Experimental Evaluation of Using Dynamic Slices for Fault Location[*]

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

ABSTRACT

Dynamic slicing algorithms have been considered to aid in debugging for many years. However, as far as we know, no detailed studies on evaluating the benefits of using dynamic slicing for detecting faulty statements in programs have been carried out. We have developed a dynamic slicing framework that uses dynamic instrumentation to efficiently collect dynamic slices and reduced ordered Binary Decision Diagrams (roBDDs) to compactly store them. We have used the above framework to implement three variants of dynamic slicing algorithms including: data slicing, full slicing, and relevant slicing algorithms. We have carried out detailed experiments to evaluate these algorithms. Our results show that full slices and relevant slices can considerably reduce the subset of program statements that need to be examined to locate faulty statements. We expect that the observations presented here will enable development of new slicing based algorithms for automated debugging.

Categories and Subject Descriptors

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

General Terms

Algorithms, Measurement, Reliability, Verification

Keywords

data slice, full slice, relevant slice, debugging

1. INTRODUCTION

The concept of program slicing was first introduced by Mark Weiser [18]. He introduced program slicing as a debugging aid and gave the first *static slicing* algorithm. During program debugging, the objective of slicing is to reduce the debugging effort by focusing the attention of the user on a subset of program statements which are expected to contain faulty code. Since debugging is performed by analyzing the statements of the program when it is executed using a specific input, Korel and Laski proposed the idea of *dynamic slicing* [12]. There are two kinds of dynamic slices: backward dynamic slices and forward dynamic slices. A backward dynamic slice of a variable at a point in the execution trace includes all those executed statements which effect the value of the variable at that point. In contrast, the forward dynamic slice of a variable at a point in the execution trace includes all those executed statements that are affected by the value of the variable at that point. Note that throughout this paper we use the term dynamic slice to refer to a *backward* dynamic slice.

A dynamic slice identifies a subset of *executed statements* that is expected to contain faulty code. The goal of our work is to experimentally evaluate the usefulness of dynamic slicing in identifying a subset of statements that contain at least one faulty statement. Therefore, we consider the cases where at least one faulty statement is present in the static slice of the faulty output since we cannot expect to find the faulty statement in a dynamic slice if it is not present in the static slice of the faulty output. Specifically, we do not consider the problem of locating erroneous statements such as those missing from the static slice of the faulty output as a result of a mistake in the variable name appearing on the lhs of an assignment statement. Also we do not consider those faulty programs from which some code is completely missing.

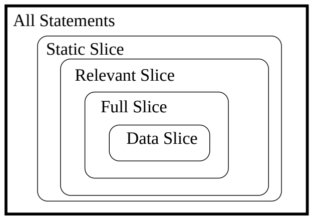
Given a faulty output value, dynamic slicing algorithms identify the subsets of executed program statements that *influenced* the computation of the faulty value. Different dynamic slicing algorithms use different notions of what they consider as *influencing*. In this paper we consider three dynamic slicing algorithms:

- *Data slicing.* Statements that directly or indirectly influence the computation of faulty output value through chains of *dynamic data dependences* are included in data slices [20].
- *Full slicing.* Statements that directly or indirectly influence the computation of faulty output value through chains of *dynamic data and/or control dependences* are included in full slices [12].
- *Relevant slicing.* While relevant slices also consider data and control dependences, in addition, they include predicates that actually did not affect the output but could have affected it had they been evaluated differently, direct data dependences of these predicates, and chains of dynamic data and control dependences of these direct data dependences [7].

The effectiveness of a given slicing algorithm in fault location

[*]This work is supported by grants from IBM, Microsoft, Intel, and NSF grants CCR-0324969, CCR-0220334, CCR-0208756, and EIA-0080123 to the Univ. of Arizona.

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?* For the class of faults considered in this paper, i.e. where the programmer has made a mistake in a predicate or an assignment statement, the faulty statement is guaranteed to be present in the static slice and relevant slice. However, it may or may not be present in the full slice or the data slice. The following relationship holds among various slices: Static Slice $\supseteq$ Relevant Slice $\supseteq$ Full Slice $\supseteq$ Data Slice.



While dynamic slicing has long been considered useful for debugging [1, 12, 2], experimental studies evaluating the effectiveness of slicing have not been carried out. The main goal of this paper is to experimentally evaluate the three dynamic slicing algorithms by considering their sizes and their ability to include faulty statements. Our results show that although data slices are small they often do not include faulty statements. Relevant slices are quite effective and only slightly larger than full slices. The experimentation is based upon a slicing tool that we have developed. This tool uses dynamic instrumentation to efficiently collect runtime information and uses the reduced ordered Binary Decision Diagrams (roBDDs) to efficiently store the collected information. Thus this tool provides a practical implementation of dynamic slicing and thus enables slicing of long program runs.

The rest of the paper is organized as follows. In section 2 we give the overview of dynamic slicing algorithms including clarifications of some issues that were not provided in other published papers. In section 3 we give the overview of our slicing tool. Section 4 presents the results of our experiments. Related work is presented in section 5 and the conclusions are given in section 6.

2. DYNAMIC SLICING ALGORITHMS

Two types of methods for computing backward dynamic slices have been proposed: *backward computation* methods [1, 20]; and *forward computation* methods [5, 21]. In backward computation methods the program dependences that are exercised during a program execution are captured and saved in the form of a dynamic dependence graph. Dynamic slices are constructed upon user's requests by backward traversal of the dynamic dependence graph. Although this approach allows computation of *all* dynamic slices of all variables at all execution points, a problem with this method is its space cost. In *forward computation* methods [5, 21] latest backward dynamic slices of all program variables are computed and maintained as sets of statements as the program executes. Advantage of this approach is that the space cost is no longer proportional to the length of execution but rather proportional to the number of variables. Therefore we decided to use forward computation method in this work.

We consider the three dynamic slicing algorithms (data, full, and relevant) because they represent different tradeoffs between slice sizes and fault location capability. Next we present an example to illustrate this point. Let us consider the program in the left hand side column of Fig. 1 – this is the correct program version. Under three different errors that we consider, we show the data slice (DS), full slice (FS), relevant slice (RS), and set of executed statements (ES). In case of Error 1, the faulty statement (13) can be found in all dynamic slices. Therefore data slice (DS) which is the smallest is the most desirable choice. One can see that DS contains far fewer statements than the number of distinct statements executed on the given input. In case of Error 2, the data slice does not contain the faulty statement (10), but it can be found in the full slice (FS). Finally in case of Error 3, the faulty statement (7) can only be found in relevant slice (RS). Thus, the different dynamic slices that we consider differ in their size and ability to capture faulty statements.

