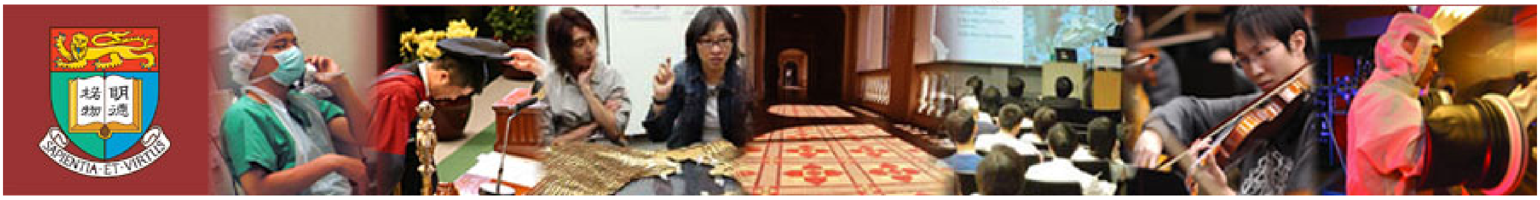


2011 35th IEEE Annual Computer Software and Applications Conference

Precise Propagation of Fault-Failure Correlations in Program Flow Graphs*

Zhenyu Zhang
State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
Beijing, China
zhangzy@ios.ac.cn

W. K. Chan
Department of Computer Science
City University of Hong Kong
Tat Chee Avenue, Hong Kong
wkchan@cs.cityu.edu.hk

T. H. Tse, Bo Jiang
Department of Computer Science
The University of Hong Kong
Pokfulam, Hong Kong
{thtse, bjiang}@cs.hku.hk

***Abstract*—Statistical fault localization techniques find suspicious faulty program entities in programs by comparing passed and failed executions. Existing studies show that such techniques can be promising in locating program faults. However, coincidental correctness and execution crashes may make program entities indistinguishable in the execution spectra under study, or cause inaccurate counting, thus severely affecting the precision of existing fault localization techniques. In this paper, we propose a BlockRank technique, which calculates, contrasts, and propagates the mean edge profiles between passed and failed executions to alleviate the impact of coincidental correctness. To address the issue of execution crashes, BlockRank identifies suspicious basic blocks by modeling how each basic block contributes to failures by apportioning their fault relevance to surrounding basic blocks in terms of the rate of successful transition observed from passed and failed executions. BlockRank is empirically shown to be more effective than nine representative techniques on four real-life medium-sized programs.**

Keyword—fault localization; graph; social network analysis

I. Introduction

Fault localization is an activity in debugging. Many statistical fault localization (SFL) techniques [1][16][18] [19] contrast the program spectra of passed and failed executions[1] to predict the fault relevance of individual program entities. They further construct lists of such program entities in descending order of their estimated fault suspiciousness. Programmer may follow the recommendation of such lists to find faults [22]. Many empirical studies [1] [15][18][19] show that this kind of semi-automatic fault localization technique has good predictive ability for faults.

A vast majority of SFL techniques, such as *Jaccard* [1], *Value Replacement* [14], *Tarantula* [15], *CBI*, [18], *SOBER* [19], and *DES* [30], propose various coefficients to measure the correlations between observed failures and the presence or absence of individual program entities in the corresponding program execution paths. Motivated by this feature, other methodologies have been proposed [5][23] to remove some available test executions with a view to improving the sensitivity of the base techniques. Although the sensitivity in differentiating program entities can be improved, the correlation obtained may be inconsistent with the intentionally omitted test executions (and hence not consistent with all the given facts). Some techniques such as [24][25] consider that simply counting the presence or absence of program entities is merely a primary step, which can be further optimized by proposing weighted coefficients.

Nonetheless, existing empirical studies (such as [15]) pointed out that faults in code regions that have been popularly executed by both passed and failed executions are hard to be located effectively. Indeed, an execution passing through a program entity (such as a compound predicate in the condition of an if-statement) may not trigger a failure even if that program entity is faulty. This is generally known as *coincidental correctness* [13]. As such, a measure of the direct correlation between execution-based failures and the coverage of individual program entities may not precisely point out the faulty positions in programs.

