Effective Software Fault Localization Using an RBF Neural Network

W. Eric Wong, Vidroha Debroy, Richard Golden, Xiaofeng Xu, and Bhavani Thuraisingham

***Abstract*—We propose the application of a modified radial basis function neural network in the context of software fault localization, to assist programmers in locating bugs effectively. This neural network is trained to learn the relationship between the statement coverage information of a test case and its corresponding execution result, success or failure. The trained network is then given as input a set of *virtual* test cases, each covering a single statement. The output of the network, for each virtual test case, is considered to be the *suspiciousness* of the corresponding covered statement. A statement with a higher suspiciousness has a higher likelihood of containing a bug, and thus statements can be ranked in descending order of their suspiciousness. The ranking can then be examined one by one, starting from the top, until a bug is located. Case studies on 15 different programs were conducted, and the results clearly show that our proposed technique is more effective than several other popular, state of the art fault localization techniques. Further studies investigate the robustness of the proposed technique, and illustrate how it can easily be applied to programs with multiple bugs as well.**

***Index Terms*—Fault location, radial basis function neural networks, software debugging.**

ACRONYMS

RBF	radial basis function
BP	backward propagation
WBC	weighted bit comparison
PDG	program dependence graph
H3C	HeuristicIII(c)

NOTATION

P	a generic program
m	the number of (executable) statements in P
n	number of test cases used to test P
x	input to radial basis function
μ	receptive field center of radial basis function
σ	width of receptive field of radial basis function
$\|\|\mathrm{x}-\mu\|\|$	generic distance between x and μ
$\|\|\mathrm{x}-\mu\|\|_{\mathrm{WBC}}$	weighted bit-comparison-based dissimilarity between x and μ
β	parameter to control number of field centers
h	number of neurons in hidden layer
$R_j(x)$	activation function of jth hidden layer neuron
w_{ji}	weight associated with the link between the jth hidden layer neuron and ith output layer neuron
y_i	output of the ith output layer neuron
t_i	the ith test case
c_{t_i}	coverage vector of t_i
r_{t_i}	execution result of t_i

I. INTRODUCTION

REGARDLESS of how much effort has gone into developing a computer program,[1] it will still contain bugs.[1] But to remove bugs from a program, we must first be able to identify exactly where the bugs are. Known as fault localization, this task can be extremely tedious and time consuming, and is recognized to be one of the most expensive activities in program debugging [41]. This growing realization has sparked the development of several fault localization techniques that aim to assist developers in finding bugs. We propose the application of a modified radial basis function (RBF) neural network as a fault localization technique. This technique is more effective at locating bugs, in that a relatively smaller amount of code needs to be examined to find bugs, compared to other state of the art contemporary techniques.

Neural network-based models have several advantages over other comparable models, such as their ability to learn. Given a sample data set, a neural network can learn rules from the data with or without supervision. Neural networks are also more tolerant by virtue of the fact that the information is distributed among the weights on the connections, and so a few errors in the network have relatively less impact on the model. In addition, they have the capability to adapt their synaptic weights to changes in the surrounding environment. That is, a neural net-

Manuscript received February 13, 2011; revised April 24, 2011; accepted May 04, 2011. Date of publication October 21, 2011; date of current version March 02, 2012. The work was supported in part by Air Force Office of Scientific Research under contract FA9550-08-1-0260. Associate Editor: S. Shieh.

W. E. Wong, V. Debroy, X. Xu, and B. Thuraisingham are with the Department of Computer Science, University of Texas at Dallas, Dallas, TX 75247 USA.

R. Golden is with the School of Behavioral and Brain Sciences, University of Texas at Dallas, Dallas, TX 75247 USA.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TR.2011.2172031

[1]In this paper, we use "programs" and "software" interchangeably, and "bugs" and "faults" interchangeably. In addition, "a statement is covered by a test case" and "a statement is executed by a test case" are used interchangeably.

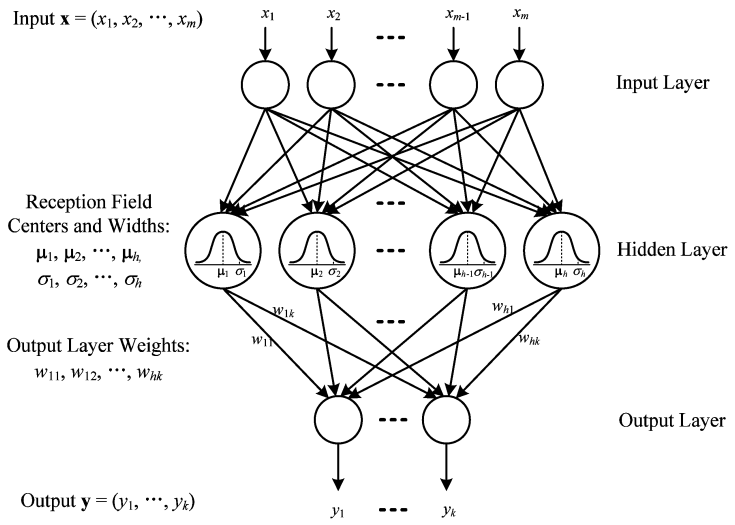

Fig. 1. A sample three-layer RBF neural network.

work trained to operate in a specific environment can be easily re-trained to deal with minor changes in the operating environmental conditions. Such qualities make neural networks popular among researchers, and therefore they have been successfully applied to many fields such as pattern recognition [10], system identification [6], intelligent control [27], and software engineering areas including risk analysis [28], cost estimation [37], reliability estimation [36], and reusability characterization [5]. However, to the best of our knowledge, they have not been applied to help find bugs except for in our previous study [48], which uses a back-propagation (BP) neural network-based technique for fault localization. In this paper, we propose to use an RBF neural network-based fault localization technique because RBF networks have several advantages over BP networks, including a faster learning rate, and a resistance to problems such as paralysis and local minima [19], [39].

A typical RBF neural network has a three-layer feed-forward structure that can be trained to learn an input-output relationship based on a data set. In this paper, the input is the statement coverage of a test case which indicates how the program is executed by the test case, and the output is the result (success or failure) of the corresponding program execution. Once the network has been trained, the coverage of a *virtual* test case with only one statement covered[1] is used as an input to compute the *suspiciousness* of the corresponding statement in terms of its likelihood of containing bugs. The larger the value of the output, the more suspicious the statement seems. Statements can then be ranked in descending order of their suspiciousness, such that programmers can examine the ranking of statements (starting from the top) one by one until the first faulty statement (statement containing bug(s)) is identified. Good fault localization techniques should rank faulty statements towards the top, if not at the very top, of their rankings. An assumption that is typically made (by all such fault localization techniques) is that programmers can correctly identify faulty statements as faulty, and by the same token, non-faulty statements as non-faulty, on examination.

There are three novel aspects to our work. First, we introduce a method for representing test cases, statement coverage, and execution results within a modified RBF neural network formalism. Moreover, the formulation of the problem in terms of training an artificial neural network with example test cases and execution results, and then testing with virtual test cases, is also novel. Second, we develop an algorithm (Fig. 2) to simultaneously determine the number of hidden neurons and their receptive field centers. Third, instead of using the traditional Euclidean distance, which has proven to be inappropriate in the fault localization context (Section III-C), a weighted bit-comparison-based dissimilarity is defined to estimate the distance between the statement coverage vectors of two test cases. Such dissimilarity is used to 1) estimate the number of hidden neurons and their receptive field centers, and 2) compute the output of each hidden neuron. We systematically evaluate the RBF technique across many different programs (*Unix suite*, *Space*, *Grep*, *Make*, *Ant* and *gcc*), each consisting of many faulty versions (we evaluate on both programs with exactly one, and programs with multiple faults in them). Previous experiments [49] have shown that the RBF technique is more effective than the Tarantula [18], SOBER [22], and Liblit05 [20] fault localization techniques. Thus, the focus of our current study is to evaluate RBF with respect to other fault localization techniques that have also reported better results than techniques such as Tarantula. Our data suggest that the RBF technique is also more effective than many other state of the art fault localization techniques that may be statistical in nature (the Crosstab technique [50]), heuristic-based (HeuristicIII [42]), or even similarity coefficient-based (Ochiai [1], and Jaccard [1]).

The remainder of the paper is organized as follows. Section II provides an overview of RBF neural networks, followed by Section III which explains the proposed fault localization technique, as well as presents an example to demonstrate its application. Section IV then reports on our case studies: we first describe our experimental design, and subsequently present data to evaluate the effectiveness of the RBF technique with respect to Crosstab [50], a technique that has reported very good fault localization results. In lieu of the fact that several other techniques have also reported good results, we compare our RBF technique to a representative set of such techniques

```
input: C = {c_t1, c_t2, ..., c_tn} and β                    output: O
1   Begin
2       O ← ∅
3       for each c ∈ C {
4           Temp ← false
5           for each μ ∈ O {
6               if (||c−μ||_WBC < β) {
7                   Temp ← true
8                   break
9               }
10          }
11          if (Temp = = false)
12              O ← {c} ∪ O
13      }
14      output O
15  End
```

Fig. 2. The algorithm to determine the number of hidden neurons, and their receptive field centers.

(Ochiai [1], Jaccard [1], and HeuristicIII [42]) in Section V. A demonstration of the application of our technique to a program with multiple bugs appears in Section VI, and the robustness of the RBF technique is evaluated in Section VII. Section VIII then discusses issues and concerns relevant to the RBF technique, and threats to validity, while Section IX overviews some related studies. Finally, our conclusions, and a discussion on future work, are presented in Section X.

II. An Overview of RBF Neural Networks

We first present a general description of neural networks, followed by a more specific discussion on the RBF neural networks which form the basis for the work presented herein.

A. Neural Networks

Traditionally, the term "neural network" has been used to refer to a network of biological neurons. The modern definition of this term is an artificial construct whose behavior is based on that of a network of artificial neurons. These neurons are connected together with weighted connections following a certain structure. Each neuron has an activation function that describes the relationship between the input and the output of the neuron [11]. The data can be processed in parallel by different neurons, and distributed on the weights of the connections between neurons. Different neural network models have been developed, including BP neural networks [11], RBF neural networks [12], self-organizing map (SOM) neural networks [14], and adaptive resonance theory (ART) neural networks [12]. A particularly important attribute of a neural network is that it can learn from experience. Such learning is normally accomplished through an adaptive process using a learning algorithm. These algorithms can be divided into two categories: *supervised,* and *unsupervised* [39]. Each network learning algorithm has certain strengths and weaknesses in the areas of reliability, performance, and generality; however, none has a clear advantage over another.

In fault localization, the output of a given input can be defined as a binary value of 0 or 1, where 1 represents a program failure

$$\begin{bmatrix} \mathbf{c}_{v_1} \\ \mathbf{c}_{v_2} \\ \mathbf{c}_{v_3} \\ \mathbf{c}_{v_4} \\ \mathbf{c}_{v_5} \\ \mathbf{c}_{v_6} \\ \mathbf{c}_{v_7} \\ \mathbf{c}_{v_8} \\ \mathbf{c}_{v_9} \\ \mathbf{c}_{v_{10}} \end{bmatrix} = \begin{bmatrix} 1&0&0&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0&0&0 \\ 0&0&0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&0&0&1 \end{bmatrix}$$

(a)

$\hat{r}_{v_1}$	0.0384	$\hat{r}_{v_6}$	0.0179
$\hat{r}_{v_2}$	0.0481	$\hat{r}_{v_7}$	0.0157
$\hat{r}_{v_3}$	0.1246	$\hat{r}_{v_8}$	0.2900
$\hat{r}_{v_4}$	0.0768	$\hat{r}_{v_9}$	0.0066
$\hat{r}_{v_5}$	0.0173	$\hat{r}_{v_{10}}$	0.0782

(b)

Fig. 3. Inputs and outputs (statement suspiciousness) based on the example. Part (a): Input coverage vectors. Part (b) Outputs of the trained network are the suspiciousness of the corresponding statements.

on this input, and 0 represents a successful execution. With this definition, the expected output of each network input (the statement coverage of each test case) is known because we know exactly whether the corresponding program execution fails or succeeds. Moreover, two similar inputs can produce different outputs because the program execution may fail on one input but succeed on the other. This makes unsupervised learning algorithms inappropriate for us because they adjust network weights so that similar inputs produce similar outputs. Therefore, neural networks using supervised learning algorithms are better candidates to solve the fault localization problem. Although BP networks are widely used networks for supervised learning, RBF networks, whose output layer weights are trained in a supervised way, are even better in our case as they can learn faster than BP networks, and do not suffer from pathologies like paralysis and local minima problems as BP networks do [19], [39].

B. RBF Neural Networks

An RBF is a real-valued function whose value depends only on the distance from its receptive field center $\boldsymbol{\mu}$ to the input $\mathbf{x}$. It is a strictly positive radially symmetric function, where the center has the unique maximum, and the value drops off rapidly to zero away from the center. When the distance between $\mathbf{x}$ and $\boldsymbol{\mu}$ (denoted as $\|\mathbf{x}-\boldsymbol{\mu}\|$) is smaller than the receptive field width σ, the function has an appreciable value.

