SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2014; **00**:1–17

Preprint: This is a pre-peer reviewed version of a manuscript submitted to the Journal of Software Testing, Verification and Reliability.

Fault Localization for Automated Program Repair: Effectiveness and Performance

Fatmah Yousef Assiri*and James M. Bieman

Computer Science Department, Colorado State University, Fort Collins, CO 80523-1873, U.S.A.

SUMMARY

Automated program repair (APR) tools apply fault localization (FL) techniques to identify the locations of likely faults to be repaired. The effectiveness and performance of APR depends in part on the FL method used. If FL does not identify the location of a fault, the application of an APR tool will not be effective — it will fail to repair the fault. If FL assigns a faulty location a low priority for repair, the performance of APR will be reduced, increasing the time required to find a repair. In this paper, we evaluate the impact of five FL techniques (Jaccard, Optimal, Ochiai, Tarantula, and the GenProg Weighting Scheme) on the effectiveness and performance of a brute force APR tool when applied to faulty versions of the Siemens Suite and two other large programs: space and sed. All FL techniques were effective in identifying all faults except Optimal which failed to identify faulty locations in two faulty versions of the space subject program. We obtained the best APR performance when Optimal was used. However, Ochiai's performance was noteworthy since it always assigned faulty statements at an equal or higher priority for repair than other FL techniques with acceptable performance.
Copyright © 2014

Received . . .

KEY WORDS: automated program repair; brute-force; fault localization technique; effectiveness; performance

1. INTRODUCTION

Debugging is an expensive process [1] that includes locating software faults and fixing them. Automated program repair (APR) refers to techniques that locate and fix software faults automatically, which promises to dramatically reduce debugging costs.

APR techniques apply fault localization (FL) to guide a repair tool towards code segments that are more likely to contain faults. Then an APR tool can modify the code most likely to contain faults until a repair is found. FL techniques compute a suspiciousness score to indicate the likelihood that each statement contains a fault. A list of potentially faulty statements (LPFS), ordered by their suspiciousness, is created for use by the repair tool.

*Correspondence to: Fatmah Y. Assiri, CS Department, Colorado State University, Fort Collins, CO 80523-1873, U.S.A.
Email: fatmahya@cs.colostate.edu

Copyright © 2014

The effectiveness and performance of APR can be impacted by the selected FL technique. APR effectiveness is the ability to fix faults, while performance is the time or number of steps required to find a repair. An ineffective fault localization technique might mislead the repair process by missing the locations where a fault hides, assigning a low score to a faulty statement, or identifying too many statements that might contain the faults. Missing faulty locations will lower APR effectiveness by failing to find a repair. Assigning a low score to a faulty statement will not decrease APR effectiveness, but it will reduce APR performance since the APR will unproductively modify many fault-free statements before reaching the faulty statement. On the other hand, identifying too many locations might improve APR effectiveness by increasing the chance of finding a repair, but with potentially poor performance. In the worst case, a fault localization technique can mark all statements in a program as potentially faulty locations, which can decrease the performance of APR dramatically especially with large programs. Le Goues et al. [2] found that for APR "time is governed" by the number of potentially faulty locations rather than program size. Thus, a fault localization technique that marks fewer statements, and/or places the faulty statement at the head of the LPFS will decrease the number of variants generated by an APR technique until a repair is found, thus improving APR performance.

Different FL techniques have been used with APR to locate potential faults. Weimer et al. [3, 4] apply a simple *Weighting Scheme* that assigns weight values to statements based on their execution by passing and failing tests. Higher weight is assigned to statements that are executed only by failing tests, and lower weights are assigned to statements that are executed by both passing and failing tests. They excluded statements that are only executed by passing tests to prevent changing correct statements. Debroy and Wong [5], and Nguyenet et al. [6] use the *Tarantula* fault localization technique [7, 8, 9] to rank program statements based on their likelihood of containing faults. Using better fault localization techniques can identify faulty statements more quickly, thus improving APR effectiveness and performance.

The effectiveness of different fault localization techniques on automated program repairs has been evaluated by Qi et al. [10]. Their evaluation used GenProg, an APR tool that implements a random search algorithm (a genetic algorithm) to find the optimal solution from the solution space. The randomness of the search algorithm might affect the accuracy of the reported results since there is dependency between the accuracy of the fault localization technique and the randomness of the search algorithm. The FL technique might accurately locate a faulty statement, but the search algorithm can select mutation operators that do not fix the fault. A brute-force algorithm is guaranteed to fix a fault if a repair is possible. Thus, to evaluate the effect of FL on APR performance and effectiveness, we evaluate FL techniques with an APR that applies a brute-force search algorithm.

We studied the impact of four well known fault localization techniques: Jaccard, Optimal, Ochiai, and Tarantula. We also used the Weighting Scheme employed by GenProg as a baseline. Our evaluation was conducted on six subject programs that include large programs (more than 14K LOC). Our results showed that Optimal improved APR performance but it failed to identify faulty statements in two of the trials, which decreased APR effectiveness. All four fault localization techniques improved APR performance compared to the Weighting Scheme used by GenProg. Optimal and Tarantula assigned low priority (place them far from the head of the LPFS) to faulty statements in 6 trials, and Jaccard assigned low priority to faulty statements in 3 trials. Ochiai always

Copyright © 2014

Prepared using stvrauth.cls
DOI: 10.1002/stvr

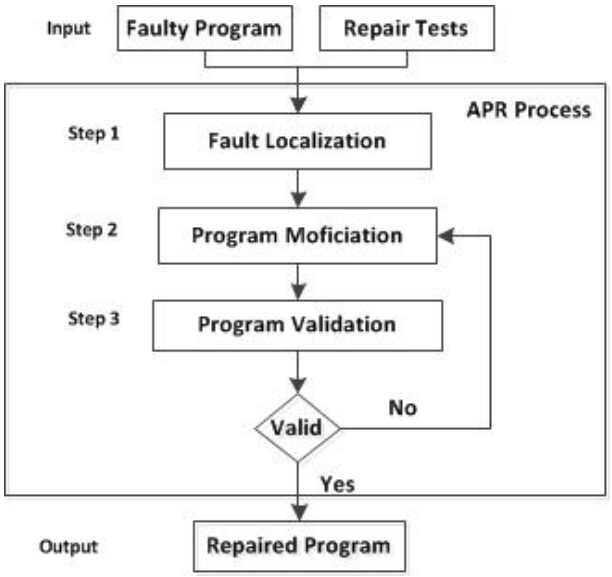

Figure 1. Overall Automated Program Repair (APR) Process

assigned the same priority or higher (near the head of the LPFS) for the faulty statements compared to other FL techniques, and it did not degrade the performance.

