Pruning Dynamic Slices With Confidence

Xiangyu Zhang Neelam Gupta Rajiv Gupta
The University of Arizona, Department of Computer Science, Tucson, Arizona 85721
{xyzhang,ngupta,gupta}@cs.arizona.edu

Abstract

Given an incorrect value produced during a failed program run (e.g., a wrong output value or a value that causes the program to crash), the backward dynamic slice of the value very frequently captures the faulty code responsible for producing the incorrect value. Although the dynamic slice often contains only a small percentage of the statements executed during the failed program run, the dynamic slice can still be large and thus considerable effort may be required by the programmer to locate the faulty code.

In this paper we develop a strategy for pruning the dynamic slice to identify a subset of statements in the dynamic slice that are likely responsible for producing the incorrect value. We observe that some of the statements used in computing the incorrect value may also have been involved in computing correct values (e.g., a value produced by a statement in the dynamic slice of the incorrect value may also have been used in computing a correct output value prior to the incorrect value). For each such executed statement in the dynamic slice, using the value profiles of the executed statements, we compute a *confidence value* ranging from 0 to 1 – a higher confidence value corresponds to greater likelihood that the execution of the statement produced a correct value. Given a failed run involving execution of a single error, we demonstrate that the pruning of a dynamic slice by excluding only the statements with the confidence value of 1 is highly effective in reducing the size of the dynamic slice while retaining the faulty code in the slice. Our experiments show that the number of distinct statements in a pruned dynamic slice are 1.79 to 190.57 times less than the full dynamic slice. Confidence values also prioritize the statements in the dynamic slice according to the likelihood of them being faulty. We show that examining the statements in the order of increasing confidence values is an effective strategy for reducing the effort of fault location.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Debugging aids, Testing tools, Tracing; D.3.4 [*Programming Languages*]: Processors—Debuggers

General Terms Algorithms, Measurement, Reliability, Verification

Keywords debugging, dynamic slicing, data and control dependences

PLDI'06 June 11–14, 2006, Ottawa, Ontario, Canada.

1. Introduction

Dynamic slicing was first proposed by Korel and Laski [13] to guide software developers in the debugging process. The dynamic slice of a value computed at a program point in the execution trace includes all those executed statements which were directly or indirectly involved in the computation of the value. A significant amount of research on algorithms for computing dynamic slices has been carried out [1, 23, 24, 25]. Computation of dynamic slices normally consists of two steps: building the dynamic dependence graph for a program execution (where dependences include both data and control dependences); and then traversing the dynamic dependence graph to compute the dynamic slice of a computed value that is observed to be incorrect by the programmer (incorrect value may correspond to an incorrect output or a value that causes the program to crash). Due to the size of the dynamic dependence graph, which keeps growing as program executes, one challenge of dynamic slicing is the cost of computing it [23]. Our prior work on dynamic slicing has already addressed this problem [25, 24]. The effectiveness of dynamic slicing in fault location is determined by two factors: *How often is the faulty statement present in the slice?* and *How big is the slice, i.e. how many statements are included in the slice?* In our previous work [27] we evaluated the effectiveness of backward dynamic slicing in fault location. We observed that dynamic slices are able to contain the faulty statement in most of the cases and in general dynamic slices are quite small compared to the number of executed statements. However, we also observed that the absolute number of statements in the slice could still be large and in addition many of the statements are apparently unlikely to be faulty even though they are present in the slice.

Given an observed incorrect value $\times_o$, in this paper, we develop an approach for computing a *Pruned Dynamic Slice* of $\times_o$, $PDS(\times_o)$, which contains a subset of statements from the *Dynamic Slice* of $\times_o$, $DS(\times_o)$, that are highly likely to include faulty statements. We observe that although $DS(\times_o)$ contains all executed statements that are involved in computing $\times_o$, not all of these statements are equally likely to be involved in causing the erroneous behavior. In particular, let us consider a common situation in which the program produces some correct outputs ($\surd_o$'s) before producing the incorrect value $\times_o$. From the perspective of the $\surd_o$'s and $\times_o$, it is possible to divide the executed statements in $DS(\times_o)$ into two sets: *May Set*, $DS_{may}(\times_o)$, containing executed statements from $DS(\times_o)$ that are also involved in computing one or more of the $\surd_o$ values; and *Must Set*, $DS_{must}(\times_o)$, containing executed statements from $DS(\times_o)$ that were involved in computing none of the $\surd_o$ values. In other words:

$$DS(\times_o) = DS_{must}(\times_o) \cup DS_{may}(\times_o)$$

$$DS_{must}(\times_o) = DS(\times_o) - \bigcup_{\forall \surd_o} DS(\surd_o)$$

$$DS_{may}(\times_o) = DS(\times_o) - DS_{must}(\times_o)$$

While the statements in the $DS_{must}(\times_o)$ are always included in the pruned slice $PDS(\times_o)$, the ones in $DS_{may}(\times_o)$ may or may not be included in $PDS(\times_o)$. We develop an analysis that computes for value v computed by each statement execution $s \in DS_{may}(\times_o)$ a *confidence value* $C(v@s)$ between 0 and 1. High confidence value for a statement execution indicates that we have a high confidence that the statement produced a correct value. The confidence values are computed using the *value profiles* of the executed statements. A *threshold confidence* τ may be set such that only statement executions in $DS_{may}(\times_o)$ that have a confidence of less than τ are included in the pruned dynamic slice $PDS_\tau(\times_o)$. In other words:

$$PDS_\tau(\times_o) = DS_{must}(\times_o) \cup DS^\tau_{may}(\times_o)$$

$$where,\ DS^\tau_{may}(\times_o) = \{s\ s.t.\ s \in DS_{may} \wedge C(v@s) < \tau\}$$

As we will show later, our analysis may yield confidence values of 1 for many statement executions and thus they are pruned from the dynamic slice irrespective of the choice of τ, i.e. they are never included in $PDS_\tau(\times_o)$ for all τ.

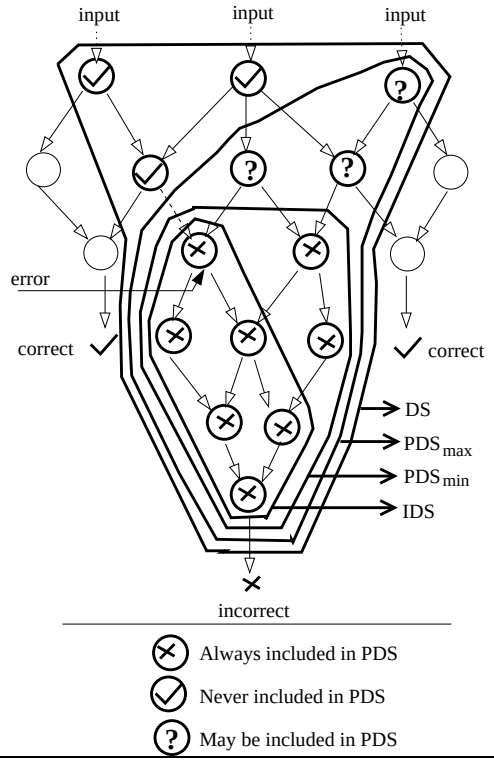


Figure 1. Pruning dynamic slice.

Figure 1 illustrates pruning of dynamic slices visually. It shows a dynamic dependence graph of a computation that produces two $\surd_o$ values before producing the incorrect value $\times_o$. DS is the *Dynamic Slice* of $\times_o$. A subset of nodes in DS that form the *Ideal Dynamic Slice* (IDS) is shown – IDS originates at the point of program error. The nodes in DS that are not present in IDS have been divided into three categories. The nodes labeled $\times$ in DS belong to DS_{must}, as they are not involved in computing the $\surd_o$ values, and thus they are always included in PDS, the *Pruned Dynamic Slice* of $\times_o$. The remaining nodes in DS are labeled with either $\surd$ or ?. The $\surd$ nodes have confidence value of 1 and thus they are never included in PDS. The nodes labeled ? have a confidence value of less than 1 and thus the value of threshold τ determines whether or not they are included in PDS. The identification of $\surd$ nodes is made possible by recognizing that *any change in the values produced by such nodes would alter the output values that were known to be correct*. Therefore it is assumed that these nodes must have produced correct values. As the figure shows, the smallest (largest) pruned dynamic slice that is produced by our algorithm corresponds to PDS_{min} (PDS_{max}). The key point to note here is that even if τ is set to 1, we obtain a pruned dynamic slice PDS_{max} which is smaller than the dynamic slice DS. We would also like to point out that PDS_{min} is actually what is known as a *dynamic dice* [3] – as our experiments later in the paper show, often when faulty code is not captured by the dynamic dice it is captured by PDS_{max}.

