Accurately Choosing Execution Runs for Software Fault Localization

Liang Guo, Abhik Roychoudhury, and Tao Wang

School of Computing, National University of Singapore, Singapore 117543
{guol, abhik, wangtao}@comp.nus.edu.sg

Abstract. Software fault localization involves locating the exact cause of error for a "failing" execution run – a run which exhibits an unexpected behavior. Given such a failing run, fault localization often proceeds by comparing the failing run with a "successful" run, that is, a run which does not exhibit the unexpected behavior. One important issue here is the choice of the successful run for such a comparison. In this paper, we propose a control flow based difference metric for this purpose. The difference metric takes into account the sequence of statement instances (and not just the set of these instances) executed in the two runs, by locating branch instances with similar contexts but different outcomes in the failing and the successful runs. Given a failing run π_f and a pool of successful runs S, we choose the successful run π_s from S whose execution trace is closest to π_f in terms of the difference metric. A bug report is then generated by returning the difference between π_f and π_s. We conduct detailed experiments to compare our approach with previously proposed difference metrics. In particular, we evaluate our approach in terms of (a) effectiveness of bug report for locating the bug, (b) size of bug report and (c) size of successful run pool required to make a decent choice of successful run.

Keywords: Programming tools, Debugging.

1 Introduction

Debugging is an important program development activity. In the past few years, substantial research has been conducted to improve debugging tools by identifying the error cause of an observable error with higher degree of automation [3, 7, 11, 12, 13, 18]. These fault localization approaches compare the failing execution run, which exhibits the observable error, with one that does not. Most of the research in this topic has focused on how to compare the successful and failing execution runs. In this paper, we present a control flow based difference metric to choose a successful run from a pool for such a comparison; the pool of successful program runs could be constructed by picking successful runs from a test-suite of program inputs. Our difference metric measures "similarity" between execution runs of a program. Given a failing run π_f and a pool of successful

A. Mycroft and A. Zeller (Eds.): CC 2006, LNCS 3923, pp. 80–95, 2006.

```
1.  while (lin[i] != ENDSTR) {
2.     m=...
3.     if (m >= 0) {
4.        ...
5.        lastm = m;
6.     }
7.     if ((m == -1) || (m == i)) {
8.        ...
9.        i = i + 1;
10.    }
11.    else
12.       i = m;
13. }
14. ...
```

Fig. 1. An example program fragment

runs S, we select the most similar successful run $\pi_s \in S$ in terms of the difference metric, and generate a bug report by returning the difference between π_f and π_s.

Our difference metric considers branch instances with similar contexts but different outcomes in two execution runs, because these branch instances may be related to the cause of error. When these branch instances are evaluated differently from the failing run, certain faulty statements may not be executed — leading to disappearance of the observable error in the successful run. Consider the program fragment (from a faulty version of `replace` program in the Siemens benchmark Suite [6, 14] — simplified here for illustration) in Figure 1, where the bug fix lies in strengthening the condition in line 3 to `if ((m >= 0) && (lastm != m))`. This piece of code changes all substrings s_1 in string `lin` matching a pattern to another substring s_2, where variable `i` represents the index to the first un-processed character in string `lin`, variable `m` represents the index to the end of a matched substring s_1 in string `lin`, and variable `lastm` records variable `m` in last loop iterations. At the ith iteration, if variable `m` is not changed at line 2, line 3 is wrongly evaluated to true, and substring s_2 is wrongly returned as output, deemed by programmer as an observable "error". The execution of the ith iteration of this failing run π_f could follow path $\langle 1, 2, 3, 4, 5, 7, 8, 9\rangle$. In this case, a successful run π_s whose ith iteration follows path $\langle 1, 2, 3, 7, 8, 9\rangle$ can be useful for error localization. By comparing π_f with π_s, we see that only the branch at line 3 is evaluated differently. Indeed this is the erroneous statement in this example, and was pinpointed by our method in the experiment. For programs whose erroneous statement is not a branch, our method will try to report the nearest branch for locating the error.

Summary of Results. The main results of this paper are as follows. We propose a control-flow based difference metric to compare execution runs (*i.e.* data flow in the runs is not taken into account). We take the view that the *difference* between two runs can be summarized by the *sequence* of comparable branch statement

instances which are evaluated differently in the two runs. This difference metric is used to choose a successful run from a pool of successful runs for automated debugging. We return as *bug report* the branch statements whose instances (1) have similar contexts, and (2) are evaluated differently in the failing run and the selected successful run. We experimentally evaluate the quality of our bug report, the volume of our bug report, and the impact of successful run pool size on the quality of our bug report. We also share some experience in using our method for debugging real-life programs.

