Spectral Debugging with Weights and Incremental Ranking

Lee Naish
University of Melbourne
Melbourne, Australia
Email: lee@csse.unimelb.edu.au

Hua Jie Lee
University of Melbourne
Melbourne, Australia
Email: leehj@csse.unimelb.edu.au

Kotagiri Ramamohanarao
University of Melbourne
Melbourne, Australia
Email: rao@csse.unimelb.edu.au

***Abstract*—Software faults can be diagnosed using program spectra. The program spectra considered here provide information about which statements are executed in each one of a set of test cases. This information is used to compute a value for each statement which indicates how likely it is to be buggy, and the statements are ranked according to these values. We present two improvements to this method. First, we associate varying *weights* with failed test cases — test cases which execute fewer statements are given more weight and have more influence on the ranking. This generally improves diagnosis accuracy, with little additional cost. Second, the ranking is computed incrementally. After the top-ranked statement is identified, the weights are adjusted in order to compute the rest of the ranking. This further improves accuracy. The cost is more significant, but not prohibitive.**

***Keywords*-software fault diagnosis; spectral debugging; weights; incremental ranking**

***Keywords*-software fault diagnosis; spectral debugging; weights; incremental ranking**

I. Introduction

Despite the achievements made in software development, bugs are still pervasive and diagnosis of software failures remains an active research area. One of many useful sources of data to help diagnosis is the dynamic behaviour of software as it is executed over a set of test cases. Software can be instrumented automatically to gather data such as the statements that are executed in each test case. A summary of this data, often called program spectra, can be used to rank the parts of the program according to how likely it is they contain a bug. Ranking is done by sorting based on the value of a numeric function (we use the term *ranking metric* or simply metric) applied to the data for each part of the program. We make the following contributions to this area:

1) We propose a method of assigning weights to failed tests which leads to more informative tests having more influence on the ranking. Existing formulas for ranking metrics can be used, but they are generalised so the inputs are real numbers instead of integers.
2) We propose a method of incrementally producing the ranking. After the top-ranked statement is determined, the weights are adjusted in order to find the next highest ranked statement, and so on.

Table I
EXAMPLE PROGRAM SPECTRA WITH TESTS $T_1 \ldots T_5$

	T_1	T_2	T_3	T_4	T_5	a_{np}	a_{nf}	a_{ep}	a_{ef}
Stat_1	1	1	1	0	1	1	0	1	3
Stat_2	1	1	0	1	0	1	1	1	2
Stat_3	1	0	1	1	1	0	1	2	2
Stat_4	0	1	0	0	1	1	2	1	1
				⋮					
Result	1	1	1	0	0				

3) We evaluate performance of several metrics using the conventional method and our proposed methods. Some metrics have not previously been evaluated for software error diagnosis. Others have been systematically evaluated, but only on programs with a single bug; here we include evaluation for programs with more than one bug.

The rest of the paper is organized as follows. Section II provides the necessary background on spectra-based diagnosis. Sections III and IV describe our methods for varying weights of failed tests and generating the ranking incrementally, respectively. Section V reports performance results. Section VI discusses related work and in Section VII we conclude.

II. Background

A program spectrum is a collection of data that provides a specific view of the dynamic behavior of software. It contains information about the parts of a program that were executed during the execution of several test cases. In general the parts could be individual statements, basic blocks, branches or larger regions such as functions. Here we use individual statements. This is equivalent to considering basic blocks, assuming normal termination (a statement within a basic block is executed if and only if the whole basic block is executed). During execution of each test case, data is collected indicating the statements that are executed. Additionally, each test case is classified as passed or failed.

