Duals in Spectral Fault Localization

Lee Naish
Department of Computing and Information Systems
The University of Melbourne, Victoria 3010, Australia
Email: lee@unimelb.edu.au

Hua Jie Lee
Dolby Laboratories, Sydney, Australia
Email: huajie.lee@gmail.com

***Abstract*—Numerous set similarity metrics have been used for ranking "suspiciousness" of code in spectral fault localization, which uses execution profiles of passed and failed test cases to help locate bugs. Research in data mining has identified several forms of possibly desirable symmetry in similarity metrics. Here[1] we define several forms of "duals" of metrics, based on these forms of symmetries. Use of these duals, plus some other slight modifications, leads to several new similarity metrics. We show that versions of several previously proposed metrics are optimal, or nearly optimal, for locating single bugs. We also show that a form of duality exists between locating single bugs and locating "deterministic" bugs (execution of which always results in test case failure). Duals of the various single bug optimal metrics are optimal for locating such bugs. This more theoretical work leads to a conjecture about how different metrics could be chosen for different stages of software development.**

I. Introduction

Bugs are pervasive in software under development and tracking them down contributes greatly to the cost of software development. 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 where it can be determined if each result is correct or not; each test case is said to *pass* or *fail*. Software can be instrumented automatically to gather data known as program spectra [1], such as the statements that are executed, for each test case. If a certain statement is executed in many failed tests but few passed tests we may conclude it is likely to be buggy. Typically the raw data is aggregated to get the numbers of passed and failed tests for which each statement is/isn't executed. Some function is applied to this aggregated data to rank the statements, from those most likely to be buggy to those least likely. We refer to such functions as set similarity metrics, or simply *metrics*. A programmer can then use the ranking to help find a bug.

We make the following contributions:

- We define three forms of *duals* of set similarity metrics which can be used for spectral fault localization.
- We define slight variants of several previously proposed similarity metrics used for fault localization, motivated by other factors identified in the literature.
- We prove that several duals of these modified metrics are optimal (or nearly optimal) for locating single bugs, and validate this empirically.
- We demonstrate a form of duality exists between locating single bugs and deterministic bugs, which always cause failure of test cases.
- We define a class of metrics and prove such metrics are optimal for locating deterministic bugs. Duals of the single bug optimal metrics are optimal for locating deterministic bugs.
- We suggest that a form of contour plot of metrics allows useful insights.
- We conjecture that the stage of software development should influence the choice of metric used for fault localization.

Section II gives a brief introduction to spectral fault localization (some of this and other background material is based on [2] without additional citation; all prior technical material is cited explicitly) and gives the definitions of metrics from the literature that we adapt, analyse and evaluate in this paper. Section III discusses some important properties metrics may have and defines three forms of duals of metrics. Section IV defines several variants of the previously defined metrics, shows that some are equivalent and discusses how contour plots are a useful tool for visualising metrics. Section V discusses previous work on optimal metrics for single bug-programs and show how several of the new metrics we propose are optimal or nearly so for single bugs. Section VI discusses the relationship between deterministic bugs and single bugs, gives a new optimality result for deterministic bugs and shows how some of the new metrics we propose are optimal or nearly so for deterministic bugs. Section VII describes some empirical experiments and their results, which validate our theoretical results. Section VIII briefly reviews some additional related work and Section IX concludes.

II. Background — Spectral fault localization

All spectral methods use a set of tests, each classified as failed or passed; this can be represented as a binary vector, where 1 indicates failed and 0 indicates passed. For statement spectra [3], [4], [5], [6], [7], [8], [2], which we use here, we gather data on whether each statement is executed or not for each test. This can be represented as a binary matrix with a row for each statement and a column for each test; 1 means executed and 0 means not executed. 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 — $\langle a_{ef}, a_{ep}, a_{nf}, a_{np} \rangle$, 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

[1] An almost identical version of this paper will appear in the proceedings of ASWEC 2013 and you will be able to get from IEEE, for a price.

TABLE I. STATEMENT SPECTRA WITH TESTS $T_1 \ldots T_5$

	T_1	T_2	T_3	T_4	T_5	a_{ef}	a_{ep}
S_1	1	0	0	1	0	1	1
S_2	1	1	0	1	0	2	1
S_3	1	1	1	0	1	2	2
				⋮			
Res.	1	1	0	0	0	$F=2$	$P=3$

failed (f). We use superscripts to indicate the statement number where appropriate. For example, a_{ep}^1 is the number of passed tests that executed statement 1. We use F and P to denote the total number of tests which fail and pass, respectively. Clearly, $a_{nf} = F - a_{ef}$ and $a_{np} = P - a_{ep}$. It is sometimes convenient to avoid explicit use of a_{nf} and a_{np}; making F and P explicit instead. Table I gives an example binary matrix of execution data and binary vector containing the test results. This data allows us to compute F, P and the a_{ij} values, $i \in \{n, e\}$ and $j \in \{p, f\}$.

Metrics, which are numeric functions, can be used to rank the statements. Most commonly they are defined in terms of the four a_{ij} values. Statements 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 2 (S_2) is executed in both failed tests and only one passed test, which is the minimum for all statements, and thus would typically be ranked highest. The relative rank of statements 1 and 3 is not so clear cut, since statement 3 is executed in more failed tests but also more passed tests. One way of viewing the ranking is that rows of the matrix are ranked according to how "similar" they are to the vector of test results, or how similar the set of failed tests is to the set of tests which execute the statement. Measures of similarity are not just important in fault localization. They are central to classification and machine learning, for example, and *many* different metrics have been proposed in a wide range of application areas. The fault localization performance of over 80 metrics gathered from the literature is evaluated in [8]. Even the small selection of metrics we use here have quite diverse origins, such as classification of alpine plants [9], marine zoology [10] and error detecting and correcting codes [11].