A typical RBF neural network has a three-layer feed-forward structure. The first layer is the input layer, which passes inputs to the (second) hidden layer without changing their values. The hidden layer is where all neurons simultaneously receive the n-dimensional real-valued input vector $\mathbf{x}$. Each neuron in this layer uses an RBF as the activation function. We made use of the Gaussian basis function [12] as it is one of the most popular choices for employment in RBF networks [35].

$$R_j(\mathbf{x}) = \exp\left(-\frac{\|\mathbf{x}-\boldsymbol{\mu}_j\|^2}{2\sigma_j^2}\right) \tag{1}$$

Usually the distance in (1) is the Euclidean distance between $\mathbf{x}$ and $\boldsymbol{\mu}$, but in this paper we use a weighted bit-comparison-based dissimilarity. To make a distinction, hereafter, we use $\|\mathbf{x}-\boldsymbol{\mu}\|$ to represent a *generic* distance, and $\|\mathbf{x}-\boldsymbol{\mu}\|_{\mathrm{WBC}}$ ((7) in

TABLE I
THE COVERAGE DATA AND EXECUTION RESULTS USED IN THE EXAMPLE

	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	r
t_1	(1)	1	1	1	0	1	0	0	1	1	(0)
t_2	1	0	0	1	1	(0)	1	0	0	1	0
t_3	1	1	1	0	0	1	0	0	1	1	0
t_4	1	0	1	0	0	1	1	0	1	1	0
t_5	1	1	1	0	1	0	0	0	1	0	0
t_6	1	1	1	1	0	0	0	1	1	1	(1)
t_7	1	0	1	1	1	1	1	1	0	1	1

t_1 is a successful test

t_6 is a failed test

s_1 is executed by t_1

s_6 is not executed by t_2

TABLE II
SUMMARY OF THE UNIX SUITE

Program	Description	Number of faulty versions	LOC	Number of test cases
Cal	Print a calendar for a specified year or month	20	202	162
Checkeq	Report missing or unbalanced delimiters and .EQ/.EN pairs	20	102	166
Col	Filter reverse paper motions from nroff output	30	308	156
Comm	Select or reject lines common to two sorted files	12	167	186
Crypt	Encrypt and decrypt a file using a user supplied password	14	134	156
Look	Find words in the system dictionary or lines in a sorted list	14	170	193
Sort	Sort and merge files	21	913	997
Spline	Interpolate smooth curves	13	338	700
Tr	Translate characters	11	137	870
Uniq	Report or remove adjacent duplicate lines	17	143	431

†LOC is the size of the program in terms of the number of lines of code.

Section III-C) for the weighted bit-comparison-based dissimilarity. The third layer is the output layer. The output can be expressed as $\mathbf{y} = [y_1, y_2, \ldots, y_k]$ with y_i as the output of the ith neuron given by

$$y_i = \sum_{j=1}^{h} w_{ji} R_j(\mathbf{x}) \quad \text{for} \quad i = [1, k] \tag{2}$$

An RBF network implements a mapping from the m dimensional real-valued input space to the k dimensional real-valued output space with a hidden layer space in between. The transformation from the input space to the hidden-layer space is nonlinear, whereas the transformation from the hidden-layer space to the output space is linear [14]. Fig. 1 shows an RBF network with m neurons in the input layer, h neurons in the hidden layer, and k neurons in the output layer. The parameters to be trained are the centers $(\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \ldots, \boldsymbol{\mu}_h)$ and widths $(\sigma_1, \sigma_2, \ldots, \sigma_h)$ of the receptive fields of hidden layer neurons, and the output layer weights. Many methods have been proposed to train these parameters [12]. Section III-B explains how they are trained in our study.

III. THE PROPOSED RBF NEURAL NETWORK-BASED FAULT LOCALIZATION TECHNIQUE

We first explain the use of an RBF neural network to compute the suspiciousness of each statement in a program for effective fault localization, and then introduce a two-stage training of the RBF network, including an algorithm for simultaneously determining both the number and the receptive field centers of hidden neurons. We also provide the formal definition of a weighted bit-comparison-based dissimilarity used by the RBF during the fault localization process, followed by an example to demonstrate the use of our proposed technique.

A. Fault Localization Using an RBF Neural Network

Suppose we have a program P with m statements, executed on n test cases. Let s_j be the jth statement of P. The vector $\mathbf{c}_{t_i}$ provides us with information on how the program P is covered by test t_i. In this paper, such coverage is reported in terms of which statements[2] in P are executed by t_i. We have $\mathbf{c}_{t_i} = [(\mathbf{c}_{t_i})_1, (\mathbf{c}_{t_i})_2, \cdots, (\mathbf{c}_{t_i})_m]$ where

$$(\mathbf{c}_{t_i})_j = \begin{cases} 0, & \text{if statement } s_j \text{ is not covered by test } t_i \\ 1, & \text{if statement } s_j \text{ is covered by test } t_i \end{cases}$$
$$\text{for } 1 \le j \le m$$

The value of r_{t_i} depends on whether the program execution of t_i succeeds or fails. It has a value of 1 if the execution fails, and a value of 0 if the execution succeeds. We construct an RBF neural network with m input layer neurons, each of which corresponds to one element in a given c_{t_i}; and one output layer neuron, corresponding to r_{t_i}, the execution result of test t_i. In addition, there is a hidden layer between the input and output layers, and the number of hidden neurons can be determined by using the algorithm in Fig. 2, which will be explained in Section III-B. Each of these neurons uses the Gaussian basis function as the activation function. The receptive field center and width of each hidden layer neuron, and the output layer weights, are established by training the underlying network.

[2]In addition to statement coverage, without loss of generality, our technique can also be applied to other program components such as functions, blocks, decisions, c-uses and p-uses [51].

TABLE III
SUMMARY OF SPACE, GREP, MAKE, AND ANT

Program	Number of faulty versions	LOC	Number of test cases
Space	38	9126	13585
Grep (version 2.2)	19	12653	470
Make (version 3.76.1)	31	20014	793
Ant (version 1.6beta)	23	80500	877

Once an RBF network is trained, it provides a good mapping between the input (in our case the coverage vector of a test case) and the output (the corresponding execution result). It can then be used to identify suspicious code of a given program in terms of its likelihood of containing bugs. To do so, we use a set of *virtual* test cases $v_1, v_2, \ldots, v_m$ whose coverage vectors are $\mathbf{c}_{v_1}, \mathbf{c}_{v_2}, \cdots, \mathbf{c}_{v_m}$, where each test v_j covers only one statement s_j (see Part (a) of Fig. 3). As reported in [3], [45], [47], if the execution of a test case fails, program bugs that are responsible for this failure are most likely to be contained in the corresponding execution slice, the statements executed by this failed test case.[3] Hence, if the execution of v_j fails, the probability that the bugs are contained in s_j is high. This result suggests that, during the fault localization, we should first examine the statements whose corresponding virtual test case fails. However, the execution results of these virtual tests can rarely be collected in the real world because it is very difficult, if not impossible, to construct such tests.[4] Nevertheless, when the coverage vector $\mathbf{c}_{v_j}$ of a virtual test case v_j is input to the trained neural network, its output $\hat{r}_{v_j}$ is the conditional expectation of whether the execution of v_j fails given $\mathbf{c}_{v_j}$. This result implies that the larger the value of $\hat{r}_{v_j}$, the more likely it is that the execution of v_j fails. Therefore, the larger the value of $\hat{r}_{v_j}$, the more likely it is that s_j contains a bug. We can treat $\hat{r}_{v_j}$ as the suspiciousness of s_j in terms of its likelihood of containing a bug. The procedure for using the RBF technique for fault localization is described as follows.

1) Build a modified RBF neural network with m input neurons, and one output neuron. Each neuron in the hidden layer uses the Gaussian basis function as its activation function.
2) Determine the number of hidden neurons h, and the receptive field center and width of each hidden neuron.
3) Use the generalized inverse (Moore-Penrose pseudo-inverse) to compute the optimal linear mapping from the hidden neurons to the output neuron.
4) Use the virtual coverage vectors $\mathbf{c}_{v_j}$, $1 \leq j \leq m$ as the inputs to the trained network to produce the outputs $\hat{r}_{v_j}$, $1 \leq j \leq m$.
5) Assign $\hat{r}_{v_j}$ as the suspiciousness of the jth statement.

The statements can now be examined one by one, in descending order of suspiciousness, until a fault is located. Hereafter, we refer to our proposed technique simply as 'RBF'. We take this opportunity to emphasize that the traditional RBF neural network has been modified to better fit our fault localization context. First, in Step 2, we develop an algorithm (Fig. 2) to simultaneously determine the number of hidden neurons and their receptive field centers. Second, we define a weighted bit-comparison-based dissimilarity to estimate the distance between two coverage vectors, as opposed to using the traditional Euclidean distance. Further details are provided in Sections III-B and III-C.

[3]In some situations, a test case may fail only because a previous test did not set up an appropriate execution environment. To account for this, we combine these test cases into a single failed test, with an execution slice consisting of the union of each test case's slice [45], [47].

[4]In general, the virtual test cases are not "real" test cases and their coverage vectors are not used as training data for the RBF network.

B. Training of the RBF Neural Network

In this section, we discuss details of the training procedure as described in Section III-A. The training of an RBF neural network can be divided into two stages [39]. First, the number of neurons in the hidden layer, the receptive field center, and width of each hidden layer neuron should be assigned values. Second, the output layer weights have to be trained. Many methods have been proposed to determine the receptive field centers. Using standard k-means clustering, input data are assigned to k clusters, with the center of each cluster taken to be the receptive field center of a hidden layer neuron [8], [12], [21], [38]. Unfortunately, this approach does not provide any guidance as to how many clusters should be used; the number of clusters (and so, the number of receptive field centers) must be chosen arbitrarily. Another disadvantage is that the k-means approach is very sensitive to the initial starting values. Its performance will significantly depend on the arbitrarily selected initial receptive field centers.

To overcome these problems, we developed an algorithm (as shown in Fig. 2) to simultaneously estimate the number of hidden neurons and their receptive field centers. The inputs to this algorithm are the coverage vectors $\{\mathbf{c}_{t_1}, \mathbf{c}_{t_2}, \cdots, \mathbf{c}_{t_n}\}$, and a parameter β $(0 \leq \beta < 1)$ for controlling the number of field centers. The output is a set of receptive field centers $\{\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \cdots, \boldsymbol{\mu}_h\}$ which is a subset of the input vectors such that $\|\boldsymbol{\mu}_i - \boldsymbol{\mu}_j\|_{\mathrm{WBC}} \geq \beta$ for any i and j, $i \neq j$, where $\| \|_{\mathrm{WBC}}$ is the weighted bit-comparison-based dissimilarity defined in Section III-C. Our algorithm not only assigns values to each receptive field center, but also decides the number of hidden neurons because each hidden neuron contains exactly one center. The larger the value of β is, the smaller the number of neurons to be used in the hidden layer, which makes the training at the second stage much faster (as explained at the end of this section). However, if the number of hidden layer neurons is too small, then the mapping between the input and the output defined by the neural network loses its accuracy.

Once the receptive field centers have been found, we can use different heuristics to determine their widths to get a smooth interpolation. Park and Sandberg [29], [30] show that an RBF neural network using a single global fixed value σ for all σ_j has

TABLE IV
PAIRWISE COMPARISON BETWEEN RBF AND CROSSTAB

		RBFBest versus CBest	RBFWorst versus CWorst	RBFWorst versus CBest
Unix	More effective	57	62	26
	Same effectiveness	89	83	12
	Less effective	26	27	134
Grep	More effective	11	10	6
	Same effectiveness	5	5	0
	Less effective	3	4	13
Space	More effective	10	10	5
	Same effectiveness	16	15	1
	Less effective	9	10	28
Make	More effective	18	19	9
	Same effectiveness	6	7	0
	Less effective	7	5	2
Ant	More effective	8	8	7
	Same effectiveness	12	10	1
	Less effective	3	5	15

the capability of universal approximation. Moody and Darken [25] suggest that a good estimate of σ is the average over all distances between the center of each neuron and that of its nearest neighbor. In this paper, we use a similar heuristic to define the global width σ as

$$\sigma = \frac{1}{h}\sum_{j=1}^{h} \left\|\boldsymbol{\mu}_j - \boldsymbol{\mu}_j^*\right\|_{\mathrm{WBC}} \tag{3}$$

where $\|\boldsymbol{\mu}_j - \boldsymbol{\mu}_j^*\|_{\mathrm{WBC}}$ is the weighted bit-comparison-based dissimilarity between $\boldsymbol{\mu}_j$ and its nearest neighbor $\boldsymbol{\mu}_j^*$.

After the centers and widths of the receptive fields of the RBFs in the hidden layer are determined, the remaining parameters that need to be trained are the hidden-to-output layer weights $(w_1, w_2, \ldots, w_h)$.[5] To do so, we first select a training set composed of input coverage vectors $(\mathbf{c}_{t_1}, \mathbf{c}_{t_2}, \cdots, \mathbf{c}_{t_n})$, and the corresponding expected outputs $(r_{t_1}, r_{t_2}, \cdots, r_{t_n})$. For an input coverage vector $\mathbf{c}_{t_i}$, its actual output from the network $\hat{r}_{t_i}$ is computed as

$$\begin{aligned}\hat{r}_{t_i} &= \sum_{j=1}^{h} w_j R_j(\mathbf{c}_{t_i}) \text{ where } R_j(\mathbf{x}) \\ &= \exp\left(-\frac{\|\mathbf{x}-\boldsymbol{\mu}_j\|_{\mathrm{WBC}}^2}{2\sigma_j^2}\right) \text{ for } 1 \le j \le h\end{aligned} \tag{4}$$

Thus, the output of the network is

$$\hat{\mathbf{r}} = \mathbf{A}\mathbf{w} \tag{5}$$

[5]In our fault localization study, the RBF network is a single-output network which produces only one output for each input coverage vector. Hence, each output layer weight (say w_j) has only one subscript, rather than two subscripts as in (2) Reference source not found., showing the connection between the output layer neuron and the corresponding hidden layer neuron (jth hidden neuron in this case).

where

$$\mathbf{A} = \begin{bmatrix} R_1(\mathbf{c}_{t_1}) & R_2(\mathbf{c}_{t_1}) & \cdots & R_h(\mathbf{c}_{t_1}) \\ R_1(\mathbf{c}_{t_2}) & R_2(\mathbf{c}_{t_2}) & \cdots & R_h(\mathbf{c}_{t_2}) \\ \vdots & \vdots & \ddots & \vdots \\ R_1(\mathbf{c}_{t_n}) & R_2(\mathbf{c}_{t_n}) & \cdots & R_h(\mathbf{c}_{t_n}) \end{bmatrix},$$
$$\mathbf{w} = [w_1, w_2, \cdots, w_h]^T, \text{ and } \hat{\mathbf{r}} = [\hat{r}_{t_1}, \hat{r}_{t_2}, \cdots, \hat{r}_{t_n}]^T.$$

Also, let the expected output $\mathbf{r} = [r_{t_1}, r_{t_2}, \cdots, r_{t_n}]^T$, and the *prediction error* across the entire set of training data be defined as $\|\hat{r} - r\|^2$ (the sum of squared error between $\hat{r}$ and $\mathbf{r}$). To find the optimal weights $\mathbf{w}^*$, we have to compute $\mathbf{w}^* = \arg\min_{\mathbf{w}} \|\hat{\mathbf{r}}-\mathbf{r}\|^2 = \arg\min_{\mathbf{w}} \|\mathbf{A}\mathbf{w}-\mathbf{r}\|^2$. For this calculation we use the generalized inverse (Moore-Penrose pseudo-inverse) of $\mathbf{A}$ [31]:

$$\mathbf{w}^* = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{r} \tag{6}$$

The complexity of computing $\mathbf{w}^*$ depends on the size of $\mathbf{A}$, which is $n \times h$, where n is the number of test cases in the training set, and h is the number of hidden neurons. For a fixed n, the smaller h is, the smaller the complexity. Therefore, an RBF network with a smaller number of hidden neurons can be trained faster than a network with more hidden neurons.