2.1 Data Slicing

Lets consider the execution of the program on an input that reveals the fault by producing an erroneous output value. Further let us assume that the presence of the faulty statement does alter the execution control flow, i.e. the set of statements executed for this input are the same whether or not the fault is present. Under these conditions, the erroneous output must have been produced by a fault in form of a computational error in one of statements whose computed value is related to the output value through a chain of dynamic data dependences. Thus, the faulty statement will be present in the data slice in this situation.

Given a statement s, let s_i denote the i^{th} execution instance of s. Let $Def[s_i]$ be the set of variables defined by s_i and $Use[s_i]$ denote the sets of variables that are read by statement execution s_i. Statement execution s_i is dynamically data dependent upon another statement execution m_n if and only if there exists a variable v such that $v \in Def[m_n]$ and $v \in Use[s_i]$. Starting from the output value, by taking the transitive closure over dynamic data dependences, we can identify the set of statements that must belong to the data slice. Because a dynamic data slice can be small and easy to understand, the faulty statement is easier to locate by examining the data slice.

Next we present the *forward computation* algorithm [5, 21] that we use to compute dynamic data slices. We use $DS[v]$ to denote the dynamic data slice for the latest definition of v. A forward computation algorithm continuously computes dynamic slices as statements are executed. Although all slices are computed, only the most recent slices of all variables are saved. After the statement execution s_i, $DS[v]$ (where $v \in Def[s_i]$) is updated to include the following: statements that belong to latest dynamic slices of variables used by s_i (i.e., variables in $Use[s_i]$) and the statement s itself. The updating of dynamic data slices following the execution of statement instance s_i is summarized below.

Algorithm 1 *Updating Data Slicing Information*

```
Procedure Update($s_i$)
1: $slice$ = {};
2: for (each use $v$ in $Use[s_i]$) do
3:     $slice$ = $slice \cup DS[v]$;
4: end for
5: for (each definition $v$ in $Def[s_i]$) do
6:     $DS[v]$ = $slice \cup \{s\}$;
7: end for
```

For example in Fig. 1, if Error 1 is introduced, the program fails on the test input shown. Forward computation of data slices for this test case is shown in Table 1. The faulty program output is -1 at statement 14_1, which is different from the correct output. For debugging we look up the data slice of z at 14_1. The value of z at this point is defined at 13_1, which is $z = x - y$. $\{x, y\}$ are the variables that are used. We can see from the Table 1 that at this point, $DS[x] = \{5\}$, and $DS[y] = \{6\}$. Therefore $DS[z] = \{13\} \cup DS[x] \cup DS[y] = \{5, 6, 13\}$. We can see the faulty statement 13 is in the data slice.

```
1.  read (a);
2.  read (n);
3.  i=0;
4.  while (i<n) {
5.      read (x);
6.      read (y);
7.      a=a/x;
8.      b=x;
9.      if (a>1)
10.         b=a-4;
11.     if (b>0)
12.         z=x+y;
        else
13.         z=x-y;
14. output (z);
15. i=i+1;
    }
```

$(Error1)$ 13. $z = x - y$ $\rightarrow$ 13. $z = x - y + 1$	$(Error2)$ 10. $b = a - 4$ $\rightarrow$ 10. $b = a - 3$	$(Error3)$ 7. $a = a/x$ $\rightarrow$ 7. $a = a/2x - 1$;
Input: $a = 2;\ n = 1;$ $x = -1;\ y = 1;$ Wrong output: $z = -1$; Correct output: $z = -2$;	Input: $a = 8;\ n = 1;$ $x = 2;\ y = 2;$ Wrong output: $z = 4$; Correct output: $z = 0$;	Input: $a = 8;\ n = 1;$ $x = 2;\ y = 2;$ Wrong output: $z = 4$; Correct output: $z = 0$;
*$DS = \{5, 6, 13\}$ $FS = \{1, 2, 3, 4, 5, 6, 8, 11, 13\}$ $RS = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 13\}$ $ES = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 13, 14, 15\}$	$DS = \{5, 6, 12\}$ *$FS = \{1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12\}$ $RS = \{1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12\}$ $ES = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 15\}$	$DS = \{5, 6, 12\}$ $FS = \{1, 2, 3, 4, 5, 6, 8, 11, 12\}$ *$RS = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12\}$ $ES = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 14, 15\}$

Figure 1: Examples of Data, Full, and Relevant Slices.

Table 1: Forward computation of data slices.

i_j	dynamic $Def[i_j]$	dynamic $Use[i_j]$	$DS[v \in Def[i_j]]$
1_1	{a}	$\emptyset$	{1}
2_1	{n}	$\emptyset$	{2}
3_1	{i}	$\emptyset$	{3}
4_1	$\emptyset$	{i,n}	n/a
5_1	{x}	$\emptyset$	{5}
6_1	{y}	$\emptyset$	{6}
7_1	{a}	{a,x}	{1,5,7}
8_1	{b}	{a}	{1,5,7,8}
9_1	$\emptyset$	{a}	n/a
11_1	$\emptyset$	{b}	n/a
13_1	{z}	{x,y}	{5,6,13}
14_1	$\emptyset$	{z}	n/a

2.2 Full Slicing

Let us consider Error 2 in Fig 1. The faulty program fails on the given input. It outputs 4 at 14_1 while the correct output value is 0. The faulty statement 10 is not in set $\{5, 6, 12\}$ which is the data slice of z at 14_1. This is because the fault does not affect value of z at 14_1 through a chain of dynamic data dependences. Instead fault in statement 10 affects the outcome of predicate at 11_1 changing the direction of the branch and thus causing statement 12 to be executed instead of statement 13. The value of z thus computed is altered. The data slice of z at 14_1 contains statement 12 which is executed by mistake but it does not contain the faulty statement 10.

Full slices correctly handle the above situation by considering control dependences. A statement s is true (false) control dependent upon a predicate p if and only if p's true (false) outcome determines whether s will be executed. This is also denoted as follows: $p^{T(F)} \in CD(s)$. Full slices are computed by taking the transitive closure over both data and control dependence edges starting from the output value. In the above example, when both types of dependences are considered, statement 10 is included in the full slice. This is because statement 12 is control dependent upon predicate 11 which is data dependent upon statement 10.

