be computed more accurately using the infeasible paths information. We will also perform experiments to determine exactly what pattern of predicate expressions should be considered to detect a large majority of statically detectable infeasible paths.

References

1. A.V. Aho, R. Sethi, J.D. Ullman, *Compilers, Principles, Techniques, and Tools*, Addison-Wesley, 1986.
2. R. Bodik, R. Gupta, and M.L. Soffa, "Interprocedural Conditional Branch Elimination," *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, June 1997.
3. E. Duesterwald, R. Gupta, and M.L. Soffa, "Interprocedural Data Flow Analysis on Demand," *The 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 37-48, San Francisco, California, January 1995.
4. E. Duesterwald, R. Gupta, and M.L. Soffa, "A Demand-Driven Analyzer for Data Flow Testing at the Integration Level," *International Conference on Software Engineering*, Berlin, Germany, March 1996.
5. P.G. Frankl, E.J. Woyeker, "An Applicable Family of Data Flow Testing Criteria," *IEEE Transactions on Software Engineering*, pages 1483-1498, Vol. 14, No. 10, October 1988.
6. C. Fraser and D. Hanson, *A Retargetable C Compiler: Design and Implementation*, Benjamin/Cummings, 1995.
7. H.N. Gabow, S.N. Maheshwari, L.J. Osterweil, "On Two Problems in the Generation of Program Test Paths," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 3, pages 227–231, September 1976.
8. R. Gupta and P. Gopinath, "Correlation Analysis Techniques for Refining Execution Time Estimates of Real-Time Applications," *11th IEEE Workshop on Real-Time Operating Systems and Software*, pages 54-58, Seattle, Washington, May 1994.
9. R. Gupta and M.L. Soffa, "Employing Static Information in the Generation of Test Cases," *Journal of Software Testing, Verification and Reliability*, Vol. 3, No. 1, pages 29-48, December 1993.
10. R. Gupta and M.L. Soffa, "Hybrid Slicing: An Approach for Refining Static Slices Using Dynamic Information," *ACM SIGSOFT Third Symposium on the Foundations of Software Engineering*, pages 29-40, Washington, DC, October 1995.
11. R. Gupta, M.J. Harrold, and M.L. Soffa, "An Approach to Regression Testing using Slicing," *Conference on Software Maintenance*, pages 299-308, Orlando, Florida, November 1992.
12. S. Horwitz, T. Reps, and M. Sagiv, "Demand Interprocedural Data Flow Analysis," *ACM SIGSOFT Third Symposium on the Foundations of Software Engineering*, Washington, DC, October 1995.
13. L.H. Holley and B.K. Rosen, "Qualified Data Flow Problems," *IEEE Transactions on Software Engineering*, Vol. SE-7, NO.1, January 1981
14. J.R. Lyle and M. Weiser, "Automatic Program Bug Location by Program Slicing," *Proc. Second IEEE Symposium on Computers and Applications*, pages 877-883, June 1987.
15. F. Mueller and D.B. Whalley, "Avoiding conditional branches by code replication," *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, *SIGPLAN Notices*, 30(6):56–66, June 1995.
16. H.S. Wang and S.R. Hsu, "A Generalized Optimal Path-Selection Model for Structural Program Testing," *The Journal of Systems and Software*, Vol. 10, pages 55–63, 1989.
17. M. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 4, pages 352–357, July 1984.