Programmers searching for a bug are expected to examine statements, starting from the highest-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*. Also, checking correctness generally cannot be done by a single statement at a time, or even one basic block at a time. Evaluation of different ranking methods generally ignores such refinement and just depends on where the bug(s) appear in the ranking.

Table II gives definitions of the established metrics used here, all of which have been evaluated for fault localization in [7] and their origins are discussed more there. We use a small sample of "interesting" metrics to illustrate our points. Three were originally proposed for spectral fault localization: Tarantula [3], which is generally credited as being the first spectral fault localization system, Ample2 [7] (based on a metrics proposed in [12] and [4]) and O^p [7]. Jaccard [9], the oldest metric here, has been used in the Pinpoint system [13]. Ochiai [10], generally referred to as Cosine in data mining and machine learning literature [14], was first evaluated for fault localization in [4], as was a metric equivalent to Hamming [11] (namely Rogers-Tanimoto) [7].

In most of the research on spectral fault localization there is an underlying assumption that we are searching for "the best" metric for this application area (based on evidence from a particular set of benchmarks). Here we challenge this view. Different metrics perform best in different situations — this is to be expected. Based on our more theoretical insights, we suggest how the current state of a software development project could be used to choose one of a set of metrics.

III. PROPERTIES AND DUALS OF METRICS

Recall that metrics can be seen as measuring similarity of the set of failed test cases and the set of test cases in which a particular statement is executed. Several general properties of set similarity (and related) measures have been investigated in the context of data mining [15], [16], [14]; here we primarily refer to the account of [14].

The first property clearly of interest is that the metrics should be well-defined for all possible values of the a_{ij}. Many metrics are undefined in some circumstances due to division by (or logarithm of) zero. It seems reasonable to constrain the problem to situations where there is at least one failed test case (otherwise there is no fault apparent). We may also want to ignore statements which are not executed in any tests and/or assume there is at least one passed test and/or tweek some definitions.

The second property is that for a fixed number of passed and failed tests, metrics strictly increase in a_{ef} and strictly decrease in a_{ep} — this is called *monotone* in [14] and *strictly rational* in [2]. Numerous metrics, including Tarantula, Jaccard and Ochiai, are not monotone due to their behaviour when $a_{ef} = 0$; they have a constant value of zero rather than decreasing in a_{ep}.

The last properties of interest are forms of symmetry. Let E be the set of test cases which execute a given statement and F be the set of test cases which fail. Saying E is similar to F is like saying saying the complement of E (the set of test cases which do not execute the statement) is dis-similar to F, or E is dis-similar to the complement of F (the set of passed test cases), or saying the complement of E is similar to the complement of F. Equivalently, we can swap ones and zeros in either the encoding of statement execution, test case failure, or both (see Table I). For ranking purposes, dis-similarity can be measured by simply negating a similarity measure. We thus define three forms of *duals* of metrics corresponding to complementing E, F and both, respectively:

Definition 1 (E-dual): The *E-dual* of a metric M, $\mathcal{D}^E(M)$, is defined as

$$\mathcal{D}^E(M)(a_{ef}, a_{ep}, a_{nf}, a_{np}) = -M(a_{nf}, a_{np}, a_{ef}, a_{ep})$$

Definition 2 (F-dual): The *F-dual* of a metric M, $\mathcal{D}^F(M)$, is defined as

$$\mathcal{D}^F(M)(a_{ef}, a_{ep}, a_{nf}, a_{np}) = -M(a_{ep}, a_{ef}, a_{np}, a_{nf})$$

TABLE II. DEFINITIONS OF RANKING METRICS USED PREVIOUSLY

Name	Formula	Name	Formula	Name	Formula
Tarantula	$\frac{\frac{a_{ef}}{a_{ef}+a_{nf}}}{\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ep}}{a_{ep}+a_{np}}}$	Jaccard	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}}$	Ochiai	$\frac{a_{ef}}{\sqrt{(a_{ef}+a_{nf})(a_{ef}+a_{ep})}}$
O^p	$a_{ef}-\frac{a_{ep}}{a_{ep}+a_{np}+1}$	Ample2	$\frac{a_{ef}}{a_{ef}+a_{nf}}-\frac{a_{ep}}{a_{ep}+a_{np}}$	Hamming	$a_{ef}+a_{np}$

Definition 3 (EF-dual): The *EF-dual* of a metric M, $\mathcal{D}^{EF}(M)$, is defined as

$$\mathcal{D}^{EF}(M)(a_{ef},a_{ep},a_{nf},a_{np}) = M(a_{np},a_{nf},a_{ep},a_{ef})$$

Metrics which are their own EF-duals are called "inversion invariant" in [14] and the term "antisymmetry for normalised measures" is used to describe metrics which are their own E-duals and F-duals (though these two forms of symmetry are generally distinct and when metrics are only used for ranking purposes is is not necessary for them to be "normalised" to a particular range). Although these forms of symmetry may be appealing when viewing set similarity from a philosophical viewpoint, in practice, most commonly used similarity metrics do not exhibit these forms of symmetry. However, the duals we have defined, which are the essence of these forms of symmetry, can be useful for spectral fault localization (and other applications).