While a statement can be statically control dependent upon multiple predicates, at runtime, each execution instance of a statement is dynamically control dependent upon a single predicate. The predicate on which the execution of a statement is control dependent is found as follows. First let us assume that there are no recursive procedures. Given an execution of a statement s, prior to its execution, the most recently executed predicate p on which s is statically control dependent is found. The execution of s is dynamically control dependent upon this execution of p. Timestamps are associated with execution instances of statements in order to evaluate the above condition. Second condition is needed in presence of recursion. We discuss this point because we have not seen this issue discussed in any other dynamic slicing paper. For a given execution of statement s to be control dependent upon an execution of a predicate p, the execution instances of both must correspond to the same function invocation. This latter condition is illustrated by the example in Fig. 2.

Here a possible execution trace involving three invocations of $f(n)$ are shown. For this execution 3_1 is dynamically control dependent upon 1_2 and 3_2 is dynamically control dependent upon 1_1. As we can see, if we had simply used the first condition to determine dynamic control dependence, we would have obtained wrong results (3_1 would have been found to be dynamically control dependent upon 1_3 and 3_2 would have been found to be dynamically control dependent upon 1_2).

```
f(n) {                    1_1
1: if (n>0) {             2_1
2:   f(n-1);                  1_2
3:   S;                       2_2
   }                              1_3
}                             3_1
                          3_2
```

Figure 2: Dynamic control dependence.

In Algorithm 2, we present the forward computation algorithm for finding full slice. Here $FS[v]$ denotes the full slice for the latest definition of variable v; $timestamp$ denotes the current time; $stack$ is the current stack frame. The $stack.slice[]$ and $stack.ts[]$ are arrays allocated in the current stack frame in the same way as a local array is allocated for a function. To correctly compute dependences in the case of recursive calls, we store the timestamps of latest executions of predicates and their corresponding full slices in $stack.ts[]$ and $stack.slice[]$ respectively. This guarantees when we search for the predicate instance with the largest timestamp in $CD(i)$, we only consider those that have the same stack frame as s_i. Note that the structure of this forward computation algorithm is similar to the forward computation algorithm for data slicing. The additional statements are present to enable handling of control

Table 2: Forward computation of full slices.

s_i	dynamic $Def[s_i]$	dynamic $Use[s_i]$	$CD(s)$	cd	$stack.ts[s]$	$stack.slice[s]$	$FS[v \in Def[s_i]$
1_1	{a}	$\emptyset$	$\emptyset$	n/a	n/a	n/a	{1}
2_1	{n}	$\emptyset$	$\emptyset$	n/a	n/a	n/a	{2}
3_1	{i}	$\emptyset$	$\emptyset$	n/a	n/a	n/a	{3}
4_1	$\emptyset$	{i,n}	$\emptyset$	n/a	0	{2,3}	{2,3,4}
5_1	{x}	$\emptyset$	$\{4^T\}$	4	n/a	n/a	{2,3,4,5}
6_1	{y}	$\emptyset$	$\{4^T\}$	4	n/a	n/a	{2,3,4,6}
7_1	{a}	{a,x}	$\{4^T\}$	4	n/a	n/a	{1,2,3,4,5,7}
8_1	{b}	{a}	$\{4^T\}$	4	n/a	n/a	{1,2,3,4,5,7,8}
9_1	$\emptyset$	{a}	$\{4^T\}$	4	1	{1,2,3,4,5,7}	{1,2,3,4,5,7,9}
10_1	{b}	{a}	$\{9^T\}$	9	n/a	n/a	{1,2,3,4,5,7,9,10}
11_1	$\emptyset$	{b}	$\{4^T\}$	4	2	{1,2,3,4,5,7,9,10}	{1,2,3,4,5,7,9,10,11}
12_1	{z}	{x,y}	$\{11^T\}$	11	n/a	n/a	{1,2,3,4,5,6,7,9,10,11,12}
14_1	$\emptyset$	{z}	$\{4^T\}$	4	n/a	n/a	n/a

Algorithm 2 *Updating Full Slicing Information*

```
Procedure Update($s_i$, $stack$)
 1: $slice$ = {};
 2: for (each use $v$ in $Use[s_i]$) do
 3:     $slice$ = $slice \cup FS[v]$;
 4: end for
 5: $cd$ = the predicate in CD(s) s.t. $stack.ts[cd]$ is maximum;
 6: $slice$ = $slice \cup stack.slice[cd] \cup \{cd\}$;
 7: if ($s$ is a predicate) then
 8:     $stack.slice[s] = slice$;
 9:     $stack.ts[s]$=$timestamp$++;
10: end if
11: for (each definition $v$ in $Def[s_i]$) do
12:     $FS[v]$ = $slice \cup \{s\}$;
13: end for
```

dependences during slicing. The lines 2-4 update the $slice$ by statements that belong to latest dynamic slices of variables used by s_i. The line 5 finds the immediate control dependence of s by picking the predicate cd in $CD(s)$ that has maximum timestamp. The line 6 updates $slice$ to include all the statements in $slice(cd)$ and cd itself. If s is a predicate, the lines 8 and 9 respectively store the $slice$ and the $timestamp$ for s in the corresponding array elements in the stack frame. If s is an assignment, then the full slice for each of the variable whose definition is generated by s is assigned the $slice$ and the statement s itself. Forward computation of full slices for execution in case of Error 2 is shown in Table 2. We can see that the faulty statement (10) is in the full slice but not in the data slice of z at 14_1. At the same time we note that the full slice, which is $\{1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12\}$, is much larger than the data slice.

2.3 Relevant Slicing

Unfortunately, in some situations erroneous statements cannot even be captured by full slices. In Fig 1, if Error 3 is considered on the given input the program fails producing output of 4 instead of 0 at 14_1. The full slice of z at 14_1 is computed as $\{1, 2, 3, 4, 5, 6, 8, 11, 12\}$, the erroneous statement 7 is not in the *full slice*. This error turns value of a at 7_1 from 4 into 1 and thus predicate at 9_1 takes the wrong branch which causes 10 not to be executed while 10 should have been executed in the correct program. Because of the missing execution of 10, 9_1 would not be in the *full slice* so that the faulty execution of 7_1 would not appear either. In general, the basic reason is that some statements which should have been executed did not get executed due to the fault.

To handle the above situation a new form of dependence needs to be introduced between certain predicate outcomes and uses. Given a use u, let us define a *potentially depends* set $PD(u)$ such that the set contains members of the form that specify predicates and their outcomes (i.e., p^T or p^F). If p^T (p^F) is present in $PD(u)$, it means that if prior to the execution of u predicate p was executed, and its outcome was T (F), then while no definition corresponding to u was encountered, it could have been encountered if p had evaluated to F (T). For the above example (whose control flow graph is given in Fig. 3) this means that $9^F \in PD(b@11)$ because when the outcome of predicate 9 is F, no definition of b is encountered after execution of 9 while if 9 had evaluated to T the definition of b at 11 would be encountered.