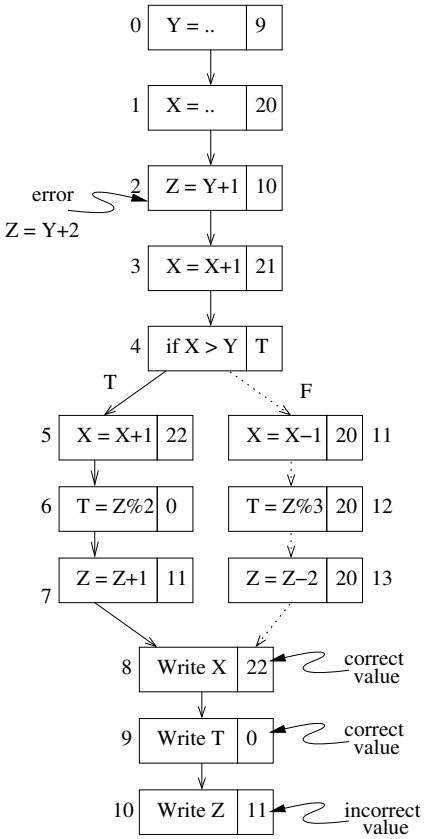
Next we present a motivating example which shows how analysis of code and runtime information can be used such that the confidence values of some statement executions in DS_{may} is determined to be 1. Figure 2(a) shows an execution of a program that follows the path corresponding to the true evaluation of the predicate at node 4. The value shown to the right of each statement is the value computed by the statement during the execution. The dynamic dependence graph of this execution is shown in Figure 2(b) – the solid edges are data dependence edges while dotted edges are control dependence edges. The nodes in the dynamic slice of the incorrect output value produced by statement 10 include $\{0, 1, 2, 3, 4, 7, 10\}$. Now let us see how the correct outputs produced by statements 8 and 9 are used to mark the nodes in DS_{may} as $\surd$ or ?.

- From the correct output value of X written by statement 8 we infer that the values produced by statements 1, 3 and 5 are also correct. The reasoning on which this inference is based is as follows. The statements 3 and 5 represent *one-to-one mappings* between the used operand values and generated result values of X. Therefore any change in the values produced by 1, 3 or 5 will cause the value output at statement 8 to change. However, the value output at statement 8 is known to be correct. Thus, we mark statements 1, 3 and 5 with $\surd$ indicating that they produce correct values. We further conclude that the *true* evaluation of predicate $X > Y$ is also correct. This is because if $X > Y$ would have evaluated to false, it would have produced a different output value for X at statement 8.
- Now let us consider the other correct output value written by statement 9. Since statement 6 does not represent a one-to-one mapping between its operand and result, even though the value of T that is produced by statement 6 is correct, we do not assume that the value of operand Z used in statement 6 is correct. As a result we conclude that values produced by statements 0 and 2 may or may not be correct and therefore we mark them with a ?. Note that the value of Y generated by statement 0 has another use in the predicate $X > Y$. Even though we have determined that the predicate correctly evaluated to true, we cannot determine from this fact that the value of Y used by the predicate is correct because many different values of Y would have produced the correct *true* evaluation of the predicate. Thus, from both uses of Y we conclude the same thing, i.e. the value of Y produced by statement 0 may or may not be correct.

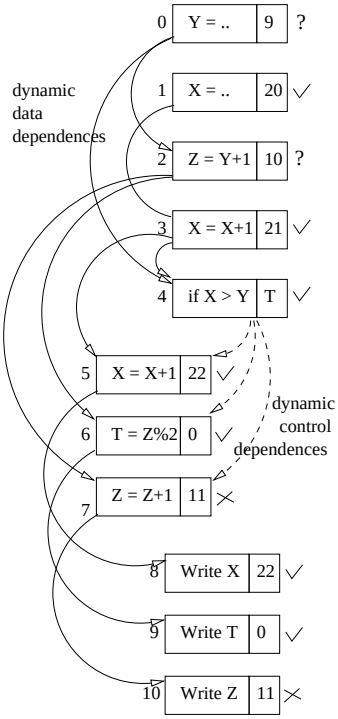
Given the above observations, the pruned dynamic slice of incorrect value output at statement 10 will always include statements 7 and 10. More importantly it will never include statements 1, 3, 4 and 5. However, it may or may not include statements 0 and 2. The confidence values for statements 0 and 2 will be compared with the threshold τ to make this determination. In other words:

$$DS = \{0, 1, 2, 3, 4, 7, 10\}; \quad IDS = \{2, 7, 10\}$$

$$PDS_{max} = \{0, 2, 7, 10\}; \quad PDS_{min} = \{7, 10\}$$



(a) Program Execution.



(b) Dynamic Dependence Graph.

Figure 2. Pruning dynamic slice.

In the remainder of this paper we will present a confidence estimation method that will produce the following results. First for the above example it will produce a confidence value of 1 for values produced by statements 1, 3, 4, and 5. Therefore our pruning algorithm will correctly remove statements 1, 3 and 4 from the dynamic slice of the incorrect output value produced by statement 10. Second it will produce confidence values of less than 1 for statements 0 and 2 such that the confidence value of statement 0 is more than confidence value of 2. Thus, depending upon the value of τ, three possible pruned slices will result: $\{0, 2, 7, 10\}$, $\{2, 7, 10\}$ and $\{7, 10\}$. The computation of confidence values will be performed using the *value profiles* of the executed statements (i.e., the operand values used and result values produced during statement executions).

Given a failed run involving the execution of a single error, our experiments show that the pruning of a dynamic slice by excluding only the statements with the confidence value of 1 is highly effective in reducing the size of the dynamic slice while retaining the faulty code in the slice. We observe that the number of distinct statements in a pruned dynamic slice PDS_{max} are 1.79 to 190.57 times less than the conventional dynamic slice. Confidence values also prioritize the statements in the dynamic slice according to the likelihood of them being faulty. We also show that locating a fault by examining the statements in the order of increasing confidence values is an effective strategy.

The rest of the paper is organized as follows. In section 2 we formally define *confidence values* and show how value profiles are used for *estimating confidence values*. In section 3 we present results of experiments to demonstrate the effectiveness of pruning dynamic slices using estimates of confidence values. Related work is discussed in section 4 and we conclude in section 5.

2. Confidence Analysis

In this section we develop an analysis that will serve as the basis for pruning a conventional dynamic slice. In this work we *assume* that for the failed program run being analyzed, the failure was caused due to the execution of a *single error* in the program (i.e., even though the program may contain many bugs, the failure was caused by encountering one of those bugs). Our goal is to develop a *heuristic* for pruning a dynamic slice such that the size of the dynamic slice is significantly reduced and very rarely is the erroneous statement pruned from the dynamic slice. In other words we would like to significantly reduce the size of the slice with minimal loss in fault location effectiveness. Before we describe our analysis, we present some basic definitions.

DEFINITION 1. *The* **Dynamic Dependence Graph** *of a program run, DDG(N, E), consists of a set of nodes N and a set of directed edges E where: each node $n_i \in N$ corresponds to i^{th} execution instance of statement n in the program; and each edge $m_j \rightarrow n_i \in E$ corresponds to a dynamic data dependence or dynamic control dependence of i^{th} execution instance of statement n on the j^{th} execution instance of statement m.*

In other words, with the execution of each statement during a program run, a new node is added to the dynamic data dependence graph and the incoming edges to the node from other nodes on which the new node is data and control dependent are introduced.

The execution of every statement during a program run results in the computation of a result value. For an assignment statement this is the value assigned to the left hand side variable during the execution while for a predicate statement the value is either true or false corresponding to the result of the predicate's evaluation. The dynamic slice of a value computed by a statement is defined as follows.

DEFINITION 2. *Given DDG(N, E), a dynamic dependence graph, the* **Dynamic Slice** *of $n_i \in N$ denoted by DS(n_i) is the subgraph of DDG(N, E) which includes n_i as well as all other nodes and edges from which n_i is reachable, i.e.*

$$DS(n_i) = (\{n_i\}, \{e | e = m_j \rightarrow n_i \in E\}) \cup \bigcup_{\forall m_j \rightarrow n_i} DS(m_j)$$

Consider a failed program run from which two kinds of evidence are collected: *negative evidence* in form of the first incorrect value $\times_o$ observed by the programmer during the program run; and *positive evidence* in form of some correct output values ($\surd_o$s) generated during the program run before the incorrect value $\times_o$ was generated. We classify each *relevant* value, i.e. value that was involved directly or indirectly in computing $\times_o$ and/or $\surd_o$ values, into three distinct categories as defined below.