2 Related Work

In this section, we discuss work on localizing software errors. There have been a lot of techniques [1, 3, 7, 10, 11, 12, 13, 18] proposed for automatic program error localization by comparing successful and failing runs of the buggy program. These techniques compare different characteristics of execution runs, *e.g.* acyclic paths [13], potential invariants [11], executed statements [1, 7, 15], basic block profiling [12], program states [3, 18], predicates [10] or return value of methods [9]. Unlike our method, most of these works focus on how to compare successful and failing execution runs to generate accurate bug reports.

The focus of our method is to choose a successful run from a given pool of successful runs, provided we have access to the failing run. In other words, we do not (semi)-automatically generate the successful run. Generating a successful run (and a corresponding input) close to a given failing run has been studied in various papers [2, 5, 19], including our past work [17].

Our difference metric bears similarities to the notion of proximity between runs proposed by Zeller et al. in [3, 18]. Their approach compares program states with similar contexts for fault localization at some control locations. Through a series of binary search over the program state and re-executing (part of) the program from "mixed" states, a set of variables which may be responsible for the bug are mined and reported. However, these "mixed" states may be infeasible. Furthermore, it may be quite costly to compare program states and to re-execute the program several times.

The work of Renieris and Reiss [12] is related to ours. They have demonstrated through empirical evidence that the successful run which is "closest" to the failing run can be more helpful for error localization than a randomly selected successful run. However, [12] measures the proximity of two runs by comparing the *set* of basic blocks[1] executed in each run. Thus, they cannot distinguish between runs which execute exactly the same statements but in different order — consider the program `for (....){ if (...) S1 else S2 }` and the two execution runs $\langle$`S1`, `S2`$\rangle$, $\langle$`S2`, `S1`$\rangle$. We consider the *sequence* of statements executed in each run for determining proximity between two runs. Detailed experiments comparing our method with [12] are reported in Section 5.

[1] Actually a sorted sequence of the basic blocks based on execution counts is used; this is different from the execution sequence of the basic blocks in the failing run.

3 Measuring Difference Between Execution Runs

We elaborate on the difference metric used for comparing execution runs in this section. We consider each execution run of a program to be a sequence of *events* $\langle e_0, e_1, ..., e_{n-1} \rangle$ where e_i refers to the ith event during execution. *Each event e_i represents an execution instance of a line number in the program*; the program statement corresponding to this line number is denoted as $stmt(e_i)$. To distinguish events from different execution runs, we denote the ith event in an execution run π as e_i^π, that is, the execution run appears as a superscript. We will drop the superscript when it is obvious from the context.

Our difference metric measures the difference between two execution runs π and π' of a program, by comparing behaviors of "corresponding" branch statement instances from π and π'. The branch statement instances with differing outcomes in π, π' are captured in $diff(\pi, \pi')$ – the difference between execution run π and execution run π'. In order to find out "corresponding" branch instances, we have defined a notion of *alignment* to relate statement instances of two execution runs. Our alignment is based on *dynamic control dependence*. Given an execution run π of a program, an event e_i^π is *dynamically control dependent* on another event e_j^π if e_j^π is the last event before e_i^π in π where $stmt(e_i^\pi)$ is statically control dependent [4] on $stmt(e_j^\pi)$. Note that any method entry event is dynamically control dependent on the corresponding method invocation event. We use the notation $dep(e_i^\pi, \pi)$ to denote the event on which e_i^π is dynamically control dependent in run π. We now present our definition of event alignment.

Definition 1 (Alignment). *For any pair of event e in run π and event e' in run π', we define $align(e, e') = true$ (e and e' are aligned) iff.*

1. $stmt(e) = stmt(e')$, *and*
2. *either e, e' are the first events appearing in π, π' or* $align(dep(e, \pi), dep(e', \pi')) = true$.

When a branch event e_i^π cannot be aligned with any event from the execution π', this should only affect alignments of events in π which are transitively dynamically control dependent on e_i^π. In addition, the ith iteration of a loop in the execution π will be aligned with the ith iteration of the same loop in the execution π', in order to properly compare events from different loop iterations.

A simple illustration of alignment appears in Figure 2; here π, π' and π'' represent three execution runs of the program segment in Figure 1 (page 81). In Figure 2, events along the same horizontal line are aligned. From this example, we can see that events in the ith loop iteration in run π are aligned with events in the ith loop iteration in run π'.

According to the notion of alignment presented in Definition 1, for any event e in π there exists *at most* one event e' in π' such that $align(e, e') = true$. The difference between π and π' (denoted $diff(\pi, \pi')$) captures all branch event occurrences in π which (i) can be aligned to an event in π' and (ii) have different outcomes in π and π'. Formally, the difference between two execution runs can be defined as follows.

