Fault Localization With Nearest Neighbor Queries

Manos Renieris and Steven P. Reiss
Brown University
Department of Computer Science
Box 1910, Providence, RI 02912, USA
E-mail: {er,spr}@cs.brown.edu

Abstract

We present a method for performing fault localization using similar program spectra. Our method assumes the existence of a faulty run and a larger number of correct runs. It then selects according to a distance criterion the correct run that most resembles the faulty run, compares the spectra corresponding to these two runs, and produces a report of "suspicious" parts of the program. Our method is widely applicable because it does not require any knowledge of the program input and no more information from the user than a classification of the runs as either "correct" or "faulty". To experimentally validate the viability of the method, we implemented it in a tool, WHITHER *using basic block profiling spectra. We experimented with two different similarity measures and the Siemens suite of 132 programs with injected bugs. To measure the success of the tool, we developed a generic method for establishing the quality of a report. The method is based on the way an "ideal user" would navigate the program using the report to save effort during debugging. The best results we obtained were, on average, above 50%, meaning that our ideal user would avoid looking at half of the program.*

1 Introduction

Programmers often write almost-correct programs. Such programs will sustain significant testing without revealing any bugs. When, eventually, use or further testing uncovers the bugs, the programmers need to localize the faulty portions of code and correct them. In this paper, we investigate techniques that leverage the successful test cases to support the first task, fault localization.

In some cases fault localization is easy. For example, consider the simple `triangleType` program in figure 1. The program fails exactly when line 5 is executed. This fault is easy to localize using the following strategy: execute the program on a set of inputs collecting information about which lines execute in each run; then observe which lines execute only on inputs for which the program fails.

This strategy fails for most programs and most faults. The reason is that single lines of code (or even short, contiguous blocks of code) are rarely decisive for the success of the program: their effect on the outcome depends heavily on the run-time context in which they execute. This is evident in the number of remaining bugs in programs that are fully tested under some code coverage criterion. In our example, the strategy works because the context is empty. In a stateless language the context problem is simpler [25], but in general differences between which blocks of code execute in failing and successful runs are potentially significant only if the runs are similar.

Research in this area has addressed the run similarity problem by minimizing differences in inputs [23, 27, 29]. These techniques require knowledge of the inputs' structure and methods to manipulate them minimally. They make the more covert assumption that similar inputs lead to similar runs. These limitations mean that the techniques are not readily applicable to arbitrary, or even most, programs.

In this paper we show how to use successful runs to address the limitations of input-based techniques. To avoid trying to localize two faults at once, we consider a single failing run at a time. The key idea is to define a similarity measure directly on the executions of the program, and then use it to select a successful run that is as similar as possible to the faulty run. The technique does not require knowledge of the bug or the input structure of the program, and is

```
triangleType(float a, float b, float c)     1
{                                           2
  if (a == b && b == c) return Equilateral; 3
  if (a == b || b == c) return Isosceles;   4
  return Isosceles; /* should be Scalene */ 5
}                                           6
```

Figure 1. A small faulty program

Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)
1527-1366/03 $ 17.00 © 2003 IEEE

therefore widely applicable with little effort.

To evaluate this technique experimentally we developed a tool, WHITHER, which we used on an established suite of programs with injected faults. WHITHER has an open architecture, allowing a number of previous approaches to be expressed in a common framework.

To measure WHITHER's success, we designed and implemented a method for the evaluation of fault localization tools. The method can assign a score to the report of a fault localization system, depending on the size of the report and how closely to the actual fault it is. Proximity to the fault is defined based on the program dependence graph. To our knowledge, this is the first such measure based on program semantics.

The rest of this paper is structured as follows: in section 2 we review previous research that lead to our work. In section 3 we describe a generic architecture for fault localization systems that use multiple successful runs and a single faulty run. Then we show the nearest neighbor concept and how it fits in the architecture. In section 4 we detail our method for quantifying the success of a fault localization system. In section 5 we describe the implementations of the architecture we have performed experiments with. Sections 6 and 7 describe our experiments and results respectively. Finally, section 8 summarizes our results and describes how our framework relates to other state-of-the-art tools.

2 Motivation

Existing fault localization systems manipulate program inputs to create two very similar inputs, one which causes the program to succeed and one which causes the program to fail. Assuming that similar inputs result in similar runs, programmers can then contrast the two runs to help locate the fault.