C. Definition of a Weighted Bit-Comparison-Based Dissimilarity

From (4), for a given test case t_i and its input coverage vector $\mathbf{c}_{t_i}$, the actual output $\hat{r}_{t_i}$ is a linear combination of the activation functions of all hidden layer neurons. Each R_j depends on the distance $\|\mathbf{x} - \boldsymbol{\mu}_j\|$ (referring to (1)). In our case, $\mathbf{x}$ is the input coverage vector $\mathbf{c}_{t_i}$, and $\boldsymbol{\mu}_j$ is the receptive field center of the jth hidden layer neuron. So, we have $\|\mathbf{x} - \boldsymbol{\mu}_j\| = \|\mathbf{c}_{t_i} - \boldsymbol{\mu}_j\|$. From the algorithm in Fig. 2, we observe that the set of receptive centers is a subset of the coverage vectors. This observation implies each μ_j by itself is also the coverage vector of a certain test case. As a result, the distance $\|\mathbf{x} - \boldsymbol{\mu}_j\|$ can also be viewed as the distance between two coverage vectors.

The most commonly used distance is the Euclidean distance. However, this distance is not suitable for our problem because it cannot represent the difference between coverage vectors accurately. For the purpose of explanation, let us use the following example. Suppose we have an RBF network trained by $\mathbf{c}_{t_1} = [0,0,1,1,0]$ and $\mathbf{c}_{t_2} = [1,0,1,1,1]$, and their execution results r_{t_1} and r_{t_2}. Suppose also the trained network has two neurons in the hidden layer with $\boldsymbol{\mu}_1 = \mathbf{c}_{t_1}$, and $\boldsymbol{\mu}_2 = \mathbf{c}_{t_2}$. When we have $\mathbf{c}_{v_1} = [1,0,0,0,0]$ as the input to the trained neural network, the output is

$$\hat{r}_{v_1} = w_1 R_1(\mathbf{c}_{v_1}) + w_2 R_2(\mathbf{c}_{v_1}) = w_1 \exp\left(-\frac{\|\mathbf{c}_{v_1} - \mathbf{c}_{t_1}\|^2}{2\sigma_1^2}\right) + w_2 \exp\left(-\frac{\|\mathbf{c}_{v_1} - \mathbf{c}_{t_2}\|^2}{2\sigma_2^2}\right),$$

where $\sigma_1 = \sigma_2 = \sigma$.

Because the first statement is covered by t_2 and v_1, but not t_1, we should have $R_1(\mathbf{c}_{v_1}) \neq R_2(\mathbf{c}_{v_1})$, which implies $\|\mathbf{c}_{v_1} - \mathbf{c}_{t_1}\| \neq \|\mathbf{c}_{v_1} - \mathbf{c}_{t_2}\|$. More precisely, v_1 is more similar to t_2 than to t_1, which means the output of the hidden neuron with $\mathbf{c}_{t_2}$ as its center should contribute more to the network output. That is, we should have $R_1(\mathbf{c}_{v_1}) < R_2(\mathbf{c}_{v_1})$, and therefore $\|\mathbf{c}_{v_1} - \mathbf{c}_{t_1}\| > \|\mathbf{c}_{v_1} - \mathbf{c}_{t_2}\|$. However, the Euclidean distance between $\mathbf{c}_{v_1}$ and $\mathbf{c}_{t_1}$ is the same as that between $\mathbf{c}_{v_1}$ and $\mathbf{c}_{t_2}$. To overcome this problem, we use a weighted bit-comparison-based dissimilarity defined as

$$\left\|\mathbf{c}_{t_i} - \boldsymbol{\mu}_j\right\|_{\mathrm{WBC}} = \sqrt{1 - \cos\theta_{\mathbf{c}_{t_i},\boldsymbol{\mu}_j}} \tag{7}$$

Where

$$\cos\theta_{\mathbf{c}_{t_i},\boldsymbol{\mu}_j} = \frac{\sum_{k=1}^{m} (\mathbf{c}_{t_i})_k (\boldsymbol{\mu}_j)_k}{\sqrt{\sum_{k=1}^{m} [(\mathbf{c}_{t_i})_k]^2} \times \sqrt{\sum_{k=1}^{m} [(\boldsymbol{\mu}_j)_k]^2}},$$

$(\mathbf{c}_{t_i})_k$ and $(\boldsymbol{\mu}_j)_k$ are the kth elements of $\mathbf{c}_{t_i}$ and $\boldsymbol{\mu}_j$, respectively. The dissimilarity measure between the two binary vectors in (7) is more desirable because it effectively takes into account the number of bits that are both 1 in two coverage vectors (those statements covered by both vectors).

In the above example, if we replace the Euclidean distance by the weighted bit-comparison-based dissimilarity, then we have $\|\mathbf{c}_{v_1} - \mathbf{c}_{t_1}\|_{\mathrm{WBC}} = 1$, which is greater than $\|\mathbf{c}_{v_1} - \mathbf{c}_{t_2}\|_{\mathrm{WBC}} = \sqrt{1/2}$.

D. An Illustrative Example of the Proposed Technique

Suppose we have a program with ten statements s_j, $1 \leqslant j \leqslant 10$, and that a total of seven test cases have been executed on the program. Table I gives the coverage vector and the execution result of each test.

We follow the steps listed in Section III-A. An RBF neural network with ten input neurons and one output neuron is constructed. Using the algorithm in Fig. 2 with $\beta = 0.1$, we find that each coverage vector (one coverage vector per test case) also serves as the receptive field center of a hidden neuron. This implies there are seven neurons in the hidden layer. The field width σ computed by using (3) is 0.395. The output layer weights are trained by the data in Table I. We have $\mathbf{w} = [w_1, w_2, w_3, w_4, w_5, w_6, w_7]^{\mathrm{T}} = [-1.326, -0.665, 0.391, -0.378, -0.308, 1.531, 1.381]^{\mathrm{T}}$. We use the coverage vectors of the virtual test cases in Part (a) of Fig. 3 as the inputs to the trained network. The outputs (i.e., the suspiciousness values of the respective statements) are shown in Part (b). Ranking the statements in descending order of their suspiciousness, we have: s_8, s_3, s_{10}, s_4, s_2, s_1, s_6, s_5, s_7, s_9. Thus, statement s_8 is the most suspicious, followed by statement s_3, and so on; and the statements would be examined in this order until the faulty statement is found.

IV. Empirical Evaluation of RBF and Comparison With Crosstab

In this section, we report our case studies using RBF for fault localization. Previous experiments have shown that the RBF technique is more effective (the effectiveness measure is formally defined in Section IV-C) than techniques such as Tarantula [49]. We now compare RBF to a statistical fault localization technique called Crosstab [50], which has also reported results that are much better (i.e., more effective) than those of Tarantula.

A. Subject Programs

We conducted case studies on five suites of programs: Unix suite, Space, Make, Grep, and Ant, the first four of which are written in C, and the fifth in JAVA.

The Unix suite consists of ten Unix utility programs, listed in Table II. Because these programs have been thoroughly used, they are a reliable basis for evaluating the behavior of fault-injected programs derived from them. These faulty versions were created using mutation-based fault injection, which has been shown in a recent study [4] as an appropriate approach for simulating real faults in software testing research. More discussion on this approach appears in Section VIII-D. The same programs and test cases were also used in other studies [44], [48]. The Space program developed at the European Space Agency provides an interface that allows the user to describe the configuration of an array of antennas by using a high level language. The correct version, its faulty versions, and a test suite were downloaded from [15]. Three faulty versions were not used in our study because no downloaded test cases could reveal the faults in these versions, whereas eight faulty versions were excluded in [18] for various reasons.

The Grep program searches for a pattern in a file. The correct and faulty source code of version 2.2, and a test suite, were downloaded from [15]. Additional bugs from [22], and faulty versions created using mutation-based fault injection, were also used. Similarly, we downloaded version 1.6beta of Ant (a very large JAVA-based build tool), and version 3.76.1 of Make (which is a software utility that manages the building of executables and other products from source code) from [15]. All faulty versions that did not fail on any of the downloaded test cases were excluded from our studies. As before, additional faulty versions were created using mutation-based fault injection. Similar to [18], multiple-line statements were combined as one source code line so that they would be counted only as one executable statement. Table III gives a summary of the Space, Grep, Make, and Ant programs. These programs

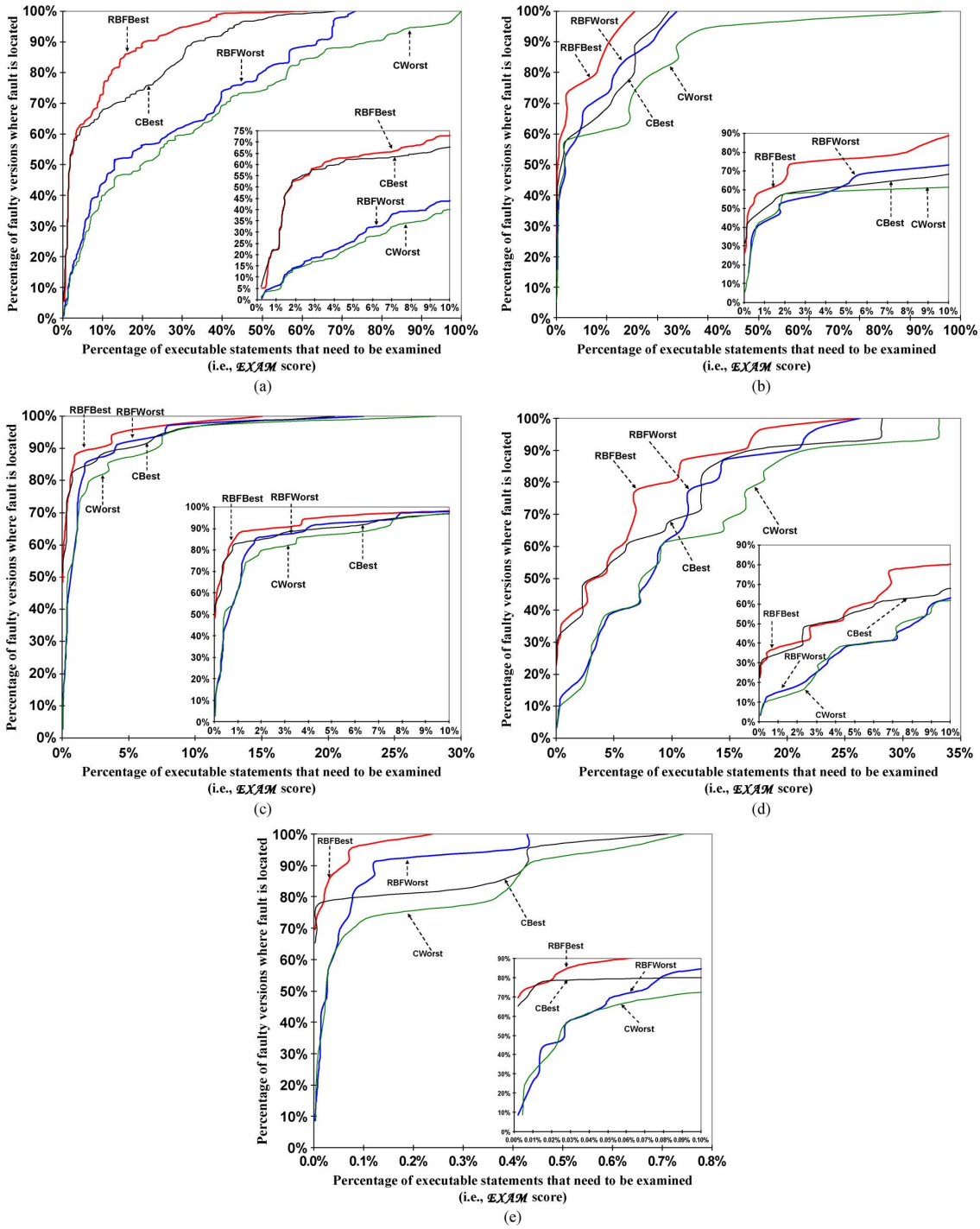

Fig. 4. Effectiveness comparison between RBF, and Crosstab. Part (a). The Unix suite. Part (b). The Grep program. Part (c). The Space program. Part (d). The Make program. Part (e). The Ant program.

vary dramatically in both their sizes and functionalities, which makes our results even more convincing and representative. As of now, each faulty version contains exactly one bug which may span multiple statements that are not necessarily contiguous or multiple functions. The same approach is also used in many fault localization studies [7], [18], [22], [45], [47], [48]. However, the RBF fault localization technique can be extended to handle programs with multiple bugs as well, and this extension is further discussed in Section VI.