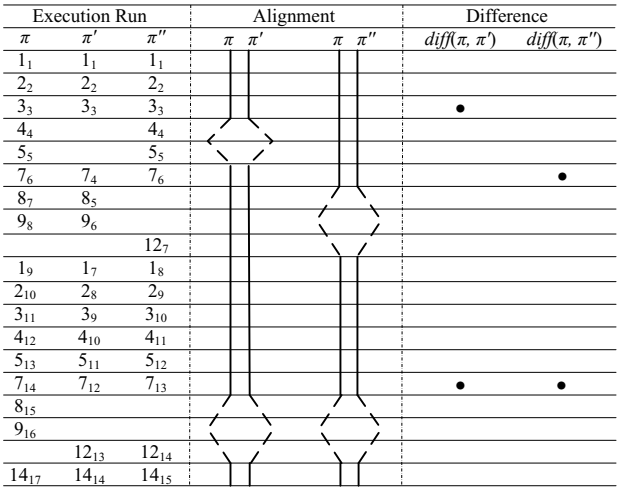


Fig. 2. Example to illustrate alignments and difference metrics. The first three columns show the event sequences of three execution runs π, π' and π'' of the program fragment in Figure 1 (page 81). Next two columns show alignments of (π, π') and (π, π''), where solid lines indicate aligned statement instances and dashed lines indicate unaligned statement instances. The last two columns show the difference between execution runs.

Definition 2 (Difference Metric). *Consider two execution runs π, π' of a program. The difference between π, π', denoted $diff(\pi, \pi')$, is defined as:*

$$diff(\pi, \pi') = \langle e^{\pi}_{i_1}, \ldots, e^{\pi}_{i_k} \rangle$$

such that

1. *each event e in $diff(\pi, \pi')$ is a branch event occurrence drawn from run π.*
2. *the events in $diff(\pi, \pi')$ appear in the same order as in π, that is, for all $1 \leq j < k$, $i_j < i_{j+1}$ (event $e^{\pi}_{i_j}$ appears before event $e^{\pi}_{i_{j+1}}$ in π).*
3. *for each e in $diff(\pi, \pi')$, there exists another branch occurrence e' in run π' such that $align(e, e')$=true (*i.e. *e and e' can be aligned). Furthermore, the outcome of e in π is different from the outcome of e' in π'*[2].
4. *all events in π satisfying criteria (1) and (2) are included in $diff(\pi, \pi')$.*

As a special case, if execution runs π and π' have the same control flow, then we define $diff(\pi, \pi') = \langle e^{\pi}_0 \rangle$.

Clearly we can see that in general $diff(\pi, \pi') \neq diff(\pi', \pi)$. The reason for making a special case for π and π' having the same control flow will be explained later in the section when we discuss comparison of differences.

Consider the example in Figure 2. The difference between execution runs π and π' is: $diff(\pi, \pi') = \langle 3_3, 7_{14} \rangle$, as indicated in Figure 2. This is because

[2] Since e, e' can be aligned, they denote occurrences of the same branch statement.

branch instances $3_3, 7_{14}$ are aligned in runs π and π' and their outcomes are different in π, π'. If the branches at lines $3_3, 7_{14}$ are evaluated differently, we get π' from π. Similarly, the difference between execution runs π and π'' is: $diff(\pi, \pi'') = \langle 7_6, 7_{14} \rangle$.

Why do we capture branch event occurrences of π which evaluate differently in π' in the difference $diff(\pi, \pi')$? Recall that we want to choose a successful run for purposes of fault localization. If π is the failing run and π' is a successful run, then $diff(\pi, \pi')$ tells us which branches in the failing run π need to be evaluated differently to produce the successful run π'. Clearly, if we have a choice of successful runs we would like to make minimal changes to the failing run to produce a successful run. Thus, given a failing run π and two successful runs π', π'', we choose π' over π'' if $diff(\pi, \pi') < diff(\pi, \pi'')$. This requires us to *compare* differences. How we do so is elaborated in the following.

Definition 3 (Comparison of Differences). *Let π, π', π'' be three execution runs of a program. Let*

$$diff(\pi, \pi') = \langle e^{\pi}_{i_1}, e^{\pi}_{i_2}, \ldots, e^{\pi}_{i_n} \rangle \quad \text{and} \quad diff(\pi, \pi'') = \langle e^{\pi}_{j_1}, e^{\pi}_{j_2}, \ldots, e^{\pi}_{j_m} \rangle$$

We define $diff(\pi, \pi') < diff(\pi, \pi'')$ iff there exists an integer $K \geq 0$ s.t.