IV. NEW METRICS FROM OLD

Based on the properties discussed above, we present slightly modified version of the previously defined metrics. Many other metrics from the literature can be modified in similar ways. Table III contains modified definitions of those in Table II which are well-defined and monotone assuming at least one passed and failed test. We allow statements which are not executed in any test, because the E-dual notion is that the statement is executed in all tests, which is often the case and such statements must be considered for fault localization. Tarantula results in the same ranking as $\frac{a_{ef}}{a_{ep}}$ (if we can obtain metric g from f by applying a monotone function, f and g are equivalent for ranking purposes)[7]. By adding a small constant ϵ to the numerator and denominator we can ensure is it monotone and defined in all relevant cases. Similary for Jaccard and Ochiai/Cosine. A more conservative modification is to add ϵ only when a_{ef} (or a_{ep}) is zero, but here we opt for simplicity. There seem no apparent good reasons to use the original versions of these metrics in preference to monotone variants. N gives the same ranking as O^p but has more desirable properties with respect to the duals. Ample2 and A are identical, as are Hamming and H; they are included in the table for completeness. In theory we can take the limit of these formulas as ϵ approaches zero, or, for a given F and P, choose some ϵ such that no smaller positive value gives a different ranking for all possible a_{ef} and a_{ep} values. In practice we can just use a small fixed constant such as 10^{-6}, which is what we use to obtain our experimental results.

As well as these definitions, we can use the three forms of duals of each of these definitions. However, in some cases we obtain metrics which are equivalent for ranking purposes.

Proposition 1: $\mathcal{D}^E(\mathrm{N})$ and N are equivalent, $\mathcal{D}^E(\mathrm{A})$, $\mathcal{D}^F(\mathrm{A})$, $\mathcal{D}^{EF}(\mathrm{A})$, and A are equivalent, $\mathcal{D}^E(\mathrm{H})$, $\mathcal{D}^F(\mathrm{H})$, $\mathcal{D}^{EF}(\mathrm{H})$, and H are equivalent, and $\mathcal{D}^F(\mathrm{T})$ and T are equivalent,

Rather than give a formal proof of these equivalences, we give a graphical illustration of these metrics and their duals. It is convenient to view metrics as contour plots, showing equal metric values as a_{ef} and a_{ep} vary from zero to F and P, respectively. Figure 1 shows plots of contours of these metrics, with rectangles showing the range of possible a_{ef} and a_{ep} values; all contours are linear for these metrics. For J, all contour lines converge where $a_{ef} = -\epsilon$ and $a_{ep} = -F$ (we include negative a_{ep} values in this plot to show the point of convergence). For T, all contour lines converge where $a_{ef} = a_{ep} = -\epsilon$. For N, A and H, all contour lines are parallel. For N the gradient is ϵ (exaggerated in the diagram), for A the gradient is F/P and for H the gradient is 1.

Several insights can be gained from plotting contours of metrics in this way. First, metrics which are equivalent for ranking purposes have the same set of contours. Equivalent metrics can be identified immediately, whereas recognising equivalence by comparing formulas (such as that for Tarantula and $\frac{a_{ef}}{a_{ep}}$) is often far from obvious. Second, monotonicity simply means that all countours have a positive (and finite) gradient. Third, the relative importance of execution in passed versus failed tests is made clear from the gradient of the contours. A low gradient means a_{ef} is more important than a_{ep} for determining the value of the metric whereas a high gradient means the opposite. Finally, the contours allow the forms of symmetry to be easily identified, modulo equivalence for ranking.

Plotting contours of the E-dual of a metric in the same way as Figure 1 results in a $180°$ rotation, since the metric at point (a_{ef},a_{ep}) has the negated value of the E-dual at point $(F-a_{ef},P-a_{ep})$. Metrics N, A, H, and all other metric for which the contours are parallel lines have this form of symmetry in their contour plots. Plotting the F-dual results in a reflection in the line $a_{ef} = a_{ep}$ with F and P also swapped. The metric at point (a_{ef},a_{ep}) has the negated value of the F-dual at point (a_{ep},a_{ef}). Metrics A, H and T have form of symmetry, so A and H are also equivalent to their EF-duals.

V. OPTIMALITY FOR SINGLE BUG PROGRAMS

In [7], optimality of metrics is introduced and "single bug" programs are the focus. In order to establish any technical results, we must be clear as to what constitutes a bug, so it is clear if a program has a single bug. In [7] a bug is defined to be "a statement that, when executed, has unintended behaviour". Note that a programmer may make a single mistake, such as

TABLE III. MODIFIED DEFINITIONS OF RANKING METRICS

Name	Formula	Name	Formula	Name	Formula
T	$\frac{a_{ef}+\epsilon}{a_{ep}+\epsilon}$	J	$\frac{a_{ef}+\epsilon}{a_{ef}+a_{nf}+a_{ep}}$	C	$\frac{a_{ef}+\epsilon}{\sqrt{(a_{ef}+a_{nf})(a_{ef}+a_{ep}+\epsilon)}}$
N	$a_{ef}-\epsilon a_{ep}$	A	$\frac{a_{ef}}{a_{ef}+a_{nf}}-\frac{a_{ep}}{a_{ep}+a_{np}}$	H	$a_{ef}+a_{np}$

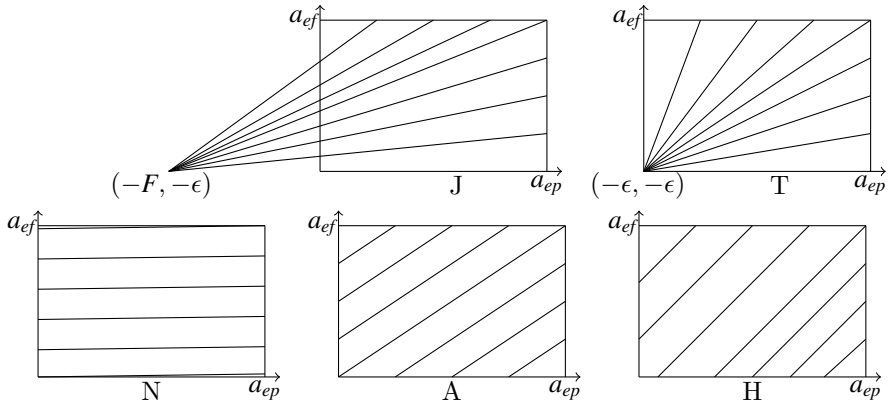

Fig. 1. Contour plots for several measures

an incorrect `#define` directive, which leads to multiple bugs according to this definition.