DEFINITION 3. *A relevant value v generated by node n is classified as:*

- *$\surd$ or correct if it is used in computing at least one of the $\surd_o$ values but it is not used in computing the incorrect value $\times_o$. Therefore values computed by all nodes in $\bigcup_{\forall \surd_o} DS(\surd_o) - DS(\times_o)$ are classified as $\surd$;*
- *$\times$ or incorrect if it is used in computing the incorrect value $\times_o$ but it is not used in computing any of the $\surd_o$ values. Therefore values computed by all nodes in $DS(\times_o) - \bigcup_{\forall \surd_o} DS(\surd_o)$ are classified as $\times$; and*
- *? or unknown if it is used in computing the incorrect value $\times_o$ and at least one of the $\surd_o$ values. Therefore values computed by all nodes in $DS(\times_o) \cap \bigcup_{\forall \surd_o} DS(\surd_o)$ are classified as ?.*

In [27] it has been shown that dynamic slice $DS(\times_o)$ typically contains the erroneous code responsible for producing the incorrect value $\times_o$; however, it also includes many statement executions that are not responsible for generating the incorrect value. According to the above definitions, statement executions in $DS(\times_o)$ will be initially classified into two categories – some will be classified as $\times$ while others will be classified as ?. The ones that are classified as $\times$ are always included in the dynamic slice. However, we perform analysis to determine what subset of statement executions classified as ? should be included in the pruned dynamic slice.

The decision as to whether the statement executions in the dynamic slice that are classified as ? should be included in the pruned dynamic slice is based upon *confidence analysis*. For every value v computed by statement execution n, confidence analysis produces a confidence value $C(v@n)$ that measures the likelihood of the value being correct. The confidence estimate $C(v@n)$ ranges from 0 to 1 where $C(v@n) = 0$ indicates that we have no confidence at all in the correctness of value v while $C(v@n) = 1$ indicates that we have the highest possible confidence in the correctness of value $v@n$. This estimate is defined as shown below.

DEFINITION 4. *Confidence estimate of value v computed by a relevant node n is defined as follows:*

- *if v is classified as $\surd$ (i.e., correct) then*

$$C(v@n) = 1$$

- *elseif v is classified as $\times$ (i.e., incorrect) then*

$$C(v@n) = 0$$

- *elseif v is classified as ? (i.e., unknown) then*

$$C(v@n) = 1 - log_{|Range(v@n)|}|Alt(v@n)|$$

where $Range(v@n)$ represents all legal values of v and $Alt(v@n) \subseteq Range(v@n)$ is a set of alternate values of v such that if any value in $Alt(v@n)$ was produced by n, the same correct $\surd_o$ values would have resulted.

Let us discuss the reasoning behind the $C(v@n)$ computation when v is classified as ?. If any change whatsoever in the value computed by n would cause at least one of the $\surd_o$ values to change and hence become incorrect, then we conclude that the value v computed by n during the program run must have been correct. In this case the set $Alt(v@n)$ contains only one value. Therefore as desired, the confidence estimate $C(v@n) = 1 - log_{|Range(v@n)|}1 = 1$. On the other hand, if changing v to other values can still yield the same $\surd_o$ values, then we have less confidence in the correctness of value v. As the set $Alt(v@n)$ increases in size, the confidence estimate $C(v@n)$ reduces and when $Alt(v@n)$ is equal to $Range(v@n)$, then $C(v@n) = 1 - log_{|Range(v@n)|}|Alt(v@n)| = 0$.

Before settling on the above definition of confidence, we considered other simpler definitions of confidence but we found them not to be nearly as effective. For example, we considered a definition in which each value's confidence was proportional to the number of correct outputs whose computation depended upon that the value. However, we observed that in many cases different outputs were derived from different values and thus many values were assigned the same confidence. In addition, this simpler method fails to exploit the knowledge that sometimes even though a value may be involved in computing a single correct output, by looking at the statements involved we may be able to definitely determine that the value is correct. For example, in Figure 2(b), since the value of X output by statement 8 is correct, we can determine that the value of X computed by statement 1 must be correct. This is because the statements along the data dependence chain (4 and 5) perform one-to-one mapping between old and new values of X.

Next we develop an algorithm for computing confidence estimates. While our definition of confidence estimates is quite simple, the computation of confidence estimates is made challenging by the need for deriving the $Range(v@n)$ and $Alt(v@n)$ sets for all nodes n that are classified as ?. There are two key problems that must be addressed. First, given a variable x referenced by a program statement s, we first define the set $Range(v@n)$, i.e. the set of *legal values* that x may be allowed to take during its reference by an execution of s. Once such a legal set of values is determined for all variable references, the $Alt()$ sets will be computed with respect to these legal values. Second, we must develop an algorithm for propagation of values. Starting from the values classified as $\surd$, we traverse the dynamic dependence graph in a bottom up fashion to compute the $Alt()$ sets of values classified as ?. The $Alt()$ set of a value classified as $\surd$ is initialized to empty while the $Alt()$ set of a value classified as ? is computed by examining the $Alt()$ sets of its child nodes in the dynamic dependence graph.

Let us first discuss how the set of *legal values* is determined for each variable reference. A simple approach would be to use all possible values a variable can take based upon its type (integer, char, boolean) or compute a more accurate set using static analysis (e.g., range propagation [2]). However, such an overestimate is not very desirable for debugging because during debugging we are interested in analyzing a single program execution (i.e. the failed run) corresponding to a specific program input. Therefore we use the *value profile* for the failed run to supply the set of legal values $Range()$.

DEFINITION 5. *Given a reference (definition or use) to a variable v in a program statement s, the value profile $VP(v@s)$ provides an ordered list of values taken by variable v during the multiple executions of s in the failed run.*

S: Y = ..
if (..)
C^1: X=Y+1
fi
if (..)
C^2: X=Y%2
fi
if (..)
C^3: X=Y+Z
fi

Reference	Value Profile
$Y@S$	{1,2,3,4,5,6,7,8,9}
$Y@C^1$	{7,8,9}
$X@C^1$	{8,9,10}
$Y@C^2$	{1,2,3,9}
$X@C^2$	{1,0,1,1}
$Z@C^3$	{9,9,5,5}
$Y@C^3$	{4,5,6,9}
$X@C^3$	{13,14,11,14}

Figure 3. Value profiles.

A program run generates a large number of values and exercises a large number of dynamic dependences. Capturing this history to perform dynamic slicing is a challenge that is already addressed in our prior work [25, 26]. We developed a highly compressed form, *whole execution traces* [26], that enables us to hold the execution trace of 1 to 2 billion instructions in 1 GB of memory.

Now let us consider the rules of propagation along data and control dependence edges in the dynamic dependence graph. Intuitively, given an execution instance of a statement, the values in the $Alt()$ set of the result computed by the statement are constrained by each of its children in the dynamic data dependence graph. Only those values can be put into the $Alt()$ set that do not adversely impact any of the $\surd_o$ values along any chain of dependence edges from the executed statement to any of the $\surd_o$ values. Therefore, we also associate $Alt()$ sets with dynamic dependence edges and then the $Alt()$ set for the result value of an executed statement is simply computed by intersecting the $Alt()$ sets of edges leaving the statement. There are two key operations involved in propagation. First from the $Alt()$ set of a result computed by an executed statement, we compute the subset of legal values that the operands can take such that these operand values produce the result values contained in the $Alt()$ set. Second the $Alt()$ set of a result is computed by examining the subset of legal values already determined at each of the uses of the result value.

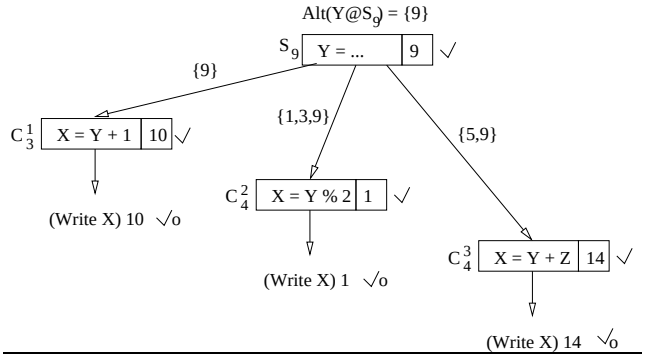


Figure 4. Dependences among assignment statements.

Let us consider propagation along dynamic data dependence edges that connect assignment statements (we will also consider predicate statements shortly). We illustrate propagation by analyzing the result value computed by 9^{th} execution instance of statement S in the example from Figure 3. The value of Y computed by S_9 is 9 and this value is used later by 3^{rd}, 4^{th}, and 4^{th} execution instances of statements C^1, C^2 and C^3 respectively. The dynamic dependence will therefore include three data dependence edges $S_9 \rightarrow C_3^1$, $S_9 \rightarrow C_4^2$, $S_9 \rightarrow C_4^3$. We further assume that the values of X computed by C_3^1, C_4^2 and C_4^3 are output and determined to be correct. Figure 4 first shows how the potential values in $Alt(Y@S_9)$ set are identified by considering each dynamic data dependence individually. Given that C^1 represents a *one-to-one mapping* between the value operand Y and result X, the $Alt(Y@S_9 \rightarrow C_3^1)$ set obtained is empty. In contrast, since statements C^2 and C^3 do not represent *one-to-one mappings* between the value of operand Y and the value of result X, the sets $Alt(Y@S_9 \rightarrow C_4^2)$ and $Alt(Y@S_9 \rightarrow C_4^3)$ corresponding to dynamic data dependence edges $S_9 \rightarrow C_4^2$ and $S_9 \rightarrow C_4^3$ contain more than one value. However, the $Alt(Y@S_9)$ is computed by intersecting the three sets for the three dynamic data dependences yielding a set with only one element. Therefore the confidence estimate $C(Y@S_9) = 1$ and therefore we mark the value computed by S_9 as $\surd$, i.e. correct.