1. *$K \leq m$ and $K \leq n$*
2. *the last K events in $diff(\pi, \pi')$ and $diff(\pi, \pi'')$ are the same, that is, $\forall 0 \leq x < K \;\; i_{n-x} = j_{m-x}$.*
3. *one of the following two conditions holds*
   - *either $diff(\pi, \pi')$ is a suffix of $diff(\pi, \pi'')$, that is, $K = n < m$*
   - *or the $(K+1)$th event from the end in $diff(\pi, \pi')$ appears later in π as compared to the $(K+1)$th event from the end in $diff(\pi, \pi'')$, that is, $i_{n-K} > j_{m-K}$.*

Thus, given a failing run π and two successful runs π', π'' we say that $diff(\pi, \pi') < diff(\pi, \pi'')$ based on a combination of the following criteria.

- Fewer branches of π need to be evaluated differently to get π' as compared to the number of branches of π that need to be evaluated differently to get π''. This is reflected in the condition $K = n < m$ of Definition 3.
- The branches of π that need to be evaluated differently to get π' appear closer to the end of π (where the error is observed), as compared to the branches of π that need to be evaluated differently to get π'' . This is reflected in the condition $i_{n-K} > j_{m-K}$ of Definition 3.

To illustrate our comparison of differences, consider the example in Figure 2. Recall that $diff(\pi, \pi') = \langle 3_3, 7_{14} \rangle$, and $diff(\pi, \pi'') = \langle 7_6, 7_{14} \rangle$, as illustrated by the "•" in the last two columns of Figure 2. Comparing $\langle 3_3, 7_{14} \rangle$ with $\langle 7_6, 7_{14} \rangle$, we see that $\langle 7_6, 7_{14} \rangle < \langle 3_3, 7_{14} \rangle$ since statement instance 7_6 occurs after statement instance 3_3 in execution run π.

According to the comparison of differences in Definition 3, we favor last differing branch instances instead of early ones. This is because the early branch instances (where the two runs are different) are often not related to the error. For example, many programs check whether the input is legal in the beginning. If we favor early branch instances, we may get failing and successful runs which only differ in whether the input is legal for such programs. Comparing such runs is unlikely to produce a useful bug report.

Comparing Runs with Identical Control Flow. Using Definitions 2 and 3 we can see that if π is the failing run, π_1 is a successful run with same control flow as that of π (*i.e.* same sequence of statements executed by a different input) and π_2 is a successful run with control flow different from π we will have $diff(\pi, \pi_2) < diff(\pi, \pi_1)$. As a result, our method for choosing a successful run will avoid successful runs with same control flow as that of the failing run. This choice is deliberate; we want to find a successful run with minimal difference in control flow from the failing run, but not with zero difference. Recall here that we construct bug report by comparing the control-flow of the selected successful run with the failing run. If the two runs have the same control flow, the bug report is null and hence useless to the programmer. In our experiments, we encountered few cases where there were some successful runs with same control flow as the failing run; these were not chosen due to our method of comparing differences between runs.

4 Experimental Setup

In order to experimentally validate our method for fault localization, we developed a prototype implementation and conducted detailed experiments. We have also implemented the Nearest Neighbor method with permutations spectrum, which performs best in [12], for a comparison with our method. We employed our prototype on the Siemens benchmark suite [6, 14] and used the evaluation framework in [12] to quantitatively measure the quality of bug reports generated by both methods. The Siemens suite has been used by other recent works on fault localization [3, 12]. In this section, we introduce the subject programs (Section 4.1) and the evaluation framework (Section 4.2).

4.1 Subject Programs

Table 1 shows the subject programs from the Siemens suite [6, 14] which we used for our experimentation. There are 132 buggy C programs in the Siemens suite, each of which is created from one of seven programs, by manually injecting defects. The seven programs range in size from 170 to 560 lines, including comments. The third column in Table 1 shows the number of buggy programs created from each of the seven programs. Various kinds of defects have been injected, including code omissions, relaxing or tightening conditions of branch statements, and wrong values for assignment statements.

In the experiments, we found that there was no input whose execution run observed the error, for two out of the 132 programs. Code inspection showed

Table 1. Description of the Siemens suite

Subject Pgm.	*Description*	*# Buggy versions*
schedule	priority scheduler	9
schedule2	priority scheduler	10
replace	pattern replacement	32
print_tokens	lexical analyzer	7
print_tokens2	lexical analyzer	10
tot_info	information measure	23
tcas	altitude separation	41

that, these two programs are syntactically different from, but semantically the same as correct programs. Actually, these two programs are not buggy programs, so we ruled out them from our experiments. We slightly changed some subject programs in our experiments. In particular, we rewrote all conditional expressions into *if* statements. This is because our prototype collects execution traces at the statement level, and cannot detect branches inside conditional expressions which are evaluated differently.