The main contributions of this paper are the following:

- A framework for comparing FL techniques in terms of their impact on the performance and effectiveness of automated program repair. The MUT-APR evaluation framework is built by adapting GenProg to (1) fix operator faults, (2) use a brute-force search, and (3) apply four additional FL techniques.

- A determination of the impact of FL techniques on an APR tool's ability to repair a faulty statement. We found that Optimal decreased APR effectiveness compared to the other four alternatives.

- Our results show that the differences in performance between the GenProg Weighting Scheme and the four alternative FL techniques (Jaccard, Optimal, Ochiai, and Tarantula) are significant, while differences between the alternatives are not significant.

2. BACKGROUND

2.1. Automated Program Repair (APR)

An automatic program repair process takes a faulty program and set of repair tests, and produces a repaired program. Figure 1 describes the overall organization and activities of APR techniques. An APR technique consists of three main steps: fault localization (step 1), program modification (step 2), and variant validation (step 3).

Copyright © 2014

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table I. The dynamic behavior of the faulty program *gcd* when executed against tests in $T1$, ..., $T5$. *Sus. Score* is the suspiciousness score computed using Tarantula.

Stmt ID	Stmt	T_1	T_2	T_3	T_4	T_5	Sus. Score
	gcd (int a, int b) {						
1	if(a < 0) //fault					✓	1.0
2	{ printf("%g \n ", b) ;						0.0
3	return 0 ; }						0.0
4	while(b ! = 0)	✓	✓	✓	✓	✓	0.5
5	if(a > b)	✓	✓		✓	✓	0.57
6	a = a − b ;		✓		✓		0.0
7	else	✓	✓		✓	✓	0.57
8	b = b − a ;	✓	✓		✓	✓	0.57
9	printf("%g \n ", a) ;	✓	✓	✓	✓		0.0
10	return 0 ;	✓	✓	✓	✓		0.0
	}						

Faulty locations that are more likely to contain a fault are identified by a fault localization technique. A faulty program is modified by using a set of mutation operators that change the code in the faulty locations to generate a new copy of the faulty program, which is called a variant or patch. Some APR techniques generate a variant from a variant produced in prior iterations. The variant is validated by executing it against a set of repair tests, regression tests, or formal specifications. The variant is called a *repair* if it passes all repair tests. The repair process stops when it finds a validated variant producing a repaired program, or when the predefined parameters have reached their limits.

2.2. *Fault Localization Techniques (FL)*

Fault localization techniques were introduced in order to guide developers towards the most suspicious statements to check during debugging. Lately, FL techniques are employed by APR to guide search algorithms towards statements that are more likely to hide faults than other non faulty statements (step 1 in Figure 1). Thus, applying FL helps to fix faults faster without breaking other required functionality.

FL techniques locate potentially faulty locations in the source code by computing a *suspiciousness score* for each statement that indicates its likelihood of containing a fault. Then, statements are ordered based on their suspiciousness. Developers can use the suspiciousness score to order their search for a fault to debug.

Spectrum-based fault localization (SBFL) [11, 12, 13, 8, 14, 15, 16] is a common approach that compares the program behavior of a passing execution to a failing execution. SBFL collects information on the dynamic behavior of program statements when they are executed against each test in a test suite. SBFL methods record the number of passing and failing tests executed for each statement, and compute the suspiciousness score for each statement. Statements that are executed more during a failing run are considered to be more likely to contain faults, thus are assigned a higher suspiciousness score than other statements in the program. Many heuristics have been proposed to compute statement suspiciousness scores [11, 7, 8, 17, 15, 15, 18].

To illustrate how FL techniques rank program statements using the computed suspiciousness scores, we used the C program in Table I, which computes the Eculid's greatest common divisor.

FL techniques record the dynamic behavior of program statements when a program is executed against a set of tests by counting the number of passing and failing tests for each statement. Each

Copyright © 2014

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table II. List of Potentially Faulty Statements (LPFS) in format used by APR tool

Statement ID	Suspiciousness score
1	1.0
5	0.57
7	0.57
8	0.57
4	0.5

FL technique uses a formula to compute suspiciousness scores. This example uses five test inputs $TS = T_1, T_2, T_3, T_4, T_5$ in which T_1, T_2, T_3, and T_4 are passing tests, and T_5 is a failing test, and Tarantula computes suspiciousness score for each statement. A list of potentially faulty statement (LPFS), which consists of statement IDs and their suspiciousness scores is created and sorted (Table II). The LPFS contains all statements with a suspiciousness score greater than zero, and will be used by the APR tool.

3. BRUTE-FORCE APR

APR processes apply a search algorithm to select a mutation operator from a pool of mutation operators to modify a suspicious statement (step 2 in Figure 1). A brute-force search algorithm applies all possible changes to the program until a repair is found. In contrast, a genetic algorithm applies mutation and crossover operators to modify a faulty program. A genetic algorithm randomly selects mutation operators from a pool of mutation operators, and a crossover operator combines changes from two parent variants to generate a new child variant. A genetic algorithm does not guarantee a repair due to its randomness.

In this study, we apply a brute-force APR process to eliminate the randomness of a genetic algorithm, and to guarantee a repair when the FL technique identifies the faulty statements and the repair is supported by the set of mutation operators.