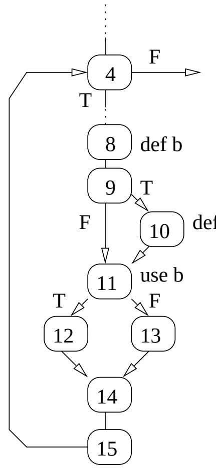


Figure 3: Control flow graph.

The potentially depends property is a static property of u which is precomputed and later used at runtime to compute relevant slices. Now let us see how the PD sets are used at runtime to compute relevant slices. When a use u is encountered at runtime we first determine the corresponding definition's execution instance from which the use gets its value. Then the execution instances of predicate outcomes indicated in $PD(u)$ that are executed before the use and after its corresponding definition are identified. Only these instances could have caused a different definition to dynamically reach the execution of use u under consideration. Thus these are also included in the slice. Let $ts(s_i)$ denote the timestamp of s_i. $LDT(v)$ denotes the latest definition's timestamp for variable v. A statement execution s_i is potentially dependent upon p_j^x (where

$x \in \{T, F\}$ and j is the instance number of the predicate), if and only if there exists $v \in Use(s_i)$ such that

$$(LDT(v) < ts(p_j^x) < ts(s_i)) \ \wedge \ p^x \in PD(v@s).$$

Next we describe the forward computation algorithm for updating relevant slices. The Algorithm 3 follows same structure as the full slicing algorithm. The changes reflect that not only data and control dependences are to be considered, but in addition, the relevant *potential dependences* are also considered. The contribution of PD dependences to the relevant slice are denoted by PS slices in the algorithm. The PS slices include potentially dependent predicates, their direct data dependences, and the relevant slices of these direct dependences. The relevant slice for the latest definition of variable v is denoted by $RS[v]$. For each use v, the lines $4-6$ in the Algorithm 3 search through all the instances of predicates in $PD(v)$ and find those that are executed between the current time and the definition time. The current $slice$ is unioned with the PSs for those selected predicate instances. Before line 8, $slice$ contains the contributions of data dependence and potential dependence to the current slice. The lines $8-9$ compute the contribution of control dependence and finally get the current relevant slice RS. When s is a predicate, a couple of things need to be done to support the control dependence and the potential dependence computation in the future. As mentioned in the full Slicing algorithm, the $stack.slice[]$ and $stack.ts[]$ are arrays allocated in the current stack frame. The line 11 in the Algorithm 3 updates the relevant slice for the latest instance of predicate s to the computed RS. The line 12 updates the timestamp which will be used later on in the comparisons at 8. Similarly, PS and the timestamp for the current predicate instance need to be stored to facilitate the future computation in line $4-6$. One thing we need to point out is that PS is updated to $slice$ but not RS because the assumption for a predicate being a potential dependent is that the branch outcome of that predicate could be wrong, which implies the fault could only contribute to the predicate outcome via data dependence or potential dependence but not control dependence. The line 18 updates the relevant slices of all the variables defined at s to RS and the statement s itself. Note that the union of PS is computed for only those instances of p^x whose execution times are between the definition time and use time of the variable which potentially depends upon p^x.

Algorithm 3 *Updating Relevant Slicing Information*

```
Procedure Update($s_i$, $stack$)
 1: $slice$ = {};
 2: for (each use $v$ in $Use[s_i]$) do
 3:     $slice = slice \cup RS[v]$;
 4:     for (each $p_j^x$ s.t. $p^x \in PD(v@i) \wedge LDT(v) < ts[p_j^x]$) do
 5:         $slice = slice \cup PS[p_j^x]$;
 6:     end for
 7: end for
 8: $cd$ = the predicate in CD(s) s.t. $stack.ts[cd]$ is maximum;
 9: $RS = slice \cup stack.slice[cd] \cup \{cd\}$;
10: if ($s$ is a predicate) then
11:     $stack.slice[s] = RS$;
12:     $stack.ts[s] = timestamp$;
13:     Let $x$ be $s$'s branch outcome.
14:     $PS[s_i^x] = slice \cup \{s_i^x\}$;
15:     $ts[s_i^x] = timestamp$;
16: end if
17: for (each definition $v$ in $Def[s_i]$) do
18:     $RS[v] = RS \cup \{s\}$ ;
19:     $LDT(v) = timestamp$;
20: end for
21: $timestamp$++;
```

3. SLICING TOOL

We have developed a dynamic slicing tool which was used to conduct the experiments described in the next section. Our tool executes `gcc` compiler generated binaries for Intel x86 and computes dynamic slices based upon forward computation algorithms described in the preceding section. Even though our tool works on binary level, the slices can be mapped back to source code level using the debugging information generated by `gcc`.

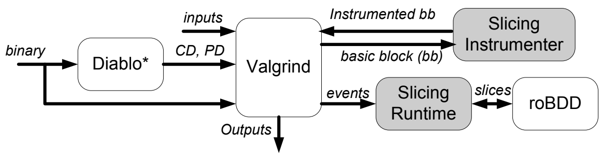


Figure 4: Slicing Infrastructure.

Fig. 4 shows the main components of the tool. The *static analysis* component of our tool computes static control dependence (CD) and potential dependence (PD) information required during full and relevant slice computations from the binary. The static analysis was implemented using the *Diablo* [24] retargetable link-time binary rewriting framework as this framework already has the capability of constructing the control flow graph from x86 binary.

The *dynamic profiling* component of our system which is based upon the *Valgrind* memory debugger and profiler [25] 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 slicing runtime were developed by us to enable collection of dynamic information and computation of dynamic slices. 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 instrumentation is dynamic in the sense that the user can enforce the expiration of any instrumented basic block such that the original basic block has to be instrumented again (i.e., instrumentation can be turned on and off as desired). Thus, we can easily turn off/on the slicing instrumentation for sake of time performance or for certain code, e.g. library code. 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.). The CD and PD information computed by the static analysis component is represented based on the virtual addresses which can be understood by Valgrind.

The forward computation algorithms maintain the latest dynamic slice for each variable/location. These dynamic slices are stored in *reduced ordered Binary Decision Diagram* (roBDD) [14] component of our system. In our previous work [21], we identified three characteristics of dynamic slices: same dynamic slices tend to *reappear* from time to time during execution, different slices tend to *share statements*, and *clusters of statements* located near each other in the program often appear in a dynamic slice. These characteristics resulted in our observation that roBDD representation of sparse sets was suitable for storing dynamic slices as it was both space and time efficient. The roBDD benefits us in the following respects. Each unique slice is presented by unique integer number in roBDD, which implies that if and only if two slices are identical, they are represented by the same integer number. The whole set of statements in the slice can be recovered from roBDD using that number. This is critical to our design because now for each variable (memory location) we only need to store one integer. Use of roBDD achieves space efficiency because roBDD is capable of removing duplicate, overlapping, and clustered sets which are exactly the characteristics of slices. Using roBDD also provides time efficiency because roBDD implementations of set operations are very efficient. More details about why and how we use roBDD can be found in [21].