Researchers have proposed techniques to alleviate the issue of coincidental correctness. For instance, in order to make the correlation assessment more precise, *DES* [30] studies the atomic units of each program entity and differentiates various sequences of such atomic units, which essentially isolates some coincidental correctness scenarios. Wang et al. [23] assess whether a particular fragment of an execution is suspicious with respect to a predefined fault pattern to determine whether the fragment should be accessed by SFL techniques. *CP* [28] backward propagates the measured correlation strengths to other program entities based on a program dependency graph, and is empirically evaluated to be promising. However, the accuracy of *CP*'s model is affected by execution crashes (due, for example, to null pointer assignments), which is a kind of failure such that a chain of propagations stops at a block.

In this paper, we propose a novel fault localization technique, known as BlockRank, to alleviate the adversarial effects of coincidental correctness and execution crashes in statistical fault localization. Like *CP*, BlockRank first computes the central tendency of the edge profiles [4][28] for passed executions and that for the failed executions. BlockRank next estimates the difference between these two mean edge profiles by finding out the initial fault relevance of each edge in the faulty program. It then transfers the initial fault relevance of edges to their directly connected basic blocks, back and forth and iteratively, by setting up an equation set to apportion the fault relevance scores among

* This research is supported in part by a grant from the Natural Science Foundation of China (project no. 61003027), grants from the General Research Fund of the Research Grants Council of Hong Kong (project nos. 111410 and 717308), and a strategic research grant from City University of Hong Kong (project no. 7002673).

[1] A failed execution means that a program execution reveals a failure (such as an incorrect output or a crash). A passed execution is the opposite.

0730-3157/11 $26.00 © 2011 IEEE
DOI 10.1109/COMPSAC.2011.16

IEEE computer society

Part I

Block		Statement	Execution Counts (Crashes)					
			t_1	t_2	t_3	t'_1	t'_2	t'_3
b_1	s_1	`if (block_queue) {`	22	17	22	25	29	25

b_2	s_2
	s_3
	s_4
	s_5
		`count = block_queue->mem_count + 1; /* fault */`
		`n = (int) (count*ratio);`
		`proc = find_nth(block_queue, n);`
		`if (proc) {`
			3 (0)	5 (0)	2 (0)	7 (0)	11 (2)	10 (4)

b_3	s_6
	s_7
	s_8
		`block_queue = del_ele(block_queue, proc);`
		`prio = proc->priority;`
		`prio_queue[prio]`
		`= append_ele(prio_queue[prio], proc);`
			1	2	1	5	8	2

b_4	s_9	`} }`	22	17	22	25	27	21

Code examining effort to locate b_2 (containing s_2):

Part IV

Tarantula		*Jaccard*		*Ochiai*		*SBI*	
susp*	r**	susp	r	susp	r	susp	r
0.50	4	0.50	4	0.71	4	0.50	4
0.50	4	0.50	4	0.71	4	0.50	4
0.50	4	0.50	4	0.71	4	0.50	4
0.50	4	0.50	4	0.71	4	0.50	4
100%		100%		100%		100%	

Part II

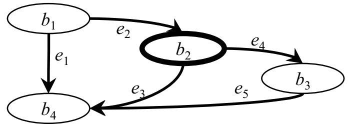
Predicate (also Condition)		Evaluation Results (true:false)					
		t_1	t_2	t_3	t'_1	t'_2	t'_3
p_1 also c_1 (at s_1)	In this example, each predicate happens to have only one condition.	3:19	5:12	2:20	7:18	11:18	10:15
p_2 also c_2 (at s_5)		1:2	2:3	1:1	5:2	8:1	2:4

Code examining effort to locate p_2 (most close to s_2):

Part V

CBI		*SOBER*		*DES-CBI*		*DES-SOBER*	
susp	r	susp	r	susp	r	susp	r
0.00	2	1.03	2	0.00	2	1.03	2
0.00	2	4.17	1	0.00	2	4.17	1
100%		100%		100%		100%	

Part III

Edge		Execution Counts					
		t_1	t_2	t_3	t'_1	t'_2	t'_3
e_1 (b_1→b_4)		19	12	20	18	18	15
e_2 (b_1→b_2)		3	5	2	7	11	10
e_3 (b_2→b_4)		2	3	1	2	1	4
e_4 (b_2→b_3)		1	2	1	5	8	2
e_5 (b_3→b_4)	We add a dummy block b_4 containing s_9 to make a complete CFG.	1	2	1	5	8	2