For each mutable statement (mutable statement is a program construct that can be changed by one of the supported mutation operators) from the LPFS, brute-force applies all possible mutation operators. Mutation operators are applied in a predefined order. If no repair is found by changing the first potentially faulty statement to all its possible alternatives (w.r.t. the set of mutation operators that are supported by the APR tool), the next statement is modified and so forth. A brute-force search algorithm runs until a repair is found or all possible combinations are executed.

Algorithm 1 describes how the brute-force algorithm fixes faults. It produces a variant (repaired program) from a faulty program, an LPFS produced by an FL technique, and a maximum fitness value that determines if a repair is found or not. A variant is a copy of the faulty program with one modification. The algorithm modifies statements sequentially. It takes the first statement in the LPFS, and checks if it contains an operator (line 4). If the operator is mutable, it applies all possible alternatives (line 5-10). Each change creates a variant (line 7). The fitness value is computed for each generated variant (line 8) by executing it against the repair tests (one of the inputs to the repair process in Figure 1). If a variant that passes all repair tests is found (in other words, the variant has a fitness value equal to the maximum fitness value), a repair is found. If not, the process continues

Copyright © 2014

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Algorithm 1 Brute-Force Pseudocode

```
1:  Inputs: Program P , List of Potentially Faulty Statements LPFS, and maximum fitness value
2:  Output: Variant
3:  for i=0 to length(LPFS)-1 do
4:    let stmtOp = checkOp(stmti),
5:    for all  mutation operators mOp for stmtOp do
6:      repeat
7:        let variant= apply(stmti, stmtOp,mOp)
8:        let variant_fitness= computeFitness (variant)
9:      until variant_fitness= maximum fitness || mOp is the last mutation operator for stmti
10:   end for
11:   if i != last index in the LPFS then
12:     i++
13:   end if
14: end for
15: return variant
```

with the next statement in the LPFS until a repair is found (line 9) or the algorithm reaches the last statement in the LPFS without a repair.

The brute-force search algorithm guarantees a repair if the fault is related to one of the mutation operators supported by APR technique, but it can be infeasible with large programs. A good FL technique assigns higher weight to the faulty statement (places it near the head of the LPFS). Then, APR modifies the most suspicious statements before modifying other non-faulty statements. Thus it finds a repair by modifying fewer statements. A good FL technique improves performance by decreasing the number of variants generated and total time required to find a repair.

Using a brute-force APR process allow us to accurately measure the impact of the FL technique since a repair is guaranteed. We conduct an evaluation to measure the impact of FL techniques on the brute-force APR process.

4. STUDY AND METHOD

In this section we report the results of evaluating the impact of five FL techniques on the performance and effectiveness of brute-force APR using six C programs of different sizes. We used many faulty versions for each subject program. Nineteen faulty versions with a total of 22,298 lines of code are used in our evaluation.

4.1. Research Questions and Evaluation Metrics

Our evaluation study is designed to answer the following research questions:

RQ1: What is the relative APR effectiveness when different FL techniques are employed?

APR effectiveness is the ability to fix faults. We compare the effectiveness of an APR tool when using five different FL techniques to identify a faulty location. An FL technique that successfully determines faulty locations improves APR effectiveness. On the other hand, an FL technique that

Copyright © 2014

Prepared using stvrauth.cls
DOI: 10.1002/stvr

fails to identify faulty locations limits APR effectiveness.

RQ2: Which FL technique assigns higher priority to faulty statements?

We are concerned with the accuracy of FL techniques in identifying faulty statements. We measure the priority of a statement by its position in the LPFS produced by the FL technique. The position of a statement in the LPFS is its LPFS rank. Statements with higher suspiciousness scores are placed near the head of the list, thus have a lower LPFS rank compared to other statements. For example, in Table II statement ID 1 has the lowest (best) LPFS rank (LPFS rank =1) since it has the highest suspiciousness score. On the other hand, statement ID 4 has the highest (worst) LPFS rank (LPFS rank = 5).

We compared the LPFS rank of the faulty statement using the LPFS that is created by each FL technique. An FL technique that assigns higher suspiciousness score to the faulty statement, placing it near the head of the LPFS (lower LPFS rank), improves APR performance compared to another FL technique that places the faulty statement far from the head of the LPFS (higher LPFS rank).

RQ3: How does the use of different FL techniques affect the number of generated variants (NGV) until a repair is found?

NGV, defined by Qi et al. [10], measures the number of generated variants until a repair (a variant that passes all repair tests) is found. We compared NGV when applying a brute-force APR using different FL techniques (lower NGV is better). An FL technique that assigns a low LPFS rank to a faulty statement requires fewer statements to be modified, thus creating fewer variants (reducing NGV).

The difference between the LPFS rank metric (used for RQ2) and NGV is that the LPFS rank metric is totally dependent on the FL technique; however, NGV can be influenced by the number of mutable statements with lower LPFS ranks than the faulty statement. For example, consider the use of two FL techniques (FL1 and FL2) to identify a faulty statement in the *gcd* program. FL1 creates List1 (Table III), and FL2 creates List2 (Table IV). Both techniques assign the same LPFS rank for the faulty statement (the faulty statement, statement 1, in both lists has a rank = 3). However, NGV depends on the number of mutable statements prior to the faulty statement. List1 consists of two statements prior to the faulty statement (statement 2 and 3) but neither can be mutated by MUT-APR mutation operators; thus NGV can be equal to any value between 1 and 5 (depending on the order of the application of alternative mutation operators that transform faulty operator $<$ into the correct one $==$). On the other hand, List2 consists of two mutable statements prior to the faulty statement (statement 5 and 8), thus NGV can be equal to any value between 2 and 10.

RQ4: Does the use of different FL techniques affect the total time required to find a repair?

We computed the total time required to find a repair. Total time includes the total time needed to generate a new variant, compile and execute each generated variant on the repair tests, and compute its fitness values. We compared the total time, measured in seconds, when each FL technique is used.