In order to understand the fault localization problem better, a very simple model program, with just two if-then-else statements and a single bug is proposed in [7], along with a very simple way of measuring performance of a metric with a given set of test cases, based on whether the bug is ranked top, or equal top. A set of test cases corresponds to a multiset of execution paths through the program. Performance depends on the multiset, but *overall* performance for K tests is determined by the average performance over all possible multisets of K execution paths. Using a combinatorial argument, a class of metrics, including O^p, is shown to be "optimal", meaning its overall performance is at least as good as any other metric, for any number of tests. The definition of the class of metrics is as follows, where it is assumed F and P are fixed:

Definition 4 (Single bug optimality [7]): A metric M is *single bug optimal* if

1) when $a_{ef} < F$, the value returned is always less than any value returned when $a_{ef} = F$, and
2) when $a_{ef} = F$, M is strictly decreasing in a_{ep}.

The first condition is motivated by the fact that for single bug programs, the bug must be executed in all failed tests. Since $a_{ef} = F$ for the bug, statements for which $a_{ef} < F$ are best ranked strictly lower. The second condition is motivated by the same intuition behind monotonicity. This optimality result was extended further in [2], using a more realistic performance measure (a minor variation of several others in the literature) and making the result independent of the set of test cases. The result required restricting attention to monotone (strictly rational) metrics but is was argued that this restriction is not of practical importance. The definition of performance and the optimality proposition are as follows:

Definition 5 (rank cost [2]): Given a ranking of S statements, the *rank cost* is

$$\frac{GT + EQ/2}{S}$$

where GT is the number of correct statements ranked strictly higher than all bugs and EQ is the number of correct statements ranked equal to the highest ranked bug.

Proposition 2 ([2]): Given any program with a single bug, any set of test cases and any single bug optimal metric M used to rank the statements, the rank cost using M is no more than the rank cost using any other strictly rational metric.

We now show that duals of several previously defined metrics, when tweeked to ensure monotonicity and well-definedness, are single bug optimal and thus lead to maximal performance for localizing single bugs.

Proposition 3: $\mathcal{D}^E(\mathrm{J})$ and $\mathcal{D}^E(\mathrm{C})$ are both single bug optimal.

Proof: The formulas for these duals are $-\frac{F-a_{ef}+\epsilon}{F+P-a_{ep}}$ and $-\frac{F-a_{ef}+\epsilon}{\sqrt{F(F-a_{ef}+P-a_{ep}+\epsilon)}}$, respectively. For $a_{ef} = F$ the numerators are ϵ and the denonimators are always much larger, so the result is a very small negative number, whereas the result is a larger negative number when $a_{ef} < F$. Also, when $a_{ef} = F$ the denonimators, and hence the overall value of the (negated) fraction decrease in a_{ep}. ■

Note that the E-duals of the original (non-monotone) versions of these metrics are not single bug optimal, since when $a_{ef} = F$ the numerators are zero, hence the overall value does not decrease in a_{ep}. Ochiai/Cosine performs quite well for fault localization (it is the best of the metrics evaluated in [4]) and Jaccard also performs reasonably well. It is interesting that a slight modification to ensure monotonicity and taking the E-dual results in optimal metrics for the single bug case.

Futhermore, this applies to numerous other metrics — of the large number of metrics evaluated in [8], a_{ef} divided by some other formula features prominently and very similar methods can be used to obtain optimal single bug metrics.

Proposition 4: $\mathcal{D}^E(\mathrm{T})$ is single bug optimal if statements which are executed in all tests are ignored.

Proof: The dual is $\frac{F-a_{ef}+\epsilon}{P-a_{ep}+\epsilon}$. The same reasoning as the previous proof applies, except when $a_{ef} = F$ and $a_{ep} = P$, that is, the statement is executed in all tests. ■

There is only a single case which is not ranked optimally: $\mathcal{D}^E(\mathrm{T})$ may rank a statement which is executed in all tests below some statements which are not executed in all failed tests. However, it is a particularly common case in practice, including "initialization" code, for example. In our empirical experiments (see Section VII) $\mathcal{D}^E(\mathrm{T})$ gives significantly lower performance than the optimal single bug metrics. However, our experiments suggest that $\mathcal{D}^E(\mathrm{T})$ still performs much better than Tarantula. Also, statements which are executed in all test cases could be treated specially in the bug localization regime rather than just relying on a metric to rank them amongst the other statements.