TABLE V
TOTAL NUMBER OF STATEMENTS EXAMINED BY RBF, AND CROSSTAB

	RBFBest	CBest	RBFWorst	CWorst
Unix	1302	2462	4758	6030
Grep	20 5	4005	3964	7443
Space	1337	2016	2417	3093
Make	9 88	12403	14590	18142
Ant	233	1076	757	1578

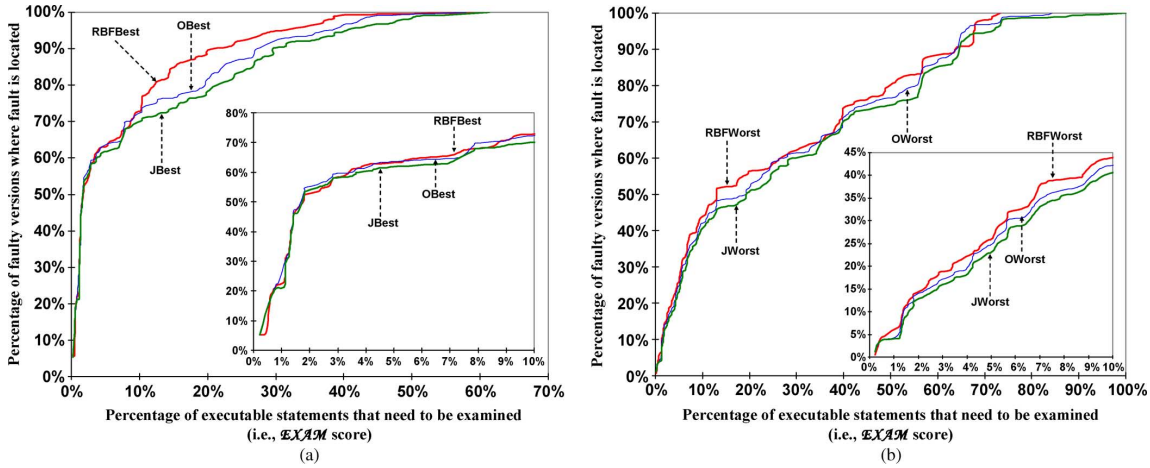

Fig. 5. Effectiveness comparison of RBF against Ochiai, and Jaccard on the Unix suite. Part (a). Best case comparison. Part (b). Worst case comparison.

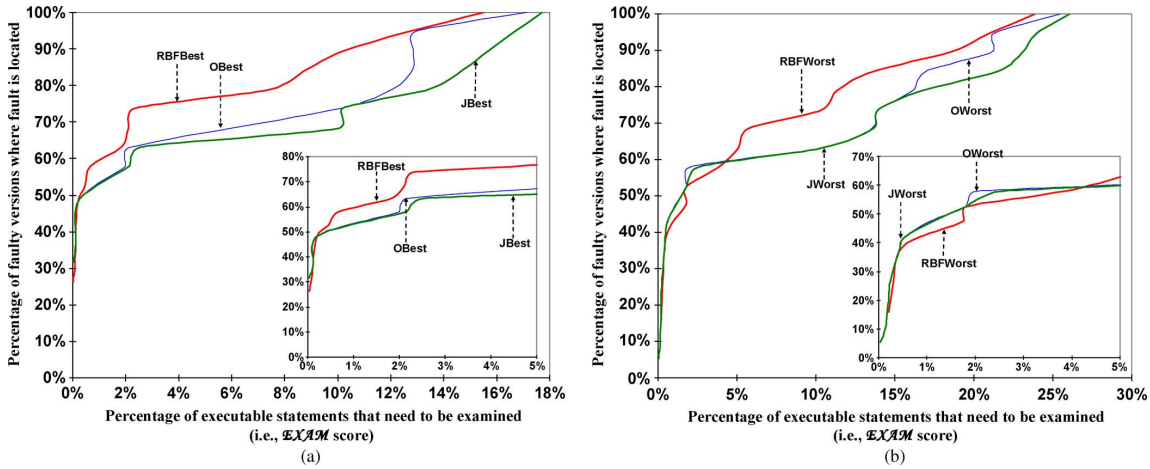

Fig. 6. Effectiveness comparison of RBF against Ochiai, and Jaccard on the Grep program. Part (a). Best case effectiveness comparison. Part (b). Worst case effectiveness comparison.

B. Data Collection

For the Unix suite and Space program, all executions were on a PC with a 2.13GHz Intel Core 2 Duo CPU, and 8GB physical memory. The operating system was SunOS 5.10 (Solaris 10), and the compiler used was gcc 3.4.3. For Grep and Make, the executions were on a Sun-Fire-280R machine with SunOS 5.10 as the operating system, and gcc 3.4.4 as the compiler. The Ant data was collected on the same machine as for the Unix suite and Space program, and the JAVA compiler version was 1.5.0_06. Each faulty version was executed against all its available test cases. The statement coverage with respect to each test case was measured by using a revised version of χSuds [51], which could collect runtime trace correctly even if a program execution was crashed due to a segmentation fault, for the C programs; and Clover for the JAVA-based Ant program [16]. The success or failure of an execution was determined by comparing the outputs of the faulty version with the correct version of a program.

We make several assumptions at this point.

- Although a bug may span multiple statements, which may not be contiguous, or even multiple functions, the fault localization stops when the first statement containing the bug is reached. Note that in no way does this mean the proposed RBF technique is limited to faults that are only across a single line.
- We also assume perfect bug detection, that is, a bug in a statement will be detected by a programmer if the statement is examined. If such perfect bug detection does not hold, then the number of statements that need to be examined to find the bug may increase. The same concern applies to all the fault localization techniques discussed in this paper.
- In addition, we assume the cost of examining each statement is fixed.

In Section IV-D, the results of our RBF technique are compared with those of Crosstab to evaluate the relative effectiveness of each technique. For a fair comparison, we compute the effectiveness of both techniques (RBF, and Crosstab) using the same data. Note that statistics such as fault revealing behavior and statement coverage of each test can vary under different compilers, operating systems, and hardware platforms. Also, the ability of the coverage measurement tool (revised version of χSuds in our experiments versus gcc with gcov in [18], and

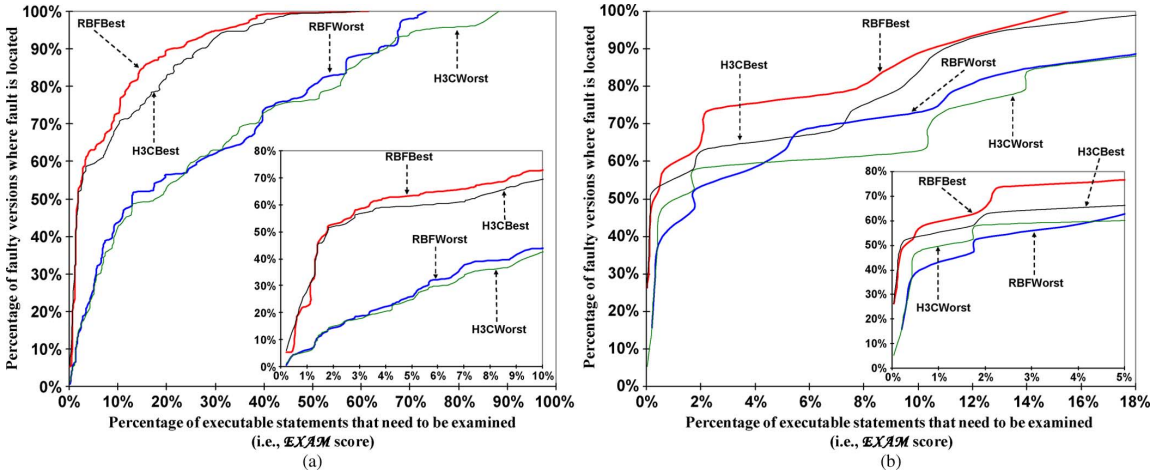

Fig. 7. Effectiveness comparison of RBF against H3C. Part (a). Comparison on the Unix suite. Part (b). Comparison on the Grep program.

Clover [16]) to properly handle segmentation faults has an impact on the use of certain faulty versions. However, such variance is not expected to be very large in nature, and does not detract from the validity of our experiments, or the accompanying results.

C. Criteria to Evaluate Fault Localization Effectiveness

In previous studies, Renieris *et al.* [33] assign a score to every faulty version of each subject program, which is defined as the percentage of the program that need not be examined to find a faulty statement in the program, or a faulty node in the corresponding program dependence graph. This score (or effectiveness measure) is later adopted by Cleve and Zeller in [7], and is defined as $1 - |\mathrm{N}|/|\mathrm{PDG}|$ where N is the set of all nodes examined, and PDG is the set of all nodes in the program dependence graph. Instead of the program dependence graph, Tarantula directly uses the program's source code, and therefore, to make their effectiveness computations comparable to those of the program dependence graph, Jones *et al.* [18] consider only executable statements to compute their score. The comparison now becomes a fair one; only statements that can be represented in the program dependence graph are considered. Because the RBF fault localizer also operates directly on the program's source code, we follow the same strategy and consider only executable statements in all of our experiments. However, while the authors of [18] define their score to be the percentage of code that need not be examined to find a fault, it is more straightforward to present the percentage of code that *must be examined* to find the fault. This modified score is hereafter referred to as $\mathcal{EXAM}$, and is defined as the percentage of executable statements that have to be examined until the first statement containing the bug is reached. We note that the two scores are equivalent, and it is easy to derive one from the other. A similar modification is made by the authors of [22] where they define their effectiveness (T-score) as $T = (|V_{examined}|/|V|) \times 100\%$ where $|V|$ is the size of the program dependence graph, i.e., the total number of statements, and $|V_{examined}|$ is the number of statements examined in a breadth first search before a faulty node is reached.

In this paper, the effectiveness of various fault localization techniques is compared on the basis of the $\mathcal{EXAM}$ score. For a faulty program α, if its $\mathcal{EXAM}$ score assigned by technique A is less than that assigned by technique B (that is, A can guide the programmer to the fault in α by examining less code than B), then A is said to be more effective (better) than B for locating the bug in α. If there is more than one faulty program, technique A can be considered more effective than B, if A assigns a smaller $\mathcal{EXAM}$ score to more faulty programs than B. Although this second criterion quantifies the number of faulty programs on which one technique is better than another (or vice-versa), it does not count the magnitude of the difference in the respective EXAM scores. This approach leads to a third criterion which measures the cumulative magnitude of differences between fault localization techniques across all the faulty versions under study. Suppose we have k faulty programs; and $A(i)$, and $B(i)$ give the total number of statements that must be examined to locate the fault in the ith faulty program by techniques A, and B, respectively. Then we can say A is more effective than B if $\sum_i^k A(i) < \sum_i^k B(i)$.

The second and the third criteria (which are both based on the original EXAM score) are somewhat complementary to each other because each is able to quantify what the other cannot. The second criterion can reveal how many faulty versions one technique may perform better against than another without reflecting the *magnitude of the betterment*, which is done by the third criterion. On the other hand, the third criterion reflects this magnitude across all of the versions, without necessarily specifying *how often* one technique is better than the other, i.e., on how many faulty versions (which is done by the second criterion). Instead of conjecturing on which criterion holds more practical value, and is therefore a better mode of comparison, we opt to provide evaluations using all of the three criteria discussed above. This choice allows for a more comprehensive evaluation of the proposed RBF with respect to other fault localization techniques.

Note that it is not necessary that the suspiciousness assigned to a statement by a fault localization technique be unique. Thus, the same suspiciousness value may be assigned to multiple statements, thereby yielding two different types of effectiveness: the "best," and the "worst." The "best" effectiveness assumes that the faulty statement is the first to be examined among all the

TABLE VI
PAIRWISE COMPARISON PART (a). RBF VERSUS OCHIAI. PART (b). RBF VERSUS JACCARD

(a)

		RBFBest versus OBest	RBFWorst versus OWorst
Unix	More effective	44	46
	Same effectiveness	91	88
	Less effective	37	38
Grep	More effective	9	9
	Same effectiveness	5	5
	Less effective	5	5

(b)

		RBFBest versus JBest	RBFWorst versus JWorst
Unix	More effective	49	51
	Same effectiveness	92	89
	Less effective	31	32
Grep	More effective	9	9
	Same effectiveness	5	5
	Less effective	5	5

statements of the same suspiciousness. Supposing there are ten statements of the same suspiciousness of which one is faulty, the "best" effectiveness is achieved if the faulty statement is the first to be examined of these ten statements. Similarly, the "worst" effectiveness occurs if the faulty statement is the last to be examined of these ten statements. Hereafter, we refer to the effectiveness of the RBF technique under the best, and the worst cases as RBFBest, and RBFWorst. Similarly, we have CBest, and CWorst as the best, and the worst effectiveness of the Crosstab technique, respectively. Data corresponding to both the best, and worst effectiveness, according to each of the evaluation criteria discussed above, is provided for each of the fault localization techniques evaluated in this paper.

D. Results

Fig 4 gives the effectiveness of the RBF and Crosstab techniques for all the programs used in our studies. Zoom-ins are also included to provide an enhanced view. The curves labeled RBFBest, and RBFWorst are for the best, and the worst effectiveness of the RBF technique; and those labeled CBest, and CWorst are for the best, and the worst effectiveness of the Crosstab technique.

For a given x value (percentage of executable statements examined), its corresponding y value is the percentage of the faulty versions whose EXAM score is less than or equal to x. For example, in the case of the Unix suite (Fig 4(a)) we see that RBF can locate 125 (72.67%) of the 172 faults in the Unix suite in the best case, and 75 (43.60%) in the worst, when up to 10% of the code is examined. In contrast, Crosstab can only locate 115 (66.86%) of the 172 faults in the best case, and 68 (39.53%) in the worst, when up to 10% of the code is examined, in the Unix suite. In fact, in the Ant program (Fig. 4(e)), RBF is able to locate all 23 faults by requiring the examination of less than 0.25% of the code in the best case, and less than 0.5% of the code in the worst. Crosstab on the other hand requires the examination of 0.71% of the code in the best case, and 0.74% of the code in the worst, to detect all the faults on the Ant program. From Fig. 4, it is therefore clear that RBFBest is more effective than CBest, and RBFWorst is more effective than CWorst. Another significant point is that, in many cases, even RBFWorst is more effective than CBest (as is seen in the case of the Ant, and Grep programs, for example).