Whalley [27] presents *vpoiso*, a tool to localize bugs in a compiler. A bug is detected by comparing the output of a program compiled with the buggy compiler with one compiled with a "correct" compiler. For bugs in the optimizer runs are contrasted at the optimization phase level. If the optimization phases are $p_1, p_2, \ldots, p_n$ *vpoiso* orders the set $\{\{p_1\}, \{p_1, p_2\}, \ldots, \{p_1, p_2, \ldots, p_n\}\}$, under the usual subset relation. Then, for every such set, it turns the included optimizations on, and checks the result. *vpoiso* performs a binary search on the set, isolating a fault inducing set $\{p_1, p_2, \ldots, p_m\}$ such that $\{p_1, p_2, \ldots, p_{m-1}\}$ is not faulty. *vpoiso* assumes the phases independent, therefore it blames phase p_m. For non-optimizing bugs, *vpoiso* does not localize the bug, although it isolates a minimal input for which the compiler fails. It orders the functions $\{f_1, f_2, \ldots, f_n\}$ of the subject program, and considers the subsets $\{f_1\}, \{f_1, f_2\}, \ldots, \{f_1, f_2, \ldots, f_n\}$. For each one of them, it compiles the set's members with the suspect compiler, and the other functions of the subject program with a trusted compiler. Similar to the optimizing case, a function set $\{f_1, f_2, \ldots, f_m\}$ is isolated such that compiling all of its functions with the suspect compiler reveals the fault, while compiling only $\{f_1, f_2, \ldots, f_{m-1}\}$ with the suspect compiler does not. Compilation of f_m is blamed.

Whalley's techniques work under strict rules: the optimizing phases are assumed totally independent, and an error in compiling one function cannot be masked by wrongly compiling another one. Essentially, the power set of $\{p_1, \ldots, p_m\}$ has to be totally ordered. Moreover, its mapping to failure or success has to be monotonic: if a set fails, all its supersets fail, and if it succeeds, all its subsets succeed. Zeller [29] extended these input minimizing techniques to handle cases where there is no monotonicity, while giving weaker guarantees on the minimality of the input. His technique, *delta debugging*, is a greedy algorithm that examines potentially a quadratic number of input subsets, although it is cleverly tuned to make full use of Whalley's assumptions if they hold. The first application of the technique on input was on the input of a web browser, but Zeller has applied this technique in other debugging domains, such as comparing thread schedules and looking at changes between two versions of a program.

Reps et al. [23] present DYNADIFF, a tool that isolates faults at the level of acyclic, intraprocedural control paths. They start with a business program they suspect of the Y2K bug, and run it twice, once with the system clock set to the end of 1999, and once with the clock set to the beginning of 2000. Then they inspect the control flow paths that the program takes, trying to find paths that were never taken in the former case, but were taken in the later. The idea is that paths taken only after the crucial date are suspect of bugs.

All three of these techniques depend strongly on specific kinds of input. It is hard to see how similar techniques would apply to arbitrary programs with generally unstructured inputs such as sets of numbers. For example, the *tcas* program, part of the Siemens suite [16] commonly used in testing research simulates the decision process that planes follow about ascending or descending. Its input is a sequence of 13 numbers, describing the plane's state.

A fault localizer that does not depend on knowledge of the input structure must rely on features of the program execution that are universal among programs. At the same time, these features must be attributable to specific portions of the source code.

Collections of such features are called *program spectra*. The term "spectrum" was introduced in Reps et al. [23], for (acyclic, intraprocedural) path spectra, and generalized in Harrold et al. [11]. One example of a program spectrum is profiling data that shows the number of times each line of the program is executed. This spectrum concentrates on control flow and abstracts the overall execution into a rel-

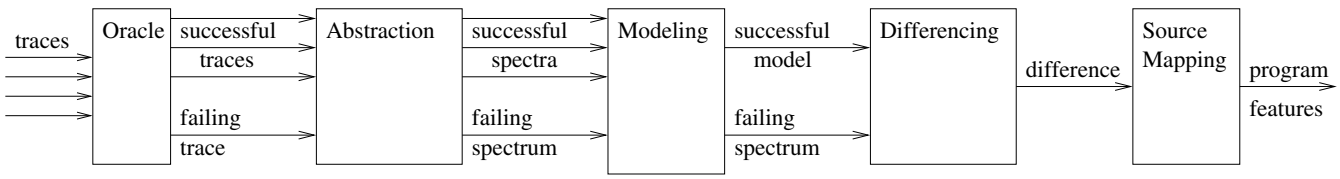

Figure 2. A pipe-and-filter architecture for fault localization

atively small set of counts. Other proposed spectra consider function call counts, program paths, program slices, and use-def chains.

Program spectra are a natural basis for fault localization. Harrold et al. [11] performed a large scale experiment on program spectra. The result was that, although an unusual spectrum does not necessarily signal a faulty run, faulty runs tend to have unusual spectra. Further work by Podgurski et al. [5] applies clustering techniques to observation testing. They find that the profiles of failing runs tend to belong in smaller clusters and be further away from their nearest neighbor than successful runs.

In a sense, these results are not surprising. Programs that succeed on a large number of inputs (almost-correct programs) must encompass a lot of knowledge about the typical inputs and consequently, typical runs.

Our general fault localization strategy uses program spectra to identify a failing run that is similar to a successful run and then uses the difference between the spectra of these runs to help isolate the location of the bug. In the next section, we describe this strategy in detail.

3 Architecture

We first define a general architecture for fault localization. Our architecture is essentially a typed pipe-and-filter architecture [26]. The input of the system is a set of program traces, containing a single trace from a failing run. A trace is simply as much information about a run as we can collect.