We also implemented a simple debugging interface which provides limited capabilities including setting breakpoints, continuing execution, stopping after certain steps of execution, slicing on a register, slicing on a memory location, and slicing on the latest instance of a predicate.

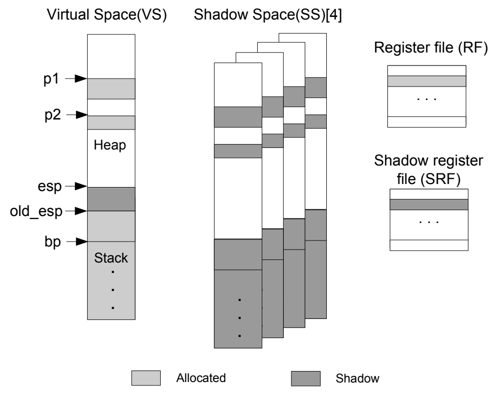
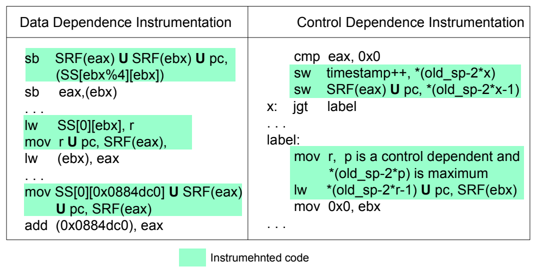


Figure 5: Storing slices.

We discuss instrumentation in more detail next. The basic idea of forward computation is that when some operation is performed on operands, the slices of source operands are fetched and unioned together with the current statement. The resulting slice is assigned to the destination operand. Although one slice can be represented as one integer, we need to store one integer for each operand which could be memory location, register, or predicate. Fig. 5 explains how we manage this. For memory, we use shadow space to store the slices. For each stack/heap byte allocated in application's virtual space, a corresponding word is allocated in the shadow space. The shadow space can be accessed using the same virtual address. As we can see in Fig. 5, we need four shadow spaces, one for each byte in a word of the virtual space. For register, we use the shadow register file to store the slices for registers. For predicate, as we described earlier, a predicate and its control dependent statement must correspond to the same function invocation. Therefore we allocate some space in the stack frame to store the slices for predicates. This can be done by shifting the stack pointer from its original position old_esp to esp as shown in Fig. 5.


Data Dependence Instrumentation
Control Dependence Instrumentation
Instrumehnted code

```
sb    SRF(eax) U SRF(ebx) U pc,
      (SS[ebx%4][ebx])
sb     eax,(ebx)
. . .
lw     SS[0][ebx], r
mov  r U pc, SRF(eax),
lw     (ebx), eax
. . .
mov SS[0][0x0884dc0] U SRF(eax)
      U pc, SRF(eax)
add  (0x0884dc0), eax
```

```
      cmp  eax, 0x0
      sw    timestamp++, *(old_sp-2*x)
      sw    SRF(eax) U pc, *(old_sp-2*x-1)
x:    jgt    label
. . .
label:
      mov  r,  p is a control dependent and
            *(old_sp-2*p) is maximum
      lw     *(old_sp-2*r-1) U pc, SRF(ebx)
      mov  0x0, ebx
. . .
```

Figure 6: Instrumentation example.

Fig. 6 shows examples for instrumentation. Left hand side is the instrumentation for data slice computation. We can see for the sb instruction, the operands are eax and ebx. The instrumentation first takes the slices for these registers from the shadow register file SRF, and then computes the union of these two slices with current pc. The computed slice is stored in the shadow space indexed by the same virtual address which indicates the corresponding memory byte in application's virtual space. Right hand side shows how the instrumentation for predicate looks like. The variable x denotes the index of the predicate, the first predicate in pc order has index of 0 and so on. After the cmp instruction gets executed, the pre-allocated shadow space in the stack frame is updated with the current timestamp and slice. These information are later used when the instruction at $label$ turns out to be control dependent on the predicate.

4. EXPERIMENTAL EVALUATION

Table 3 shows the benchmarks we used for our experimentation. The first five programs are from the Siemens suite [9, 23] and the remaining three are medium sized linux utility programs from the same code repository [23] as the Siemens suite. We exclude the program *tcas* and *tot_info* from the Siemens suite because *tcas* is too small and our tool currently does not support floating point programs like *tot_info*. The faulty programs (Num. Versions) and the test cases (Num. Tests) are provided at the website [23]. Not all the provided faulty programs are used in our experiments. Some faulty programs produce no output and thus it is unclear how a proper slicing criterion should be defined. There are also a certain number of faults which are simple code omissions. Therefore, the faulty omitted statement will not be present in any slice. We simply exclude those versions. For the rest of the versions, the slicing criteria we choose are either the points at which either the *first incorrect output is produced* or at which a *segmentation fault* occurs. In our experiments, each faulty version of the program has exactly one fault injected.

Table 3: Overview of benchmark programs.

Program	Description	Num. Versions	LOC	Num. Tests
`print_tokens`	lexical analyzer	5	565	4072
`print_tokens2`	lexical analyzer	8	510	4057
`replace`	pattern replacement	19	563	5542
`schedule`	priority scheduler	6	412	2627
`schedule2`	priority scheduler	3	307	2683
`gzip2`	compression utility	1	7199	217
`gzip5`	compression utility	2	8009	217
`flex`	lexical analyzer generator	5	12418	525

We instrument the faulty programs in minor ways when we run them through our tool. For example, we intercept almost all the output functions like *fprintf, printf, fputc* etc. and redirect them to our customized functions in which it is more convenient for our tool to find slicing criterion.