VI. Single bugs and deterministic bugs

Deterministic bugs are those which cause test case failure whenever they are executed [17] (note that our use of the term "deterministic" here only relates to this definition; it is not about whether the behaviour of a program is completely determined by the test case). The relationship between performance of spectral fault localization and how consistently bugs lead to test case failure has been studied [4], [8]. The term "error detection accuracy", q_e, is used to describe bug consistency in [4], defined by $q_e = a_{ef}/(a_{ef} + a_{ep})$. Although deterministic bugs (where $q_e = 1$) have been noted in the literature, this class of bugs has attracted relatively little attention in fault localization research. In part, this is due to the fact that they are often elimination early in software development. The vast majority of fault localization effort goes into finding bugs which are not deterministic, so execution of them results in test case failure in only some cases (and quite often a very small proportion of cases). The average q_e value for bugs in the Siemens test suite (see Section VII), for example, is only 0.12 [8]. There are no established benchmark sets of deterministic bug programs.

In contrast, single bug programs have attracted far more interest and most of the larger sets of benchmark programs have at least a strong bias towards single bugs. Although most fault localization effort is also directed towards programs with multiple bugs, we can at least hope to be hunting down the last bug (or at least the last bug exposed by our test suite) at some stage, and this hunt can take considerable effort. Here we show an interesting relationship between programs with a single bug and programs with only deterministic bugs. This leads to an optimality result for deterministic bug programs analogous to that for single bug programs.

For the bug in a single bug program $a_{ef} = F$ and (equivalently) $a_{nf} = 0$, as mentioned earlier. For deterministic bugs, $a_{ep} = 0$ (and $a_{np} = P$). Thus there is a form of EF-duality relating these two cases. Optimal single bug metrics have a contour with a small positive gradient close to the line $a_{ef} = F$. By reflection and rotation we obtain a contour with a very high gradient close to the line $a_{ep} = 0$. We can define the class of deterministic bug optimal metrics based intuitively on the EF-dual of the class of single bug optimal metrics:

Definition 6 (Deterministic bug optimality): A metric M is *deterministic bug optimal* if

1) when $a_{ep} > 0$, the value returned is always less than any value returned when $a_{ep} = 0$, and
2) when $a_{ep} = 0$, M is strictly increasing in a_{ef}.

This class of metrics in indeed optimal for programs with only deterministic bug, in that no other monotone metric can result in a smaller rank cost:

Proposition 5: Given any program with only deterministic bugs, any set of test cases and any deterministic bug optimal metric M used to rank the statements, the rank cost using M is no more than the rank cost using any other monotone metric.

Proof: Let O be a deterministic bug optimal metric, b be a statement which is ranked (equal) top by O and M be some other monotone metric. The rank cost depends only on the top-most ranked bug(s), so it is sufficient to show that

1) no (correct) statement s which is ranked higher than b by O is ranked lower than b by M, and
2) no buggy statement b' which is ranked lower than b by O is ranked higher than b by M.

For 1), we know $a^b_{ep} = 0$ (since b is a deterministic bug) and so $a^s_{ep} = 0$ (since s is ranked higher by O, which is deterministic bug optimal), so $a^s_{ef} > a^b_{ef}$ (similarly), so s is ranked higher than b by M (since M is monotone). For 2), $a^{b'}_{ep} = 0$ (since b' is a deterministic bug), so $a^{b'}_{ef} < a^b_{ef}$ (since b is ranked higher by O, which is deterministic bug optimal), so b is ranked higher by M (since M is monotone). ■

Note that in this proof we rely on the performance measure being based only on where the top-most bug is ranked, whereas no such assumption is necessary for the proof of single bug optinality (since there is only a single bug in the ranking). Most proposed performance measures have this characteristic, the reasoning being that after a bug is located, it can be corrected and the tests re-run to find any further bugs. In practice, re-running all the tests may be expensive and parallel search for multiple bugs may be desirable. However, even for performance measures which are based on the ranking of multiple bugs, it seems that deterministic bug optimal metrics are likely to perform best for this class of bugs.

The EF-duals of the new single bug optimal metrics discussed previously are deterministic bug optimal:

Proposition 6: $\mathcal{D}^F(\mathrm{N})$ (and, equivalently, $\mathcal{D}^{EF}(\mathrm{N})$), $\mathcal{D}^F(\mathrm{J})$, and $\mathcal{D}^F(\mathrm{C})$ are all deterministic bug optimal.

Proof: $\mathcal{D}^F(\mathrm{N})$ is defined as $-(a_{ep} - \epsilon a_{ef})$ and the proof for this and the other metrics is straightforward. ■

Note that $\mathcal{D}^F(O^p)$ is not deterministic bug optimal. It is necessary that the coefficient for a_{ep} is "sufficiently small", which ϵ is but the coefficient used in O^p may not be (depending on the F and P values).

Proposition 7: T is deterministic bug optimal if statements which are executed in no tests are ignored.

Proof: Straightforward. ■

Although T is not deterministic bug optimal in the strict sense, it does give optimal rank cost. Since we assume there is at least one failed test, there must be at least one bug which is executed in some test and the rank cost depends only on the top-most ranked bug. For performance measures which depend on the ranking of all bugs (or if we drop the assumption that there is a failed test), T may perform less well than deterministic bug optimal metrics. It may rank a statement which is never executed below some statements which are used in a passed test (which is sub-optimal for deterministic bugs).

Based on our optimality results for deterministic bugs and single bugs, and the discussion on when these bug classes are most important during software development, we conjecture the following:

Conjecture 1: In early stages of software development, metrics that have contours with relatively high gradients are preferable, whereas in late stages of software development, metrics that have contours with relatively low gradients are preferable.