Copyright © 2014

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table III. List1: List of Potentially Faulty Statements (LPFS) for *gcd* created by FL1

Statement ID	Suspiciousness score
2	0.96
3	0.91
1	0.80
5	0.71
7	0.68
8	0.57
4	0.5

Table IV. List2: List of Potentially Faulty Statements (LPFS) for *gcd* created by FL2

Statement ID	Suspiciousness score
5	0.96
8	0.91
1	0.80
2	0.72
3	0.6
4	0.5
10	0.5

Table V. Benchmark programs. Each *Program* is an original program from the SIR [19]. *LOC* is the number of lines of codes. *#Faulty Versions* is the number of faulty versions. *Average # Repair Tests* is the average number of repair tests for each faulty version.

Program	LOC	# Faulty Versions	Average # Repair Tests
tcas	173	4	6.4
replace	564	5	19
schedule2	374	2	9
tot_info	565	4	8
space	6195	2	38.4
sed	14427	2	28
Total	22298	19	108.9

4.2. Subject Programs

To evaluate our approach, we used six C programs from the Software artifacts Infrastructure Repository (SIR) [19] along with a comprehensive set of test inputs. We used the Siemens Suites: tcas, replace, schedule2, and tot_info. We also used two larger programs: space and sed. Subject programs have sizes ranging from 173 to 14K lines of code; each program is seeded with a single fault. We used multiple faulty versions for each subject program. Faulty versions are taken from the SIR. We also used Proteum/IM 2.0 [20], which is a C mutation tool, to create additional faulty versions. Our study includes nineteen faulty versions. Table V identifies the subject programs along with their size, the number of faulty versions (after removing equivalent mutants), and the average size of repair tests.

4.3. Repair Tests

One of the inputs to an APR tool is a set of repair tests. We selected a set of repair tests that have at least one failing test, and one passing test. Failing tests execute faults, and passing tests protect

Copyright © 2014

Prepared using stvrauth.cls
DOI: 10.1002/stvr

program functionality. Repair tests for the Siemens Suites are taken from the SIR repository. Test suites for the large programs provided by the SIR contain too many test inputs, which will slow the APR process, since repair tests are used to validate each generated variant. We created repair tests for each large program containing at least one failing test and 20 passing tests following Qi et al.[10]. In addition, a study by Abreu et al. [15] found that fault localization techniques give a stable behavior when no less than 20 test inputs are used. To validate our results, we repeated the study for each faulty version using five different repair tests that are selected/created randomly using test data provided by the SIR except for two versions of tcas program (tcas-v5 and tcas-25), which used three and one repair tests, respectively, since there are no other test suites that execute faults. The average number of repair tests can be found in the last column of Table V.

4.4. Fault Localization Techniques

We used the FL technique employed by GenProg [3, 4, 21, 2], called the *Weighting Scheme*, as a baseline. We compared the results of using the Weighting Scheme to the four other fault localization techniques: Jaccard, Ochiai, Optimal, and Tarantula. We selected these four FL techniques for the following reasons:

1. Ochiai was identified as a highly effective FL technique from a developer point of view [15, 22].
2. Jaccard was identified as the most effective FL technique with GenProg [10].
3. Tarantula was used in prior work with APR techniques [5, 6].
4. Optimal is one of the recent proposed heuristics which was found to be more effective than Ochiai and Jaccard from a developer point of view [18].

We used the implementation of the Weighting Scheme from the GenProg framework, and we implemented Jaccard, Ochiai, Optimal, and Tarantula using about 300 LOC of OCaml code.

4.5. Automated Program Repair Tool

To fix faults, we used our MUT-APR repair tool [23]. MUT-APR was build by adapting the GenProg version 1 framework [3, 4, 21, 2]. Unlike GenProg which makes use of existing code in the subject program to repair faults, MUT-APR applies a set of mutation operators that construct new operators to replace faulty ones within a genetic algorithm. For this paper, we replaced the genetic algorithm in MUT-APR with a brute-force search algorithm, and added support for the use of different FL techniques applying the changes developed by Qi et al. [10].

4.6. Study Design

For each faulty version, we used each FL technique to create an LPFS. Then the list is used by MUT-APR to find a repair. We executed each FL technique five times on each faulty version with five different repair tests except for tcas. On one version of tcas we used three different repair tests, and on the other version of tcas we used one repair test. In total, our evaluation includes 79 trials (faulty versions $\times$ number of test suites for each faulty version). We compute the average value of our measurements for each faulty version across the multiple test suites. Our experiments were

Copyright © 2014

Prepared using stvrauth.cls
DOI: 10.1002/stvr

conducted on a fedora Linux machine with an Intel Xeon R 2.67GHz CPU and 7.7 GB memory size.

5. RESULTS

We compared the values of each individual subject program using our measurements. Then, we compared the mean values of each metric (LPFS rank, NGV, and time) to measure the impact when using each FL technique. We excluded the trials in which at least one FL technique did not identify faulty statement.

Then, to measure the statistical difference we applied Mixed Model [24], which is preferable over ANOVA, to study the effect of many measurements using the same subject program. We studied the difference at the 0.95 confidence level.

RQ1: Relative APR Effectiveness

To compare APR effectiveness when different FL techniques are used, we studied the ability of FL techniques to identify faulty statements. If an FL technique failed to identity faulty location, APR will fail to repair the faults. The four FL techniques successfully identified faulty statements for all faulty versions except that Optimal failed to identity the faulty locations for two trials of the space program.

All FL techniques failed to identify faulty statement in two faulty versions of replace (version 25 and version 31). We checked these two versions of replace to investigate why FL techniques failed to identify faulty statements. In both versions, faults were in an *if* statement nested inside a *switch* statement. When the program is executed against the failing tests, the *if* statement is checked and it returns false. Thus, the execution jumps to the *default* statement, which caused failures. However, the faulty *if* statements were not recorded as one of the executed statements. We are not certain of the reason; it might be a problem in the code that creates the instrumented version of faulty programs which records coverage information.

RQ2: Which FL Technique Ranked Faulty Statements lowest?