The architecture then comprises the following phases, as shown in figure 2:

1. A classification phase, in which an oracle classifies the traces as successful or failing.

2. A trace abstraction phase, that converts traces to abstract representations of runs. Following [11, 23], we will call these abstract representations *spectra*. The distinction between successful and failing traces carries over to spectra.

3. A modeling phase, that converts the successful spectra into a *model* of successful runs, perhaps taking into account the failing spectrum. The outputs of this phase are the model and the spectrum of the faulty run, passed intact.

4. A differencing phase, which contrasts the model and the spectrum of the failing run to produce a *difference*.

5. A source mapping phase, which maps the difference to a *report*, a set of program locations.

The data type flowing on any pipe must lend itself to all the succeeding operations. In particular, every modeling process demands that the spectra afford specific operations. For example, a naive model could simply include a random successful spectrum. Such a model demands a kind of spectrum that affords defining a difference operation on two spectra. The later phases demand that this difference is attributable to source code.

We differentiate between kinds of spectra according to the operations that we can perform on them, and therefore the kinds of models we can build from them. In the rest of this section, we discuss two possible kinds of spectra and the models they support.

3.1 Set Spectra and the Union and Intersection Models

A simple kind of spectra afford a single operation: conversion to a set of program features. We will call this kind *set spectra*. For example, a set spectrum could encode the pieces of code that executed during the run it represents. Such pieces of code could be basic blocks, functions, paths, or slices.

Based on set spectra, a simple model is one that, given a failing run f and a set of successful runs S, computes the union of all the successful runs $\bigcup_S s$ and the difference $f - \bigcup_S s$. In essence, this "union model" tries to find features that are unique to the faulty run. A complementary model is the "intersection model", which tries to find features whose absence is discriminant for the faulty run: $\bigcap_S s - f$. Extensions to these models are proposed in [3, 21] based on "soft" unions and intersections that incorporate frequency criteria.

3.2 Distance Spectra and the Nearest Neighbor Model

They key idea in DYNADIFF and *vpoiso* is the careful selection of a single successful run to contrast with the faulty run. Selecting the run according to its input has the disadvantages we discussed in section 1.

A *distance spectrum* is a spectrum that affords a distance operation, which allows us to lift both difficulties. The distance operation should be a measure of dissimilarity. Ideally the distance operation implements a metric; in other words, it is symmetric and obeys the triangle inequality. A plausible model consists of a single successful spectrum, the one closest to the failing spectrum. We will call this the nearest neighbor model.

It becomes essential that the distance spectra supply a contrast operation, which attributes the distance between two spectra to a set of program features. This way, the two spectra can be used to produce a fault localization report.

A set spectrum can be trivially augmented with a distance function, using the Hamming distance[1] over the representation of the two sets as binary vectors.

The difference function that is most consistent with the Hamming distance computation is the symmetric set difference between the two sets, putting the blame on all features that appear in one but not the other (the Hamming distance is the size of the symmetric set difference). However, this would mean that features absent in the failing run get reported, which is generally not desirable. Instead, we can just use the non-symmetric difference $f - s$ and report features that are only present in the failing run. The trivial fix to this asymmetry is to compute the distance in the space defined by the features present in the successful run. But this space is smaller, and we will see that the size of the space is important.

4 Evaluation Framework

To evaluate different instances of the architecture we need a quantitative measure of the quality of a fault localization report. Previous research does not provide such measures, relying instead on case studies for evaluation. This hinders comparing fault localization systems.

Formally, given a report R, that is, a list of program features the localizer indicates as possible locations of the bug, we want to compute and assign a score to it. We want the scoring function to represent an estimate of the effort to find the bug starting with the report. To allow for direct comparison between reports on different programs, we want the scoring function to assign a perfect score, say 1, to perfect reports, and 0 to the worst possible reports. We now need to define a perfect report, and to define a way to measure how far any report is from a perfect report.

For the purpose of evaluating a fault localizer, we can assume the existence of a correct program; the differences between the correct program and its faulty version point to where the fault is. The general problem of taking differences between programs is difficult [12], but in fault localization the granularity of differences is dictated by the tracing process. This makes the problem much easier than the general case. A perfect report should directly point to the fault and should also be as small and specific as possible: a report that includes the whole program would definitely include the bug, but it would be far from perfect. However, the exact location of the bug can be difficult to pin down, either because there can be a lot of ways to fix the bug or because fixing the bug requires changes scattered throughout the code. In the first case, we would say that the bug is at any of those locations; in the second, that the bug is at all of them. To handle all cases in a uniform and simple manner, we say that a report is still perfect if it points to at least one faulty location, and no correct locations. Then, the worst possible report is the one that points to all the correct locations and none of the faulty ones.

Deciding how close a report is to a perfect report is more difficult. At any level of granularity, the perfect report and the report we are trying to assign a score to are sets of program features. The correspondence of such sets is commonly measured by comparing the size of their intersection with the sizes of the sets themselves. But this treats programs as unstructured sets, which is not an accurate representation.