Early in software development, when new code is written and unit tested, for example, deterministic bugs are relatively likely and multiple bugs are common. Metrics which are deterministic bug optimal (or similar) are thus likely to perform relatively well, whereas metrics which are single bug optimal (or similar) are likely to perform relatively poorly. Late in development, when regression testing of a whole system is performed, for example, bugs tend to be much less consistent and there are likely to be fewer bugs. Metrics which are similar to single bug optimal metrics are thus likely to perform relatively well. In the past, researchers have tended to focus on finding "the best" metric to be used for spectral fault localization. Here we are suggesting there is no single best metric, but by understanding certain properties of metrics and the current state of a software development project, a good choice of metric can be made. Unfortunately, detailed empirical investigation of this conjecture is beyond our current resources. We have, however, performed some experiments to validate our theoretical contributions.

VII. Experimental results

To validate our theoretical results we have performed some simple model-based experiments in the style of [7]. The simple deterministic model program of [7] was used. This has two if-the-else constructs in sequence and each of the four branches is an instrumented statement, executed in half the tests, on average. One of the statements is buggy and leads to test case failure half the time it is executed, on average. In addition, we used a model with three if-the-else constructs in sequence and each of the six branches is an instrumented statement, executed in half the tests, on average. Two statements, in different if-the-else constructs, are deterministic bugs. Table IV gives the average rank cost (as a percentage) for each model and each metric, using 20 and 50 test cases. For 20 test cases, performance was averaged over every possible set of test cases (every multiset of 20 execution paths). For 50 test cases we used ten million randomly selected multisets. Test sets that lead to no failed tests or no passed tests were ignored. As well as the new metrics, the table includes the original version of Jaccard and its duals and the original version of Tarantula, modified to avoid division by zero ($x/0$ is considered to be 0.5 if $x = 0$ and 9999 otherwise).

TABLE IV. Experimental results using simple models

	Single bug		Det. bug	
Num. Tests	20	50	20	50
N	3.01	1.47	2.85	2.79
$\mathcal{D}^E(\mathrm{C})$	3.01	1.47	2.36	1.98
$\mathcal{D}^E(\mathrm{J})$	3.01	1.47	2.64	2.33
$\mathcal{D}^E(\mathit{Jaccard})$	5.30	2.78	2.76	2.35
$\mathcal{D}^E(\mathrm{T})$	3.07	1.48	1.49	1.06
C	3.79	2.63	1.71	1.32
J	4.28	3.27	1.77	1.43
Jaccard	4.22	3.25	1.77	1.43
A	4.33	3.04	0.87	0.49
T	6.14	4.92	0.76	0.35
Tarantula	6.12	4.90	3.50	1.85
H	9.98	8.85	1.47	1.16
$\mathcal{D}^{EF}(\mathrm{C})$	12.27	10.69	0.88	0.52
$\mathcal{D}^{EF}(\mathrm{J})$	13.69	12.33	0.94	0.61
$\mathcal{D}^{EF}(\mathit{Jaccard})$	13.76	12.33	0.93	0.61
$\mathcal{D}^F(\mathrm{C})$	13.72	12.32	0.76	0.35
$\mathcal{D}^F(\mathrm{J})$	14.93	13.71	0.76	0.35
$\mathcal{D}^F(\mathit{Jaccard})$	15.07	13.72	3.54	1.85
$\mathcal{D}^F(\mathrm{N})$	17.77	17.43	0.76	0.35

As expected, N, $\mathcal{D}^E(\mathrm{J})$ and $\mathcal{D}^E(\mathrm{C})$ give equal-best performance for the single bug model. $\mathcal{D}^E(\mathrm{T})$ gives somewhat worse performance for the smaller number of tests but for the larger number of tests the divergence from optimal performance is very small. The reason for this is that (for this model and a uniform distribution of test cases), with a reasonably large number of test cases it is very rare that any statement is executed in all test cases. Also as expected, $\mathcal{D}^F(\mathrm{N})$, $\mathcal{D}^F(\mathrm{J})$ and $\mathcal{D}^F(\mathrm{C})$ perform equal best for the deterministic model. T also gives optimal performance for this model since the performance measure depends only on the top-most ranked bug and there is at least one failed test (see the discussion following Proposition 4).

Comparison of J and Jaccard, and their duals, allows us to see the effect of adding ϵ to achieve monotonicity. For the single bug optimal version, $\mathcal{D}^E(\mathrm{J})$, and deterministic bug optimal version, $\mathcal{D}^F(\mathrm{J})$, monotonicity significantly improves the performance for single bug and deterministic bug models, respectively. In other cases the performance differences are small, though (perhaps surprisingly) Jaccard performs slightly better than J for the single bug model. This is because adding ϵ increases the gradient of all the contours. Although the increase is tiny, the effect on performance is noticable because of ties in the ranking. There are cases where a bug and a non-bug, with lower a_{ef} but higher a_{ep}, are tied in the ranking using Jaccard but using J the non-bug is always ranked higher. If ϵ was added only when $a_{ef} = 0$ we would not see this effect. Similarly, Tarantula performs slightly better than T for this model (some contours have a higher gradient with T), though significantly worse for the deterministic bug model, as expected. $\mathcal{D}^E(\mathrm{T})$ performs significantly better than Tarantula for both models.