For each statement, four numbers are ultimately produced. They are the number of passed/failed test cases in which the statement was/wasn't executed. Adapting Abreu et al. [1] we use the notation $\langle a_{np}, a_{nf}, a_{ep}, a_{ef} \rangle$, where the first

part of the subscript indicates whether the statement was executed (e) or not (n) and the second indicates whether the test passed (p) or failed (f). We use superscripts to indicate the statement number where appropriate. For example, a_{ep}^3 is the number of passed tests that executed statement 3. The raw data can be presented as a matrix of binary numbers $e_{s,t}$, with one row for each program statement, s, and one column for each test case, t, where each cell indicates whether a particular statement is executed (the value is 1) or not (the value is 0) for a particular test case. Additionally, there is a (binary) vector indicating the result (0 for pass and 1 for fail) of each test case. Here we assume the failed tests appear first, numbered 1 to F, the total number of tests is T and the number of statements is S. This data allows us to compute the a_{ij} values, $i \in \{n, e\}$ and $j \in \{p, f\}$. Table I gives an example with five tests, the first three of which fail.

Applying a function that maps the four a_{ij} values to a single number (we call such functions *ranking metrics*) for each statement allows us to rank the statements: those with the highest metric values are considered the most likely to be buggy. We would expect buggy statements to generally have relatively high a_{ef} values and relatively low a_{ep}. In the example in Table I, statement 1 is used in all failed tests and the lowest number of passed tests and thus has the highest value for all sensible metrics. Statement 2 has the next highest ranking since it is used in the next highest number of failed tests and is also used in the lowest number of passed tests. The best ordering of statements 3 and 4 is less clear — statement 3 is used in more failed tests but also more passed tests. Some metrics which perform well overall rank statement 3 higher, others rank statement 4 higher and others give them equal rank. Another way of viewing the ranking is that rows of the matrix are ranked according to how "similar" they are to the result vector. Many clustering and classification problems can be formalised in this way, and many ranking metrics come from areas other than software diagnosis.

Diagnosis can proceed by the programmer examining statements starting from the top-ranked statement until a buggy statement is found. In reality, programmers are likely to modify the ranking due to their own understanding of whether the code is likely to be buggy, based on other information such as static analysis, the history of software changes, *et cetera*, and checking correctness generally cannot be done by a single statement (or even one basic block) at a time. Evaluation of different ranking methods generally ignores such refinements and just depends on where the bug(s) appear in the ranking.

The metrics we use in this paper are defined in Table II. In the experiments we report on we have also used many other metrics (most are described in more depth and evaluated for software diagnosis in [2]) but for reasons of space we restrict the number here. We include metrics which were originally proposed for software diagnosis: Tarantula (a graphical debugger [3], [4]), Zoltar [5], Wong3 (the best of several metrics proposed in [6]), O^p (the best of several metrics proposed in [2]), and Ample2 (the better of the two generalisations of the metric used in the AMPLE system [7] which were evaluated in [2]). We also include all other metrics which perform particularly well in any of our experiments: McConnaughey (McCon) [8] (the best performing metrics in our multiple-bug experiments, originally developed for the study of plankton communities), Kulczynski2 [9], Fager [10] and Pearson (from [11]). Finally, we include two particularly well known metrics: Jaccard [12] (used in the PINPOINT Java debugger [13], originally developed for the study of plant communities) and Ochiai [14] (the best metric for debugging evaluated in [1], originally developed for the study of fish, also known as Cosine).

There is no single metric which is best in all situations. The O^p metric has been shown optimal with respect to a simple performance measure for a class of simple single-bug programs [2]. Wong3 and Zoltar are close to optimal and these three metrics out-perform other metrics on realistic single-bug programs using more reasonable measures of performance. It seems likely that no substantially better metrics exist for the single-bug case. However, we have little understanding of the (more realistic) multiple bug case: performance of previously proposed metrics have not been systematically evaluated and it is unknown whether there are substantially better metrics. One contribution we make here is to evaluate performance of all these metrics on programs with more than one bug (the McCon and Fager metrics have not previously been evaluated at all for debugging). However, our main contributions are methods which can be applied with *all* such metrics.