From the above analysis we observe two things. First, the presence of one-to-one mappings are greatly beneficial in pruning a dynamic slice since they prevent $Alt()$ sets from expanding as propagation proceeds. Second, we observe that as long as there is one data dependence edge along which a computed value can be verified (i.e., its $Alt()$ set contains one value), the value is considered verified. As we will show later, our approach is very effective because programs often contain many statement executions that correspond to one-to-one mappings (e.g., copy operations, expressions with two operands one of which is a constant etc.).

In the above example we considered propagation along dynamic data dependence edges and these edges were present between assignment statement executions. Next we see how to handle the situation in which predicate evaluations are present and hence dynamic control dependence edges are also present. There are two points to be made here. First we classify the value of a predicate as being correct ($\surd$) if the value of one of its direct or indirect control dependent assignment statements has been determined to be $\surd$. This is because if the predicate would have evaluated differently the variable assigned by the control dependent assignment would have had a different value and hence it would have adversely affected one of the $\surd_o$ values through its further uses. Second it should be noted that when the result value of a predicate is classified as correct, it only means that the outcome of the predicate evaluation (true or false) is correct. However, since a predicate usually represents a *many-to-one* mapping between its operand values and true/false result, we cannot infer that the operand values are necessarily correct. The only thing we can say is that the operand values are the subset of legal values for which the predicate produces the same desired result, i.e. true/false. To illustrate the above points we use a fragment of the previous example as shown in Figure 5. The

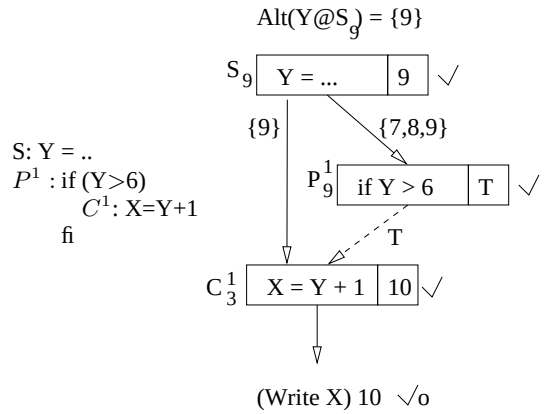


Figure 5. Dependences involving predicates.

dynamic dependence graph and the results of analysis are shown in the figure. Note that the predicate evaluation P_9^1 is marked as

Initialize: $Alt(*) \leftarrow \phi$;
for each relevant node S_i marked ? in *bottom-up order* do
  if S_i is an *assignment* $X = ..$ then
    $ComputeAlt(Alt(X@S_i))$;
    if $|Alt(X@S_i)| = 1$ then
      $C(X@S_i) = 1$; *mark* S_i as $\surd$;
    else
      $C(X@S_i) = 1 - log_{|Range(X@S_i)|}|Alt(X@S_i)|$
    endif
  elseif S_i is a *predicate* then
    if $\exists\ S_j$ st S_j dynamically control dependent upon S_i
        and S_j is marked $\surd$
    then *mark* S_i as $\surd$ endif
  endif
endfor

$ComputeAlt(Alt(X@S_i))$
  Let the following dynamic dependence edges lead
  from S_i to nodes marked $\surd$ or ?:
  to assignments: $S_i \rightarrow C^1_{1i}, S_i \rightarrow C^2_{2i}, ... S_i \rightarrow C^n_{ni}$;
  to predicates: $S_i \rightarrow P^1_{1i}, S_i \rightarrow P^2_{2i}, ... S_i \rightarrow P^m_{mi}$.

  for each $C^j : Y = f(X)\ st\ \exists\ S_i \rightarrow C^j_{ji}$ do
    $Alt(X@S_i \rightarrow C^j_{ji}) = \{v :$
      $v \in VP(X@C^j) \wedge C^j(X = v) \in Alt(Y@C^j_{ji})\}$
  endfor
  for each $P^j : f(X)\ st\ \exists\ S_i \rightarrow P^j_{ji}$ do
    $Alt(X@S_i \rightarrow P^j_{ji}) = \{v :$
      $v \in VP(X@P^j) \wedge P^j(X = v) = P^j_{ji}\}$
  endfor
  $Alt(X@S_i) = \bigcap_{\forall j, S_i \rightarrow C^j_{ji}} Alt(X@S_i \rightarrow C^j_{ji})$
    $\cap \bigcap_{\forall j, S_i \rightarrow P^j_{ji}} Alt(X@S_i \rightarrow P^j_{ji})$
end$ComputeAlt$

Figure 6. Confidence computation algorithm.

$\surd$ because its dynamic control dependent child C^1_3 is marked $\surd$. $Alt(Y@S_9 \rightarrow P^1_9)$ also includes values 7 and 8 in addition to 9 as for these legal values of Y, the predicate $Y > 6$ evaluates to true just as it evaluates to true for the value 9 produced by S_9.

The process we have described is summarized fully in the algorithm presented in Figure 6. All nodes in the dynamic dependence graph that have been marked as ? are the ones that are processed to compute their confidence estimates. The $Alt()$ sets for all nodes are initialized to the empty set. The nodes marked ? are then processed in a bottom-up order one by one. If a node being processed is an assignment statement then the $Alt()$ set for its result value is computed, from which then its confidence estimate is derived. Predicate nodes are processed by considering the markings on their dynamically control dependent assignment statements. In Figure 6, the function $ComputeAlt()$ presents the details of the $Alt()$ set computations which were described intuitively earlier.

3. Experimental Results

3.1 Implementation and Benchmarks

We have developed a dynamic slicing framework which was used to conduct experiments. Our tool executes gcc compiler generated binaries for Intel x86 and captures dynamic information including dependence, value, and control flow traces [26]. Even though our tool works at binary level, the dynamic information is easily mapped back to source code level using the debugging information generated by gcc. Figure 7 shows the main components of our tool. The *static analysis* component of our tool computes static control dependence (CD) required for forward/backward slice computations from the binary. The static analysis was implemented using the *Diablo* [32] retargetable link-time binary rewriting framework as this framework already has the capability of constructing the control flow graph from an Intel x86 binary. The *dynamic profiling* component of our system which is based upon the *Valgrind* memory debugger and profiler [33] accepts the same gcc generated binary, instruments it by calling the *slicing instrumenter*, and executes the instrumented code with the support of the *slicing runtime*. The slicing instrumenter and the slicing runtime were developed by us to enable the collection of dynamic information. Valgrind's kernel is a dynamic instrumenter which takes the binary and before executing any new (never instrumented) basic blocks it calls the instrumentation function, which is provided by the slicing instrumenter. The instrumentation function instruments the provided basic block and returns the new basic block to the Valgrind kernel. The kernel executes the instrumented basic block instead of the original one. The instrumented basic block is copied to a new code space and thus it can be reused without calling the instrumenter again. The *slicing runtime* essentially consists of a set of call back functions for certain events (e.g., entering functions, accessing memory, binary operations, predicates etc.). It also manages the shadow memory which is used to capture dynamic dependences. More details on the working and use of shadow memory can be found in [27]. We intercept the output system call (*_WRITE etc.*) and then augment the original output with its corresponding position in the DDG. The *confidence component* implements the analysis in this paper. It receives dynamic information from the slicing runtime and stores it as DDG, which is a variant of our prior WET representation [26].

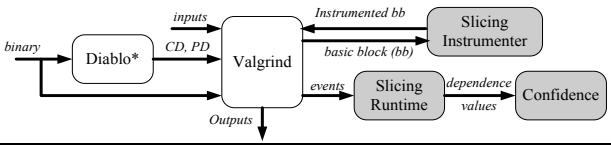


Figure 7. Tool infrastructure.

Table 1 shows the benchmarks used in our experimentation. The first five are known as the Siemens suite programs [11]. The last two unix utilities are also available from the same website [31]. We use this suite of programs because it provides several faulty versions of the programs which have exactly one fault injected in each one of them. The versions used in our experiments are also indicated in Table 1. For each faulty version many test inputs are also provided in [11]. Different inputs often result in different positions for the first incorrect output in the output stream. The column *position range* of Table 1 gives the range of the positions of the first observed wrong output. The greater the position number, the greater is the number of correct outputs produced before the incorrect output. We can see that it is common for a certain number of correct outputs to be generated in a failed run. In fact these numbers can be very high for some failed runs. We exclude the program *tcas* and *tot_info* from the Siemens suite because *tcas* is too small and *tot_info* has floating point operations, which are currently not supported by our tool. We do have some real world benchmarks. However, most of them are memory corruption errors for which program executions terminated before any output was produced.