$P(t_1) = \{e_1 \mapsto 19, e_2 \mapsto 3, e_3 \mapsto 2, e_4 \mapsto 1, e_5 \mapsto 1\}$ $P(t_2)$ $P(t_3)$ $P(t'_1)$ $P(t'_2)$ $P(t'_3)$

Part VI

	CP		Our Method	
	susp	r	susp	r
	0.00		0.00	
	0.47		6.00	
	0.12		0.33	
	0.58		3.67	
	0.88		3.67	
b_1	0.34	4	4.00	3
b_2	0.73	3	4.32	1
b_3	1.02	2	3.67	4
b_4	1.16	1	4.00	3

Code examining effort to locate b_2 (containing s_2):

CP	Our Method
75%	25%

*** susp: suspiciousness computed for block/predicate/condition/edge**

**** r: ranking for block/predicate/condition/edge**

Fig. 1. Faulty version v2 of program "schedule".

basic blocks. Finally, BlockRank sorts the basic blocks of the program in descending order of their fault relevance and maps the rankings of each basic block to its statements.

BlockRank is innovative in that if a basic block is important in correlation to any observed failure, and a direct incoming block of this basic block is its *popular source* of entrance point, the importance of this particular incoming block should *indirectly* correlate to the observed failure. Therefore, a highly popular source (i.e., basic block) to a highly fault-correlated basic block can reveal its fault relevance even though this popular basic block has been heavily executed by passed executions and rarely executed by failed executions. Different from many existing techniques that use vertex profiles to assess the correlation of individual program entities to failures, BlockRank uses edge profiles to express program executions and computes mean edge profiles to alleviate the impact of coincidental correctness. Moreover, unlike *CP*, BlockRank uses the observed transition rates among blocks to handle execution crashes, and models popularity using every such transition rate. Our experiment shows that BlockRank is more effective than *Tarantula*, *Jaccard*, *Ochiai*, *SBI*, *CBI*, *SOBER*, *DES-CBI*, *DES-SOBER*, and *CP* on four real-life programs.

The main contribution of this work is twofold. First, it proposes a novel and precise propagation model of fault-failure correlations. It points out a method to factor in both coincidental correctness and execution crashes. Second, it reports an experiment that verifies this innovative approach.

The rest of the paper is organized as follows. Section II uses an example to motivate this work. Section III presents our technique, which is evaluated in Section IV. Section V reviews related work. Section VI concludes the paper.

II. Motivating Example

A. Example Program

Part I of Fig. 1 shows a code excerpt from a faulty version v2 of the program "schedule" (from SIR [10]). This code excerpt manages a process queue. It first computes the index of a target process, and then moves it along priority queues. There is a fault in statement s_2, which causes the program to generate an incorrect index *n* in s_3. It may finally lead to a program failure, or immediately crashes the program execution when executing s_4 (due to a null-pointer error in the function call `find_nth`).

In this example, the two "if" statements s_1 and s_5 divide the code excerpt into three basic blocks [5] b_1, b_2, and b_3.[2] For ease of explanation, we add a dummy basic block b_4 to form a complete control-flow graph (CFG) [4] and assume

[2] Each basic block consists of program statements that will share the same execution count [30] in any execution, unless the program crashes (as in the example of Fig. 1) or the system API `exit` is invoked.

that each execution starts from b_1 and ends at b_4. The CFG representing the code excerpt is shown in Part III of Fig. 1. In this graph, each ellipse represents a basic block. We use thick border to highlight b_2 because it contains the fault. The four basic blocks are connected via five control-flow edges (e_1, e_2, e_3, e_4, and e_5, shown as lines with arrows). Let us use e_1 as an illustration. It indicates that s_1 has been evaluated to be false in an execution, and b_4 will be next executed. Further, statements s_1 and s_5 contain predicates [18][19] p_1 and p_2, respectively, as shown in Part II of Fig. 1.