III. Varying weights for failed tests

The idea behind varying weights comes from the following observations. First, some failed tests provide more information than other failed tests. Consider the extreme example of two failed tests, where one executes almost every statement and the other executes only one statement. The first test gives us little information whereas the second test allows us to conclude with certainty that the executed statement is buggy. By only using the number of failed tests in which a statement is executed, this information is lost. Second, although ranking metrics are normally applied to natural numbers, they can be defined and used on real numbers. With varying weights for failed tests the a_{ef} and a_{nf} values we compute can be any real number between zero and the number of failed tests. Of the dozens of metrics proposed, none use integer-specific operations such as modulo; no adaptation is needed for use with non-integral a_{ef} and a_{nf} values.

The weights we use depend on the set of *suspect* (possibly buggy) statements, B. Initially this is the statements executed in at least one failed test: $\{s|\exists t, t \leq F, e_{s,t} = 1\}$.

Table II
DEFINITIONS OF NEW RANKING METRICS USED

Name	Formula	Name	Formula	Name	Formula
Kulczynski2	$\frac{1}{2}\left(\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ef}}{a_{ef}+a_{ep}}\right)$	Zoltar	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}+\frac{10000a_{nf}a_{ep}}{a_{ef}}}$	Jaccard	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}}$
Ochiai	$\frac{a_{ef}}{\sqrt{(a_{ef}+a_{nf})(a_{ef}+a_{ep})}}$	Fager	$\frac{a_{ef}}{\sqrt{(a_{ef}+a_{nf})(a_{ef}+a_{ep})}}-\frac{1}{2\sqrt{a_{ef}+a_{ep}}}$	Ample2	$\frac{a_{ef}}{a_{ef}+a_{nf}}-\frac{a_{ep}}{a_{ep}+a_{np}}$
Tarantula	$\frac{\frac{a_{ef}}{a_{ef}+a_{nf}}}{\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ep}}{a_{ep}+a_{np}}}$	Pearson	$\frac{(a_{ef}a_{np})-(a_{nf}a_{ep})}{\sqrt{(a_{ef}+a_{nf})(a_{ef}+a_{ep})(a_{nf}+a_{np})(a_{ep}+a_{np})}}$	O^p	$a_{ef}-\frac{a_{ep}}{a_{ep}+a_{np}+1}$
Wong3	$a_{ef}-h$, where $h=\begin{cases} a_{ep} & \text{if } a_{ep}\le 2 \\ 2+0.1(a_{ep}-2) & \text{if } 2<a_{ep}\le 10 \\ 2.8+0.001(a_{ep}-10) & \text{if } a_{ep}>10 \end{cases}$			McCon	$\frac{{a_{ef}}^2-a_{nf}a_{ep}}{(a_{ef}+a_{nf})(a_{ef}+a_{ep})}$

In the incremental ranking method described later, this is refined. We first define a relative weight, w_t, for each test t, which is (almost) inversely proportional to the number of suspect statements executed in the test minus one:

$$w_t = \frac{1}{\sum_{s\in B} e_{s,t} - 1 + \epsilon}$$

We use a small constant, ϵ (we use 0.0001 in our code), to avoid division by zero when there is only one suspect statement executed. The weight given to a failed test t is $w_t F/W$, where F is the number of failed tests and W is the sum of the relative weights w_t over all failed tests: $\sum_{t=1}^{F} w_t$. The total weight of all failed tests is thus unaffected by the weighting, so the overall impact of the failed tests (compared to passed tests) is not changed. Also, if all failed tests execute the same number of statements, all weights are 1 (and the method is equivalent to the conventional method). To compute a_{ef} for a statement we take the sum of the weights of failed tests in which it was executed, rather than simply the number of tests:

$$a^s_{ef} = \sum_{e_{s,t}=1} w_t F/W$$

For example, with the data of Table I, if we ignore ϵ, the relative weights for tests 1–3 are 1/2, 1/2 and 1, respectively, and the weights are 0.75, 0.75 and 1.5, respectively. The a_{ef} values for statements 1–4 are 3, 1.5, 2.25 and 0.75, respectively. The a_{nf} values can be computed using a similar weighted sum (over the failed tests where the statement is not executed), or we can simply use the total number of failed tests minus a_{ef}. In this example, the a_{nf} values are therefore 0, 1.5, 0.75 and 2.25, respectively. The greater weight for test 3 leads to a^3_{ef} being greater than a^2_{ef} and for most of the better metrics, this is sufficient to raise the ranking of statement 3 above that of statement 2.