Data are also provided in Table IV to show the pairwise comparison between the effectiveness of the RBF technique and Crosstab (as per the second criterion discussed in Section IV-C) to decide for how many faulty versions one technique is better than, the same as, and worse than another. As an example, for the Make program, RBFBest is more effective (i.e., examining fewer statements before the first faulty statement is identified) than CBest for 18 of the 31 faulty versions, as effective (i.e., examining the same number of statements) for 6 faulty versions, and less effective (i.e., examining more statements) for 7 faulty versions. The table also lists the comparisons between RBFWorst and CWorst, and shows that the former is more effective than or as effective as the latter for the majority of the faults. RBFWorst is also often as effective as, or more effective than, CBest.

Table V presents the effectiveness comparison in terms of the total number of statements that need to be examined to locate all the bugs (as per the third criterion discussed in Section IV-C). For each program, RBFBest requires the examination of fewer statements than CBest. The same applies to the comparison between RBFWorst and CWorst. For example, the ratio of the number of statements examined by RBFBest to the number of statements examined by CBest for all 23 faulty versions of Ant is 21.65% (i.e., 233/1076), and the ratio between RBFWorst and CWorst is 47.97%. This result also implies that RBFBest examines 78.35% fewer statements than CBest, and RBFWorst examines 52.03% fewer statements than CWorst on the Ant program. Moreover, for two of the programs (Grep and Ant) even RBFWorst examines fewer statements than CBest. Note that there may not be any subset to superset relationship between the statements examined by RBF and Crosstab because their rankings can be very different. Thus, when we say RBF requires the examination of only a fraction (percentage) of the statements that Crosstab requires, this fraction is based purely on the number of statements, and not on the sets of statements, examined. From the data collected on the Unix suite, Space, Grep, Make, and Ant programs, we observe that not only is RBFBest more effective than CBest, and RBFWorst more effective than CWorst, but also RBFWorst is more effective than CBest in many cases. This result clearly indicates that the RBF technique is more effective in fault localization because less code needs to be examined to locate faults by using RBF than Crosstab.

V. RBF VERSUS OTHER FAULT LOCALIZATION TECHNIQUES

Recently, other fault localization techniques have been proposed, and reported good results. It is important to compare RBF to the other techniques, but it is impossible to do so against them all. Hence, we only present details of our evaluation on a representative set of techniques. Also, due to space limitations, we

TABLE VII
TOTAL NUMBER OF STATEMENTS EXAMINED BY RBF, OCHIAI, AND JACCARD PART (a). BEST CASE EFFECTIVENESS. PART (b). WORST CASE EFFECTIVENESS

(a)

	RBFBest	OBest	JBest
Unix	1302	1876	2361
Grep	2075	3092	3458

(b)

	RBFBest	OBest	JBest
Unix	4758	5247	5777
Grep	3964	4825	5191

cannot show the data on all subject programs. Instead, we select one set of small-sized programs (the Unix suite), and one large-sized program (Grep) for the purposes of the discussion.

A. RBF Versus Similarity Coefficient-Based Techniques

In [1], Abreu *et al.* propose and evaluate the use of the Ochiai similarity coefficient when applied to the context of software fault localization. Based on their case studies, the authors conclude that the Ochiai coefficient performs better than not just Tarantula, but also over several other similarity coefficients in terms of its fault localization effectiveness. We therefore also present a comparison between RBF and some of the techniques discussed in [1]. More specifically, we compare RBF with the Ochiai coefficient-based and Jaccard coefficient-based fault localization techniques (hereafter, referred to simply as Ochiai, and Jaccard, respectively). Prior to that, we briefly describe how the Ochiai and Jaccard techniques work. The Ochiai technique assigns a suspiciousness value $(susp(s))$ to each statement s as

$$susp(s) = \frac{failed(s)}{\sqrt{(totalfailed) \times (failed(s) + passed(s))}}. \tag{8}$$

The Jaccard technique assigns the suspiciousness as

$$susp(s) = \frac{failed(s)}{(totalfailed) + passed(s)} \tag{9}$$

where $failed(s)$, and $passed\ (s)$ are the number of failed, and successful test cases that execute statement s, respectively; and the quantities *totalfailed,* and *totalpassed* correspond to the total number of failed, and successful test cases, respectively. Once the suspiciousness values for each statement have been computed, the statements are sorted in descending order of their suspiciousness to produce rankings in the same way as RBF, and Crosstab. Also, based on the discussion towards the end of Section IV-C, similar to RBF and Crosstab, the Ochiai and Jaccard techniques may also result in two levels of effectiveness, the best, and the worst, because the rankings produced may not necessarily be free of ties, i.e., two (or more) statements may have the same Ochiai or Jaccard computed suspiciousness value. As with RBFBest, RBFWorst, CBest, and CWorst defined before, the best cases of Ochiai, and Jaccard are annotated as OBest, and JBest, respectively; and the worst cases as OWorst, and JWorst, respectively. In the interests of clarity and readability, we separate the best case effectiveness from the worst case effectiveness, i.e., Fig. 5(a) corresponds to the best case of each technique on the Unix suite, and Fig. 5(b) corresponds to the worst. Similarly, Fig. 6(a) corresponds to the best case evaluation on the Grep program, and Fig. 6(b) corresponds to the worst.

From Figs. 6 and 7, we observe that RBF generally does better than both Ochiai and Jaccard irrespective of whether the best or the worst case is considered, and this result is especially true for the Grep program (Fig. 6). It is only in the case of the zoom-in with regard to the worst case effectiveness on the Grep program (Fig. 6(b)) that RBF is slightly worse than Jaccard and Ochiai initially, but it quickly catches up, and in fact beyond the 5% mark (i.e., when more than 5% of code is examined) RBFWorst is always better than OWorst and JWorst. When the best case effectiveness on the Grep program is considered (Fig. 6(a)), RBF is always better than (or at least equal to) both Ochiai and Jaccard.

Table VI presents the pairwise comparison between RBF and Ochiai, as well as RBF and Jaccard, in terms of the number of faulty versions where one technique might perform better, worse, or equal to the other. From the table, we observe that RBF performs better than both Ochiai and Jaccard (regardless of best or worst case) in the sense that it performs better on more faulty versions than the other techniques. Data corresponding to the third criterion in Section IV-C (namely, the total number of statements that need to be examined to locate faults in all of the faulty versions) for the RBF, Ochiai, and Jaccard techniques (both best, and worst cases) are presented in Table VII. We conclude that RBF performs better than Ochiai and Jaccard (irrespective of whether the best or worst case is considered) as fewer statements need to be examined to locate all of the faults.

B. RBF Versus a Heuristic-Based Technique

In [42], Wong *et al.* propose that, in the context of fault localization, if a statement has been executed by many successful test cases, then the contribution of each additional successful execution to the suspiciousness of the statement is likely to be less than that of the first few successful tests. Similarly, if a statement has already been executed by many failed test cases, the contribution of each additional failed execution to its suspiciousness is likely to be less than the contribution of the first few failed tests. They also propose that, if a statement has been executed by at least one failed test case, then the total contribution from all the successful tests that execute the statement should be less than the total contribution from all the failed tests that execute it. To this effect, several heuristics are proposed in [42], of which the one that yields the best performance is referred to as HeuristicIII (different from the heuristic with the same name in [46]). Note that the general form of HeuristicIII involves the use of certain parameters, and we select the same values of the parameters as in [42] that result in the best performance (therefore, HeuristicIII(c) of [42] instead of just HeuristicIII). For further details on HeuristicIII, we direct the readers to [42]. Because it is reported that HeuristicIII(c) performs better than Tarantula, we also compare RBF to HeuristicIII(c). Once again, due to the space limitation, we present results on the 10 programs in the Unix suite, and the Grep program. In addition, for brevity, we

TABLE VIII
PAIRWISE COMPARISON BETWEEN RBF, AND H3C

		RBFBest versus H3CBest	RBFWorst versus H3CWorst
Unix	More effective	43	44
	Same effectiveness	82	80
	Less effective	47	48
Grep	More effective	9	9
	Same effectiveness	5	5
	Less effective	5	5

refer to HeuristicIII(c) as H3C in this paper. Keeping with previous conventions, the best case is referred to as H3CBest, and the worst as H3CWorst.

Fig. 7 presents the effectiveness comparison between RBF and H3C based on the $\mathcal{EXAM}$ score (i.e., the first criterion in Section IV-C) across the Unix suite, and Grep. Based on the figures, we find that in general RBF performs better than H3C. However, in such cases where curves sometimes intersect one another (consider RBFWorst and H3CWorst on the Unix suite, i.e., Fig. 7(a)) it is hard to determine which technique performs better using such a comparison alone. Thus, additional criteria (namely, the second, and the third criteria) in Section IV-C should also be used.

Table VI presents a comparison between RBF and H3C in terms of the number of faulty versions where one technique performs better, worse, or equal to another. We observe that RBF performs slightly worse than H3C (irrespective of best or worst case) in the case of the Unix suite. Yet at the same time RBF performs much better than H3C (also irrespective of best or worst case) in the case of the Grep program. However, it is important to appreciate that the degree of superiority of H3C over RBF on the Unix suite is not as significant as that of RBF over H3C on the Grep program. To better understand this result, we note that H3CBest on the Unix suite performs better than RBFBest by four more faulty versions than when RBFBest performs better than H3CBest (47 versus 43). However, as per Table II, there are a total of 172 different faulty versions of the Unix suite used in this study. In the case of the Grep program, RBFBest performs better than H3CBest on four more faulty versions than when H3CBest performs better than RBFBest (9 versus 5), but the Grep program has just 19 faulty versions as per Table VIII. The difference in the case of the Unix suite $(4/172 = 2.33\%)$ does not nearly seem as significant as the difference in the case of the Grep program $(4/19 = 21.05\%)$. Even so, this criterion still does not allow us to make any decisive conclusions regarding the comparison of RBF and H3C. Data corresponding to the third criterion is presented in Table IX, which clearly shows that RBF performs better than H3C (in that fewer statements need to be examined by using RBF as opposed to if H3 were used). This result is true regardless of whether the best or worst case effectiveness is considered, and more importantly it is true across both the Unix suite and Grep. We thus conclude that RBF is more effective than H3C (by taking the various criteria into account) based on our data.

Experiments reported in Section V evidence the superior performance of the RBF technique, and results indicate that it performs better than similarity coefficient-based techniques such as Ochiai [1] and Jaccard [1], and heuristic-based techniques such as H3C (i.e., HeuristicIII(c) in [42]). Combined with the experiments reported in Section IV, and the results from our previous study [49], it has now been shown that RBF is more effective than Tarantula [18], SOBER [22], Liblit05 [20], Crosstab [50], Ochiai [1], Jaccard [1], and H3C (HeuristicIII(c)) [42].

TABLE IX
TOTAL NUMBER OF STATEMENTS EXAMINED BY RBF, AND H3C

	Best Case Effectiveness		Worst Case Effectiveness	
	RBFBest	H3CBest	RBFWorst	H3CWorst
Unix	1302	1655	4758	5026
Grep	2075	2702	3964	4435

VI. PROGRAMS WITH MULTIPLE BUGS

Thus far, the discussion and evaluation of the RBF technique has been with respect to programs that categorically are single-bug in nature, i.e., each has exactly one bug. In this section, we discuss and demonstrate how the RBF technique may very easily be applied to programs with multiple bugs in them as well.

In the case of programs with multiple bugs, two source-independent execution failures may not necessarily occur due to the same causative fault. However, if failed test cases could be segregated into groups such that the tests in one group all fail due to the same fault, then each group of failed test cases could be used to locate that particular fault, thereby reducing the multi-bug problem to a set of single bug problems. This reduction is done in two steps. The first step is to conduct an appropriate clustering on failed executions or failed tests. Different techniques have been proposed for this purpose [17], [23], [32], [56]. For example, we can group failed test cases into *fault-focused* clusters such that those in the same cluster are related to the same fault as per [17]. The second step is to combine failed tests in each cluster with the successful tests for debugging a single fault. We follow a similar approach as described in [17] to generate a specialized test suite for each fault. However, instead of using Tarantula's ranking (as was done in [17]), we follow the procedure discussed in Section III to rank the statements. Consequently, the RBF technique can easily be extended to handle programs that may contain multiple bugs.

We conducted a case study using gcc, the GNU Compiler Collection, to illustrate the concept discussed above. There are multiple releases of gcc posted at the GNU website (http://gcc.gnu.org). We used gcc 3.4.1, and the 29 C files in its sub-directory gcc/cp were instrumented using a revised version of χSuds [51]. These files contain 95218 lines. We do not have a separate version of gcc, each of which contains exactly one bug. Instead, we

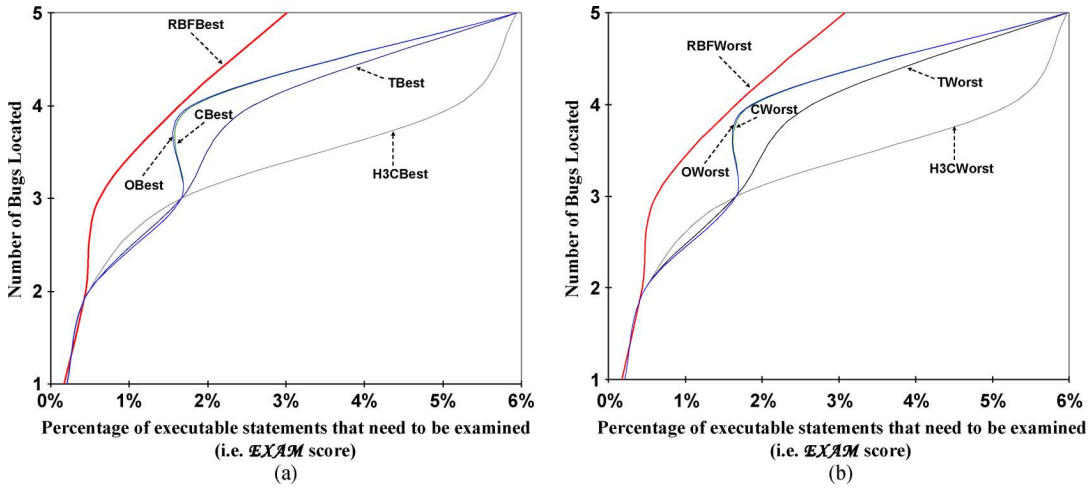

Fig. 8. Effectiveness comparison between RBF, Crosstab, Tarantula, Ochiai, and H3C for the 5-bug version of gcc. (a) Best case effectiveness comparison. (b) Worst case effectiveness comparison.

have only one gcc program with multiple bugs. There are 9489 successful test cases for this version.