We propose an approach based on static program dependencies. At the level of granularity we choose, we construct a program dependence graph (pdg) [14], a graph that contains a node for each expression in the program, and two kinds of edges: data dependency edges between two nodes that use the same value, as reflected in their use of variables, and control-dependency edges, if one node controls the execution or not of the other. We mark the nodes of pdg as "faulty", if they were reported by differencing the correct and the faulty versions of the program, and "blamed" if they are reported by the localizer. We assign to all edges the same arbitrary, constant weight.

Then, for each node n, we define the *k-dependency sphere set* (DS_k). The DS_k of a node n is the set of nodes m for which there is a directed path of length no more that k that joins m and n. For example, the DS_0 is n itself. DS_1 includes the node itself, and also all the nodes m such that there is a edge from m to n, or from n to m. DS_2 includes DS_1, and also all the nodes m such that there is a directed path of length 2 from m to n or from n to m. The k dependency sphere of a report is simply the union of the DS_ks of

[1] The Hamming distance is defined as the number of positions at which the two vectors disagree.

all the nodes in the report.

Let us consider a report R, and let us call $DS_\star(R)$ its smallest dependency sphere that includes a faulty node. The score we assign to R is based on the size $|DS_\star(R)|$ of that sphere. We want to normalize the scores with respect to the number of nodes in the overall graph $|PDG|$, and we also want a higher score to signify a better report. Therefore the formula we use for the score is

$$1 - \frac{|DS_\star(R)|}{|PDG|}$$

As a special case, we assign a zero score to empty reports.

If a report R includes a faulty node, then its score will be $1 - \frac{|R|}{|PDG|}$. A report that includes every node would therefore get a score of 0. A report that includes a single node which is far from faulty nodes will get a score close to, or sometimes exactly, zero. The scoring function assigns higher scores to reports that contain the bug, but a report cannot achieve that by including as many nodes as possible. At the same time, the scoring function will assign a high score to a report that includes a few nodes, at least one of which is very close to a faulty node.

The scoring function reflects the amount of code an ideal user would have to read. Our ideal user starts with the report and then, using knowledge of the program, works outwards from the locations in the report. This corresponds to doing a breadth-first search of the dependency neighborhoods. We assume that the ideal user recognizes faulty nodes on sight and thus stops as soon as a faulty node is encountered. Thus the score reported is proportional to the amount of the program left unexamined by the ideal user.

5 Implementations of the Architecture

In order to explore different fault localization approaches, we implemented four instantiations of the general fault localization architecture. One of these uses the union model, one uses the intersection model, and two use the nearest neighbors model. The first three use a simple binary coverage spectrum, while the fourth uses a more detailed coverage spectrum. This lets us compare the nearest neighbor model with other models, and evaluate the impact of the choice of spectrum and the availability of trace information on the results.

In the binary coverage spectrum, a run is represented as the set of basic blocks (pieces of code that execute atomically) that executed during it. We used *gcov*, the basic block level profiler of the GNU compiler suite, to get counts of executions for each basic block, and mapped the zero counts to zero and the non-zero counts to one. The binary coverage spectrum is trivially a set spectrum, and therefore the union, intersection, and Hamming distance nearest neighbor models are supported as discussed in section 3.1. The choice of the Hamming distance (a symmetric operation) and the asymmetric set difference operation for the nearest neighbor model means that it is possible for a pair of runs to have non-zero distance and an empty difference.

The binary coverage spectrum is extremely simple, and we use it to provide a base case and to provide rough comparisons. A more interesting spectrum would make use of the actual basic block execution counts. There is negligible extra cost in collecting them, and representing runs as vectors of integers allows a multitude of choices for the distance function, such as the Euclidean distance.

However, using such vectors for determining the difference between a successful and a failing run in order to find the potentially faulty locations is difficult. The standard elementwise vector difference will not do; long runs can actually be very close to shorter runs (going though a loop 10 times in one example and 10,000 in another) but elementwise difference will not preserve this information. The same is true for the case that two loops in the program execute an equal number of times within each run, but the numbers are different between the two runs. We could normalize the vectors before subtracting them. But then the components that execute the same numbers of times (for example, the first line of the main function in a C program, which executes exactly once) would appear different for any run of even miniscule length differences. What we need is a technique that would keep the relevant execution counts of the runs, but not the counts themselves.

We therefore represent each run as the sorted sequence of its basic blocks, where the sorting key is the number of times that basic block executed, and measure distance between the sequences as the distance between two permutations. Distances between permutations are generally based on the cost of transforming one to the other given a certain set of operations (for a quick introduction, see [4]). This is equivalent to considering one of the permutations as sorted, and counting the number of operations we need to sort the other one. If the operations we allow are only adjacent exchanges (as when we sort by insertsort or bubblesort), the resulting distance is called Kendall's τ. If we allow arbitrary exchanges, the distance is called Cayley's distance. Ulam's distance is the distance we get if we allow arbitrary *moves*. For example, the permutations a, b, c, d and a, c, d, b are at distance 1, because we can transform the first one to the second by simply moving the b to the end. Allowing such operations captures the phenomenon of executing the body of one loop more or fewer times than the body of another. Additionally, we want to attribute the difference between two spectra to the basic blocks involved in the editing operations; Kendall's and Cayley's methods involve too many. For these reasons, we use Ulam's distance as our measure. Ulam's distance is also easy to implement efficiently [15].