In general, the weightings result in more rational behaviour than the conventional method. If there is one failed test which executes a single statement s, that test will have relative weight of $1/\epsilon$ and a weight close to one whereas other weights will be close to zero. Thus a^s_{ef} will be close to F and a_{ef} for other statements will be close to zero. The better ranking metrics will rank s highest (it is as if s is used in all the failed tests and no other statement is used in any failed test). Note that it is not rational to apply weighting to passed tests in the same way. If a passed test executes a single statement we cannot conclude the statement is correct, since buggy statements sometimes produce correct results. The weighted method we propose is not just a generic method for determining "similarity" of a row of the matrix with the result vector — it uses knowledge of the software diagnosis domain.

Using weights does not change the algorithmic complexity of ranking. If the whole matrix of execution data is stored, computing the unweighted a_{ij} values takes $\Theta(ST)$, processing one statement at a time. The weighted method requires additional processing of each failed test to determine the weights, which takes $O(ST)$. The unweighted a_{ij} values can be computed without storing the whole matrix, resulting in $\Theta(S)$ space complexity instead of $\Theta(ST)$, by simply keeping accumulators for each value and processing one test at a time. The weighted values can be computed with the same space complexity. After each failed test is run the relative weight can be computed. Accumulators for relative weights can be maintained as well as an accumulator for the number of failed tests. After the last test is processed, each accumulator of relative weights can be multiplied by the total number of failed tests. Having computed the a_{ij} values, the metric values can be determined and then sorted in $O(S \log S)$ time to give the ranking, giving an overall complexity of $O(ST + S \log S)$ time.

IV. INCREMENTAL RANKING

The model of debugging suggested by ranking statements (by whatever method) is that the top-ranked statement is considered first and the second-ranked statement is only considered when the programmer has established the top-ranked statement is correct. Although naive, this model is typically assumed when evaluating performance of ranking methods. It also allows us to use additional information to

refine the ranking, generating it incrementally. Once the top-ranked statement has been decided, the other statements can be ranked under the *assumption* that the top-ranked statement is *not* buggy. With a probabilistic approach, for example, the statement with the highest probability of being buggy would be ranked highest. The next highest ranking would be the statement with the highest probability of being buggy *given that* the top-ranked statement is correct, and so on.

Generating a ranking incrementally in this way is not a particularly novel technique. However, it cannot refine traditional (unweighted) spectral ranking because there is no method of incorporating the additional information: the metric value returned for a statement does not depend on information about other statements. In contrast, the weighted method introduced above depends on the set of suspect statements. By no longer considering the top-ranked statement to be suspect, the number of suspect statements executed in a test decreases and the weights change, potentially changing the ranking. For example, with the data of Table I, having computed the metric values using the weighted method, statement 1 is ranked highest with all sensible metrics. If it is then not considered suspect, the number of suspect statements executing tests 1–3 are 2, 2 and 1, respectively. Thus test 3 has a relative weight of $1/\epsilon$ and statement 3 (the only remaining suspect statement executed in test 3) is ranked highest by all sensible metrics. Note this ranking is different (and more rational) than the ranking produced by the traditional unweighted method discussed in Section II.

The algorithm is described in Figure 1. It uses the matrix and result vector as inputs and outputs the ranking, a sequence of statements. It terminates when all statements executed in some failed test have been ranked. This ensures a buggy statement appears in the ranking, and prevents weights from becoming negative. In our example, it terminates after ranking statements 1 and 3 — one of these must be buggy so it is not necessary to consider the other two statements at all in our search for a bug. When several statements have the maximal metric value an arbitrary one is chosen (our implementation picks one pseudo-randomly).