We evaluated the accuracy of FL techniques in identifying faulty statements by comparing the LPFS rank of the faulty statement produced by each FL technique. We excluded the two versions of the replace program in which the faulty statements were not identified. We compared the LPFS rank of faulty statements using seventeen faulty versions (79 trials).

First, we compared the LPFS rank of faulty statements when using Jaccard, Optimal, Ochiai and Tarantula to that of the Weighting Scheme. We found that in 12 out of 79 (15.18%) of the trials the Weighting Scheme assigned the lowest LPFS rank to faulty statements, but in 67 out of 79 (84.8%) of the trials Jaccard, Optimal, Ochiai and Tarantula assigned the lowest LPFS rank to faulty statements.

Then, we compared the behavior of FL techniques when using different test suites for each faulty version. We compared the LPFS ranks given by Jaccard, Optimal, Ochiai and Tarantula, since they

Copyright © 2014

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table VI. Result summary of running Jaccard, Optimal, Ochiai, Tarantula and the Weighting scheme on each faulty version. *Measurement* is the metric used for comparison.

Measurement		Jaccard	Optimal	Ochiai	Tarantula	Weighting Scheme
LPFS Rank	Mean	26.6	25.3	26.1	25.9	144.4
	Median	24	23	24	24	55
	Std Err	4.4	4.4	4.4	4.4	29.8
NGV	Mean	36.6	34.4	35.4	35.6	211.8
	Std Err	17.9	17.9	17.9	17.9	39.8
Time	Mean	73.5	27.9	71.8	66.9	502.8
	Std Err	67.3	63.5	67.6	65.5	204.5

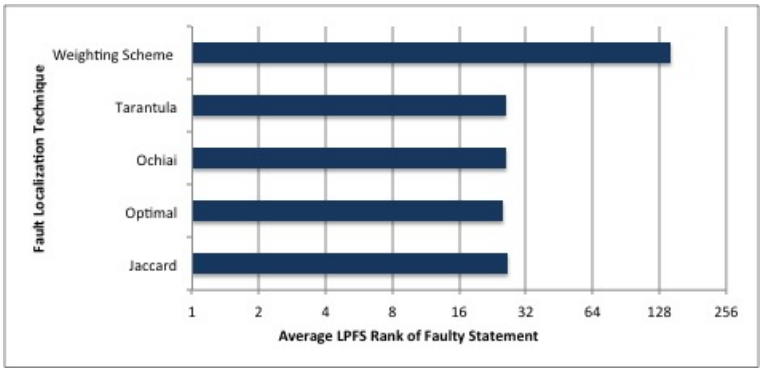

Figure 2. Average LPFS Rank for each FL Technique across all trials

have similar behavior. For the small subject programs (58 trials), all four FL techniques assigned the same LPFS rank to the faulty statements across the five test suites used for each faulty version. For replace version 5 and tot_info versions 7 and 13, all four FL techniques performed equally by assigning the same LPFS rank for faulty statements in most (89.66%) of the trials. In 6 out of 58 (10.34%) of the trials Optimal and Tarantula assigned the lowest (best) LPFS rank to faulty statements compared to Jaccard and Ochiai, but in these trails Ochiai performed better than Jaccard by assigning a lower LPFS rank to faulty statements. In 3 out of 58 (5.17%) trials, Jaccard assigned the highest (worst) LPFS rank to faulty statements compared to Optimal, Ochiai and Tarantula.

For the space program, Optimal failed to identify faulty statements in two trials (one trial for each faulty version), but it assigned the same or lowest LPFS rank to faulty statements compared to Jaccard, Ochiai and Tarantula in the other trials. For the sed program, Optimal assigned the same or lowest (best) LPFS rank to faulty statements compared to other FL techniques.

We compared the average LPFS rank for all FL techniques (Table VI). On average, Jaccard, Optimal, Ochiai and Tarantula assigned lower (better) LPFS ranks to faulty statements than the Weighting Scheme. The average LPFS ranks of Optimal and Tarantula is slightly better than Jaccard and Ochiai. Figure 2 graphically shows the average LPFS ranks of faulty statements for each FL technique across all trials. Since LPFS rank is an ordinal measure, we also compared the median. Optimal assigned a lower LPFS rank to faulty statements, but the difference is just one position compared to Jaccard, Ochiai and Tarantula.

In summary, all four FL techniques assigned lower (better) LPFS ranks to faulty statements compared to the Weighting Scheme, and the difference is significant between the Weighting

Copyright © 2014

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table VII. P-values of applying the Mixed Model for LPFS Rank, NGV, and time Metrics at 0.95 confidence level.

Methods	LPFS Rank	NGV	Time
Jaccard-Optimal	0.508	0.941	0.595
Jaccard-Ochiai	0.802	0.980	0.985
Jaccard-Tarantula	0.740	0.979	0.939
Jaccard-Weighting Scheme	0.0001	0.0001	0.046
Optimal-Ochiai	0.679	0.966	0.609
Optimal-Tarantula	0.743	0.969	0.643
Optimal-Weighting Scheme	< 0.0001	< 0.0001	0.027
Ochiai-Tarantula	0.934	0.996	0.955
Ochiai-Weighting Scheme	0.0001	< 0.0001	0.046
Tarantula-Weighting Scheme	0.0001	< 0.0001	0.043

Scheme and all four alternative FL techniques (Jaccard, Optimal, Ochiai, and Tarantula) at the 0.95 confidence level. In two trials Optimal failed to identify faulty statements. Optimal and Tarantula had the worst performance in 6 trials, and Jaccard was the worst in 3 trials. Although Ochiai was not the best FL technique on average, it always identified faulty statements, and it never assigned the highest (worst) LPFS rank to faulty statements across all trials. The differences are not significant between Jaccard, Optimal, Ochiai, and Tarantula (p-values are shown in Table VII).

RQ3: Number of Generated Variants (NGV)

In order to repair faults, APR tools apply mutation operators to modify a location in the faulty program generating variants. Modifying a non-faulty location generates an invalid variant (a variant that does not pass all repair tests). If the faulty statement is placed earlier in the LPFS, then the APR will change the faulty statement and produce fewer invalid variants.