4.2 Evaluation Framework

In order to evaluate the effectiveness of a defect localizer, an evaluation framework has been proposed by Renieris and Reiss [12]. This framework assigns a score to each bug report to show the quality, defined as follows:

$$score = 1 - \frac{|DS_*|}{|PDG|} \tag{1}$$

where PDG refers to the program dependence graph of the buggy program. Let $DS(n)$ be the set of nodes that can reach or be reached from nodes in the bug report by traversing at most n directed edges in the PDG. Then DS_* is the $DS(n)$ with the smallest n which contains the observable error statement (or at least one error statement if there are more than one observable errors).

The score measures the percentage of code that can be ignored for debugging. Clearly, higher score indicates bug report with higher quality. Note that the score only measures the utility of the bug report for debugging, *it does not necessarily correlate a good quality bug report with a lean bug report*. To address this weakness, we conducted separate experiments to measure bug report size.

5 Experimental Results

We employed the prototype implementation of both our method and the Nearest Neighbor method with permutations spectrum (NN method) [12][3] to 130

[3] We used the accurate permutations spectrum for NN method and considered all failing runs which had *some* successful run with a different spectrum. So, we can study all the 130 programs compared to the 109 programs studied in [12] where certain programs were ruled out based on a coarser spectrum (coverage).

buggy programs from the Siemens suite. The NN method compares code coverage between a failing run and the "nearest" successful run from a pool of successful runs. Through the experiments, we validate our method by answering the following three questions.

- Is our method effective for fault localization?
- Is the size of generated bug report voluminous and overwhelming?
- How many successful runs are required available to make a decent choice of successful run?

In this section, we present experimental results for these questions.

5.1 Locating the Bug

In the Siemens benchmark suite, each buggy program P comes with a large pool of inputs, some of which result in successful runs, and others result in failing runs. For each failing run π_f, there is a set of successful runs $Closest(\pi_f)$ which are closest to π_f, in terms of our difference metric or that of the NN method. The score for a failing run π_f averages scores of comparing π_f against each successful run π_s in $Closest(\pi_f)$, *i.e.*

$$score(\pi_f) = \frac{\sum_{\pi_s \in Closest(\pi_f)} score(\pi_f, \pi_s)}{|Closest(\pi_f)|}$$

where the quantity $score(\pi_f, \pi_s))$ is defined in Equation (1) in Section 4.2. The score for a buggy program P averages scores of all failing run π_f of P, *i.e.*

$$pgm\_score(P) = \frac{\sum_{\pi_f \in Failing(P)} score(\pi_f)}{|Failing(P)|}$$

where $Failing(P)$ refers to the set of failing runs of program P. Our method differs from the NN method in which successful runs are selected for comparison, and (hence) which statements are reported in bug report.

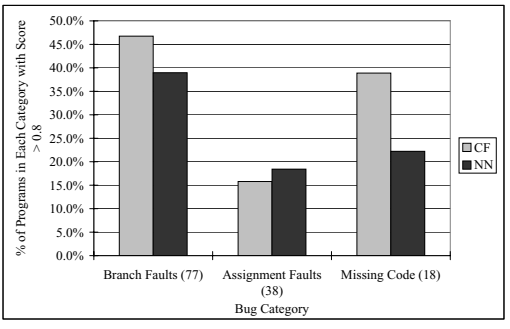
Table 2(a) shows the distribution of *pgm_score* for two methods. Our method is shown as CF, an abbreviation for Control Flow based difference metric. As we can see, our method performs a little better than the NN method on the Siemens suite. Bug reports returned by our method achieved a score of 0.8 or better for more than 37% of all the buggy programs, while the NN method achieved a score of 0.8 or more for about 31% of the programs. Note that a bug report with score of 0.8 or more indicates that programmer needs to inspect at most 20% of a buggy program for fault localization using this bug report.

In the above experiments, we computed the score of program P by averaging scores w.r.t. all π_s and π_f. However, the programmer will often choose one closest successful run π_s and one failing run π_f for comparison. Is our method sensitive to the choice of π_s and π_f? First, our method is not sensitive to the choice of closest successful run π_s since any $\pi_s \in Closest(\pi_f)$ returns the same bug report, that is, $score(\pi_f, \pi_s)$ is the same for all $\pi_s \in Closest(\pi_f)$. Secondly, our method

Table 2. (a) Distribution of scores, and (b) Locating different kinds of errors, where each category has 77, 38 and 18 programs, respectively. Note that the sum of programs in each category is more than 130. This is because 3 programs have two bugs of different kinds, and are counted in two categories.