6 Experimental Setup

Using the four instances defined in the previous section, we wanted to validate experimentally three hypotheses:

- It is possible to locate bugs with the spectra we have.
- The nearest neighbor model outperforms the union and intersection models.
- Comparing a failing run with its nearest neighbor is more beneficial to fault location than comparing the failing run with some other, random run.

Throughout, we will explore how the actual choice of spectrum affects the answers to these questions. As mentioned previously, we developed a tool, WHITHER, that implements the union, intersection and nearest neighbor models, the last one with two spectra, the coverage spectrum and the permutations spectrum. WHITHER is about 1000 lines of Ocaml [19].

Our subject faulty programs come from the GeorgiaTech version [24] of the Siemens suite [16]. The Siemens test suite consists of 132 C programs with injected faults (Table 1). Each program is a variation of one of seven "golden" programs, ranging in size from 170 to 560 lines, including comments. Each faulty version has exactly one fault injected, although the faults are not necessarily localized; some of them span multiple lines or even functions. A significant number of the faults are simple code omissions: some of them make some condition in the program stricter, some make a condition laxer, and some omit a statement from a sequence. We made some slight cosmetic modifications to the programs, largely to tune them to the available tools. For example, we joined lines that were part of the same statement. The most important change we performed was to align the programs with the correct one, so that all bugs were just line modifications, as opposed to line insertions and deletions.

To obtain the pdg, we used CodeSurfer [1], a commercial program slicing tool, which exports a pdg. We had to convert the exported pdg to a graph over lines, our level of profiling. We did this by adding a node to the pdg for every line, and connecting all line nodes to the pdg nodes that represented the line. We gave such edges weight 0. The distance between any two lines is the length of the shortest directed path between them.

Each of the seven program families comes with a test suite that exposes the bugs in each of the faulty versions. To separate the fault inducing inputs from the non-fault inducing inputs, we run the golden version on each of the inputs and compared the result with the result of running a faulty program on the same input. An overwhelming number of tests succeed for each version. Column 6 (#Failed Tests) of table 1 shows the range of the number of faulty runs for each program family. For example, a version of *print_tokens* failed on only 6 of 4072 inputs, and another version of the same program failed on 186 inputs. Two programs (version 32 of replace and version 9 of schedule2) gave us no faulty runs. In the first one, the inserted bug was the exchange of a logical and operation for a bitwise and operation, but on the system we run the programs this had no effect. For the second program, the inserted bug exchanges a function call for another, but the intended function is called transitively (and the result discarded). We excluded these two programs from the rest of our experiments. To examine our third hypothesis, we need to consider every pair consisting of a successful and a failing run of every specific program version. The number of such pairs per program version ranges from about 1500 to about two million. The total number of such pairs over all programs exceeds 34 million.

While collecting traces, we observed that in some cases, spectra of successful and failing runs collided. That is, the spectra of some failing runs were indistinguishable from the spectra of some successful runs for the same version of the program. We observed 2888 such collisions with the coverage spectrum, and 1088 such collisions with the permutation spectrum. Naturally, all the runs that collide in the permutations spectrum also collide for the binary coverage spectrum. We chose to exclude all the failing runs with collisions in the binary coverage spectrum from our experiments, for three reasons. First, when two spectra collide, all the techniques we are concerned with will produce empty reports, and therefore those runs provide no comparison points. Second, any score we would assign to empty reports would be arbitrary since empty reports are obviously not good reports, but at least they would not mislead the programmer. Last, we expect the occurrence of collisions to be rare when using more elaborate spectra. Once we exclude all failing runs with collisions, there are no failing runs left for some programs. This left 109 programs that we actually used in the experiment.

7 Results

We are interested in the average behavior of each method. We cannot simply average all the scores that each method achieves for each failing run, because certain programs have many more runs than others, and the simple average would be skewed to reflect those programs more. Instead, we will average the scores in stages, obtaining a weighted average with every program version having the same weight.

Let us focus on a program version P, and let us call F its set of failing runs, S its set of successful runs and $score_U(f)$ the score of the union method, when it uses a failing run f. Then the score of the union method for P is the average of the scores of the reports based on each failing

Program	Description	Versions	LOC	#Tests	#Failed Tests	Versions with collisions only
print_tokens	lexical analyzer	7	565	4072	6–186	0
print_tokens2	lexical analyzer	10	510	4057	33–518	0
replace	pattern replacement	32	563	5542	3–309	1
schedule	priority scheduler	9	412	2627	7–293	1
schedule2	priority scheduler	10	307	2683	2–65	1
tcas	altitude separation	41	173	1592	1–131	18
tot_info	information measure	23	406	1026	3–251	2

Table 1. Overview of the Siemens suite.