We compared the number of generated variants (NGV) until a repair is found to determine the most effective FL technique for use in APR. An FL technique that gives lower NGV is better since it decreases the chances of producing invalid variants before finding the correct one. First, we compared the NGV of Jaccard, Optimal, Ochiai and Tarantula to that of the Weighting Scheme. In 5 out of 79 (6%) trials the Weighting Scheme decreased NGV compared to other four FL techniques, and in 4 out of 79 (5%) trials the Weighting Scheme generated the same NGV as other FL techniques. The Weighting Scheme increased the NGV in 68 out of 79 trials (86.1%).

Then we compared the NGV for Jaccard, Optimal, Ochiai and Tarantula. For the small benchmarks (59 trials), all FL techniques generated the same NGV except in 4 out of 59 (6.78%) trials in which Jaccard generated more variants (high NGV) until a repair is found. For space and sed programs, in 6 out of 20 (30%) trials Optimal decreased NGV, and in one out of 20 (5%) trials Jaccard decreased NGV. In 11 out of 20 trials (55%) Jaccard, Optimal, Ochiai, and Tarantula generated the same number of variants until a repair is found.

Table VI shows the average NGV for each FL technique across all trials. NGV is similar for all techniques except the Weighting Scheme which generated an average of 211 variants. Figure 3 shows the average NGV for each technique. The difference is significant between the Weighting Scheme and all alternative FL techniques at the 0.95 confidence level (Table VII).

Copyright © 2014

Prepared using stvrauth.cls
DOI: 10.1002/stvr

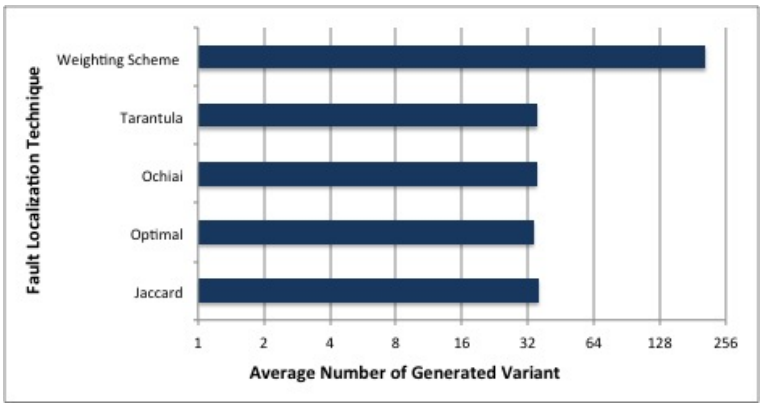

Figure 3. Average Number of Generated Variants (NGV) for each FL Technique across all trials

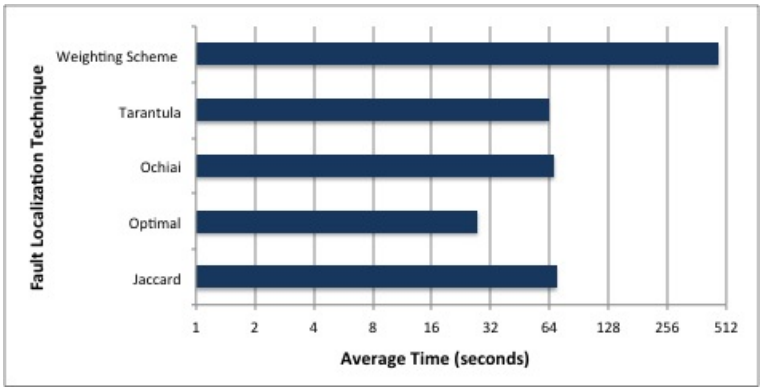

Figure 4. Average Time for each FL Technique across all trials

RQ4: Total Time

We compared the total time required until a repair is found when each FL technique is used to generate the LPFS. We compared the average time for each FL technique. Table VI shows that Optimal required the shortest average time (27.9 seconds) to repair faults. Jaccard, Ochiai and Tarantula required similar times (69.5, 67.8, 63.3 seconds respectively). The Weighing Scheme required on average 502.8 seconds to find a repair. Figure 4 shows the average time required to repair faults when each FL technique is used to identify faulty statements. The time improvement of Jaccard, Optimal, Ochiai, and Tarantula over the Weighting Scheme (Table VII) is significant at the 0.95 confidence level.

6. DISCUSSION

We evaluated the impact of different FL localization techniques on the effectiveness and performance of APR techniques. We compared APR performance when different FL techniques are used in terms of three metrics: LPFS rank, NGV, and time. We studied the relationship between these metrics. There is a high correlation (0.949, 0.872, 0.985, 0.986, and 0.988 for Jaccard, Optimal, Ochiai, Tarantula and Weighting Scheme respectively) between the LPFS rank of faulty statements

Copyright © 2014

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table VIII. Correlation results between performance metrics: LPFS Rank, Number of Generated Variants (NGV), and Time of Jaccard, Optimla, Ochiai, Tarantula and the Weighting Scheme.

Correlation	Jaccard	Optimal	Ochiai	Tarantula	Weighting Scheme
LPFS Rank-NGV	0.949	0.872	0.985	0.986	0.988
LPFS Rank-Time	0.050	0.332	0.277	0.342	0.784
NGV-Time	0.110	0.492	0.309	0.375	0.776

and the NGV until a repair is found (Table VIII). FL techniques that assigned lower LPFS ranks to faulty statements decreased the NGV until a repair is found. Decreasing the NGV improved APR performance in terms of time.

The correlation between time and other two metrics: LPFS rank and NGV is weak. This might be due to the influence of other factors that affect the time to find repair such as the number of repair tests, and the compilation and execution time of variants. In addition, time can be affected by the resources used to run the experiments.

7. THREATS TO VALIDITY

Our evaluation studied the impact of different FL techniques on the performance and effectiveness of automated program repair. To mitigate threats to internal validity, we applied a brute-force APR to eliminate the randomness of the search algorithm used in a previous study [10]. We also selected many faulty versions of each subject program.