Score	CF	NN
0.9 - 1	23.1	10.8
0.8 - 0.89	13.8	20.0
0.7 - 0.79	8.5	20.8
0.6 - 0.69	10.8	13.8
0.5 - 0.59	14.6	10.8
0.4 - 0.49	9.2	10.0
0.3 - 0.39	6.2	2.3
0.2 - 0.29	9.2	3.1
0.1 - 0.19	2.3	0.8
0 - 0.09	2.3	7.7

(a)



(b)

is less sensitive to the choice of failing run π_f than the NN method. We validated this by computing variances w.r.t. choice of failing run for each fault program in our experiments. Given a set of failing runs $Failing(P)$ of a faulty program P, the variance of P is defined as:

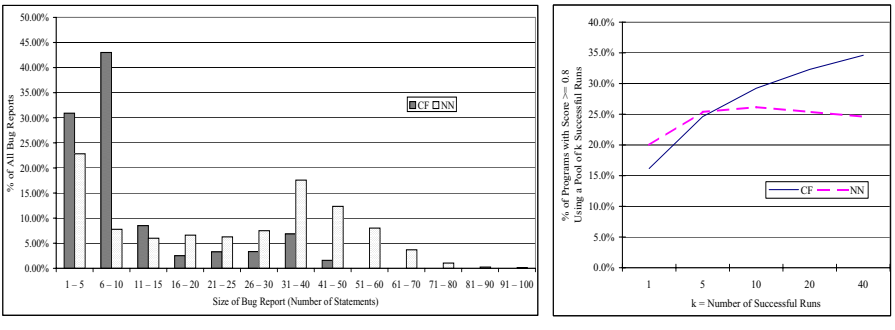
$$variance(P) = \frac{\sum_{\pi_f \in Failing(P)} (score(\pi_f) - pgm\_score(P))^2}{|Failing(P)|}$$

where $score(\pi_f)$ and $pgm\_score(P)$ are defined in the preceding. Using our method, we found that the score's variance was small (less than 0.01) for 56.6% of all 130 faulty programs. On the other hand, using the NN method, only 42.3% of 130 faulty programs had small variances (less than 0.01) in their scores.

Next we study the effectiveness of our technique in locating different kinds of errors. We classified all the errors in the faulty programs into three categories: *Assignment Faults*, *Branch Faults* and *Missing Code*, where *Assignment Faults* refer to errors in assignment and return statements, *Branch Faults* refer to errors in conditional branch statements and *Missing Code* refers to errors due to missing program statements. Table 2(b) shows percentage of faulty programs in each category where the bug reports got a score of 0.8 or better. We see that our method was more effective in locating branch faults. For almost half of the programs with branch faults, our method got a score of at least 0.8; this is not surprising since the difference metric returned by our method contains only branch statements with different outcomes in failing and successful runs. For the same reason, our method did not fare as well in locating faulty assignments. Since we report only branches in the bug report, the programmer has to follow dependencies from these branches to the faulty assignment – thereby reducing the score of our bug report. In presence of "missing code" errors, our method may report branch statements on which missed code would have been (transitively) control dependent.

5.2 Size of Bug Report

In the above experiments, we used scores to measure the quality of bug report according to the evaluation framework in Section 4.2. *The reader should note that there is a fundamental difference between the bug report statements and the statements that a programmer should inspect for debugging according to the evaluation framework.* Clearly, measuring the amount of code to be inspected for debugging (captured by the bug report score) is important. However, we feel that measuring the bug report size is also important. If the programmer is overwhelmed with a voluminous bug report (*e.g.* 50 statements for a 500 line program), he/she may not even get to the stage of identifying which code to inspect using the bug report.



(a) (b)

Fig. 3. (a) Size of bug report and (b) Impact of successful run pool-size

Figure 3(a) shows sizes of bug reports produced by our method and NN method. We can see the bug reports produced by our method are relatively small. For example, more than 80% (40%) of bug reports in all the 130 faulty programs contained less than 15 statements using our method (NN method). Considering that programs in the Siemens suite are relatively small, reports with more than 15 statements may be too voluminous. The choice of the cutoff number 15 is not crucial as can be seen in Figure 3(a); similar trends are observed for any small cut-off number on the bug report size.

Recall from Table 2(a) that our method and NN method produced roughly the same scores – 37 % (31 %) of all the 130 programs produced a score of 0.8 or more with our (NN) method. However, if we study these buggy programs which produced a high score (of 0.8 or more) with the two methods — we see that bug report in 83% of them had less than 15 statements for our method, compared to only 28% for the NN method.

5.3 Size of Successful Run Pool

In the Siemens suite, each faulty program has a large set of test inputs (1000 – 5000). The successful run pool is constructed out of these inputs. How many

successful runs are required for the programmer to make a decent choice of successful run? We study this in the following.