We also performed empirical evaluation using a collection of small C programs with single bugs: the Siemens Test Suite (STS), from the Software Information Repository [18], plus several small Unix utilities, from [19]. These, particularly STS,

TABLE V. DESCRIPTION OF SIEMENS + UNIX BENCHMARKS

Program	versions	LOC	Tests
tcas	37	173	1608
schedule	8	410	2650
schedule2	9	307	2710
print_tok	6	563	4130
print_tok2	10	508	4115
tot_info	23	406	1052
replace	29	563	5542
Col	28	308	156
Cal	18	202	162
Uniq	14	143	431
Spline	13	338	700
Checkeq	18	102	332
Tr	11	137	870

TABLE VI. EXPERIMENTAL RESULTS USING STS AND UNIX

Metric	STS	Unix	Combined
N	14.78	19.37	16.87
$\mathcal{D}^E(\mathrm{C})$	14.78	19.37	16.87
$\mathcal{D}^E(\mathrm{J})$	14.78	19.37	16.87
$\mathcal{D}^E(\mathit{Jaccard})$	27.46	30.90	29.02
$\mathcal{D}^E(\mathrm{T})$	16.85	21.52	18.98
C	19.26	22.28	20.63
J	22.57	22.75	22.65
Jaccard	22.57	22.75	22.65
A	21.30	25.23	23.09
T	23.22	28.17	25.47
Tarantula	23.22	31.49	26.98
H	43.57	29.37	37.10
$\mathcal{D}^{EF}(\mathrm{C})$	43.87	32.51	38.70
$\mathcal{D}^{EF}(\mathrm{J})$	44.11	32.93	39.02
$\mathcal{D}^{EF}(\mathit{Jaccard})$	44.12	32.93	39.02
$\mathcal{D}^{F}(\mathrm{C})$	44.04	34.92	39.88
$\mathcal{D}^{F}(\mathrm{J})$	44.35	35.23	40.20
$\mathcal{D}^{F}(\mathit{Jaccard})$	44.35	38.54	41.71
$\mathcal{D}^{F}(\mathrm{N})$	44.74	36.54	41.00

are widely used for evaluating spectral ranking methods. Table V gives the names of the programs (the first seven are from STS), and the numbers of versions, lines of code (LOC) and test cases. A small number of programs in the repository were not used because there was more than one bug according to our definition (for example, a `#define` was incorrect) or we could not extract programs spectra. We used the `gcov` tool, part of the `gcc` compiler suite (version 4.4.5 on Ubuntu), and it cannot extract spectra from programs with runtime errors.

Table VI gives the results for these benchmarks. In computing the rank cost, lines of code which were not executed in any tests were ignored. The results are very similar to those from the single bug model. As with this model, N, $\mathcal{D}^E(\mathrm{J})$ and $\mathcal{D}^E(\mathrm{C})$ give equal-best performance. $\mathcal{D}^E(\mathrm{T})$ gives significantly worse performance (though still better than the other metrics), due to bugs which are executed in all test cases. The expected performance of $\mathcal{D}^E(\mathrm{T})$ for such cases is substantially lower than that of the optimal metrics, leading to the gap in overall performance for the benchmark set. $\mathcal{D}^E(\mathrm{T})$ still performs much better than T and Tarantula. Again, $\mathcal{D}^E(\mathrm{J})$ performs significantly better than $\mathcal{D}^E(\mathit{Jaccard})$. In other cases the effect of ϵ is quite small. As expected, the deterministic bug optimal metrics (along with some others) perform very poorly, particularly for Siemens, with its low average bug consistency. Spectral fault localization performance for the Unix benchmark is not affected so much by bug consistency, partly due to the large number of ties in the ranking (there are far fewer test cases than STS and they were not developed with the same technique) [8]. This explains why the variation in performance between the different metrics is less for Unix than for STS.

VIII. OTHER RELATED WORK

In Section II we referred to several papers which introduced new metrics for spectral fault localization, or evaluated metrics which had previously been introduced for other domains. Here we briefly review other related work. There are a couple of approaches which post-process the ranking produced which are equivalent to adjusting the metric. The post-ranking method of [6] essentially drops any statement which is not executed in any failed test to the bottom of the ranking. That of [20] ranks primarily on the a_{ef} value and secondarily on the original rank. Thus if the original ranking is done with a monotone metric, the resulting ranking is single bug optimal. An alternative way of ensuring single bug optimality is to adapt metric definitions so the case when $a_{ef} = F$ is treated specially [2].

Other variations on the statement spectra ranking method described in this paper attempt to use additional and/or different information from the program executions. Execution frequency counts for statements, rather than binary numbers, are used in [21] to weight the different a_{ij} values and in [22] aggregates of the columns of the matrix are used to adjust the weights of different failed tests. The RAPID system [23] uses the Tarantula metric but uses branch spectra rather than statement spectra.

The CBI [17] and SOBER [24] systems use predicate spectra: predicates such as conditions of if-then-else statements are instrumented and data is gathered on whether control flow ever reaches that point and, if it does, whether the predicate is ever true. CBI uses sampling to reduce overheads but aggregates the data so there are four numbers for each predicate, which are ranked in a similar way to how statements are ranked using statement spectra. SOBER uses frequency counts and a different form of statistical ranking method. The Holmes system [25] uses path spectra: data is collected on which acyclic paths through single functions are executed or "reached ", meaning the first statement is executed but not the whole path, and the paths are ranked in a similar way to predicate ranking in CBI. Statement and predicate spectra are compared in [26], and it is shown that the aggregate data used in predicate spectra methods is more expressive than that used for statements spectra and modest gains in theoretical performance are demonstrated. The data collected for path spectra contains even more information and thus could potentially be used to improve performance further.

IX. CONCLUSION

Spectra-based techniques are a promising approach to software fault localization. Here we have used one of the simplest and most popular variants: ranking statements according to some metric, a function of the numbers of passed and failed tests in which the statement is/isn't executed. Metrics can be viewed as a way to measure similarity of sets, a generic problem applicable to many areas of science. For this reason, a very large number of metrics have been proposed over many years and general properties of such metrics have been studied. One important property is that metrics should be monotone. For example, if we fix the number of passed and failed tests, the suspiciousness of a statement should increase in the number of failed tests it is executed in and decrease in the number of passed tests it is executed in. Many set similarity metrics

are not monotone but can be made monotone with a small adjustment, such as adding a small constant to the numerator of the metric definition.