Score	Inter.	Union	NN/Cov	NN/Perm
0-10%	108	101	31	19
10-20%	0	0	7	0
20-30%	0	0	15	2
30-40%	0	0	15	7
40-50%	0	0	10	4
50-60%	0	0	8	21
60-70%	0	0	4	15
70-80%	0	1	5	13
80-90%	0	1	9	10
90-99%	1	4	5	18
100%	0	2	0	0

Table 2. Distribution of scores per method.

run of that program:

$$score_U(P) = \frac{\sum_F score_U(f)}{|F|}$$

The formula for the intersection model is similar.

For the nearest neighbor techniques, the score of the technique depends not only on the failing run, but also on the nearest neighbor the technique picks to contrast with it. If there are multiple nearest neighbors the technique could uniformly pick any of them. Therefore, to compute the score for the failing run, we average over the scores obtained by selecting each neighbor. More formally, if each failing run f in the set of all failing runs F, has a set of nearest neighbors N_f, then the nearest neighbor score for the run is

$$score_{NN}(f) = \frac{\sum_{N_f} score(f, n)}{|N_f|}$$

Then the score for a program is defined as in the union and intersection cases.

7.1 Technique Performance

Table 2 shows the distribution of scores for the four techniques.

The intersection technique, gives an empty report and achieves a zero score for all programs bar one (version 9 of schedule), for which it gives an almost perfect report. The reason for the proliferation of empty reports is the following: our test suites assure program coverage. Therefore, every top-level condition in every program has to be true in some execution and false in some other. When it is true, the parts of the code that are in the false branch are excluded from the intersection. Conversely, when the condition is false, the parts of the code that are in the true branch are excluded from the intersection. Therefore, every line of code that is guarded by at least one top-level condition, cannot appear in the intersection. Note that this is dependent only on branch coverage of top-level conditional statements, which even a rudimentary testing suite would include. The only parts of the program that could appear in the intersection under top-level coverage are the ones that are not guarded by anything. But those statements will be always executed, even in the failing runs we examine. The intersection technique achieves a high score for version 9 of schedule for an interesting reason: the program ends prematurely (with a segmentation fault) producing an empty profile. The bug now is actually in the first executable line of the main function. Obviously, all successful runs execute that line; the failing run also executes it, but because of the segmentation fault, it is not reflected in the spectrum.

The union model did succeed in finding some bugs. The interesting thing about it though, is its almost bimodal behavior. The union model reports either nothing, or things very close to the bug. It depends on almost the reverse of what the intersection depends on: it has to be impossible to achieve full code coverage with only successful runs. This means that the bug has to be very well localized at the level of abstraction the spectra provide.

The average score for the nearest neighbor model was 56% with the permutation spectrum, and 35% with the coverage spectrum. The nearest neighbor models give us con-

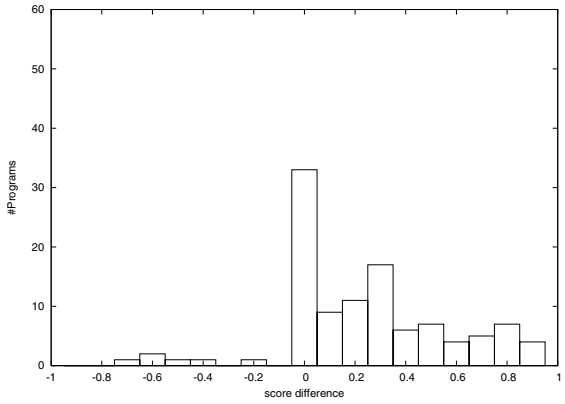

Figure 3. Distribution of the difference in scores between the nearest neighbor model (with the coverage spectrum) and the union model for our subject programs

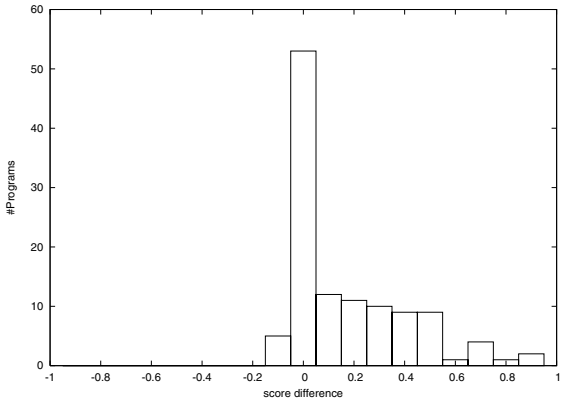

Figure 4. Distribution of the difference in scores between the permutation spectrum and the coverage spectrum, with the nearest neighbor model

sistently better results than the union and intersection models, even if the bugs are not found exactly. Nearest neighbors are not hindered by coverage issues. Having a large number of runs helps, because it is easier to find a close neighbor of failing run, but it is the existence of a close neighbor, not the number of runs that matters. This is an important property. For the union and intersection models the set of successful runs has to be large enough (to exclude all irrelevant parts of the program) but not too large, because then the successful runs shadow the bug. This is why previous work that is based on these models has to use slicing, introducing an a priori relevance of the spectra to the bug. Thanks to the nearest neighbor models we can answer affirmatively our first question, about the possibility of locating bugs with the given spectra.