Given a program P, we selected the failing run π_f whose score $score(\pi_f)$ (using both our method and NN method) is closest to the score of the program $pgm\_score(P)$) (again using both our and NN methods). The selected failing run π_f was used to study both our method and the NN method. We did not conduct experiments w.r.t. all failing runs because it was too expensive.

Next, for every successful run π_s in the available pool of the Siemens suite, we computed the difference between π_f and π_s, generated a bug report by comparing π_f and π_s, and computed $score(\pi_f, \pi_s)$ (refer Equation (1)). After all successful runs were processed, their differences were sorted in ascending order. Let π_i be the successful run with ith smallest difference w.r.t. π_f. The *parameterized mean score* of a faulty program P for a successful run pool-size of k is:

$$par\_score(P,k) = \sum_{i=1}^{n} score(\pi_f, \pi_i) \cdot p(i,k) \qquad p(i,k) = \frac{^{n-i}C_{k-1}}{^{n}C_{k}}$$

where π_f is the failing run chosen in P as mentioned above, n is the number of available successful runs in Siemens suite, and $p(i,k)$ is defined above. Here $^{n}C_{k}$ denotes a well-known quantity — the number of ways of choosing k items from n distinguishable items. Clearly, $p(i,k)$ denotes the probability that the ith-closest successful run of the failing run is chosen as the nearest successful run of a failing run from a pool of k different successful runs. Hence $par\_score(P,k)$ captures the statistical expectation of the score obtained for failing run π_f using any pool of k successful runs. Calculating the parameterized mean score $par\_score(P,k)$ allows us to avoid exhaustively enumerating the score of P for different successful run pools of size k.

Figure 3(b) presents the parameterized mean scores for different values of k, the successful run pool size. We see that both our method and NN method made a decent choice of successful run from a pool of 5 runs and thereby achieved a score of at least 0.8 in 25% of the 130 faulty programs. However, as the pool size increases to 40, our method achieved a score of 0.8 or more for larger number of faulty programs (for 35% of faulty programs). This is not the case for the NN method, which in fact needed even larger pool sizes.

5.4 Threats to Validity

In our experiments, we used the evaluation framework of Section 4.2 to measure the quality of bug report. However, the score computed by the framework of Section 4.2 may not accurately capture the human efforts for fault localization in practice. First, the framework assumes that the programmer can find the error when he/she reads the erroneous statements. This assumption may not hold for non-trivial bugs, where the programmer has to analyze program states. Secondly, the evaluation framework requires the programmer to perform pure breadth-first search for fault localization starting from statements in the

bug report. However, the programmer usually has some understanding of the buggy program, and he/she can prune some irrelevant statements from bug report.

There are also threats to the validity of the study on successful run pool size. In this experiment, we chose one failing run for each faulty program, instead of studying all failing runs. Thus, if we chose some other failing run, the *parameterized mean score* for a pool of k successful runs may change. We expect that such changes will not be significant (though it is possible), because the variances w.r.t. all failing runs were small for most faulty programs.

6 Experience and Discussion

In this paper, we present a control flow based difference metric to compare execution runs. This difference metric can be used to choose a successful run from a pool of program inputs, and compare the given failing run with the chosen successful run for fault localization. Our experiments with the Siemens suite indicate that our difference metric produces bug reports which are small in size and effective in fault localization.

One important issue in a method like ours is the choice of the successful run pool. In the last section, we reported experiments to measure the required size of the successful run pool. However, even for a given pool-size many choices of the pool are possible. So, how do we construct the pool? There are two solutions to this problem. One possibility is to have a pre-defined large set of program inputs *Inp*; this set of test-cases might have been generated using some notion of coverage. Now given a failing run, we find out which of the inputs in *Inp* produces a successful run — thereby getting a pool of successful runs. In our experiments with the Siemens suite, we followed this approach by using the pre-defined pool of inputs provided with each benchmark. Another way of constructing the successful run pool is to use the input for the given failing run. We can slightly perturb this failing input to generate a set of program inputs; we then classify which of these perturbed inputs produce a successful run — thereby getting a pool of successful runs. The main drawback of this approach is that it relies too much on the programmer's intuition in deciding what to perturb in the failing input. Although automatic techniques such as Delta Debugging [19] exist, they cannot be used for arbitrary programs. This is because these approaches construct an input by removing part of the erroneous input. *This is indeed suitable for debugging programs like compilers, web-browsers — where the program input is a large file. However, for other programs (*e.g. *programs with integer inputs) this approach may be problematic.*