The test suite provides more versions than those used in our experiments. We excluded some of the versions as they are not appropriate for experimentation. Some versions produce no output or the very first output produced is wrong. Therefore our approach is not applicable. In two kinds of situations the faulty statement is not present in the dynamic slice itself and thus we cannot study the ef-

Benchmark	Version	Error in	Failed Cases	Position Range
print_tokens (565 LOC)	1	switch-case	6	[14-495]
	2	switch-case	143	[17-1707]
	4	constant	23	[17-1209]
	6	constant	143	[13-2714]
	7	predicate	28	[8-1271]
print_tokens2 (510 LOC)	4	assignment	268	[20-394]
	5	return	67	[20-1106]
	6	parameter	329	[20-870]
	7	predicate	158	[27-486]
	8	predicate	194	[60-928]
replace (563 LOC)	1	predicate	24	[2-20]
	3	predicate	130	[2-666]
	6	loop condition	92	[2-609]
	9	predicate	92	[2-609]
	14	predicate	92	[3-49]
	18	predicate	190	[2-380]
	21	predicate	2	[18-40]
	25	predicate	2	[3-11]
schedule (412 LOC)	2	assignment	200	[2-38]
	4	predicate	267	[2-39]
	7	added code	20	[2-14]
schedule2 (307 LOC)	5	added code	32	[5-28]
	6	constant	2	[10-18]
	7	predicate	20	[2-16]
gzip (7199 LOC)	1	predicate	6	[19-19]
flex (12418 LOC)	4	constant	12	[16885-53109]
	5	constant	257	[7130-9056]
	7	constant	97	[6164-6164]
	10	array index	6	[7142-7144]
	11	predicate	513	[6867-43647]
	15	constant	515	[13430-53895]
	17	constant	315	[10632-51067]
	19	constant	343	[20495-61777]

Table 1. Characteristics of benchmarks

fectiveness of pruning in such cases. First, *code omission* faults are present in some versions. Since such faults are not even captured in the static slice of the output, they cannot be caught by any dynamic slicing algorithm. Second, it is known that the dynamic slice of the incorrect output does not always include the erroneous statement executed. This can happen when the erroneous output is produced due to an incorrect evaluation of a branch predicate causing the execution of some statements to be incorrectly bypassed. This situation can be handled by constructing an expanded dynamic slice called the *relevant slice* [6, 27]. While in our experiments we omit such cases, later we show how they can be handled by extending our technique.

3.2 Confidence-based Pruning

Since for some faulty versions there are many test inputs, and some of these may not differ much in their behavior, for each faulty version we selected three test inputs such that varying number of correct outputs are generated before the incorrect output is produced. Whenever possible, we selected three runs such that the wrong output was observed at: the lower bound of *position range* in the first run; closest to the middle of *position range* in the second run; and at the upper bound of *position range* in the third run. For each run, we first computed the dynamic slice of the wrong output and then pruned the slice using confidence analysis. We present six numbers about the slice sizes in Tables 2 and 3. $All.PDS_{min}$, $All.PDS_{max}$, and $All.DS$ represent the number of DDG nodes in PDS_{min}, PDS_{max}, and DS. The corresponding *distinct* numbers ($D.PDS_{min}$, $D.PDS_{max}$, and $D.DS$) denote the number of unique statements in them (note that one unique statement may get executed many times and result in many nodes in DDG). We also present the fault location effectiveness in column *Error In*. Here I, X, and D indicate the presence of erroneous statement in PDS_{min}, PDS_{max}, and DS respectively. The results are also summarized by taking averages across different versions of each benchmark in Table 4.

From these two tables, we make the following observations:

(1) The confidence analysis greatly reduces the size of dynamic slice without sacrificing the fault location effectiveness. Table 4 shows the average factor by which PDS_{max} is smaller than DS ranges from 4.31 to 87514.33 (all) and 1.79 to 190.57 (distinct). For *flex*, the slices are so precisely reduced that they simply contain the chain of dependences from the erroneous statement to the incorrect output – this chain includes only a few statements.

(2) For most of the versions, we used three runs and studied the relation between the pruning capability and the number of correct outputs. The absolute sizes of the PDSs appear to be independent of the number of correct outputs. However, the reductions in the sizes of PDSs with respect to the sizes of DSs increase as the number of correct outputs grow because of the increases in the sizes of DSs.

(3) We observe that the fault location effectiveness of PDS_{max} is very good. Even though it is much smaller than DS, only in one case the erroneous statement is removed during pruning – this happened in *replace* version v9 run r2. Fig. 8 explains how this happened. In this run, statement $i = i + 1$ is wrong such that 'D' is assigned to the wrong position in array pat. However, statement *return flag* is verified and thus *flag=true;* is verified, which means the predicate is correct. Since the predicate represents a one-to-one mapping to its operand when it evaluates to *true*, $pat[j]$ contains the correct value 'D'. According to our analysis, the store to $pat[i]$ will get verified and so will the wrong index. As is illustrated in the right hand side of Fig. 8, $pat[j]$ being correct is the result of both array pat and j being wrong.

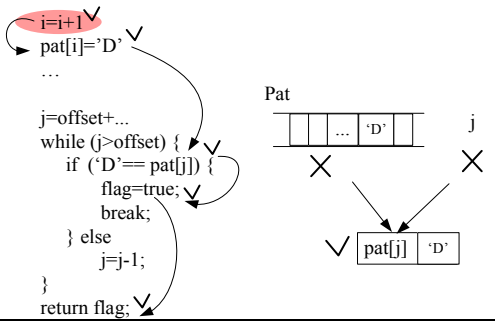


Figure 8. Replace version v9 run r2

(4) Let us compare PDS_{max} with PDS_{min}. Although PDS_{min} works for a large number of test cases, we did observe that in several cases, such as *replace v1, v3, v9, v21* and *schedule v7*, it prunes the erroneous statement while PDS_{max} does not do so. On the other hand, PDS_{max} works almost equally well for the cases in which PDS_{min} also works. As shown in Table 4, when the erroneous statement is captured in both PDS_{max} and PDS_{min}, corresponding to the IX columns, PDS_{max}/PDS_{min} is roughly one, i.e. their sizes are nearly the same (the entries marked NA are ones where there were no slices in that category). Thus, using confidence analysis to obtain PDS_{max} is an effective method for both pruning the slice and maintaining the fault location effectiveness.

(5) In some cases such as *flex v15*, part of the wrong output appears to be correct which may cause some confusion. For example, *flex v15* has the error of *printf ("YY_USER_ACTION")* missing a

Benchmark	Version	Wrong Output Pos.	($All.PDS_{min}$-$All.PDS_{max}$)/$All.DS$	($D.PDS_{min}$-$D.PDS_{max}$)/$D.DS$	Error In
print_tokens	1	14	(310-310)/712	(41-41)/72	IXD
		301	(239-240)/4582	(40-40)/86	IXD
		495	(317-317)/13603	(41-41)/134	IXD
	2	17	(70-70)/429	(19-19)/61	IXD
		231	(68-69)/3605	(18-18)/86	IXD
		1707	(70-70)/44158	(19-19)/149	IXD
	4	17	(246-246)/603	(40-40)/69	IXD
		91	(212-212)/1965	(35-35)/92	IXD
		1206	(263-295)/28513	(43-43)/141	IXD
	6	13	(1457-1470)/1804	(44-44)/71	IXD
		109	(214-214)/1993	(35-35)/97	IXD
		2714	(432-432)/66651	(36-36)/145	IXD
	7	$8^{(1)}$	(399-400)/698	(41-41)/74	IXD
		$92^{(1)}$	(423-436)/1486	(41-41)/94	IXD
		$1271^{(1)}$	(390-391)/27274	(37-37)/136	IXD
print_tokens2	4	20	(174-174)/902	(40-40)/99	IXD
		47	(447-447)/1561	(50-50)/95	IXD
		394	(770-770)/8364	(44-44)/138	IXD
	5	20	(499-499)/850	(58-58)/97	IXD
		79	(364-364)/1013	(59-59)/109	IXD
		1106	(285-285)/27841	(56-56)/154	IXD
	6	20	(208-208)/680	(61-61)/95	IXD
		34	(208-208)/770	(61-61)/97	IXD
		870	(208-208)/18602	(61-61)/143	IXD
	7	27	(697-698)/1290	(59-60)/96	IXD
		75	(329-329)/1140	(53-53)/83	IXD
		486	(1105-1105)/10630	(67-67)/148	IXD
	8	$60^{(1)}$	(377-377)/2091	(59-59)/100	IXD
		63	(377-406)/1676	(48-51)/105	IXD
		928	(367-413)/20738	(48-51)/151	IXD
replace	1	2	(192-494)/2212	(38-77)/147	XD
		9	(241-461)/1625	(53-81)/130	XD
		20	(179-408)/1687	(44-64)/128	XD
	3	2	(160-671)/1012	(32-86)/136	IXD
		18	(89-89)/1997	(21-21)/155	IXD
		666	(17-868)/18522	(3-45)/125	XD
	6	2	(371-780)/1166	(45-62)/136	IXD
		19	(216-648)/2129	(28-50)/132	IXD
		609	(325-605)/20525	(46-49)/153	IXD
	9	2	(180-357)/889	(40-61)/115	XD
		$26^{(2)}$	(48-243)/3047	(18-42)/125	D
	14	3	(289-656)/1187	(55-88)/138	IXD
		9	(1006-1689)/2515	(73-117)/161	IXD
		49	(103-112)/3021	(23-28)/111	IXD
	18	2	(106-107)/669	(26-27)/109	IXD
		35	(152-152)/4145	(37-37)/143	IXD
		380	(194-194)/12588	(37-37)/127	IXD
	21	18	(390-781)/2372	(53-86)/132	XD
		40	(502-783)/3501	(42-59)/102	XD
	25	3	(321-531)/975	(55-78)/120	IXD
		11	(450-552)/2952	(72-84)/165	IXD
schedule	2	2	(464-465)/1046	(65-66)/93	IXD
		10	(621-623)/2155	(69-69)/118	IXD
		38	(295-359)/6176	(55-55)/119	IXD
	4	2	(1225-1468)/2605	(88-98)/119	IXD
		10	(1025-1029)/2155	(85-89)/117	IXD
	7	2	(386-399)/726	(67-68)/90	IXD
		6	(83-284)/1124	(24-65)/105	XD
		14	(84-330)/2146	(24-59)/97	XD
schedule2	5	5	(1152-1152)/1823	(64-64)/83	IXD
		14	(195-195)/2594	(34-34)/73	IXD
		28	(1896-1896)/5639	(60-60)/79	IXD
	6	10	(230-230)/1611	(40-40)/67	IXD
		18	(254-254)/2526	(42-42)/67	IXD
	7	2	(80-145)/696	(27-36)/67	IXD
		6	(113-129)/2871	(25-27)/94	IXD
		16	(693-709)/3311	(59-61)/84	IXD