Set similarity is normally applied to spectral debugging by considering the similarity of the set of failed tests and the set of tests each statement is executed in. However, we can equally consider the similarity of the complements of these two sets, or the dis-similarity of one set and the complement of the other. Thus there are three variant ways of applying set similarity. Equivalently, we can define three "duals" of any given set similarity measure. For several existing similarity metrics, if we modifiy the definitons slightly to ensure monotonicity, then a dual of the metric is optimal for single bug programs. We have also shown that a form of duality exists between single bug programs and programs which only contain "deterministic" bugs, and defined a class of metrics which is proven to be optimal for localizing fault in such programs. These two classes of programs can be seen as the two extreme cases for spectral fault localization. Because deterministic bugs tend to be more prevalent in early stages of software development and single bugs occur late software development, we conjecture that metrics should be chosen based on the stage of software development, rather than aiming for a single "best" metric to be used in all situations.

REFERENCES

[1] 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 Conference held jointly with the 5th ACM SIGSOFT*. New York, USA: Springer-Verlag New York, Inc. New York, 1997, pp. 432–449.

[2] L. Naish, H. J. Lee, and R. Kotagiri, "Spectral debugging: How much better can we do?" in *35th Australasian Computer Science Conference (ACSC 2012), CRPIT Vol. 122*. CRPIT, 2012.

[3] J. Jones and M. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," *Proceedings of the 20th ASE*, pp. 273–282, 2005.

[4] R. Abreu, P. Zoeteweij, and A. van Gemund, "An evaluation of similarity coefficients for software fault localization," *PRDC'06*, pp. 39–46, 2006.

[5] W. E. Wong, Y. Qi, L. Zhao, and K. Cai, "Effective Fault Localization using Code Coverage," *Proceedings of the 31st Annual IEEE Computer Software and Applications Conference*, pp. 449–456, 2007.

[6] X. Xie, T. Y. Chen, and B. Xu, "Isolating Suspiciousness from Spectrum-Based Fault Localization Techniques," in *10th International Conference on Quality Software , 2010. QSIC 2010*, 2010.

[7] L. Naish, H. J. Lee, and R. Kotagiri, "A model for spectra-based software diagnosis," *ACM Transactions on software engineering and methodology (TOSEM)*, vol. 20, no. 3, August 2011.

[8] H. J. Lee, "Software Debugging Using Program Spectra," Ph.D. dissertation, University of Melbourne, 2011.

[9] 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.

[10] 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.

[11] R. Hamming, "Error detecting and error correcting codes," *Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[12] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight bug localization with AMPLE," in *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*. ACM, 2005, pp. 99–104.

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

[14] P.-N. Tan, V. Kumar, and J. Srivastava, "Selecting the right interestingness measure for association patterns," in *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, ser. KDD '02. New York, NY, USA: ACM, 2002, pp. 32–41. [Online]. Available: http://doi.acm.org/10.1145/775047.775053

[15] L. Geng and H. J. Hamilton, "Interestingness measures for data mining: A survey," *ACM Comput. Surv.*, vol. 38, no. 3, Sep. 2006. [Online]. Available: http://doi.acm.org/10.1145/1132960.1132963

[16] P. Lenca, P. Meyer, B. Vaillant, and S. Lallich, "On selecting interestingness measures for association rules: User oriented description and multiple criteria decision aid," *European Journal of Operational Research*, vol. 184, no. 2, pp. 610–626, 2008.

[17] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan, "Scalable statistical bug isolation," *Proceedings of the 2005 ACM SIGPLAN*, vol. 40, no. 6, pp. 15–26, 2005.

[18] 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.

[19] W. Wong, J. Horgan, S. London, and A. Mathur, "Effect of Test Set Minimization on Fault Detection Effectiveness," *Software-Practice and Experience*, vol. 28, no. 4, pp. 347–369, 1998.

[20] V. Debroy, W. Wong, X. Xu, and B. Choi, "A Grouping-Based Strategy to Improve the Effectiveness of Fault Localization Techniques," in *10th International Conference on Quality Software , 2010. QSIC 2010*, 2010.

[21] H. J. Lee, L. Naish, and R. Kotagiri, "Effective Software Bug Localization Using Spectral Frequency Weighting Function," in *Proceedings of the 2010 34th Annual IEEE Computer Software and Applications Conference*. IEEE Computer Society, 2010, pp. 218–227.

[22] L. Naish, H. J. Lee, and R. Kotagiri, "Spectral debugging with weights and incremental ranking," in *16th Asia-Pacific Software Engineering Conference, APSEC 2009*. IEEE, December 2009, pp. 168–175.

[23] H. Hsu, J. Jones, and A. Orso, "RAPID: Identifying bug signatures to support debugging activities," in *23rd IEEE/ACM International Conference on Automated Software Enginering, 2008. ASE 2008*, 2008, pp. 439–442.

[24] 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.

[25] T. Chilimbi, B. Liblit, K. Mehra, A. Nori, and K. Vaswani, "HOLMES: Effective statistical debugging via efficient path profiling," in *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 34–44.

[26] L. Naish, H. J. Lee, and R. Kotagiri, "Statements versus predicates in spectral bug localization," in *Proceedings of the 2010 Asia Pacific Software Engineering Conference*. IEEE, December 2010, pp. 375–384.