We now conclude the paper by sharing some experience in this regard that we gained by debugging a widely used Unix utility – the *grep* program. The correct version of *grep* has 13,286 lines, without header files. The *grep* program searches text files for a pattern and prints all lines that contain that pattern. Faulty versions of the *grep* program and test cases are provided at [16]. For the sake of illustrating our point about the successful run pool, here we only report our experience in

```
1.  char ch[2];
2.  ch[0] = c;
3.  ch[1] = '\0';
4.  if (strcoll (ch, lo) <= 0 && strcoll (hi, ch) <= 0)
5.  { ...
```

Fig. 4. Fragment of a faulty version of the *grep* program

debugging a particular failing run of a particular buggy version. Figure 4 presents a faulty version of `grep`, where the branch in line 4 should be `if (strcoll (lo, ch) <= 0 && strcoll (ch, hi) <= 0)`. We consider the failing run corresponding to the input `grep -G '[1-5\]' grep1.dat`. This failing run contains 800,738 statement instances. This run did not return all lines which contain numbers between 1 and 5 in the `grep1.dat` file — an observable error.

When we ran our debugging method against 35 selected successful runs from the given test inputs of *grep* (provided in [16]) we got a bug report containing 11 statements. A line very close to the faulty branch statement in Figure 4 was included in the report; the score of the bug report is 0.977. This means that programmer needs inspecting about 100 lines in the worst case, considering that the *grep* program has many blank lines. In practice, some statements contained in the bug report may be pruned, depending on programmer's understanding of the program. This will lead to ever fewer statements for inspection.

On the other hand, if we perturb the failing input to get various sub-intervals of `[1-5]` as the first argument of *grep*, only the following five are encountered as successful inputs.

$$\texttt{grep} - \texttt{G}\ \ '[\texttt{i} - \texttt{i}\backslash]'\ \ \texttt{grep1.dat} \qquad \texttt{i} \in \{1, 2, 3, 4, 5\}$$

When we applied our debugging method to this pool of five successful runs we observed the following. (1) Depending on the choice of the successful run, there was substantial variation in the bug reports and their scores (the score varied from 0.288 – 0.998). Thus choosing a successful run seems to be important even if the successful run pool is manually generated using programmer's intuition. (2) The difference corresponding to the chosen successful run produced a bug report of 15 statements, which included the buggy statement (thereby obtaining a nearly perfect score 0.998).

Thus, the score was slightly better than the score produced using the test input pool provided with the *grep* program. However, significant intuition was needed to manually construct the successful input pool for a specific failing run. In practice, we feel that the choice of successful run will always benefit from the programmer's intuition. However fault localization methods — such as the one described in this paper — can substantially increase the degree of automation in this debugging task.

Future Work. In terms of future work, we note that our prototype implementation currently has limitations w.r.t. tracing overheads. Since our difference

metric uses more information (traces of the failing and successful runs) than the NN method (which uses sets of statements in the two runs), therefore the issue of tracing overheads becomes important. Indeed, it was costly to collect and store execution traces using our prototype. To make our method scalable to large programs, sophisticated instrumentation techniques (*e.g.* [8]) need to be employed. We are currently working in this direction.

Acknowledgments

This work was partially supported by a Public Sector Research Grant from Agency of Science Technology and Research (A*STAR), Singapore.

References

1. T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 97–105, 2003.
2. S. Chaki, A. Groce, and O. Strichman. Explaining abstract counterexamples. In *ACM SIGSOFT Symp. on the Foundations of Software Engg. (FSE)*, 2004.
3. H. Cleve and A. Zeller. Locating causes of program failures. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2005.
4. J. Ferrante, K.J. Ottenstein, and J.D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.
5. A. Groce. Error explanation with distance metrics. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 108–122, 2004.
6. M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 191–200, 1994.
7. J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 467–477, 2002.
8. J. R. Larus. Whole program paths. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 259–269, 1999.
9. B. Liblit, A. Aiken, A. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2003.
10. B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan. Scalable statistical bug isolation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.
11. B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss. Automated fault localization using potential invariants. *CoRR*, cs.SE/0310040, Oct, 2003.
12. M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *Automated Software Engineering (ASE)*, pages 30–39, 2003.
13. T. W. Reps, T. Ball, M. Das, and J. R. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *ACM SIGSOFT Symp. on the Foundations of Software Engg. (FSE)*, 1997.

14. G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering*, 24, 1998.
15. J. Ruthruff, E. Creswick, M. Burnett, C. Cook, S. Prabhakararao, M. Fisher II, and M. Main. End-user software visualizations for fault localization. In *ACM Symposium on Software Visualization*, pages 123–132, 2003.
16. `http://www.cse.unl.edu/~galileo/sir`.
17. T. Wang and A. Roychoudhury. Automated path generation for software fault localization. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pages 347–351, 2005.
18. A. Zeller. Isolating cause-effect chains from computer programs. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 1–10, 2002.
19. A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28, 2002.