(1). Part of the wrong output appeared to be correct;
(2). The root cause was pruned.

Table 2. Pruning effectiveness results of faulty versions for up to three test inputs.

Benchmark	Version	Wrong Output Pos.	$(All.PDS_{min}$-$All.PDS_{max})/All.DS$	$(D.PDS_{min}$-$D.PDS_{max})/D.DS$	Error In
gzip	1	19	(82-394520)/1699490	(10-121)/357	XD
flex	4	$16885^{(1)}$	(13-14)/62235	(7-8)/692	IXD
		$19825^{(1)}$	(16-17)/42823	(9-9)/648	IXD
		$53109^{(1)}$	(13-14)/1120244	(7-8)/889	IXD
	5	7130	(17-76)/23292	(6-18)/542	IXD
		8925	(4-4)/81991	(3-3)/681	IXD
		9056	(4-4)/59501	(3-3)/709	IXD
	7	6164	(17949-18026)/22886	(217-229)/280	IXD
	10	7142	(76-86)/84210	(19-23)/730	IXD
		$8925^{(1)}$	(74-75)/1021249	(17-18)/786	IXD
	11	6867	(15-15)/5756	(10-10)/81	IXD
		16092	(15-15)/39484	(10-10)/552	IXD
		43647	(15-15)/254532	(10-10)/720	IXD
	15	$13430^{(1)}$	(71-71)/30002	(14-14)/824	IXD
		$16092^{(1)}$	(71-71)/72756	(14-14)/988	IXD
		$53859^{(1)}$	(96-96)/1120987	(19-19)/941	IXD
	17	10632	(1-1)/22093	(1-1)/632	IXD
		11584	(1-1)/86515	(1-1)/813	IXD
		51067	(1-1)/1118733	(1-1)/864	IXD
	19	$20495^{(1)}$	(35-54)/32219	(16-20)/764	IXD
		$21955^{(1)}$	(35-35)/98133	(16-16)/947	IXD
		$61777^{(1)}$	(32-33)/1130822	(15-16)/981	IXD

(1). Part of the wrong output appeared to be correct;

Table 3. Pruning effectiveness results of faulty versions for up to three test inputs.

Benchmark	$(All.PDS_{min} - All.PDS_{max})/$ $All.DS$	$(D.PDS_{min} - D.PDS_{max})$ $D.DS$	$All.DS/$ $All.PDS_{max}$	$D.DS/$ $D.PDS_{max}$
print_tokens	(341-345)/1320	(35-35)/100	73.4	3.12
print_tokens2	(428-433)/6543	(55-55)/114	19.53	2.09
replace	(310-546)/4112	(43-60)/131	13.14	2.52
schedule	(454-596)/3188	(56-70)/117	9.41	1.79
schedule2	(562-630)/2358	(50-58)90	6.58	1.69
gzip	(82-394520)/1699490	(10-121)/357	4.31	2.95
flex	(1232-1240)/342692	(25-27)727	87514.33	190.57

Benchmark	$All.PDS_{max}/All.PDS_{min}$		$D.PDS_{max}/D.PDS_{min}$	
	IX	X	IX	X
print_tokens	1.01	NA	1	NA
print_tokens2	1.01	NA	1.01	NA
replace	1.78	8.55	1.38	3.36
schedule	1.08	3.68	1.03	2.58
schedule2	1.54	NA	1.29	NA
gzip	NA	4811.22	NA	12.1
flex	1.05	NA	1.04	NA

Table 4. Summary of results across all versions.

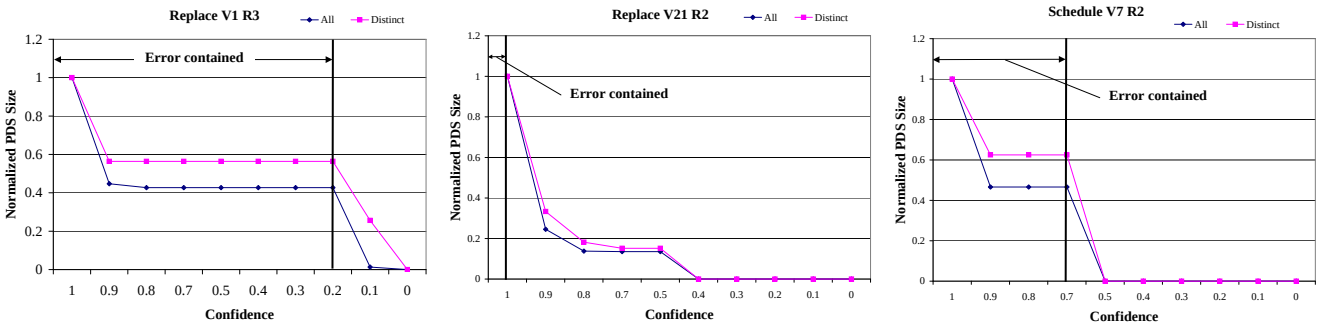


Figure 9. Pruned dynamic slice for varying threshold (version Vi run Rj).

'$\backslash n$' at the end of the string. If we assume the "YY_USER_ACTION" is correct, the wrong *printf* will get verified. To solve this problem, we divide the output into units, which is lines in this case, and compute slice on the first character of the wrong unit.

Interactive Pruning. It is possible that PDS_{max} is still quite big. However, pruning can be further carried out during debugging. During the course of debugging the programmer usually investigates the values in *gdb* and decides if they are correct or wrong. This information can be fed back to our confidence analysis to enable further pruning. Similarly, the user can also look at the slice and tell our system if certain values seem to be correct. We conducted an experiment trying to simulate this procedure. We picked *replace* version v14, one of whose three prunings (the third run) is quite successful and so we are able to understand the relation from the error to the wrong output. We use the most conservative pruned slice in the third run, PDS^3_{max}, as a reference when we start examining the PDS^1_{max} of the first run. We find the first statement which is in PDS^1_{max} but not in PDS^3_{max} and mark it as correct in our system. PDS^1_{max} is further pruned to 587/74 (*all*/*distinct*) from 656/88. After another two interactions, we are able to reduce it to 93/23, which is very close to dependence chain along which error is propagated. We also tried the same experiment with *replace v3* – we used the second run as a reference to prune the first run and found that in only one step, we reduce the slice from 671/86 (*all*/*distinct*) to 33/15 and it still contained the error.

Varying threshold. So far we have been looking at either PDS_{min} or PDS_{max}. In this experiment, we study the relationship between the threshold τ and the corresponding PDS_τ's size and its fault location effectiveness. The results for three different runs are plotted in Fig. 9. As we expected, the PDS_τ drops in both size and fault location effectiveness as τ decreases. However, we did not observe the existence of a value of τ that nicely balances between the size and the fault location effectiveness.