The time taken to find the top-ranked statement is $\Theta(ST)$, and the time taken to produce the entire ranking is $O(S^2T)$. This is a substantial increase in overall CPU time. However, it is possible to compute the lower parts of the ranking in parallel with manually checking correctness of the higher ranked statements (though this does constrain the user interface in a debugging tool). The manual checking is likely to take more time so the main bottle-neck is determining the top-ranked statement, which has acceptable complexity. Space is another additional cost. Because the weights depend on which statements are suspect, and this changes during execution, the matrix (for failed tests at least) is required. If the matrix is stored in main memory the space complexity is $\Theta(ST)$. The alternative is re-reading it from disk in each iteration, reducing space complexity to $\Theta(S)$ but slowing the algorithm by a substantial constant factor. Overall, although the decrease in efficiency is significant, it does not make the algorithm infeasible.

One way to reduce the CPU time is to use the incremental method to rank only some statements, and rank the remaining statements using the simpler weighted method (with the remaining set of suspect statements). In Section V we provide performance figures for ranking the top 10% of statements incrementally. Another variation is to generate the ranking "bottom-up" — starting with the lowest ranked statement. This requires the whole ranking to be produced before manual checking starts and our experiments (not reported here) suggest the performance is poorer than the "top-down" method.

V. Experiments

We now discuss two sets of experiments performed to evaluate the effectiveness of our algorithms. We compare the traditional (unweighted) method with our weighted method and our iterative method (used to rank all statements or just the top 10%).

A. Benchmarks

We used two well-known benchmark sets, the Siemens Test Suite and "Unix" (a collection of Unix utilities) [15] which have been used in previous bug localization work [1], [4], [16], [2]. These benchmarks contain multiple buggy versions of several small C programs. In [2] Space (a significantly larger C program) is used, along with the Siemens Test Suite. The relative performance of the different metrics was similar in both benchmarks sets, though the absolute performance was substantially better for Space. The compiler that we used was GCC Version 4.2.1 and the Gcov tool was used to extract runtime statistics to compute the program spectra. Gcov generates statistics for each line of the program (including blank lines, *et cetera*); we ignored lines that were never executed. All experiments were carried out on a Pentium 4 PC running Ubuntu 6.06.

The first set of experiments we report on are for programs with a single bug defined as follows (from [2]): a bug is a statement that, when executed, has unintended behaviour. For such programs, the buggy statement must be executed in every failed test (this was used in the design of O^p and is another example of asymmetry between passed and failed tests). Programs with multiple bugs according to this definition were eliminated from the benchmark set, along with programs which had runtime errors (which prevent Gcov returning statistics) and programs for which there were no failed test cases. Table III lists the programs (the first seven are the Siemens Test Suite programs; the rest are from Unix), the number of versions of each in our single-bug ("1 Bug") experiments and the number of test cases used.

```
incremental_ranking:
    input: program spectra (binary matrix of execution data,
           binary vector of test results)
    output: ranking (sequence of statements)

    ranking = empty
    suspects = {all statements used in a failed test}
    repeat
        compute failed test weights using spectra and suspects
        compute a_ij and metric values for each statement
        s = a statement with maximum metric value
        append s to ranking
        suspects = suspects \ s
    until there is a test that executes no suspect statements
```

Figure 1. Incremental ranking algorithm

Table III
NUMBER OF PROGRAM VERSIONS IN BENCHMARKS

Program	1 Bug	2 Bugs	Test Cases
$tcas$	37	604	1608
tot_info	23	244	1052
$schedule$	8	—	2650
$schedule2$	9	29	2710
$print\_tokens$	6	—	4130
$print\_tokens2$	10	10	4115
$replace$	29	34	5542
Col	28	147	156
Cal	18	115	162
$Uniq$	14	14	431
$Spline$	13	20	700
$Checkeq$	18	56	332
Tr	11	17	870
TOTAL	224	1290	24458

Unfortunately, there are no good established multiple bug program benchmarks. We generated programs with exactly two bugs by taking pairs of single-bug versions of the same program. We used single-bug programs where the bug was in a single line of source code and eliminated versions with runtime errors and those with no failed test cases, as before. Table III summarises this "2 Bugs" benchmark set. Although the benchmark set can clearly be improved, it gives us a reasonable indication of what happens when we move away from the single-bug case.