7.2 Technique Comparison

The second question is how the nearest neighbor techniques perform in comparison with the union and intersection techniques. The comparison with the intersection technique is not very interesting; in the one case it locates the bug, the bug is also found by the nearest neighbor techniques (for the same reasons). On the other hand, the intersection technique produces mostly empty reports, so at least it does not mislead the user.

The question of the behavior of the nearest neighbors with respect to the union technique is a little harder to answer. The union technique gets a perfect score for two programs, and gets more scores above 90% than the nearest neighbor with the coverage spectrum. However, it achieves fewer scores above 80%. The distribution of the difference of scores between the union and the nearest neighbor method is shown in figure 3. The average difference is 27 points. The nearest neighbor does better in all cases but 7. Not surprisingly, 6 of these cases are the ones where union gets a score above 90%. However, it is also true that for those cases, the difference in scores is large, which means that the nearest neighbor is not performing well. Note though that this is the average nearest neighbor, and that among these cases, in two of them there is a particular run for which the nearest neighbor finds the bug.

Figure 4 shows a similar graph for the difference between the nearest neighbor with the coverage spectrum and the nearest neighbor with the permutation spectrum. The permutation spectrum performs considerably better. On average, it achieves a score 10 points higher. Still, there are few cases in which the simpler spectrum does better. The reason is that, in these cases the more complex spectrum, which is symmetric, gives us a few more nodes than just the faulty ones, and therefore the score is a little lower.

7.3 Nearest Neighbor vs. Random Selection

The third hypothesis we wanted to test requires that we evaluate the utility of selecting the nearest neighbor for comparison with the failing run as opposed to selecting a run randomly. Figure 5 shows the distribution of the differences of scores between using the nearest neighbor and using any run. That is, given a failing run, what score would we get if instead of going through the trouble of choosing the nearest neighbor, we chose some arbitrary successful run?

Surprisingly, in the simple spectrum case, the average run performs a bit better! The simple explanation we have is that the space that the coverage spectrum defines is too compact. Because of this, there is always a run that, when compared with our failing run, it will isolate the faulty line.

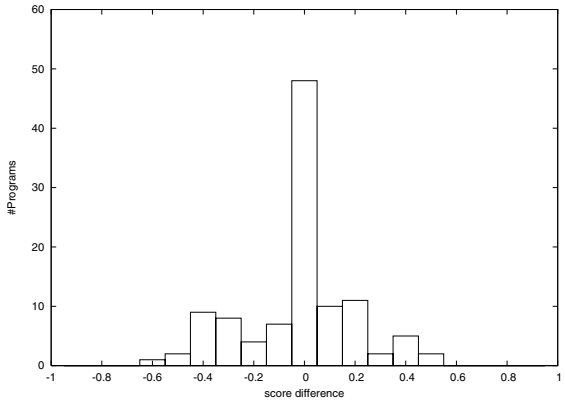

Figure 5. Distribution of the difference in scores between choosing the nearest neighbor and choosing any successful run uniformly (coverage spectrum)

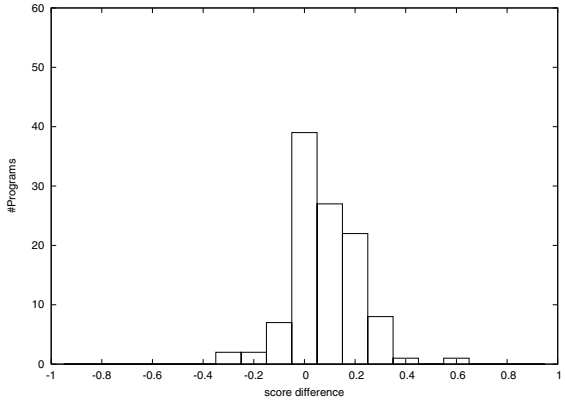

Figure 6. Distribution of the difference in scores between choosing the nearest neighbor and choosing any successful run uniformly (permutation spectrum)

If this intuition is correct, then enlarging the space, by using a more complete spectrum, should increase the difference from the average. Indeed, in figure 6 we show the distribution of the difference between the score of the nearest neighbor using the more complex spectrum and the average score of using any run, instead of the nearest neighbor. This difference is 10 points on average, suggesting that indeed, in more elaborate spaces, selecting the nearest neighbor significantly outperforms selecting a random run.

8 Discussion

We presented a technique for fault localization based on distances and differences between abstractions of program runs. We defined a generic architecture for fault localization systems based on *spectra* and *models*. We defined a novel evaluation strategy for fault localization systems, and we used it to compare four instantiations of the architecture: three based on coverage spectra (using a union, an intersection, and a nearest neighbor model), and one based on a new abstraction of profile data and the nearest neighbor model. Our experiments showed that the intersection model performs badly, the union model is bimodal, that the nearest neighbor model outperforms them on average, but it only becomes effective for the more elaborate spectra.