To demonstrate previous techniques and motivate our approach, we randomly select three passed test cases (t_1, t_2, and t_3) and three failed test cases (t'_1, t'_2, and t'_3) from the test pool of the program "schedule" [10]. The execution counts of the blocks, as well as the number of times that the program crashes within b_2, are shown in Part I of Fig. 1. The evaluation results of the predicates and the frequencies of the edges (that is, how many times each edge is exercised in a program execution [28]) are shown in Parts II and III, respectively, of Fig. 1. Let us take the gray column as an example. It shows that blocks b_1, b_2, b_3, and b_4 are executed 22, 3, 1, and 22 times, respectively, for test case t_1. The numbers "3 (0)" mean that, in the three times when b_2 is executed, the program never crashes. Furthermore, during these three times, the predicate p_2 is evaluated as true and false for twice and once, respectively. In the 22 times that b_1 is executed, the predicate p_1 is evaluated as true and false for 3 and 19 times, respectively. The frequencies of edges e_1, e_2, e_3, e_4, and e_5 with respect to the execution of t_1 are 19, 3, 2, 1, and 1, respectively. We represent them as $P(t_1) = \{e_1 \mapsto 19, e_2 \mapsto 3, e_3 \mapsto 2, e_4 \mapsto 1, e_5 \mapsto 1\}$. Following Ball et al. [4], we refer to $P(t_1)$ as the *edge profile* of test case t_1. The other edge profiles $P(t_2)$, $P(t_3)$, $P(t'_1)$, $P(t'_2)$, and $P(t'_3)$ can be similarly explained. Given a frequency relation $e_i \mapsto n_i$, we define $|e_i \mapsto n_i| = n_i$.

B. Previous Techniques Revisited

In Part IV of Fig. 1, we show the effectiveness of four statement-level techniques, *Tarantula* [15], *Jaccard* [1], *Ochiai* [1], and *SBI* [26], on locating the fault in this code excerpt. By applying *Tarantula*, we can compute the *fault relevance score* [15][28] for each block and accordingly compute the *code examining effort to locate fault* [26][29]. This effort is extensively used to evaluate the effectiveness of fault localization techniques in previous studies [26][27] [29]. Similarly, the effectiveness of techniques *Jaccard*, *Ochiai*, and *SBI* are also evaluated and shown in Part IV. We observe from the results that none of the peer techniques can locate the fault until all the code in the excerpt has been examined (in the order of examination recommended by each technique). This is because all blocks (and statements) are exercised to the same extent by the set of the passed and failed executions. In general, if a faulty is executed but no failure is revealed (and consequently the test case is marked as a passed test case), the phenomenon is known as *coincidental correctness* [13] in testing. Coincidental correctness makes the execution counts of program entities indistinguishable, and lowers the effectiveness of fault localization techniques. Unfortunately, previous studies also show that coincidental correctness occurs frequently in real-life programs [23].

In Part V of Fig. 1, we show the effectiveness of two predicate-based techniques (*CBI* [18] and *SOBER* [19]) and two condition-based techniques (*DES-CBI* [30] and *DES-SOBER* [30]) in locating the fault in this code excerpt.[3] [4] Here, we omit the tedious computation process and directly show the code examining effort to locate the fault. Interested readers may follow the references to find the details of these methods. *CBI* investigates whether a predicate is evaluated (to be true or false) [18] and ignores the detailed evaluation results (such as how many times it is evaluated). As a consequence, *CBI* and *DES-CBI* cannot distinguish between the evaluation results of p_1 and p_2, and need 100% code examining effort to locate the fault. *SOBER* uses *evaluation bias* [19] to partially capture the execution spectra of predicates, but inaccurately gives predicate p_2 a higher fault relevance score than p_1. As such, *SOBER* and *DES-SOBER* still need to examine 100% of all code to locate the fault. The unsatisfactory results are also due to the unexpected coincidental correctness issues.

In part VI of Fig. 1, we evaluate the technique *CP* [28]. *CP* lets the fault relevance scores of blocks propagate via edges, so that they can be found by solving an equation set containing fault relevance scores of both blocks and edges (with the former unknown and the latter known). However, the propagation model of *CP* cannot handle program crashes (at t'_2 and t'_3), which stop the propagation of errors to any other basic blocks, thus causing inaccurate result in this example. Because of this problem, *CP* has to evaluate 75% of all the code before the fault is located. As we will present later, BlockRank alleviates this problem.