B. Performance measure

The performance measure we use is *rank percentages*, used in various studies [1], [6], [2]. This is the rank of the highest ranked buggy statement, expressed as a percentage of the program size. For example, if the program has 200 lines of code and the highest ranked bug is the tenth in the ranking, the rank percentage is 5%. If there are several statements with equal rank, including the bug, we use the average of the ranks. Because our incremental method uses pseudo-random numbers to break ties, we ran each of these experiments twenty times and took the average as the result. The average rank percentages over all runs of all versions of all programs are reported.

C. Hypothesis

Our main hypothesis is that the proposed incremental method gives better performance (in terms of average rank percentages) than the traditional (unweighted) method. We test this hypothesis separately for each of the metrics considered. We provide additional figures so the main sources of performance gain can be determined.

D. Results for single-bug programs

Table IV gives average rank percentages for single-bug programs using the traditional (unweighted) ranking method, our weighted method, the incremental algorithm for the top 10% of the ranking (and weighted method for the remainder), "10% Inc.", and the incremental algorithm for the entire ranking. The relative performance of metrics for the unweighted method is consistent with that observed in [2], except that Wong3 is (slightly) better than Zoltar. All metrics show a small increase in performance for the weighted method, with the exception of O^p, which has a very slight decrease. There is also a small increase in performance for the iterative method for all metrics except Tarantula. The 10% incremental method performs between the weighted and incremental methods in most cases, as expected, but there are several exceptions. Overall, the results give only weak support for our hypothesis in the single bug case, but we can say with reasonable confidence that performance does not significantly degrade in this case.

The proof of optimality of O^p in [2] used a simple model program and a performance measure which depended only on how often the bug was ranked top. It was conjectured that ranking methods which use the whole matrix (rather than just the a_{ij} values) could not improve on O^p for this case, and our weighted and incremental methods do not improve the results in this case. However, it was noted that the matrix contains more information than the a_{ij} values and for more realistic single-bug programs it may be possible to improve performance (measured in a more reasonable way). Our

results here show a very small increase in performance but it is not statistically significant. However, we are continuing a more theoretical investigation.

Table IV
AVERAGE RANK PERCENTAGES FOR SINGLE-BUG PROGRAMS

Metric	Unweighted	Weighted	10% Inc.	Incremental
O^p	17.86	17.87	17.73	17.76
Wong3	18.19	18.06	18.00	17.80
Zoltar	18.23	18.22	18.15	18.10
Kulczynski2	19.06	18.91	18.86	19.03
McCon	19.06	18.91	18.80	19.05
Fager	21.26	20.81	20.75	20.49
Ochiai	21.63	21.18	21.20	21.09
Pearson	23.56	23.27	23.24	23.00
Jaccard	23.64	23.09	23.16	23.06
Ample2	24.08	23.65	23.64	23.75
Tarantula	27.17	26.56	27.43	27.28

E. Results for two-bug programs

Table V
AVERAGE RANK PERCENTAGES FOR TWO-BUG PROGRAMS

Metric	Unweighted	Weighted	10% Inc.	Incremental
Kulczynski2	19.53	19.35	17.98	17.86
McCon	19.53	19.35	18.11	17.96
Fager	20.01	19.57	18.17	18.15
Ochiai	20.17	19.62	18.20	18.18
Zoltar	20.52	20.46	19.09	19.20
Pearson	21.08	20.26	18.86	18.82
Jaccard	21.10	20.38	19.03	18.99
Ample2	21.37	20.59	19.28	19.22
Wong3	22.54	22.44	21.07	20.97
O^p	22.84	22.72	21.42	21.34
Tarantula	23.34	22.44	21.21	21.20

Table V gives average rank percentages for two-bug programs. There is a consistent increase in performance for all metrics from unweighted, through weighted and 10% incremental to incremental, except for Zoltar, for which 10% incremental has slightly better performance than incremental. The more consistent pattern apparent with 10% incremental, compared to the single-bug figures, may be partly due to the larger number of programs in the benchmark set. However, both the weighted and incremental methods do appear to increase performance significantly more for two-bug programs than for single-bug programs. Overall there is good support for our hypothesis in the two-bug case.