There are some threats to the validity of our results: the choice of programs, the choice of spectra, and the fact that we used our own evaluation strategy. The programs in the Siemens suite are relatively small, and the faults are injected; there is definitely need for more realistic experiments. The spectra we use are simplistic, as witnessed by the number of spectrum collisions, and there is no guarantee that the results will hold for more elaborate spectra. Still, we feel that the idea of using a single similar run to help isolate faults will hold with more elaborate spectra, as witnessed by previous research that exploited input structure. Perhaps the strongest threat is that the performance evaluation method is our own. However, its simplicity leads us to believe that there are no hidden biases in favor of the nearest neighbor model.

Our technique assumes the availability of a large number of runs. This need can be addressed by keeping the results of beta-testing, by a test case generator [18], or by driving a manual testing process [13], perhaps based on the fault localization report. We have claimed that our technique is independent of the input structure. If the test cases come from a test case generator, the advantage is smaller: the existence of a test generator assumes knowledge about the input structure, but our technique alleviates the need for a distance metric on inputs and it discharges the assumption that similar inputs lead to similar runs.

More static approaches to fault localization are possible: Engler et al. [7] discover code structure rules in programs similar to the subject program, and then check for violations of those rules.

Our architecture is simply a filtering architecture [6], and it provides no support for explaining the causes of the error. Existing work in model checking [2, 9] addresses this need for faults against a specification. Zeller's technique for isolating causes by manipulating two program runs [28] presumably works better when the runs are similar, and therefore could use our technique as a preparation step. Our technique also provides no hint as to which parts of the report are more likely to be faulty; Jones et al. [17] provide such a ranking, but their technique assumes multiple faulty runs, which is problematic in the case of multiple faults.

In future work, we would like to explore the use of more elaborate spectra. For example, recent work [10, 20, 22]

abstracts runs as collections of predicates or *invariants* [8] and we are investigating better models for them.

Acknowledgment

This research was supported by the National Science Foundation and a gift from Grammatech, Inc.

References

[1] P. Anderson and T. Teitelbaum. Software inspection using codesurfer. In *Proceedings of the 1st Workshop on Inspection in Software Engineering*, 2001.

[2] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 97–105, Jan. 2003.

[3] T. Y. Chen and Y. Y. Cheung. On program dicing. *Software Maintenance: Research and Experience*, 9(1):33–46, Feb. 1997.

[4] D. E. Critchlow. *Metric Methods for Analyzing Partially Ranked Data*, volume 34 of *Lecture Nodes in Statistics*. Springer-Verlag, 1985.

[5] W. Dickinson, D. Leon, and A. Podgurski. Pursuing failure: The distribution of program failures in a profile space. In *Proceedings of the Joint 8th European Software Engeneering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engineering*, pages 246–255, Sept. 2001.

[6] M. Ducassé. A pragmatic survey of automated debugging. In *Proceedings of the 1st Workshop on Automated and Algorithmic Debugging*, volume 749 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[7] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Symposium on Operating Systems Principles*, 2001.

[8] M. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, pages 213–225, 1999.

[9] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *Proceedings of the 10th International SPIN Workshop on Model Checking of Software*, volume 2648 of *Lecture Notes Computer Science*. Springer-Verlag, 2003.

[10] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, 2002.

[11] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability*, 10(3):171–194, 2000.

[12] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 234–245, 1990.

[13] S. Horwitz. Tool support for improving test coverage. In *Proceedings of the European Symposium on Programming*, April 2002.

[14] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *Proceedings of the 14th International Conference on Software Engineering*, pages 392–411, May 1992.

[15] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, May 1977.

[16] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, pages 191–200, May 1994.

[17] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of Test Information to Assist Fault Localization. In *Proceedings of the 24th International Conference on Software Engineering*, May 2002.

[18] B. Korel. Automated software test generation. *IEEE Transactions on Software Engingeering.*, 16(8):870, Aug. 1990.

[19] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system. `http://caml.inria.fr/`.

[20] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 141–154, 2003.

[21] H. Pan and E. H. Spafford. Heuristics for automatic localization of software faults. Technical Report SERC-TR-116-P, Purdue University, 1992.

[22] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss. Automated fault localization using potential invariants. In *Proceedings of the Workshop on Automated and Algorithmic Debugging*, 2003.

[23] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of the 6th European Software Engineering Conference*, pages 432–449, Sept. 1997.

[24] G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *Software Engineering*, 24(6):401–419, 1998.

[25] E. Y. Shapiro. *Algorithmic Program Debugging*. The ACM Distinguished Dissertation Series. The MIT Press, Cambridge, Massachusetts, 1983.

[26] M. Shaw and D. Garlan. *Software Architecture—Perspectives on an Emerging Discipline*. Prentice–Hall, 1996.

[27] D. B. Whalley. Automatic isolation of compiler errors. *ACM Transactions on Programming Languages and Systems*, 16(5):1648–1659, Sept. 1994.

[28] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering*, 2002.

[29] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.