C. Motivating Our Approach

We first compute a mean edge profile $P^{\surd}$ [4][28] for all the passed executions. It is a frequency relation showing a mean frequency of passed executions with respect to every edge. In the motivating example, $P^{\surd} = \frac{P(t_1)+P(t_2)+P(t_3)}{3} = \{e_1 \mapsto 17.00, e_2 \mapsto 3.33, e_3 \mapsto 2.00, e_4 \mapsto 1.33, e_5 \mapsto 1.33\}$. Here, each arithmetic operation (such as addition and scalar division) on the edge profiles $P(t_i)$ represent the element-wise operation of every individual entry in $P(t_i)$ (see Definition 1 in Section III.C). Similarly, we compute a mean edge profile $P^{\times}$ for all failed executions. In our example, $P^{\times} = \frac{P(t'_1)+P(t'_2)+P(t'_3)}{3} = \{e_1 \mapsto 17.00,\ e_2 \mapsto 9.33, e_3 \mapsto 2.33, e_4 \mapsto 5.00, e_5 \mapsto 5.00\}$. Such mean edge profiles stand for the execution spectra producing no failure and the execution spectra correlating to failures, respectively. We therefore use $P^{\Delta} = P^{\times} - P^{\surd}$ to calculate the increase from

[3] Note that, in this example, each predicate happens to have only one condition, so that the effectiveness of the each condition-based technique is identical to that of its predicate-based counterpart (that is, the effectiveness of *DES-CBI* is the same as that of *CBI*, and the effectiveness of *DES-SOBER* is the same as that of *SOBER*).

[4] After predicates have been assigned fault relevance scores, programmers are suggested to follow a breath-first search (starting from predicates having the highest fault relevance scores) to locate faults [22].

$P^{\surd}$ to $P^{\times}$, statistically modeling the *net contribution to failures* [28] by all branch transitions (that is, edges). In our example, $P^{\Delta} = \{e_1 \mapsto 0.00, e_2 \mapsto 6.00, e_3 \mapsto 0.33, e_4 \mapsto 3.67$ $e_5 \mapsto 3.67\}$.

The five values in P^{Δ} are used to calculate the fault relevance scores of the basic blocks b_1, b_2, b_3, and b_4 by equations (1)–(4):

$$BR(b_1) = BR(b_2) + \frac{0.00}{0.00 + 0.33 + 3.67} BR(b_4) = 4.00 \tag{1}$$

$$BR(b_2) = BR(b_3) + \frac{0.33}{0.00 + 0.33 + 3.67} BR(b_4) = 4.00 \tag{2}$$

$$BR(b_3) = \frac{3.67}{0.00 + 0.33 + 3.67} BR(b_4) = 3.67 \tag{3}$$

$$BR(b_4) = 0.00 + 0.33 + 3.67 = 4.00 \tag{4}$$

Let us first take equation (1) to illustrate the basic idea. We recall that in an execution, the execution of b_1 can be immediately followed by the execution of either e_1 or e_2. On the other hand, b_2 will only be reached directly from b_1 via the edge e_2. An execution passing through b_2 must also pass through b_1. As such, the fault relevance of b_2 is deemed to come from that of b_1 due to the investigation on their spectra in execution. Suppose that, when calculating the fault relevance scores of b_1 and b_2, we find that the fault relevance of b_2 totally contributes to that of b_1. On the other hand, b_4 may be reached from e_1, e_3, or e_5. Hence, the fault relevance of b_4 partially contributes to b_1. We apportion the fault relevance of b_4 to b_1 based on the edges from other blocks to b_4. Block b_4 has three incoming edges e_1, e_3 and e_5. From the edge profile P^{Δ}, the net contributes of these edges to failures are 0.00, 0.33, and 3.67, respectively. Since block b_1 can directly reach b_4 only via e_1, block b_4 contributes $\frac{0.00}{0.00+0.33+3.67}$ of its fault relevance to block b_1.[5] For each block having outgoing edge(s) (namely, block b_2 or b_3) we set up a formula similar to equations (2) and (3). Block b_4 has no outgoing edge, and hence we use the sum of frequencies in P^{Δ} of all incoming edges to estimate its fault relevance score, as in equation (4). Thus we obtain $BR(b_1) = 4.00$, $BR(b_2) = 4.00$, $BR(b_3) = 3.67$, and $BR(b_4) = 4.00$. Here, $BR(b_x)$ denotes the fault relevance score of block b_x.