3.3 Confidence-based Prioritization

In the preceding section we explored the use of confidence values to carry out pruning of the dynamic slice. We observed that the most effective pruning strategy is one in which only the statements with confidence values of 1 are pruned from the dynamic slice to produce PDS_{max}. In this section we study an additional use of confidence values. The statements in PDS_{max} are *prioritized* in the order of increasing confidence values. To locate faulty code, the statements are then examined by the programmer in the order of increasing confidence values till the faulty code is encountered. The effectiveness of this strategy is measured in terms of the percentage of executed statements that are examined by the user before encountering the faulty code.

In prior work we have shown that an effective strategy for exploring dynamic slices to locate the faulty code is to examine the statements in the dynamic slice in increasing order of their dependence distance from the point at which the erroneous value is encountered during a failed run [27, 28]. We conducted an experiment in which we compared the effectiveness of the two strategies: exploring dynamic slice in order of increasing *dependence distances* (DD); and exploring pruned dynamic slice PDS_{max} in the order of increasing *confidence values* (CV). When using the confidence value based strategy, if two statements with same confidence value are present, then the dependence distance is used as the tie-breaker.

The results of this experiment are given in Figure 10. For a given point in each graph, the y-axis represents the fraction of faults located while the x-axis represents the percentage of executed statements examined to locate these faults. The results are for the same failed runs that were used in the experiments presented in the preceding section. As we can observe, for a given percentage of executed statements examined, typically the fraction of faults that are located is higher for CV in comparison to DD. It should be noted that there are other works (e.g., [12, 14]) that employ statistical analysis to prioritize statements for fault location. However, these techniques perform prioritization based upon dynamic information collected from multiple program runs. As far as we know, our technique is the only one that performs prioritization of statements based upon dynamic information collected from a *single* failed run.

3.4 Relevant Slicing

It is known that a dynamic slice may not be able to capture the error even though the wrong output is actually caused by the error [6]. Figure 11 gives such an example. It is taken from version v3 run r1 of *gzip*. The error is in the assignment to *save_orig_name*. The correct code is *save_orig_name=!no_name*. Since *save_orig_name* contains the wrong value $False$, branch S3 is not taken and thus $flags$ has the wrong value 0 while it should have been defined as $ORIG\_NAME$ at S3. This wrong $flags$ value is finally propagated to the output file. The dynamic slice DS of the wrong output does not contain the error because S4 depends on S2. The fact that S4 could have had a different value if S3 had taken the other branch cannot be captured by dynamic slicing technique itself.

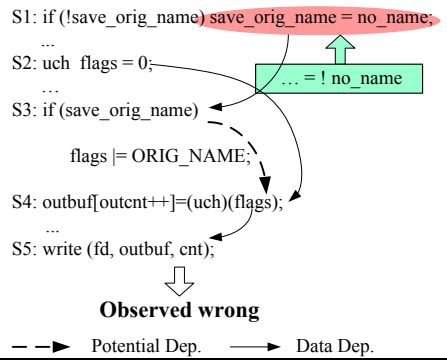


Figure 11. Gzip version v3 run r1

One proposed solution is to introduce *potential dependence* [6] between S4 and S3 such that S1 is reachable from the wrong output. However, these potential dependence edges are introduced for each node in the DDG which can result in a much larger slice. In our example, the computed slice has the size of 1809771/348 (all/distinct). Being presented with even a larger slice than this may not be appreciated by the programmer.

With our confidence analysis, we are able to select a more reasonable strategy. Even though the pruned slice may not capture the erroneous statement, it does capture a part of the dependence chain from the erroneous statement to the incorrect output. If the pruned slice is small enough, the programmer is able to inspect the entire pruned slice and determine the error is not present. Next the programmer can request for expansion of the pruned slice using the *potential dependence of the root of the dependence chain in the pruned dynamic slice*. This approach results in a small increase in the size of the pruned dynamic slice and also appears to be quite effective. In our example, the PDS_{max} has the size of 15/6 (*all*/*distinct*). It takes just seconds to figure out that the chain in PDS_{max} that should be expanded is S4→S5. By adding the potential dependence for the root of S4, which is S3 here, the error is reachable by just one dependence edge. In other words, with confidence analysis, the programmer is able to search for the erroneous statement by considering *potential dependences* in a *demand-driven* fashion. The programmer only asks for potential dependences after exploring the pruned dynamic slice.

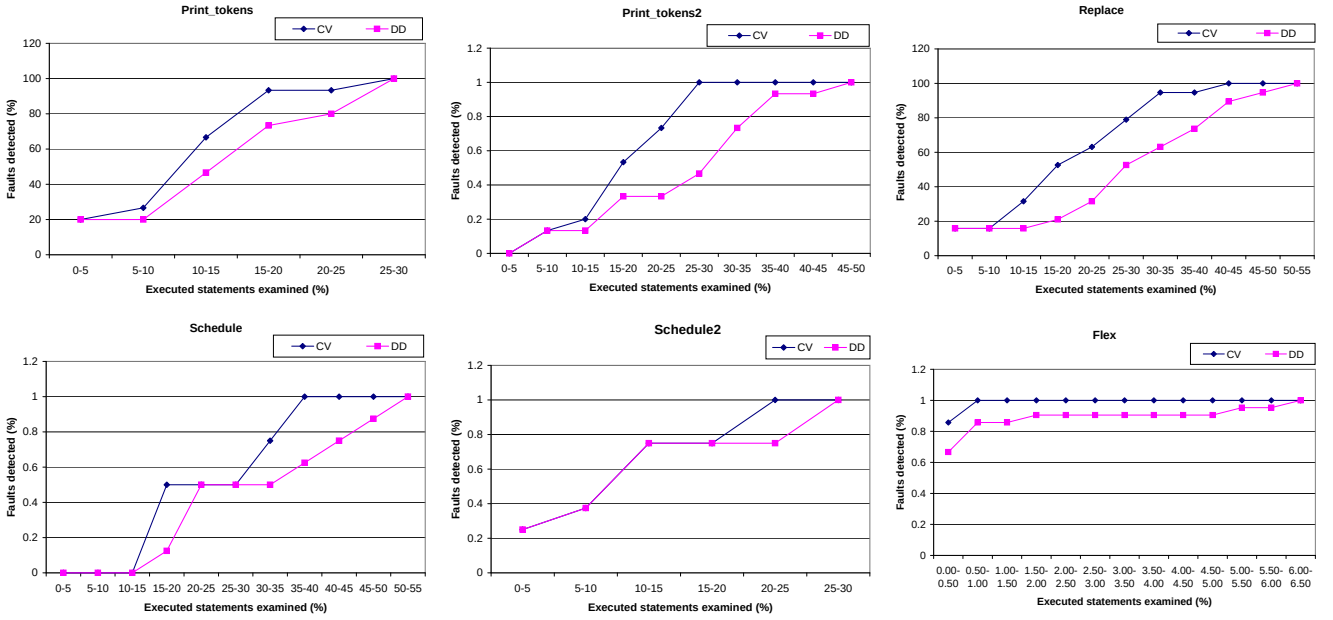


Figure 10. Locating fault by examining statements in increasing order of confidence values.

4. Related Work

Dynamic Slicing: Dynamic slicing was introduced as an aid to debugging [13, 1]. Our recent works [24, 25] have greatly reduced the space and time cost of dynamic slicing. In [27], we evaluated the effectiveness of *backward dynamic slices* in fault location. Our result showed that even though dynamic slices can capture the faulty code, identifying the faulty code from the set of statements in the slice still requires non-trivial human effort. We first narrowed the scope of potentially faulty code in [5] by, for the first time, using *forward dynamic slices* of failure-inducing input difference. We further narrowed the scope of potentially faulty code in [28] by identifying *bidirectional dynamic slices* of critical predicates. The intersection of backward, forward, and bidirectional slices yielded the *Bidirectional Chop* which was our smallest estimate of potentially faulty code. The above prior work is based upon identifying multiple kinds of *negative evidence*, i.e. program entities related to the execution of faulty code. In contrast, in this paper, we have demonstrated the use of *positive evidence* in form of correct portions of the output produced during a failing run to reduce the scope of potentially faulty code. While our prior techniques carried out coarse-grained pruning of potentially faulty code by intersecting different dynamic slices, the technique presented in this paper represents a fine-grained pruning of the backward dynamic slice. Moreover, confidence analysis is also useful in prioritizing the potentially faulty statements which was not possible using our prior techniques.

The confidence based approach we have presented analyzes both the *program state* (i.e., values of variables at various program points) and the relationships among the values (i.e., *program dependences*). Analysis of program state in conjunction with dependences is the reason why our approach is so effective. As we saw, *dynamic dicing* [3] (PDS_{min}) is not always effective. This is because while it analyzes dynamic dependences, it does not carefully consider the program state.