External validity relates to the ability to generalize the results. To mitigate external threats, our evaluation consists of programs of different sizes including two large C programs (more than 14K LOC). However, our results might not be generalized to other fault types or to programs from other domains.

The accuracy of FL techniques depends in part on the test inputs used to identify faulty statements, which is a threat to construct validity. To mitigate this threat we used five different sets of repair tests (test inputs). For the small programs, we selected test suites randomly from the set of suites provided by the SIR. For the large programs, we created independent repair tests for each faulty version with no less than 20 passing test cases to achieve the best accuracy as reported in [15].

Conclusion validity is another threat to our results. We applies statistical tests to measure the consistency of the results across all faulty versions, and to measure the statistical difference. To limit the threats to the conclusion validity, we used the same trials (combination of faulty versions and repair tests) with all FL techniques. In addition, we ensured randomness in the experimental setting when selecting/creating tests inputs.

Copyright © 2014

Prepared using stvrauth.cls
DOI: 10.1002/stvr

8. RELATED WORK

Fault localization tools are introduced to decrease the cost of finding faults and improving software quality. These tools are based on the spectrum-based fault localization (SBFL) techniques to identify faulty locations in source code. The Tarantula tool [9] was developed to locate faults in C programs, and AMPLE [13] is an Eclipse plug-in for object-oriented software. Ochiai is the formula that is used in the molecular biology domain [25], and Jaccard is the formula used in the Pinpoint tool [26].

FL techniques have been evaluated in terms of their accuracy to locate faults. Abreu et al. [27] evaluated the effectiveness of four FL techniques (Pinpoint, Ochiai, Tarantula and AMPLE) in terms of the LPFS rank of the faulty statement. Experiments conducted on the Siemens Suites and space program showed that the Ochiai FL technique is the most effective of those evaluated. Abreu et al. [15, 28] studied the impact of the quality and the quantity of passing and failing tests on the accuracy of FL techniques. They found that Ochiai, Jaccard, and Tarantula provide accurate results with low quality tests that include only 1% of failing tests that propagate faults to the outputs. They studied the impact of the number of passing and failing tests on the accuracy of FL techniques, and found that adding more failing tests will always improve the accuracy of FL techniques. Adding more than 6 failing tests has a minimal affect but it does not lower the accuracy of fault diagnosis. On the other hand, adding more than 20 passing tests decreased the accuracy of FL techniques. Lee et al. [29] and Naish et al. [18] conducted a more comprehensive study to compare the accuracy of the formulas used to locate faults. Naish et al. also proposed two *Optimal* metrics to locate faults, which outperform other metrics. The latest study by Xie et al. [30] performed a theoretical analysis to evaluate FL techniques, and found that the *Optimal* metrics by Naish et al. have similar behaviors.

Lately, research has been directed toward automated program repair (APR) techniques to reduce debugging costs. GenProg is a well known APR technique developed by Weimer and his colleagues [3, 4, 21, 2]. It uses a genetic programming algorithm to fix faults automatically in C programs. GenProg can fix a variety of faults including segmentation faults and infinite loops. Arcuri [31, 32] proposed an approach and tool, called JAFF, that use genetic programming for automatic bug fixing for Java programs. Ackling et al. [33] developed *py*EDB tool to automate repairs of Python software. SemFix [6] is a tool for fixing faults through semantic analysis; Debroy and Wong [5] applied a brute-force search method to repair faults using first order mutation operators. Wei et al. [34] developed a tool, called *AutoFix-E*, to automate fault fixing in Eiffel programs equipped with contracts. Kim et al. [35] describe the Pattern-based Automatic program Repair tool (*PAR*), which repairs faults by generating patches using fix patterns. Ten patterns are created based on patches commonly written by humans.

APR techniques must locate faults in order to fix them. Existing FL techniques are employed by APR tools to locate and fix faults automatically. Debroy and Wong [5], and Nguyen et al. [6] use Tarantula heuristics to identify faulty statements. GenProg uses a basic Weighting Scheme to locate faults, and some APR techniques [23, 35] apply the Weighting Scheme following the GenProg approach. Only one prior study evaluates the effectiveness of different FL techniques employed by APR techniques [10]. However, this work evaluates the effectiveness of FL techniques using the GenProg random search algorithm, which can affect the accuracy of the results.

Copyright © 2014

Prepared using stvrauth.cls
DOI: 10.1002/stvr

9. CONCLUSION AND FUTURE WORK

Fault localization (FL) techniques are employed by APR tools to reduce the number of potentially faulty locations to be modified in order to find a repair. FL techniques that identify faulty statements will improve APR effectiveness. An FL technique that places faulty statements at the head of the list of potentially faulty statements (LPFS), will improve APR performance since fewer variants will be generated until a repair is found, thus decreasing the time required to fix faults.

Our evaluation shows that Optimal is the least effective and the Weighting Scheme has the lowest performance of the FL techniques studied. However, Ochiai never failed to identify faulty statements or assigned the worst (highest) LPFS rank to faulty statements compared to the other FL techniques.

APR performance is measured in terms of LPFS rank, NGV, and time. LPFS Rank and NGV are strongly correlated, and they are not platform dependent compared to the time metric.

Our results contribute towards improving effectiveness and performance of APR techniques. We provide a framework for evaluating alternative FL techniques along with an evaluation of these techniques. We plan to use the framework to study different search algorithm and identify heuristics to improve the performance of APR techniques without decreasing their effectiveness.

ACKNOWLEDGMENT

This project was supported by the Ministry of Higher Education, Saudi Arabia. The authors would like to thank Westley Weimer and his research group for sharing GenProg, and Yuhua Qi et al. [10] for sharing the GenProg-FL tool.