Because the execution of block b_2 is observed to have chances to crash, leading to abnormal execution termination before reaching the block b_3 or b_4, we distinguish crashing cases from non-crashing cases. The probability of b_2 not crashing is given by:

$$T(b_2) = \frac{2 + 3 + 1 + 2 + 1 + 4 + 1 + 2 + 1 + 5 + 8 + 2}{3 + 5 + 2 + 7 + 11 + 10} = 0.84$$

which computes the ratio of the number of times that the outgoing edges of b_2 are exercised in all executions to the number of times that the incoming edges of b_2 are exercised in all executions (the values used in computation can be obtained from Part III of Fig. 1). We further use equation (5) to calibrate $BR(b_2)$ to $BR'(b_2)$:

$$BR'(b_2) = BR(b_2) \times T(b_2) + I(b_2) \times [1 - T(b_2)] = 4.32 \tag{5}$$

$BR'(b_2)$ is the weighted sum of fault relevance scores of b_2 in crashing and non-crashing cases. In a non-crashing case, the fault relevance score of b_2 is calculated as $BR(b_2)$. In a crashing case, because the execution of b_2 must not be transferred via any outgoing edge, the fault relevance score is calculated as the sum of net contributions to failures (frequencies in P^{Δ}) of all its incoming edges, as in equation (6):

$$I(b_2) = 6.00 \tag{6}$$

The basic block b_2 containing the faulty statement s_2, is given a highest fault relevance score $BR'(b_2) = 4.32$ and ranked the highest. As a result, BlockRank needs to examine 25% of all code to locate fault in the example of Fig. 1.

This example motivates a novel approach that alleviates the impacts from coincidental correctness and program crashing. However, there are still many challenges with such an approach. Apparently, readers may be interested in the mathematical basis of the proposed approach. Furthermore, there are also practical issues. For example, what if a loop exists in a control-flow graph so that the calculation of $BR(b_x)$ in equations (1)–(4) may rely on the results of one another? We will elaborate on our model in the next section.

III. Our Model

A. Problem Settings

Let M denote a faulty program. $T \cup T'$ is a set of test cases, where $T = \{t_1, ..., t_i, ..., t_u\}$ is the set of all passed test cases and $T' = \{t'_1, ..., t'_i, ..., t'_v\}$ is the set of all failed test cases. Our aim is to estimate the extent that each statement s in M is related to faults. In this paper, we use the term *fault relevance score* of s to denote such a value. We then sort the statements into a list in the descending order of their fault relevance scores thus calculated. In previous studies, such a list is deemed useful to facilitate programmers in locating faults in programs [22][26].

B. Preliminaries

1) Control flow graph

Following existing work [1][3], we use $G(M) = \langle E, B \rangle$ to denote the control flow graph (CFG) [3] of a given program M, where $E = \{e_1, e_2, ..., e_m\}$ is the set of control flow edges of M, and $B = \{b_1, b_2, ..., b_n\}$ is the set of basic blocks of M. In particular, we use $e_i = edge(b_{i1}, b_{i2})$ to denote an edge from block b_{i1} pointing to block b_{i2}. Edge e_i is called an outgoing edge of b_{i1} and an incoming edge of b_{i2}. We say that b_{i1} is a predecessor of b_{i2}, and b_{i2} is a successor of b_{i1}. We further use the notation $edges(b_j, *)$ and $edges(*, b_j)$ to represent, respectively, the set of outgoing and incoming edges of b_j.

For example, Part III of Fig. 1 gives a CFG, in which edges e_1, e_2, e_3, e_4, and e_5 represent branch transitions, and nodes b_1, b_2, b_3, and b_4 represent basic blocks. Edges e_3 and e_4 are incoming edges of block b_3. Block b_3 is a successor of b_2 in relation to edge e_4. The set $edges(*, b_4)$ stands for the set of incoming edges of block b_4, which is $\{b_1, b_2, b_3\}$.

[5] We also notice that the edge frequency of e_1 in P^{Δ} is 0, which means that edge e_1 is not relevant to the fault. Accordingly, b_4 does not contribute to b_1.

2) Edge Profile

The frequency of an edge [4][28] is the number of times that the edge has been exercised in a program execution. In this paper, we use $\theta(e_i, t_k)$ to denote the frequency relation with respect to the execution of edge e_i over a passed test case t_k. Similarly, $\theta(e_i, t'_k)$ is the frequency relation with respect to the execution of e_i over a failed test case t'_k.