4.1 Data, Full, and Relevant Slices

The first experiment we carried out compares the three basic slicing algorithms that we have discussed. We compare these algorithms both from the perspective of slice sizes and whether the slices include faulty statements. For each faulty program, we take all the test cases which produce wrong outputs, and then for each test case we compute *data slice* (DS), *full slice* (FS), and *relevant slice* (RS) for the first wrong output. We classified all the errors in the faulty versions of the programs into two categories, *Assignment Faults* and *Predicate Faults*. If the error was in a definition, it was classified as an assignment fault, otherwise it was classified as a predicate fault. We did not compute DS for predicate faults because DSs never contain branch predicates. The averages over all the failed test cases for each faulty program are presented in Table 4. In this table, the column labeled *Fail Cases* shows the number of failed test cases for the faulty version corresponding to a given row. The column labeled *Avg. Exec.* shows the average number of distinct statements that are executed at least once for the failed test cases. The columns labeled *Avg. DS*, *Avg. FS* and *Avg. RS* respectively show the average number of statements in DS, FS and RS. Similarly, the columns labeled *In DS*, *In FS* and *In RS* respectively show the *fraction* of data slices and full slices that contained the faulty statement. The results are also summarized according to their *fault types* for each benchmark in Table 5 and Table 6.

Note that since there was only one faulty statement in each faulty version, the relevant slice always contained the faulty statement. However, in general when there are multiple faulty statements in a program, a relevant slice may contain only some of the faulty statements. This is because a relevant slice is computed by considering the contributions of potentially dependent predicates that were *actually executed* for an input. To illustrate this, let us consider the simple code segment below.

```
1:   read(m,n);
2:   x:=2*m;
3:   w:=5;
4:   a:=10;
5:   if (w > n)
6:        b:=15;
7:   else
8:        if (x >5)
9:             a:=20;
10:       else
11:            b:=25;
12:       endif
13:  endif
14:  output(a);
```

Let us consider the execution of above code segment for the input (m=1, n=2). The lines 1, 2, 3, 4, 5, 6, 14 in the code segment above are executed for the above input. Let us assume statements on lines 2 and 3 are faulty. The full dynamic slice of variable a at line 14 for this input will consist of only statement at line 4. However, for computing the relevant slice for this input, the predicate at line 5 will be identified as a potential dependence since if it had evaluated to false outcome, the value of variable a output at line 14 could be affected. Therefore, the relevant slice will contain the statements at lines 5, 3 and 1 in addition to the statement at line 4 that is in the full dynamic slice. Thus, the relevant slice of variable a at line 14 for this input will contain only one faulty statement i.e., the statement at line 3. The other faulty statement (at line 2) will be missing from this relevant slice. This is because the predicate at line 8 was not identified as a potential dependence since it was *not executed*.

However, if we consider an input (m=1, n=6) for which the predicate at line 5 evaluates to false, then the predicate at line 8 will be executed and it will evaluate to its false outcome. Although the full dynamic slice for this input will again consist of only the statement at line 4, the predicate at line 8 will be identified as a potential dependence because if it evaluated to true, the value of variable a at line 14 would be affected. Therefore, the statement at line 2 will be included in this relevant slice since predicate at line 8 is data dependent on it. However, the faulty statement in line 3 will not be included in the relevant slice because the direct control dependences of the potentially dependent predicates are not included in the relevant slice. Thus, in general if multiple erroneous statements are present in the static slice of an output variable, a relevant slice for a given input may contain only a subset of the faulty statements.

Next we describe our experimental results and compare the effectiveness of data, full and relevant slices in capturing faulty statements. We first discuss our observations from the experiments with capturing assignment faults.

Assignment Faults: From Table 5, we can see the average data slice sizes (*Avg. DS*) are very small. They are only $0.5 - 28\%$ of the average executed (*Avg. Exec.*) statements. Data slices do not always capture the faults as can be seen from the column labeled *In DS*. In some benchmarks (e.g., $replace$), none of data slices we computed contained the faulty statements. On the other hand, we do observe that for $flex$, the largest benchmark we have, DS successfully located all the faulty statements. For this program DS has the average size of around 8 statements, which is only 0.5% of the executed statements.

For comparison, we also compute FS and RS for assignment faults. Although the size of FS increased by the factors of $1.74 - 8.39$ when compared with the size of DS, FS covered almost all the faulty statements including those missed by DS except for benchmark $schedule2$. Compared to FS, although the size of RS increased by a factor of $1.04 - 2.59$, RS successfully contained all of the faulty statements. For most of the benchmarks, the inflations in size of RS when compared to FS are small (many of them are less than 10%) except $flex$. However, when we compare the size of RS to the executed statements (*Avg. Exec.*) for $flex$, it has just 12% of the executed statements.

Predicate Faults. For this type of faults, we compare the performance of FS and RS in terms of their sizes and how often they contained the faulty statement. We can see from the Table 6 that most of the time, FS was able to capture the faulty statement. The FS contained $4.3 - 61\%$ of the executed statements. Those faulty statements which were missed from FS were captured by RS. Compared to FS, RS are $2 - 181\%$ larger in size. However, the size of RS increased by less than 10% for many benchmarks. In addition, RS contained only $12 - 62\%$ of the executed statements as compared to FS that contained $4.3 - 61\%$ of the executed statements.

Overall, all the slicing methods including DS, FS and RS are quite effective in reducing the executed set of statements to a much smaller faulty statement candidate set. In our experiments DS was found to be very small in size and effective in containing assignment faults for large programs ($gzip5$ and $flex$). The FS showed a very high chance to capture both types of faults. Its size was normally much larger than DS's even though it was still a small subset of the executed statements. The RS was very effective in capturing faults and its size was comparable to that of FS's in most of the cases. For benchmark $flex$, out of the 1500 executed statements, the DS size is 8, the FS size is 65 and the RS size is 180 on average. This strongly supports that slicing is very effective in helping users focus their attention during debugging.

4.2 Searching for a faulty statement in Data, Full and Relevant Slices

A slice provides a fault candidate set that the programmer must examine to identify the faulty statement. Therefore smaller the set of statements that the user has to examine the better it is. Although data slices are small, our experiments show that very often the

Table 4: Comparison of Data, Full, and Relevant Slicing.