The relative performance of metrics is quite different to the single-bug case. O^p and Wong3 appear to be over-optimised for the single-bug case and do not perform particularly well for two-bug program. Zoltar performs rather better, even though the performance of these three metrics is very similar for single-bug programs. Kulczynski2 and McConnaughey perform well for both single and two-bug programs and may be a good compromise. Tarantula performs poorly. The absolute performance is better than the single-bug case for some metrics (particularly the poorer ones) because with more bugs, the top-ranked bug is likely to be higher, even with random ranking. However, the best performance in the single-bug case is better than the best performance in the two-bug case. This may be due to the fact that more effort has been put into developing metrics for single-bug programs, but in some sense the single bug case is a simpler problem, which may lead to better optimal performance.

F. Threats to Validity

In this section, we briefly discuss the potential threats to the validity of our conclusions above. Since the performance of the methods depend on the programs used, we have applied the Wilcoxon Rank-Sum test [17] to check the statistical significance of the increased performance of the incremental method compared with the unweighted method. The increased performance was significant with a confidence interval of greater than 95% for all metrics in the two-bug case (this is the normal threshold for accepting a hypothesis). In most of the single bug experiments this threshold was not achieved.

We should actually be somewhat more conservative in our statistical tests. There are two sources of variation — the choice of buggy program and the psuedo-random numbers used to break ties in the iterative method — and these should not be combined. For example, although we performed $224 \times 20 = 4480$ single-bug runs for each metric, if the statistical tests assumed there were 4480 programs (instead of 224), the confidence would be greatly exaggerated. However, the way we perfomed the tests may also exaggerate the confidence if we happened to be "lucky" with the pseudo-random numbers. Ideally, we should use more repititions and/or use more refined statistical tests.

The size of the programs is clearly too small to draw strong conclusions about typical programs, though in [2] it is shown that performance of the unweighted method for single-bug programs scales well (for the 9059 line Space program the average rank percentage was 1.62, compared with 16.00 for the Siemens Test Suite). We also have some tentative results showing improved performance with the incremental method on Space. Conclusions for multiple bug programs must also be tentative due to the limitations of our benchmark set.

VI. OTHER RELATED WORK

Instead of using information on statement executions, some systems use information on *predicates* (such as conditions of if statements) being executed and evaluating to true or false. The CBI (Collaborative Bug Isolation) system [18] gathers such information and ranks predicates using a metric, essentially the same as the unweighted method. Our weighted and incremental methods seem easy to adapt to such a system.

The SOBER system [19] also ranks predicates but uses a non-binary matrix containing frequency counts (for example,

how often a predicate evaluated to true in a test). The number of times a predicate is true in each passed test forms a distribution; similarly for each failed test. Ranking is based on the *evaluation bias* which is an established statistical measure of difference between the two distributions for each predicate. A direct comparison with the spectral approach we have investigated is difficult. However, we know there are asymmetries in the way passed and failed tests are best dealt with and an understanding of this may help improve systems such as SOBER. We are considering methods which use a non-binary matrix (execution counts of statements) to affect the weight of passed tests.

The work of [20] on "parallel debugging" attempts to cluster failed test cases, so the clusters correspond to different bugs. Such techniques may be help improve performance of spectral diagnosis. They generate a benchmark set with versions of the Space program with eight bugs (generated from the single-bug versions). We use essentially the same technique (but have two bugs and use Siemens Test Suite and Unix programs).

VII. Conclusion

We have proposed two enhancements to the traditional spectral ranking method for diagnosing software errors. The first is to have variable weights for different failed tests, dependent on the number of statements executed in the tests. This has a rational basis and does not affect the algorithmic complexity. The second is to compute the ranking incrementally, and relies on the first method. It gave a statistically significant improvement in performance for all metrics examined in the two-bug case. For the single-bug case it also improved performance for all but one metric, though for most metrics the improvement was not statistically significant. It makes the algorithmic complexity worse but does not seem prohibitively expensive.