Defect data retrieved from the gcc Bugzilla database are used to determine the exact location of each bug. Information on additional test case(s) created for each bug is also available from bug reports to help us determine the failed tests for each bug. We use such information to accomplish the *fault-focusing* clustering in step 1 to group failed test cases into clusters that target different bugs. A more robust clustering technique will be developed in our future study; however, that discussion is beyond the scope of this paper. Each fault-focused cluster is combined with the successful tests for locating a single bug. Five bugs are used in our study: $\mathcal{B}_1$ (ID 16637), $\mathcal{B}_2$ (ID 16889), $\mathcal{B}_3$ (ID 16929), $\mathcal{B}_4$ (ID 16965), and $\mathcal{B}_5$ (ID 18140). Each of the bugs $\mathcal{B}_1$, $\mathcal{B}_2$, $\mathcal{B}_3$, and $\mathcal{B}_4$ has one failed test, and 9489 successful tests; $\mathcal{B}_5$ has two failed tests, and 9489 successful tests. The execution environment for gcc is the same as that for the Space program (see Section IV-A).

Fig. 8 presents the effectiveness comparison between RBF and several techniques on the 5-bug version of gcc based on the description provided above. We split the figure into the best case comparison (Fig. 8(a)) and the worst case comparison (Fig. 8(b)) to enhance readability. Based on the curves in the figure, we observe that RBF performs much better than all of the other techniques, and is able to locate all of the bugs by examining much less code, regardless of whether the best case or worst case effectiveness is considered.

Table X lists the total number of statements examined by RBF and the other fault localization techniques for the five bugs of gcc. It is clear from the table that RBF is the most effective technique in that fewer statements need to be examined if RBF is used (as opposed to the other techniques), irrespective of the best case or worst case effectiveness.

To summarize, Sections IV and V show that RBF is more effective than other fault localization techniques evaluated herein, with respect to single bug programs, and the experiments in Section VI reveal that this is also true for programs with multiple bugs.

TABLE X
TOTAL NUMBER OF STATEMENTS EXAMINED FOR THE 5-BUG VERSION OF GCC

	Best Case	Worst Case
RBF	1789	1826
Crosstab	3082	3119
Tarantula	3315	3352
Ochiai	3075	3112
H3C	4104	4141

VII. EVALUATING THE ROBUSTNESS OF THE PROPOSED RBF FAULT LOCALIZATION TECHNIQUE

Thus far, the focus of this paper has been to demonstrate the fault localization effectiveness of the proposed RBF technique with respect to other fault localization techniques, i.e., to show that the RBF provides better results than other state of the art techniques (which is indicated by our case studies). However, a fault localization technique, along with being highly effective, must also be robust and able to perform well even under harsh circumstances. Therefore, in this section, we aim to investigate the robustness of RBF, and evaluate how its fault localization effectiveness changes as the conditions under which it is used deteriorate. In doing so, we evaluate RBF with respect to itself, i.e., observe how the RBF under a harsh circumstance compares to the RBF under golden circumstances. The harsh circumstances are simulated by deliberately modifying the inputs that are fed to the RBF, and the golden or baseline scenario corresponds to the use of the RBF against the unmodified input (all the data presented so far in this paper corresponds to this scenario).

A. Sensitivity to Test Set Size

Comprehensive test sets may not always be available, and therefore it is important to understand how sensitive a fault localization technique is to the size of the test set, i.e., the number of test cases available. Intuitively, the more distinct test cases we have, the more information (coverage and execution result) we can provide a fault localization technique with, and consequently, we expect the fault localization results to be better. We

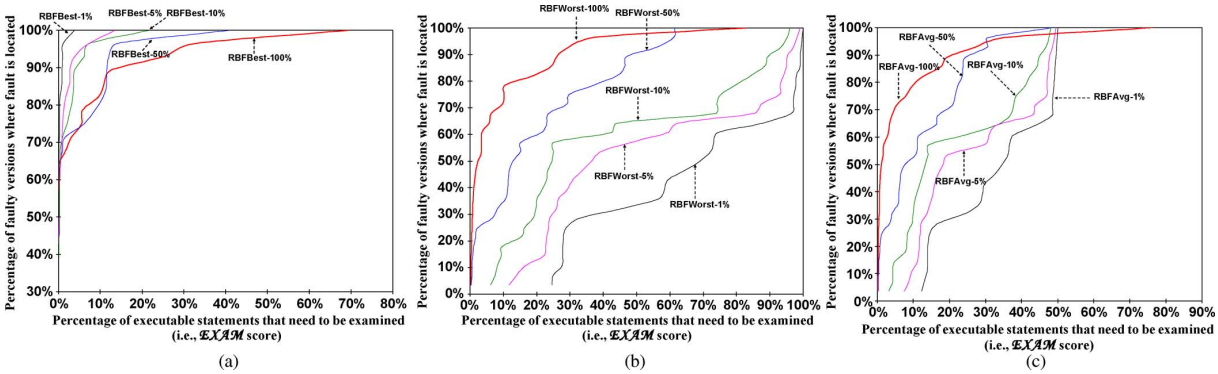

Fig. 9. Effectiveness evaluation of RBF across various sampling rates on the Gzip program. Part (a). Best case effectiveness comparison. Part (b). Worst case effectiveness comparison. Part (c). Average case effectiveness comparison.

investigate this intuition by using partial (with respect to the entire test set available) test sets, and observing the corresponding fault localization effectiveness, relative to that when the entire test available is used. Given a sampling rate ε $(0 < \varepsilon \leqslant 1)$, and an original test suite size of N, we randomly select $N \times \varepsilon$ test cases without replacement to produce a test set T_ε (we refer to such a test set as a ε-sampled test set). If a T_ε does not contain at least one failed test case, it is discarded, and the process is repeated until a T_ε containing at least one failed test case is produced. Then, the RBF technique is evaluated based on only the information in T_ε.

We note that one ε-sampled test suite may differ considerably from another, and therefore the fault localization results obtained using a test suite of a certain size may vary drastically from the results obtained from another test suite of the same size. To reduce such bias, evaluations across any sampling rate are conducted 100 times (i.e., using 100 distinct T_ε for a single ε), and the results averaged. We choose sampling rates of 90% to 10% in increments of 10%, and then 5%, and 1%. Note that $T_{100\%}$ (a sampling rate of 100%) corresponds to using the original test suite in its entirety.

Because we are investigating the link between fault localization effectiveness and test set size, it is important that the subject program not have too many test cases to begin with; the reason being that the effect of reducing the number of test cases is more appreciable in cases where the test set is small. For example, even if we restrict ourselves to sampling only 10% of the test set, i.e., a 0.1-sampled test set, on the Space program, we would still have about 1359 test cases as there are a total of 13585 test cases to work with. A good alternative would then be the Unix suite as these programs have relatively small-sized test sets (as seen from Table II). However, these programs are themselves small. With this result in mind, we downloaded version 1.1.2 of the Gzip program (along with some faulty versions) from [15], which consists of 6573 lines of code. Additional faulty versions from other studies [42] were also used, bringing the total number of faulty versions to 28. The Gzip data set consisted of 211 usable test cases, and thus has a good balance between test set size and program size, making it ideal for these experiments.

Fig. 9 presents the effectiveness comparison (the best and worst have been separated for clarity) of RBF on the Gzip program across various sampling rates. The figures only present the curves corresponding to sampling rates of 50%, 10%, 5%, and 1% (even though the experiments were performed using all the sampling rates discussed above, all of which show the same trends as the curves shown in the figure). Also, for ease of comparison, the curves corresponding to RBF on the original (un-sampled, or equivalently, 100% sampled) test set have also been provided in the figures. The curves are annotated: RBFBest-ε% corresponds to the best case effectiveness of RBF using a sampling rate of ε, and the same convention is true of the worst case (RBFWorst).

From Fig. 9, we observe that the curves corresponding to lower sampling rates seem to fan away from those corresponding to higher sampling rates. Indeed, in the case of RBFWorst (Fig. 9(b)) we observe that as the sampling rate is reduced (fewer test cases are used) the effectiveness grows worse, which is consistent with our initial intuition. However, the case of RBFBest looks quite contradictory to this intuition at first glance, as we see that a reduction in sampling rate brings about an improvement in best case effectiveness. How is this possible?

To understand this result, let us recall the definition of the best and worst case effectiveness: in the event that a faulty statement is tied with several other statements for the same suspiciousness, then in the best case we assume we examine the faulty statement before the other statements that are tied, and in the worst case, only after. This event happens when fault localization techniques cannot distinguish between faulty statements and certain non-faulty statements, and assign them the same suspiciousness. Consider that the effect of reducing the sampling rate is to reduce the number of test cases that fault localization techniques have to work with. We conjecture that this reduction adversely affects the ability of fault localization techniques to distinguish one statement from another. The smaller the sampling rate, the harder it is for a technique to tell the difference between one statement and another, and thus there may be more ties among statements, of which the faulty statement may be one. To see how this result can lead to an improvement in best case effectiveness, and deterioration at worst, consider the trivial case where we have a program with n statements, and every statement is tied with the same suspiciousness. In the best case, we shall need to examine just one statement to find the fault; and in the worst case, all n statements. Thus, we have a very high best case effec-

TABLE XI
TOTAL NUMBER OF STATEMENTS EXAMINED FOR DIFFERENT SAMPLING RATES

Sampling rate (ε)	Total number of statements examined to locate faults across all faulty versions		
	RBFBest	RBFWorst	RBFAvg
10 %	2966	4743	3854.5
90%	2806.22	5417.95	4112.085
80%	2750.45	6456.79	4603.62
70	2493.55	7448.14	4970.845
60%	2378.77	8879.38	5629.075
50%	2015.43	9873.33	5944.38
40%	1875.87	11281.12	6578.495
30%	1603.89	13413.9	7508.895
20%	1274.84	16185.07	8729.955
10%	921.98	20201.51	10561.745
5%	624.04	24829.77	12726.905
1%	169.72	31195.17	15682.445

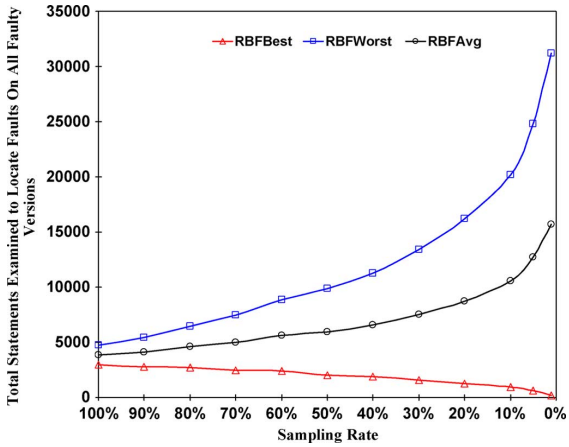

Fig. 10. Total number of statements that must be examined as a function of sampling rate (Graphical Perspective).

tiveness, and a very low worst case effectiveness. It is therefore important to point out that it may be a fallacy to assume that a reduction in sampling rate leads to better fault localization simply based on the curves for RBFBest. Rather, the curves for RBFBest (Fig. 9(a)), and RBFWorst (Fig. 9(b)) must be analyzed collectively, as for any sampling rate an improvement in RBFBest seems to be accompanied by a deterioration of RBFWorst.

For this reason, we also present the effectiveness in terms of an average (which is simply the arithmetic mean of the best and the worst effectiveness) in part (c) of Fig. 9. The notation RBFAvg- $\varepsilon\%$ denotes the average effectiveness of the RBF when a sampling rate of ε is used. The curves clearly demonstrate that the average effectiveness decreases as the sampling rate decreases. An interesting observation is that, while RBFAvg-100% is better than RBFAvg for any of the smaller sampling rates, RBFAvg-100% only converges (the y-value reaches 100%) after all the other curves. This result is because, in the case of RBFAvg-100%, 27 out of the 28 faults can be located by examining less than 35% of the code, yet for the last fault, 76.20% of the code needs to be examined before the faulty statement is located. However, this phenomenon is quite certainly dependent on the test cases and subject programs.

In terms of the total number of statements that need to be examined to locate all of the faults (namely, the third criterion in Section IV-C) based on changing sampling rates, we present this data in Table XI for RBFBest, RBFWorst, and RBFAvg. Fig. 10 provides a graphical perspective of the same. Once again, the sampling rate of 100% corresponds to using the entire available test set (i.e., no sampling involved). There may be fractional values in Table XI, whereas the number of statements that are examined should be values of integer form. The fractional component arises because, as discussed earlier, for each sampling rate ε, we generated 100 distinct ε-sampled test sets, and averaged the results.

As with Fig. 9, we observe that, in terms of the total number of statements that must be examined to find faults in all faulty versions, RBFBest seems to grow more effective, and RBFWorst less effective, as the sampling rate is decreased. Once again, based on the trend observed in the case of RBFBest alone, it would be a fallacy to conclude that the overall effectiveness of the approach increases as the sampling rate is reduced. We also observe that RBFAvg decreases as the sampling rate is reduced (though not to the extent that RBFWorst decreases). As far as the relative magnitude of change in RBFAvg with respect to the change in sampling rate is concerned (which is a good indicator of sensitivity), we can utilize the total number of statements that must be examined as per Table XI. We note that the relative change in RBFAvg is not nearly as high as the change in sampling rate. Consider for example that, with a sampling rate of just 1%, the number of statements that we must examine only goes up by a relative factor of 4.07 (15682.445/3854.5=4.0686), which is very small when we consider that we used only 1/100th of the test case information.