Program	Fault Type	Version	Fail Cases	Avg. Exec.	Avg. DS	In DS	Avg. FS	In FS	Avg. RS	In RS
print_tokens	Assignment	v4	28	116	26	0.00	74	1.00	77	1.0
		v6	186	119	28	0.00	76	1.00	78	1.0
	Predicate	v1	6	124	-	-	86	1.00	88	1.0
		v2	48	127	-	-	74	1.00	75	1.0
		v7	28	149	-	-	82	1.00	85	1.0
print_tokens2	Assignment	v4	332	153	18	0.00	71	0.91	75	1.0
		v5	173	134	18	0.00	70	1.00	72	1.0
	Predicate	v3	33	149	-	-	86	1.00	87	1.0
		v6	518	145	-	-	73	1.00	78	1.0
		v7	207	149	-	-	69	0.95	75	1.0
		v8	256	153	-	-	67	0.94	71	1.0
		v9	56	161	-	-	74	0.83	80	1.0
		v10	173	116	-	-	60	1.00	64	1.0
replace	Assignment	v12	301	89	7	0.00	53	1.00	56	1.0
		v15	63	102	15	0.00	50	1.00	54	1.0
	Predicate	v1	70	168	-	-	100	0.81	122	1.0
		v2	39	167	-	-	82	0.62	111	1.0
		v3	131	185	-	-	108	1.00	122	1.0
		v5	272	186	-	-	89	0.71	120	1.0
		v6	97	184	-	-	113	1.00	120	1.0
		v7	85	59	-	-	35	1.00	36	1.0
		v8	55	157	-	-	49	0.00	97	1.0
		v9	31	158	-	-	54	0.23	96	1.0
		v10	24	167	-	-	75	0.52	103	1.0
		v11	31	158	-	-	54	0.24	96	1.0
		v14	138	181	-	-	106	1.00	115	1.0
		v16	84	59	-	-	35	1.00	36	1.0
		v18	211	180	-	-	101	1.00	116	1.0
		v21	3	160	-	-	84	1.00	80	1.0
		v23	23	176	-	-	78	0.58	99	1.0
		v25	4	179	-	-	106	1.00	113	1.0
		v26	128	200	-	-	90	0.55	121	1.0
schedule	Assignment	v1	4	85	24	1.00	40	1.00	42	1.0
		v2	210	146	42	0.00	75	0.49	81	1.0
		v3	159	142	43	0.74	73	0.81	80	1.0
		v6	4	85	24	1.00	40	1.00	42	1.0
		v7	27	142	36	0.00	66	1.00	72	1.0
	Predicate	v4	294	144	-	-	70	0.57	78	1.0
schedule2	Assignment	v5	32	120	17	1.00	40	1.00	52	1.0
		v6	7	109	18	0.00	34	0.00	55	1.0
	Predicate	v7	31	126	-	-	46	1.00	60	1.0
gzip2	Predicate	v1	170	541	57	0.00	286	0.21	307	1.0
gzip5	Assignment	v1	9	272	36	1.00	74	1.00	105	1.0
	Predicate	v10	169	517	-	-	256	0.18	274	1.0
flex	Assignment	v4	74	1327	8	1.00	67	1.00	184	1.0
		v5	193	1562	8	1.00	67	1.00	186	1.0
		v17	278	1546	8	1.00	77	1.00	183	1.0
		v18	160	1607	9	1.00	66	1.00	165	1.0
	Predicate	v11	356	1467	-	-	64	1.00	180	1.0

Table 5: Comparison of Data, Full, and Relevant Slicing for Assignment Faults.

Program	Avg. Exec.	Avg. DS	In DS	Avg. FS	In FS	Avg. RS	In RS	DS/Exec.	FS/DS	RS/FS	RS/Exec.
print_tokens	117.5	27	0	75	1	80.6	1.0	0.23	2.78	1.07	0.68
print_tokens2	143.5	18	0	70.5	0.96	73.5	1.0	0.13	3.92	1.04	0.51
replace	95.5	11	0	51.5	1	55	1.0	0.12	4.68	1.07	0.58
schedule	120	33.8	0.74	58.8	0.86	63.4	1.0	0.28	1.74	1.08	0.53
schedule2	114.5	17.5	0.5	37	0.5	53.5	1.0	0.15	2.11	1.45	0.47
gzip2	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
gzip5	272	36	1	74	1	105	1.0	0.13	2.06	1.42	0.38
flex	1510.5	8.25	1	69.3	1	179.5	1.0	0.0054	8.39	2.59	0.12

Table 6: Comparison of Data, Full, and Relevant Slicing for Predicate Faults.

Program	Avg. Exec.	Avg. FS	In FS	Avg. RS	In RS	FS/Exec.	RS/FS	RS/Exec.
print_tokens	133.3	80.7	1	82.7	1.0	0.61	1.02	0.62
print_tokens2	145.5	71.5	0.95	75.8	1.0	0.49	1.06	0.52
replace	160.2	79.9	0.72	100.2	1.0	0.50	1.25	0.62
schedule	144	70	0.57	78	1.0	0.49	1.11	0.54
schedule2	126	46	1	60	1.0	0.36	1.30	0.48
gzip2	541	286	0.21	307	1.0	0.53	1.07	0.57
gzip5	517	256	0.18	274	1.0	0.50	1.07	0.53
flex	1467	64	1	180	1.0	0.043	2.81	0.12

faulty statement is not present in data slices. Thus, the user would need to examine larger full and/or relevant slices. Even though the full slices and relevant slices are small compared to the set of executed statements, it is still quite a lot of work to examine all of the statements in those slices. Therefore, to get an estimate of the number of the statements a developer will have to examine in a slice, we took the first test case which produced wrong output for each faulty program and computed the full slice for the wrong output.

In Table 7, the column labeled EFS (Explore Full Slice) shows how many statements were present in the full slice starting from the wrong output until we reached the faulty statement. The columns labeled DS and FS respectively show the number of statements in the data slice and the full slice. We excluded $Flex$ from this experiment because in most of its versions, the faulty statement was captured by the data slice which contained very few statements.

We can see from the Table 7 that the number of statements that were present in the full slice between the faulty statement and wrong output is 1.19 to 3.33 times larger than the size of data slices and they are only 32% to 71% of the statements in the full slices.

In practice, a developer may examine the statements in a slice by traversing along the data and control dependence edges from the point where a faulty output was produced. Clearly the dependence distance of the faulty statement from the point where the incorrect output is produced will be smaller the number of statements present in the slice between these two points. Therefore, in practice a developer may need to examine fewer % of statements in the full slices than shown in the Table 7 in order to locate the faulty statement. This suggests that in practice, the dynamic slices can significantly reduce the effort to search for a faulty statement.

Table 7: Exploring slices

Program	$Avg.$ DS	$Avg.$ EFS	$\frac{EFS}{DS}$	$Avg.$ FS	$\frac{EFS}{FS}$
`print_tokens`	20	35	1.75	74	0.47
`print_tokens2`	14	24	1.71	73	0.32
`replace`	9	30	3.33	86	0.35
`schedule`	31	37	1.19	52	0.71
`schedule2`	22	28	1.27	48	0.57
`gzip2`	54	94	1.74	192	0.49
`gzip5`	61	124	2.03	197	0.63

4.3 Performance

In this experiment, we measure the average timing and space overhead of dynamic slicing for the runs of $gzip$ and $flex$. We do not present this data for Siemens suite programs as they are smaller and hence most of the time and space cost are spent during the start-up and shut-down procedures of our slicing tool.