REFERENCES

1. Hailpern B, Santhanam P. Software debugging, testing, and verification. *IBM Systems Journal* 2002; **41**(1):4–12.
2. Le Goues C, Nguyen T, Forrest S, Weimer W. GenProg: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on* 2012; **38**(1):54 –72.
3. Forrest S, Nguyen T, Weimer W, Le Goues C. A genetic programming approach to automated software repair. *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, GECCO '09, ACM: New York, NY, USA, 2009; 947–954.
4. Weimer W, Nguyen T, Le Goues C, Forrest S. Automatically finding patches using genetic programming. *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, IEEE Computer Society: Washington, DC, USA, 2009; 364–374.
5. Debroy V, Wong WE. Using mutation to automatically suggest fixes for faulty programs. *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, 2010; 65 –74.
6. Nguyen HDT, Qi D, Roychoudhury A, Chandra S. Semfix: Program repair via semantic analysis. *Proceedings of the 2013 International Conference on Software Engineering*, IEEE Press, 2013; 772–781.
7. Jones JA, Harrold MJ, Stasko JT. Visualization for fault localization. *Proceedings of ICSE 2001 Workshop on Software Visualization, Toronto, Ontario, Canada*, Citeseer, 2001; 71–75.
8. Jones JA, Harrold MJ, Stasko J. Visualization of test information to assist fault localization. *Proceedings of the 24th international conference on Software engineering*, ACM, 2002; 467–477.
9. Jones JA, Harrold MJ. Empirical evaluation of the Tarantula automatic fault-localization technique. *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, ACM: New York, NY, USA, 2005; 273–282.

Copyright © 2014

Prepared using stvrauth.cls
DOI: 10.1002/stvr

10. Qi Y, Mao X, Lei Y, Wang C. Using automated program repair for evaluating the effectiveness of fault localization techniques. *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, ACM: New York, NY, USA, 2013; 191–201.
11. Agrawal H, Horgan JR, London S, Wong WE. Fault localization using execution slices and dataflow tests. *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, IEEE, 1995; 143–151.
12. Baah GK, Podgurski A, Harrold MJ. The probabilistic program dependence graph and its application to fault diagnosis. *Software Engineering, IEEE Transactions on* 2010; **36**(4):528–545.
13. Dallmeier V, Lindig C, Zeller A. Lightweight defect localization for Java. *ECOOP 2005-Object-Oriented Programming*. Springer, 2005; 528–550.
14. Renieres M, Reiss SP. Fault localization with nearest neighbor queries. *Proceedings. 18th IEEE International Conference on Automated Software Engineering*, IEEE, 2003; 30–39.
15. Abreu R, Zoeteweij P, Van Gemund AJ. On the accuracy of spectrum-based fault localization. *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, IEEE, 2007; 89–98.
16. Chilimbi TM, Liblit B, Mehra K, Nori AV, Vaswani K. Holmes: Effective statistical debugging via efficient path profiling. *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, IEEE, 2009; 34–44.
17. Renieres M, Reiss SP. Fault localization with nearest neighbor queries. *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, IEEE, 2003; 30–39.
18. Naish L, Lee HJ, Ramamohanarao K. A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.* Aug 2011; **20**(3):11:1–11:32.
19. Software-artifact infrastructure repository. http://sir.unl.edu/php/previewfiles.php.
20. Delamaro ME, Maldonado JC. Proteum/im 2.0: An integrated mutation testing environment. *Mutation testing for the new century*. Kluwer Academic Publishers: Norwell, MA, USA, 2001; 91–101.
21. Weimer W, Forrest S, Le Goues C, Nguyen T. Automatic program repair with evolutionary computation. *Commun. ACM* may 2010; **53**(5):109–116.
22. Lo D, Jiang L, Budi A. Comprehensive evaluation of association measures for fault localization. *Software Maintenance (ICSM), 2010 IEEE International Conference on*, IEEE, 2010; 1–10.
23. Assiri FY, Bieman JM. An assessment of the quality of automated program operator repair. *Proceedings of the 2014 ICST Conference*, ICST '14, 2014.
24. Brown H, Prescott R. *Applied mixed models in medicine*. John Wiley & Sons, 2006.
25. Meyer AdS, Garcia AAF, Souza APd, Souza Jr CLd. Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (zea mays l). *Genetics and Molecular Biology* 2004; **27**(1):83–91.
26. Chen MY, Kiciman E, Fratkin E, Fox A, Brewer E. Pinpoint: Problem determination in large, dynamic internet services. *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, IEEE, 2002; 595–604.
27. Abreu R, Zoeteweij P, Van Gemund AJ. An evaluation of similarity coefficients for software fault localization. *Dependable Computing, 2006. PRDC'06. 12th Pacific Rim International Symposium on*, IEEE, 2006; 39–46.
28. Abreu R, Zoeteweij P, Golsteijn R, Van Gemund AJ. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software* 2009; **82**(11):1780–1792.
29. Lee HJ, Naish L, Ramamohanarao K. Study of the relationship of bug consistency with respect to performance of spectra metrics. *Computer Science and Information Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference on*, 2009; 501–508.
30. Xie X, Chen TY, Kuo FC, Xu B. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Trans. Softw. Eng. Methodol.* Oct 2013; **22**(4):31:1–31:40.
31. Arcuri A. On the automation of fixing software bugs. *Companion of the 30th international conference on Software engineering*, ICSE Companion '08, ACM: New York, NY, USA, 2008; 1003–1006.
32. Arcuri A. Evolutionary repair of faulty software. *Applied Soft Computing* 2011; **11**(4):3494 – 3514.
33. Ackling T, Alexander B, Grunert I. Evolving patches for software repair. *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, GECCO '11, ACM: New York, NY, USA, 2011; 1427–1434.
34. Wei Y, Pei Y, Furia CA, Silva LS, Buchholz S, Meyer B, Zeller A. Automated fixing of programs with contracts. *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, ACM: New York, NY, USA, 2010; 61–72.
35. Kim D, Nam J, Song J, Kim S. Automatic patch generation learned from human-written patches. *Proceedings of the 2013 International Conference on Software Engineering*, IEEE Press, 2013; 802–811.

Copyright © 2014
Prepared using stvrauth.cls

DOI: 10.1002/stvr