B. RBF in the Presence of Noisy Data

A test set, when executed against the same program but in two different environments, may result in two different sets of failed test cases. For a fault localization technique relying on the coverage and test case execution results as its input, its effectiveness may therefore also vary depending on which environment it is employed in (or rather depending on which environment its input data is collected in). In fact, a good technique should not only be effective when considering one environment, but also in multiple environments such that, even if the information varies from environment to environment to a small degree, it is still able to localize faults effectively. To investigate if the proposed RBF fault localization technique is still able to perform effectively, even with changes in environments as discussed, we decide to simulate such circumstances by introducing the notion of *noise*.

We introduce noise into the collected data by randomly perturbing the result of a test case such that, if a selected test case was previously recorded as failed, it is now recorded as successful, and vice versa. This perturbation directly affects fault localization techniques such as RBF, and Crosstab, among many others, because it changes one or more variables that are used as input to each technique. To evaluate how much RBF is affected, we use the Gzip program because it only has 211 test

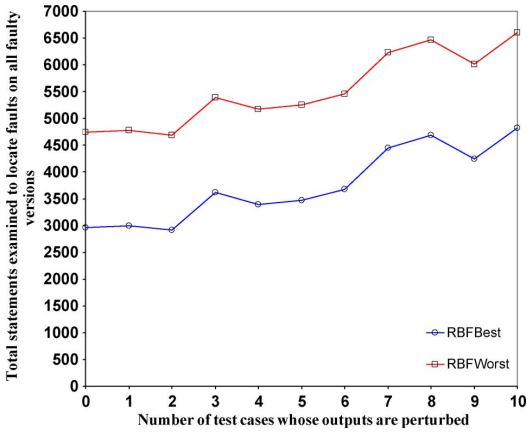

Fig. 11. The number of statements that require examination versus the level of noise for RBF on the Gzip program.

cases. Switching the result of one test case implies that we alter a larger fraction of the input information as opposed to a program with many more test cases (such as the Space program which has 13585 test cases). We want to evaluate the performance of RBF under harsh circumstances, and therefore we decided to use the Gzip program for this portion of the experiment.

Note that this experiment is considerably different from the one in Section VII-A in that the previous experiment involved investigating how the effectiveness of the RBF changed with respect to a reduction of the input. Test cases were randomly sampled (ensuring the presence of at least one failed test each time), and the effectiveness of the RBF technique was re-evaluated using the sampled test cases. In contrast, this experiment preserves the number of inputs (test cases) to the RBF, but perturbs data contained in the inputs. Recall that the RBF technique involves training a neural network on the input data, and the net effect of perturbing the data in the manner discussed is to purposefully mislead the RBF technique (by training on the modified data) to evaluate how it performs in such situations. We perturb the outputs of up to 10 test cases for this experiment. The reason the maximum number of test cases whose results are switched is set at 10 (about $5\% = 10/211$ of all the test cases) is because it is very unlikely that the switching of environments will result in a larger difference in terms of the set of failed test cases. Therefore, to differentiate between noisy data and unreliable data, we set the limit at 10 test cases.

Fig. 11 presents the results of RBF against noisy data, once again making use of the third criterion in Section IV-C (namely, the total number of statements examined to locate faults on all faulty versions) as a means of comparison. The x-axis in the figure represents the amount of noise (in terms of the number of test cases that have their execution results switched) that is seeded into the input data. The y-axis represents the number of statements that must be examined to locate all the bugs. The first data point $(x - \text{value} = 0)$ means no noise has been seeded yet, and has been provided for reference purposes to observe the change in effectiveness as the noise in the input data is increased.

The first thing we observe based on Fig. 11 is that the effectiveness of both RBFBest and RBFWorst seems to decline as the amount of noise that is seeded (number of test cases that are perturbed) is increased, as more statements need to be examined to locate all the faults. Second, we note that this trend is not strictly monotonic. For example, the effectiveness of the RBF (both best and worst) is slightly better when two tests have been perturbed than when none or one test is perturbed. Similarly, the effectiveness of the RBF (both best and worst) when three test cases are perturbed is worse than when four or five test cases are perturbed. Third, the curves for RBFBest and RBFWorst follow nearly the same pattern, meaning that the seeding of noise does not have any special impact on the number of tied statements with respect to the rankings produced by the RBF. With respect to the degree, and rate of change in effectiveness as a result of seeding noise, we find that appreciable differences in effectiveness do not arise until at least three or more test cases have had their results changed. However, the effect of changing the results for three test cases (3614 statements in the best case, and 5391 in the worst) is about the same (i.e., the difference is quite small) as that of changing the results for six test cases (3678 in the best case, and 5455 in the worst). After seven or more test cases have had their results changed, the differences in effectiveness start to become significantly apparent. Yet when as many as seven test cases have had their results swapped, and the fault localization effectiveness starts to decline, we must question whether the fault localization technique is to blame, or if the reliability of the input data itself is suspect.

In summary, the experiments in this section reveal that, along with being highly effective at fault localization, the RBF technique is also robust in the sense that it performs well, even with inadequate test sets, or noisy data.

VIII. Discussion

This section discusses important aspects related to the proposed RBF technique, and addresses threats to validity.

A. RBF, and Other Machine Learning Algorithms

The RBF approach is similar to a support vector machine (SVM) methodology in that an input vector is mapped into a feature space (in our case the features are the neurons at the hidden layer of the modified RBF), and then a linear model is used to compute a weighted sum of features [13]. An SVM is generally used for classification. Also, SVMs typically use Vapnik's epsilon-insensitive loss function for parameter estimation. However, we use a negative log-likelihood loss function with respect to a conditional Gaussian regression model for parameter estimation. Furthermore, our focus is not on the *classification* of inputs, but on the value of an SVM linear discriminant function, $\hat{r}_{v_i}$, which is the measure of suspiciousness of the ith statement.

Decision tree methods partition the input vector space into a set of hyper-rectangles, and a linear model is used to compute a weighted sum of features. An important aspect of decision tree methods is that algorithms have been developed for splitting rectangles and creating decision trees with different types of methodologies. Our modified RBF may be interpreted as partitioning the input space into a set of hyper-ellipsoids, and again a linear model is used to compute a weighted sum of features [13]. There is no intrinsic advantage in using a decision space of hyper-rectangular features versus one of hyper-ellipsoidal features. The advantage of one decision space over another will ultimately be determined by characteristics of the sta-

tistical data. In general, there is no single machine learning algorithm which provides the optimal generalization performance for all data sets. Our motivation for using neural networks in the context of fault localization was primarily due to the many benefits offered by neural networks; and we specifically focused on RBF networks as they have several advantages over BP networks. However, we intend to investigate the use of other machine learning techniques in the context of fault localization in the future.

B. Logistic Versus Linear Regression

Along the same lines as the machine learning algorithms discussed in Section VIII-A, generalized logistic regression methods also involve having an input vector mapped into a feature space, and then a weighted sum of features is computed. In the case of such methods, the weighted sum of features is mapped via a logistic sigmoid transformation into the probability of a particular classification [13]. We experimented with the use of a logistic function to perform the mapping, but it did not have any beneficial impact on the fault localization results whatsoever. Moreover, such experimentation is beyond the scope of this paper. Thus, instead of distracting readers with more complicated logistic regression modeling, we decided to employ a linear approach.

C. Sequential Estimation of Hidden Neurons, Receptive Field Centers, and Weights

As per the RBF technique which has been proposed in this paper, the number of hidden neurons, receptive field centers, and weights are not simultaneously estimated. A potential problem with sequential estimation is that the performance of the network is suboptimal. However, we make no claims of the optimality of the proposed technique (or its individual aspects). We recognize that what was presented herein is an instance of a more general RBF neural network-based fault localization model, and other variations of our approach certainly exist. However, we did not want to distract readers from the context in which the RBF is being applied, namely software fault localization. Besides, the method implemented here can be interpreted as a generalized linear regression model with highly nonlinear preprocessing transformations, which is a reasonable baseline approach.

D. Threats to Validity

We now discuss some possible threats to the validity of our experiments and results.

Our use of the $\mathcal{EXAM}$ score, and the other criteria described in Section IV-C (such as the total number of statements examined, and the number of faulty versions where one technique performs better than, the same as, or worse than another) represents a threat to construct validity. While such criteria are suitable measures of fault localization effectiveness, by themselves they do not provide a complete picture of the effort spent, as developers rarely examine statements one at a time, and may not spend the same amount of time examining different statements. We also assume that, if a developer examines a faulty statement, then they will identify the corresponding fault(s). By the same token, a developer shall not identify a non-faulty statement as faulty. If such perfect bug detection does not hold in practice, then the effort spent in examination may increase. However, such concerns apply to all fault localization techniques, and not just the RBF technique proposed here.

As far as internal validity goes, as discussed in Section IV-A, not every downloaded faulty version of a program was usable for our experiments, and some of the faulty versions had to be created using mutation-based fault injection. However, such injection has been shown to be an effective approach to simulating realistic faults that can be used in software testing research to yield trustworthy results [4], [9], [26]. Other researchers, such as those of [22], also argue that, although these faults are manually injected, they do resemble common logic errors. A good example is that errors like "off-by-one" may sneak in when developers are handling obscure corner conditions. Therefore, while our seeding of faults may represent a threat to internal validity, it was, nevertheless, a necessary step to make our evaluations more comprehensive, and does not detract from the validity and correctness of our results.

Regarding external validity, because our evaluation of the proposed RBF technique has primarily been based on empirical data, arguably our results may not be generalized to all programs. To alleviate this threat, we used many different sets of faulty programs in our evaluation. Furthermore, each program varies greatly from the other in terms of size, function, number of faulty versions, number of test cases, etc., and this variability helps us have confidence in the general applicability of the RBF to different programs.

Finally, we would also like to emphasize that, similar to any other fault localization techniques, the effectiveness of RBF varies for different programs, bugs, and test cases. The abilities of the coverage measurement tools, such as whether the runtime trace can be correctly collected even if a program execution is crashed due to a segmentation fault, and environments, including compilers, operation systems, hardware platforms, etc., also have an impact.

IX. Related Studies

In the recent years, many studies have been performed, and several techniques proposed, in the area of fault localization.

Renieris and Reiss [33] propose a nearest neighbor debugging technique that contrasts a failed test with another successful test which is most similar to the failed one in terms of the "distance" between them. If a bug is in the difference set between the failed execution and its most similar successful execution, it is located. For a bug that is not contained in the difference set, the technique continues by first constructing a program dependence graph, and then including adjacent un-checked nodes in the graph step by step, and checking to see if they are faulty, until the bug is located. The set union and set intersection techniques are also reported in [33]. The former computes the set difference between the "program spectra" of a failed test and the union spectra of a set of successful tests. It focuses on the source code that is executed by the failed test, but not by any of the successful tests. The latter is based on the set difference between the intersection spectra of successful tests and the spectra of the failed test. It focuses on statements that are executed by all successful tests, but not by the failed test case.

In [7], Cleve and Zeller report a program state-based debugging technique, cause transition, to identify the locations and times where a cause of failure changes from one variable to another. This technique is an extension of their earlier work with delta debugging [52], [53]. Tarantula [18] follows the intuition that statements primarily executed by failed test cases are more likely to be faulty than those executed by successful test cases, and assigns suspiciousness to each statement. Tarantula performs better than techniques such as set union, set intersection, and cause transitions; and previously it has been shown that RBF performs better than Tarantula [49], which means RBF also performs better than the techniques less effective than Tarantula such as set union and intersection, cause transitions, etc.

Liblit *et al.* propose a statistical debugging technique (which we refer to as Liblit05 in this paper) that can isolate bugs in the programs with instrumented predicates at particular points [20]. As an extension (and improvement) to Liblit05, Liu *et al.* propose the SOBER technique to rank suspicious predicates [22]. All instrumented predicates can be ranked in order of their scores, and examined in order of their fault-relevance. RBF has been shown to be more effective than the SOBER and Liblit05 techniques in our previous study [49].

Zhang *et al.* [54] present a technique such that, for a given failed test, they perform multiple executions against that test. In each execution, the outcome of one predicate is switched, and this process continues until the program produces the correct output as a result of the switch; then the corresponding predicate is a critical predicate. Bidirectional dynamic slices of such critical predicates are then computed to help programmers locate the bugs. There are also many slicing-based studies which can be further classified as static slicing-based [24], [40], dynamic slicing-based [2], and execution slicing-based [3], [45], [47] fault localization techniques. As opposed to directly assessing the suspiciousness of individual program entities, in [55], the authors instead focus on the propagation of infected program states.

In [34], the original Tarantula technique [18] is extended by making use of the Ochiai coefficient [1] (which is also compared to RBF in Section V-A) to evaluate the quality of fault localization with respect to multiple coverage types, namely statements, branches, and data dependencies. Their study shows that no single coverage type performs best for all studied faults, and that different kinds of faults are best localized by different coverage types. A new coverage-based approach to fault localization that leverages each coverage type via combination is also presented in [34]. For the purposes of this paper, only statement-based coverage is considered as an input to RBF, and the effects of using multiple coverage types with RBF is deferred to future study.

We direct readers interested in learning more about fault localization, and fault localization techniques, to our report, *A Survey on Software Fault Localization* [43], where a comprehensive survey on the state of the art in fault localization can be found.