From Table 8, we can see that the *data slicing* algorithm is the fastest and the *relevant slicing* is the slowest. The slow down factor (SD) varies a lot from $gzip$ to $flex$. The slow down for $gzip$ is very high, this is because the uninstrumented runs terminate within $0.005sec$, which makes the $Valgrind$ start-up time very significant. For $flex$, whose uninstrumented runs take several seconds, the slow down for our tool is lower. We can also see the memory consumption for our tool does not vary too much. One explanation could be the memory used by $Valgrind$ is the dominant factor which tends to be constant for different benchmarks and different runs.

Table 8: Performance

Program	DS		FS		RS	
	SD	Mem. (MB)	SD	Mem. (MB)	SD	Mem. (MB)
gzip2	1040	221	1172	222	1624	228
gzip5	827	220	948	222	1428	227
flex	59	221	77	222	102	228

5. RELATED WORK

In [7], Agrawal et al. first proposed the concept of *relevant slicing*. They expanded the *Dynamic Dependence Graph* by introducing dependence edges between predicates and the statements which potentially depend on those predicates. They suggested using relevant slicing in incremental regression testing. Gyimothy et al. presented a forward computation algorithm for computing *relevant slices* in [7]. Even though they suggested using relevant slices for debugging, they did not experimentally evaluate the effectiveness of relevant slicing for debugging. Wang et al. also talked about how to compute *relevant slices* in [17]. They followed a method similar to Agrawal's by considering potential dependence edges. The difference is that instead of constructing the whole *DDG*, they maintain slices, which is basically a partial *DDG*, by backward traversal of a compressed java bytecode trace. They do not evaluate relevant slicing either.

Dynamic slicing was introduced as an aid to debugging [2, 11, 13, 4, 15]. Agrawal et al. [4] proposed subtracting a single correct execution trace from a single failed execution trace. In [15], Pan and Spafford presented a family of heuristics for fault localization using dynamic slicing. Compared to these previous works, we are the first one to compare the effectiveness of different dynamic slicing algorithms in fault location.

A lot of interesting research other than dynamic slicing have been carried on in fault location. Zeller has presented a series techniques [8, 19, 6] from isolating the critical input to isolating cost-effect chains in both space and time. The basic idea is to find the specific part of the *input/program state* which is critical to the program failure by minimizing the difference between the *input/program state* leading to a passing run and that leading to a failing run. We believe our technique can be combined with Zeller's technique in many aspects, for instance, the isolated *causes* are perfect slicing criteria starting from which dynamic slicing may produce a much smaller fault candidate set than from the failure point. Renieris and Reiss [16] presented technique that selects the single passing run that most resembles to the failing run and reports the difference between these two runs. Jones [10] presented a technique that uses software visualization to assist fault location. Their technique provides a ranking of each statement according to its ratio of failing tests to correct tests.

6. CONCLUSIONS

The development of dynamic slicing was motivated by the problem of locating the faulty code when an execution of a program fails. There has been a lot of research in developing algorithms for computing different types of dynamic slices. The contribution of this paper is to present an experimental evaluation of three different types of dynamic slices for the benefit of using them to locate a faulty statement in a program. We have presented experimental studies for several large programs. In our experiments, data slices were found to be much smaller than full slices and relevant slices. However, they captured very few faulty statements on average. The full slices were significantly bigger than data slice and captured much larger number of faulty statements than data slice. Interestingly, we found the relevant slices performed best in capturing faulty statements. The relevant slices were found to be only slightly larger than full slices, however they captured all the faulty statements in our experiments.

Acknowledgements

We would like to thank Dr. Gregg Rothermel, Dept. of Computer Science and Engineering, University of Nebraska, Lincoln, for providing the programs with the associated faulty versions and test suites.

7. REFERENCES

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

[2] H. Agrawal, R. DeMillo, and E. Spafford, "Debugging with Dynamic Slicing and Backtracking," *Software Practice and Experience*, Vol. 23, No. 6, pages 589-616, 1993.

[3] H. Agrawal, J.R. Horgan, E.W. , and S.A. London, "Incremental Regression Testing", *IEEE Conference on Software Maintenance*, pages 348-357, Montreal, Canada, 1993.

[4] H. Agrawal, J. Horgan, S. London, and W. Wong, "Fault Localization Using Execution Slices and Dataflow Tests," *6th IEEE International Symposium on Software Reliability Engineering*, pages 143-151, 1995.

[5] A. Beszedes, T. Gergely, Z.M. Szabo, J. Csirik, and T. Gyimothy, "Dynamic Slicing Method for Maintenance of Large C Programs," *5th European Conference on Software Maintenance and Reengineering*, pages 105-113, March 2001.

[6] H. Cleve and Andreas Zeller, "Locating Causes of Program Failures", *27th International Conference on Software Engineering*, pages 342-351, 2005.

[7] T. Gyimothy, A. Beszedes, I. Forgacs, "An Efficient Relevant Slicing Method for Debugging", *7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 303-321, Toulouse, France, 1999.

[8] R. Hildebrandt and A. Zeller, "Simplifying Failure-inducing Input", *International Symposium on Software Testing and Analysis*, pages 135-145,2000.

[9] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the Effectiveness of Dataflow- and Controlflow-based Test Adequacy Criteria," *16th International Conference on Software Engineering*, pages 191-200, 1994.

[10] J.A. Jones, "Fault Localization Using Visualization of Test Information", *26th International Conference on Software Engineering*, page 54-56,2004.

[11] M. Kamkar, "Interprocedural Dynamic Slicing with Applications to Debugging and Testing," *PhD Thesis*, Linkoping University, 1993.

[12] B. Korel and J. Laski, "Dynamic Program Slicing," *Information Processing Letters*, Vol. 29, No. 3, pages 155-163, 1988.

[13] B. Korel and J. Rilling, "Application of Dynamic Slicing in Program Debugging," *3rd International Workshop on Automatic Debugging*, pages 43-58, Linkoping, Sweden, 1997.

[14] J. Lin-Nielsen. "BuDDy, A Binary Decision Diagram Package," Department of Information Technology, Technical University of Denmark, *http://www.itu.dk/research/buddy/*.

[15] H. Pan and E. H. Spafford, "Heuristics for Automatic Localization of Software Faults", *Technical Report SERC-TR-116-P*, Purdue University, 1992.

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

[17] T. Wang and A. Roychoudhury, "Using Compressed Bytecode Traces for Slicing Java Programs", *26th International Conference on Software Engineering*, pages 512-521, Edinburgh, Scotland, UK, 2004.

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

[19] A. Zeller, "Isolating Cause-effect Chains from Computer Programs", *10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 1-10, Charleston, South Carolina, 2002.

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

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

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

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

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

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