Delta Debugging: In a series of articles [22, 21, 20], the *delta debugging* algorithm has been developed to automatically simplify or isolate a failure-inducing input [22, 21], produce cause effect chains [20] and to link cause transitions [4] to the faulty code. In [4] delta debugging algorithm is used to analyze *program state changes* during the execution of a failed run to identify points of *cause transitions*. Code executed at the points of cause transitions is expected to be relevant to the fault. Comparing and changing memory states of C program executions at a point is difficult due to pointers [4]. In addition, to identify points of cause transitions, the above state-based analysis has to be performed at a large number of points along the failed run. Therefore, program state based analysis is difficult and time consuming for C programs [4]. In comparison our approach is inexpensive in terms of time taken.

Statistical Approaches: A number of statistical approaches that analyze program spectra of program runs for multiple inputs, including inputs corresponding to both failed and successful runs, are being employed for fault location. Harrold et al. [8] compared the spectra of passing and failing runs and found that failing runs tend to have unusual coverage spectra. Jones et al. [12] ranked each statement according to its ratio of failing tests to correct tests and used this information to assist fault location. Liblit et al. [14] describe a sampling framework and present an approach to guess and eliminate predicates to isolate a deterministic bug. For isolating nondeterministic bugs, they use statistical regression techniques to identify predicates that are highly correlated with the program failure. In contrast, Renieris and Reiss [17] focused on the difference between the failing run and a *single* passing run with similar spectra as a means to narrow down the search space for faulty code.

Our work differs from the above work in significant ways. First, it focuses on a single failed run corresponding to a single input for fault location. It is able to prioritize the potentially faulty statements using confidence analysis based upon a single run while the above techniques require multiple runs. Second, an advantage of our approach is that it provides dependence relationships between the faulty code (i.e., the cause) and the erroneous output (i.e., the effect). This information is very useful during debugging.

Other Works: Some additional works include the following. Xie et al. show that many redundancies [19] in programs correspond to hard program errors. Hangal et al. [7] identified the causes of

some programming errors in Java programs by observing violations of program invariants. In [9], we developed a technique that used a notion of path based weakest preconditions to automatically locate faulty code in a function when the precondition and postcondition of the function are available as first order predicate logic formulas.Recently hardware support for assisting in program debugging has been proposed to increase the efficiency of debugging [29, 30, 15].

5. Conclusion

We have developed a novel approach for pruning dynamic slices that exploits program state information in terms of observed values of variables in addition to the dynamic dependence information as is done traditionally in dynamic slicing. We developed a simple analysis that estimates confidence in computed values. Due to a fairly large number of executed statements that represent one-to-one mappings between an operand and the result, we are able to obtain the highest confidence value of one for a large number of computed values. Therefore, even the largest pruned dynamic slice that we obtain is significantly smaller than the conventional dynamic slice. The number of distinct statements in PDS_{max} is 1.79 to 190.57 times less than the corresponding number in DS. We show that our approach is more effective than *dynamic dicing* because pruning performed by dynamic dicing can often prune the faulty code from the dynamic slice. As mentioned earlier, it is well known that dynamic slices do not always capture the erroneous statements [6]. We described a programmer assisted demand-driven strategy for expanding the pruned dynamic slice to handle this problem. Our ongoing work is focused on automating this approach.

Acknowledgments

This work is supported by grants from Microsoft, IBM, and NSF grants CCF-0541382, CCF-0324969, and EIA-0080123 to the University of Arizona.

References

[1] Agrawal, H. and Horgan, J., "Dynamic Program Slicing," *SIGPLAN Conference on Programming Language Design and Implementation*, pages 246-256, 1990.

[2] Blume, W. and Eigenmann, R., "Symbolic Range Propagation," *International Symposium on Parallel Processing*, 1995.

[3] Chen, T.Y. and Cheung, Y.Y., "Dynamic Program Dicing," *International Conference on Software Maintenance*, pages 378-385, 1993.

[4] Cleve, H. and Zeller, A., "Locating Causes of Program Failures," *International Conf. on Software Engineering*, pages 342-351, 2005.

[5] Gupta, N., He, H., Zhang, X., and Gupta, R., "Locating Faulty Code Using Failure-Inducing Chops," *IEEE/ACM International Conference on Automated Software Engineering*, pages 263-272, Nov. 2005.

[6] Gyimothy, T., Beszedes, A., and Forgacs, I., "An Efficient Relevant Slicing Method for Debugging," *European Software Engineering Conference/ ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 303-321, 1999.

[7] Hangal, S. and Lam, M.S., "Tracking Down Software Bugs Using Automatic Anomaly Detection," *International Conference on Software Engineering*, pages 291-301, May 2002.

[8] Harrold, M.J., Rothermel, G., Sayre, K., Wu, R., and Yi, L., "An Empirical Investigation of the Relationship Between Spectra Differences and Regression Faults," *Journal of Software Testing Verification and Reliability*, 10(3):171-194, 2000.

[9] He, H. and Gupta, N., "Automated Debugging using Path-Based Weakest Preconditions," *Fundamental Approaches to Software Engineering*, Springer, LNCS 2984, pages 267-280, 2004.

[10] Hildebrandt, R. and Zeller, A., "Simplifying Failure-inducing Input," *International Symposium on Software Testing and Analysis*, 2000.

[11] Hutchins, M., Foster, H., Goradia, T., and Ostrand, T., "Experiments on the Effectiveness of Dataflow- and Controlflow-based Test Adequacy Criteria," *International Conference on Software Engineering*, pages 191-200, 1994.

[12] Jones, J.A., "Fault Localization Using Visualization of Test Information," *International Conf. on Software Engineering*, 2004.

[13] Korel, B. and Laski, J., "Dynamic Program Slicing," *Information Processing Letters*, 29(3):155-163, 1988.

[14] Liblit, B., Aiken, A., Zheng, A.X., and Jordan, M.I., "Bug Isolation via Remote Program Sampling," *SIGPLAN Conference on Programming Language Design and Implementation*, pages 141-154, 2003.

[15] Narayanasamy, S., Pokam, G., Calder, B., "BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging," *International Symp. on Computer Architecture*, pages 284-295, 2005.

[16] Pytlik, B., Renieris, M., Krishnamurthi, S., and Reiss, S., "Automated Fault Localization Using Potential Invariants," *Fifth International Workshop on Automated and Algorithmic Debugging*, Sept. 2003.

[17] Renieris, M. and Reiss, S., "Fault Localization with Nearest Neighbor Queries," *IEEE International Conference on Automated Software Engineering*, pages 30-39, 2003.

[18] Weiser, M., "Program Slicing," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 4, pages 352-357, 1982.

[19] Xie, Y. and Engler, D., "Using Redundancies to Find Errors," *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 51-60, 2002.

[20] Zeller, A., "Isolating Cause-effect Chains from Computer Programs," *SIGSOFT Symposium on Foundations of Software Engineering*, 2002.

[21] Zeller, A., "Yesterday, my program worked. Today, it does not. Why?," *European Software Engineering Conference/ ACM SIGSOFT Symp. on Foundations of Software Engineering*, pages 253-267, 1999.

[22] Zeller, A. and Hildebrandt, R., "Simplifying and Isolating Failure-inducing Input," *IEEE Transactions on Software Engineering*, Vol. 28, No. 2, pages 183-200, Feb. 2002.

[23] Zhang, X., Gupta, R., and Zhang, Y., "Precise Dynamic Slicing Algorithms," *International Conference on Software Engineering*, pages 319-329, May 2003.

[24] Zhang, X., Gupta, R., and Zhang, Y., "Effective Forward Computation of Dynamic Slices Using Reduced Ordered Binary Decision Diagrams," *International Conf. on Software Engineering*, pages 502-511, May 2004.

[25] Zhang, X. and Gupta, R., "Cost Effective Dynamic Program Slicing," *SIGPLAN Conference on Programming Language Design and Implementation*, pages 94-106, June 2004.

[26] Zhang, X. and Gupta, R., "Whole Execution Traces," *IEEE/ACM 37th International Symposium on Microarchitecture*, 2004.

[27] Zhang, X., He, H., Gupta, N., and Gupta, R., "Experimental Evaluation of Using Dynamic Slices for Fault Location," *International Symposium on Automated and Analysis-Driven Debugging*, 2005.

[28] Zhang, X., Gupta, N., and Gupta, R., "Locating Faults Through Automated Predicate Switching," *International Conference on Software Engineering*, 2006.

[29] Zhou, P., Liu, W., Long, F., Lu, S., Qin, F., Zhou, Y., Midkiff, S., and Torrelas, J., "Accmon: Automatically Detecting Memory-related Bugs via Program Counter-based Invariants," *International Symposium on Microarchitecture*, pages 269-280, 2004.

[30] Zhou, P., Qin, F., Liu, W., Zhou, Y., and Torrelas, J., "Iwatcher: Efficient Architecture Support for Software Debugging," *International Symp. on Computer Architecture*, pages 224-237, 2004.

[31] *http://www.cse.unl.edu/∼galileo/sir*

[32] *http://www.elis.ugent.be/diablo/*

[33] *http://valgrind.org/*