X. Conclusion and Future Work

An RBF (radial basis function) neural network-based fault localization technique is presented in this paper. The network is trained on coverage information for each test case paired with its execution result, either success or failure, and the network so trained is then given as input a set of virtual test cases, each of which covers a single statement. The output of the network is considered to be the suspiciousness of the statement corresponding to the virtual test. Statements with a higher suspiciousness should be examined first as they are more likely to contain program bugs. Empirical data (in Section IV) based on the Unix suite, Space, Grep, Make, and Ant programs (i.e., both small and large-sized programs) indicates that RBF is more effective in fault localization than Crosstab [50]. Section V shows that RBF is also more effective than similarity coefficient-based techniques such as Ochiai [1], and Jaccard [1]; and Heuristic-based techniques such as H3C [42]. Combining the results presented in this paper with those from a previous study [49] which shows that RBF is more effective than techniques such as Tarantula [18], Liblit05 [20], and SOBER [22], we have shown that RBF is more effective than 7 different competing fault localization techniques: Tarantula, SOBER, Liblit05, Ochiai, Jaccard, Crosstab, and H3C.

Experiments that both illustrate and evaluate the RBF technique on programs with multiple bugs (based on the very large gcc program) are also performed. Our data suggests that the RBF technique can not only easily be applied to localize bugs in a multi-bug program, but is also more effective than techniques such as Crosstab, Tarantula, Ochiai, and H3C in such situations. Further experiments that evaluate the RBF technique in the presence of inadequate test sets and noisy data in the Gzip program show that RBF is not just effective at fault localization, but is also highly robust.

Studies that target a wider range of application domains are currently in progress to further validate the general effectiveness of the RBF technique. As stated earlier, what we have presented in this paper is an instance of a much larger model where RBF neural networks are used in the context of fault localization. In the future, we intend to investigate other instances of the same technique by varying our parameters and algorithms, and evaluating with respect to other distance measures (as opposed to the weighted-bit comparison based dissimilarity). We also plan to extend our studies using other machine learning algorithms (e.g., support vector machines, decision trees, logistic regression, etc.), and observe the potential variation of the performance, if any. Additionally, we are currently working with our industry partners to apply the RBF fault localization technique in their environments.

References

[1] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund, "A Practical Evaluation of Spectrum-based Fault Localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, 2009.

[2] H. Agrawal, R. A. DeMillo, and E. H. Spafford, "Debugging with dynamic slicing and backtracking," *Software: Practice & Experience*, vol. 23, no. 6, pp. 589–616, Jun. 1996.

[3] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault localization using execution slices and dataflow tests," in *Proceedings of the 6th International Symposium on Software Reliability Engineering*, Toulouse, France, Oct. 1995, p. 143-15.

[4] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?," in *Proceedings of the 27th International Conference on Software Engineering*, St. Louis, Missouri, USA, May 2005, pp. 402–411.

[5] G. Boetticher and D. Eichmann, "A neural network paradigm for characterizing reusable software," in *Proceedings of the 1st Australian Conference on Software Metrics*, Sydney, Australia, Nov. 1993, pp. 41–54.

[6] S. R. Chu, R. Shoureshi, and M. Tenorio, "Neural networks for system identification," *IEEE Control Systems Magazine*, vol. 10, no. 3, pp. 31–35, Apr. 1990.

[7] H. Cleve and A. Zeller, "Locating causes of program failures," in *Proceedings of the 27th International Conference on Software Engineering*, St. Louis, Missouri, USA, May 2005, pp. 342–351.

[8] J. Dang, Y. Wang, and S. Zhao, "Face recognition based on radial basis function neural networks using subtractive clustering algorithm," in *Proceedings of the 6th World Congress on Intelligent Control and Automation*, Dalian, China, Jun. 2006, pp. 10294–10297.

[9] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *IEEE Trans. Software Engineering*, vol. 32, no. 9, pp. 733–752, Sep. 2006.

[10] K. Fukushima, "A neural network for visual pattern recognition," *Computer*, vol. 21, no. 3, pp. 65–75, Mar. 1998.

[11] M. T. Hagan, H. B. Demuth, and M. Beale, *Neural Network Design*. New York: PWS Publishing, 1995.

[12] M. H. Hassoun, *Fundamentals of Artificial Neural Networks*. Cambridge, MA: MIT Press, 1995.

[13] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Berlin, Germany: Springer, 2001.

[14] S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd ed. New York: Prentice Hall, 1999.

[15] [Online]. Available: http://sir.unl.edu/portal/index.html

[16] Clover: A Code Coverage Analysis Tool for Java [Online]. Available: http://www.atlassian.com/software/clover/

[17] J. A. Jones, J. Bowring, and M. J. Harrold, "Debugging in parallel," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, London, UK, Jul. 2007, pp. 16–26.

[18] J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM Conference on Automated Software Engineering*, Long Beach, California, USA, Dec. 2005, pp. 273–282.

[19] C. C. Lee, P. C. Chung, J. R. Tsai, and C. I. Chang, "Robust radial basis function neural networks," in *IEEE Trans. Systems, Man, and Cybernetics: Part B Cybernetics*, Dec. 1999, vol. 29, no. 6, pp. 674–685.

[20] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Chicago, Illinois, USA, Jun. 2005, pp. 15–26.

[21] G. F. Lin and L. H. Chen, "Time series forecasting by combining the radial basis function network and the self-organizing map," *Hydrological Processes*, vol. 19, no. 10, pp. 1925–1937, Jun. 2005.

[22] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: a hypothesis testing-based approach," *IEEE Trans. Software Engineering*, vol. 32, no. 10, pp. 831–848, Oct. 2006.

[23] C. Liu and J. Han, "Failure proximity: A fault localization-based approach," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Portland, Oregon, USA, Nov. 2006, pp. 46–56.

[24] J. R. Lyle and M. Weiser, "Automatic program bug location by program slicing," in *Proceedings of the Second International. Conference on Computer and Applications*, Beijing, China, Jun. 1987, pp. 877–883.

[25] J. Moody and C. J. Darken, "Learning with localized receptive fields," in *Proceedings of Connectionist Models Summer School*, 1988, pp. 133–142.

[26] A. S. Namin, J. H. Andrews, and Y. Labiche, "Using mutation analysis for assessing and comparing testing coverage criteria," in *IEEE Trans. Software Engineering*, Aug. 2006, vol. 32, no. 8, pp. 608–624.

[27] K. S. Narendra and S. Mukhopadhyay, "Intelligent control using neural networks," *IEEE Control System Magazine*, vol. 12, no. 2, pp. 11–18, Apr. 1992.

[28] D. E. Neumann, "An enhanced neural network technique for software risk analysis," *IEEE Trans. Software Engineering*, vol. 28, no. 9, pp. 904–912, September 2002.

[29] J. Park and I. W. Sandberg, "Universal approximation using radial-basis-function networks," *Neural Computation*, vol. 3, no. 2, 1991.

[30] J. Park and I. W. Sandberg, "Approximation and radial-basis-function networks," *Neural Computation*, vol. 5, no. 2, pp. 305–316, March 1993.

[31] R. Penrose, "A generalized inverse for matrices," in *Proceedings of the Cambridge Philosophical Society*, July 1955, vol. 51, pp. 406–413.

[32] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated support for classifying software failure reports," in *Proceedings of the 25th International Conference on Software Engineering*, Portland, Oregon, USA, May 2003, pp. 465–475.

[33] M. Renieres and S. P. Reiss, "Fault localization with nearest neighbor queries," in *Proceedings of the 18th International Conference on Automated Software Engineering*, Montreal, Canada, October 2003, pp. 30–39.

[34] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold, "Lighweight fault-localization using multiple coverage types," in *Proceedings of the 31st International Conference on Software Engineering*, Vancouver, Canada, May 2009, pp. 56–66.

[35] P. Singla, K. Subbarao, and J. L. Junkins, "Direction-dependent learning approach for radial basis function networks," *IEEE Trans. Neural Networks*, vol. 18, no. 1, pp. 203–222, January 2007.

[36] Y. S. Su and C. Y. Huang, "Neural-network-based approaches for software reliability estimation using dynamic weighted combinational models," *Journal of Systems and Software*, vol. 80, no. 4, pp. 606–615, April 2007.

[37] N. Tadayon, "Neural network approach for software cost estimation," in *Proceedings of the International. Conference on Information Technology: Coding and Computing*, Las Vegas,, Nevada, USA, April 2005, pp. 815–818.

[38] C. Wan and P. B. Harrington, "Self-configuring radial basis function neural networks for chemical pattern recognition," *Journal of Chemical Information and Modeling*, vol. 39, no. 6, pp. 1049–1056, November 1999.

[39] P. D. Wasserman, *Advanced Methods in Neural Computing*. New York: Van Nostrand Reinhold, 1993.

[40] M. Weiser, "Programmers use slices when debugging," *Communications of the ACM*, vol. 25, no. 7, pp. 446–452, Jul. 1982.

[41] I. Vessey, "Expertise in debugging computer programs," *International Journal of Man-Machine Studies: A Process Analysis*, vol. 23, no. 5, pp. 459–494, 1985.

[42] W. E. Wong, V. Debroy, and B. Choi, "A family of code coverage-based heuristics for effective fault localization," *Journal of Systems and Software*, vol. 83, no. 2, pp. 188–208, February 2010.

[43] W. E. Wong and V. Debroy, A Survey on Software Fault Localization Department of Computer Science, University of Texas, Dallas, Technical Report UTDCS-45-09, November 2009.

[44] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of test set minimization on fault detection effectiveness," *Software-Practice and Experience*, vol. 28, no. 4, pp. 347–369, April 1998.

[45] W. E. Wong and Y. Qi, "Effective program debugging based on execution slices and inter-block data dependency," *Journal of Systems and Software*, vol. 79, no. 7, pp. 891–903, Jul. 2006.

[46] W. E. Wong, Y. Qi, L. Zhao, and K. Y. Cai, "Effective fault localization using code coverage," in *Proceedings of The 31st IEEE Computer, Software, and Applications Conference*, Beijing, China, July 2007, pp. 449–456.

[47] W. E. Wong, T. Sugeta, Y. Qi, and J. C. Maldonado, "Smart debugging software architectural design in SDL," *Journal of Systems and Software*, vol. 76, no. 1, pp. 15–28, April 2005.

[48] W. E. Wong and Y. Qi, "BP neural network-based effective fault localization," *International Journal of Software Engineering and Knowledge Engineering*, vol. 19, no. 4, pp. 573–597, June 2009.

[49] W. E. Wong, Y. Shi, Y. Qi, and R. Golden, "Using an RBF neural network to locate program bugs," in *Proceedings of the 19th IEEE Intl. Symposium on Software Reliability Engineering*, Seattle, USA, November 2008, pp. 27–38.

[50] W. E. Wong, T. Wei, Y. Qi, and L. Zhao, "A crosstab-based statistical method for effective fault localization," in *Proceedings of the First Intl. Conference on Software Testing, Verification and Validation*, Lillehammer, Norway, Apr. 2008, pp. 42–51.

[51] χSuds User's Manual Telcordia Technologies, 1998.

[52] A. Zeller, "Isolating cause-effect chains from computer programs," in *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, Charleston, South Carolina, USA, November 2002, pp. 1–10.

[53] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Software Engineering*, vol. 28, no. 2, pp. 183–200, February 2002.

[54] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *Proceedings of the 28th International Conference on Software Engineering*, Shanghai, China, May 2006, pp. 272–281.

[55] Z. Zhang, W. K. Chan, T. H. Tse, B. Jiang, and X. Wang, "Capturing propagation of infected program states," in *Proceedings of the 7th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Amsterdam, The Netherlands, August 2009, pp. 43–52.

[56] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken, "Statistical debugging: Simultaneous identification of multiple bugs," in *Proceedings of the 23rd International Conference on Machine Learning*, Pittsburgh, Pennsylvania, USA, June 2006, pp. 1105–1112.

W. Eric Wong received his Ph.D. in Computer Science from Purdue University. He is currently an Associate Professor in Computer Science with the University of Texas, Dallas. Prior to joining UTD, he was with Telcordia (formerly Bellcore) as a Project Manager for Dependable Telecom Software Development. Dr. Wong received the Quality Assurance Special Achievement Award from Johnson Space Center, NASA, in 1997. His research focus is on the technology to help practitioners develop high quality software at low cost. In particular, he is doing research in software testing, debugging, safety, and reliability at the application and architectural design levels. Dr. Wong is Vice President-elect for Technical Operations of the IEEE Reliability Society, and the Secretary of the ACM Special Interest Group on Applied Computing (SIGAPP).

Vidroha Debroy received his B.S. degree in Software Engineering, M.S. degree in Computer Science, and Ph.D. degree in Software Engineering from the University of Texas, Dallas. He is currently a Software Testing Engineer at Microsoft. Dr. Debroy's research interests include software testing and fault localization, program debugging, and automated and semi-automated ways to repair software faults.

Richard Golden is a Professor of the Cognition and Neuroscience Ph.D. Program within the School of Behavioral and Brain Sciences at the University of Texas at Dallas. He is also the Program Head of the Undergraduate Cognitive Science and the Graduate Applied Cognition and Neuroscience at UTD. Dr. Golden has been a member of the editorial board of the Journal of Mathematical Psychology since 1996. He was also on the editorial board of Neural Processing Letters (1999–2004), and the journal Neural Networks (1995–2006). Dr. Golden received a Bachelor's degree in Electrical Engineering and Experimental Psychology from the University of California at San Diego, and a Master's degree in Electrical Engineering and a Doctoral degree in Experimental Psychology from Brown University.

Xiaofeng Xu is a Ph.D. student at Xiamen University, who is currently a visiting research scholar at the University of Texas, Dallas under Professor W. Eric Wong's supervision. His research interests include code coverage testing, and fault localization.

Bhavani Thuraisingham is a Distinguished Professor in Computer Science, and the Director of the Cyber Security Research Center at the University of Texas, Dallas. She is a Fellow of IEEE, AAAS (American Association for the Advancement of Science), and BCS (British Computer Society). Prior to joining UTD, Dr. Thuraisingham was a Program Director at the National Science Foundation, USA. Her work in information security and information management has resulted in over 80 journal articles, and three US patents. Dr. Thuraisingham was educated in the United Kingdom at the University of Bristol, and the University of Wales.