For single-bug programs the traditional spectral ranking method has "hit a wall" with respect to improving performance by developing better ranking metrics, and our methods may suffer from the same limitation. The results for two-bug programs are of interest for two reasons. There have been no previous experiments comparing the performance of multiple ranking metrics using large numbers of programs with more than one bug, so they are a significant contribution to our quest for the best overall ranking metric. They also show the methods we have proposed can lead to good performance improvements comparted to the traditional method.

One theme which has emerged from our work is the asymmetry between passed and failed tests. The better ranking metrics have a bias towards failed tests (the relative importance of a_{ef} and a_{ep} in the formulas, for example). For O^p this was part of the design. For others, it seems more the result of tinkering and experimentation using benchmarks that consist of mostly single-bug programs. For others, there is no consideration of the software diagnosis domain as the metrics were developed for other domains. Our proposal for varying weights of tests is also rational for failed tests but not passed tests. It works well for software diagnosis but it may not be appropriate for other domains. Although diagnosis can be thought of in the same way as other clustering techniques, domain-specific knowledge can be important in maximising performance.

References

[1] R. Abreu, P. Zoeteweij, and A. van Gemund, "An Evaluation of Similarity Coefficients for Software Fault Localization," *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, pp. 39–46, 2006.

[2] L. Naish, H. Lee, and K. Ramamohanarao, "A Model for Spectra-based Software Diagnosis," *TOSEM*, To appear.

[3] J. Jones, M. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," *Proceedings of the 24th international conference on Software engineering*, pp. 467–477, 2002.

[4] J. Jones and M. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pp. 273–282, 2005.

[5] A. Gonzalez, "Automatic Error Detection Techniques based on Dynamic Invariants," Master's thesis, Delft University of Technology, The Netherlands, 2007.

[6] W. Wong, Y. Qi, L. Zhao, and K. Cai, "Effective Fault Localization using Code Coverage," *Proceedings of the 31st Annual International Computer Software and Applications Conference-Vol. 1-(COMPSAC 2007)-Volume 01*, pp. 449–456, 2007.

[7] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight bug localization with AMPLE," *Proceedings of the Sixth sixth international symposium on Automated analysis-driven debugging*, pp. 99–104, 2005.

[8] B. McConnaughey, "The determination and analysis of plankton communities," *Marine Research, Special No, Indonesia*, pp. 1–40, 1964.

[9] F. Lourenco, V. Lobo, and F. Bação, "Binary-based similarity measures for categorical data and their application in Self-Organizing Maps," *JOCLAD*, 2004.

[10] R. Sokal and P. Sneath, *Principles of numerical taxonomy*. San Francisco: W.H. Freeman, 1963.

[11] B. Everitt, S. Landau, and M. Leese, "Cluster Analysis. 2001," *Arnold, London*, 2001.

[12] P. Jaccard, "Étude comparative de la distribution florale dans une portion des Alpes et des Jura," *Bull. Soc. Vaudoise Sci. Nat*, vol. 37, pp. 547–579, 1901.

[13] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pp. 595–604, 2002.

[14] A. Ochiai, “Zoogeographic studies on the soleoid fishes found in Japan and its neighbouring regions,” *Bull. Jpn. Soc. Sci. Fish*, vol. 22, pp. 526–530, 1957.

[15] H. Do, S. Elbaum, and G. Rothermel, “Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact,” *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.

[16] W. Wong, J. Horgan, S. London, and A. Mathur, “Effect of test set minimization on fault detection effectiveness,” *Proceedings of the 17th international conference on Software engineering*, pp. 41–50, 1995.

[17] M. Hollander and D. Wolfe, “Nonparametric statistical methods,” *New York*, p. 518, 1973.

[18] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan, “Scalable statistical bug isolation,” *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, vol. 40, no. 6, pp. 15–26, 2005.

[19] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, “Sober: statistical model-based bug localization,” *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 286–295, 2005.

[20] J. Jones, J. Bowring, and M. Harrold, “Debugging in parallel,” *Proceedings of the 2007 international symposium on Software testing and analysis*, pp. 16–26, 2007.