The frequency relation with respect to the execution of all edges is represented using an edge profile. In this paper, we also use the term *edge profile of a test case* to denote the edge profile over the execution of a test case. For example, the edge profile of a passed test case t_k is $P(t_k) = \{\theta(e_1, t_k), \theta(e_2, t_k), ..., \theta(e_m, t_k)\}$, where $\theta(e_i, t_k)$ is the frequency relation with respect to the execution of edge e_i over test case t_k. Similarly, the edge profile of a failed test case t'_k is $P(t'_k)=\{\theta(e_1, t'_k), \theta(e_2, t'_k), ..., \theta(e_m, t'_k)\}$. In Fig. 1, for instance, the edge profile for test case t_1 is $P(t_1) = \{e_1 \mapsto 19, e_2 \mapsto 3, e_3 \mapsto 2, e_4 \mapsto 1, e_1 \mapsto 1\}$.

3) PageRank

PageRank [21] is a link analysis technique to find popular Web pages. It models the Internet as a directed graph, where the nodes are the Web pages and the edges are the links. PageRank assumes that a more popular page tends to be more important and has more links towards it [21]. PageRank thus counts a link from page p to page q as a vote (by p) on the importance of q. By analyzing the constructed graph, it measures the importance of every page and accordingly estimates the popularity of each web page as follows: Let p be a page, $F(p)$ be the set of pages that p has links to, and $B(p)$ be the set of pages having links to p. It uses $PR(p)$ to denote the ranking of a page p. PageRank assumes that "*highly linked pages should be regarded as more important than pages being seldom linked*" [21]. Therefore, a link to a page is a vote of the importance of that page. $PR(p)$ is calculated by the formula:

$$PR(p) = (1-d) + d \sum_{q \in B(p)} \left[PR(q) \times \frac{1}{|F(q)|} \right]$$

Here, $|F(q)|$ means the number of pages in the set $F(q)$. The argument d is a damping factor introduced to simulate the probability that a user continues to browse pages via the links. Only in such a scenario, the links contribute to the popularity and Google ranks of their connected pages. PageRank uses a magic number 0.85 for d, which is an empirical value for search engines [21].

4) Inspirations

Infected program states may propagate via control-flow edges during program execution. We can easily think of using an execution transition via an edge to capture the propagation of infected program states, and use the frequency of an edge as "votes" to the fault relevance score of the basic block that the edge points to. However, different from the concepts in PageRank that counts the incoming links to a page as votes to the importance of that page, in our case, we let the outgoing edges of a block vote for the fault relevance score of that block. This is because the root cause of observed failures is the block from which the infected program states propagate (as well as the outgoing edges of that block).

Since a program may crash during the execution of some basic blocks and that stops the propagation of infected program states, we intuitively employ a damping factor to simulate such a case. However, directly applying $d = 0.85$ in fault localization has no scientific ground. In our work, we do not use any magic number. We calculate the ratio of the number of times the execution leaves a block to that of entering that block, to estimate the probability of that block propagating infected program states via its outgoing edges.

C. Our Model Proposal — BlockRank

Since we aim at ranking the basic blocks, our model is named BlockRank, which is a three-stage process. ***S1: Construct the suspicious edge profile.*** We use the edge profile $P(t_k)$ of each passed test case t_k for $k = 1, 2, ..., u$ to compute the mean edge profile $P^{\surd}$. In the same manner, we compute the mean edge profile $P^{\times}$ from the edge profile $P(t'_k)$ for each failed test cases t'_k for $k = 1, 2, ..., v$. By comparing $P^{\surd}$ with $P^{\times}$, we compute the *suspicious edge profile* P^{Δ}. ***S2: Calculate the fault relevance scores.*** In this stage, BlockRank estimates how much each basic block is fault-relevant by assigning a fault relevance score $BR(b_j)$ to each of them. $BR(b_j)$ is computed as the sum of the fault relevance scores related to two chances: (a) the chance that the program terminates within the block b_j (e.g., the basic block b_j involves an exit function call statement), and (b) the chance that the program does not terminate within b_j. ***S3: Sort the basic blocks.*** BlockRank then sorts the basic blocks in the descending order of their fault relevance scores, and generates a list of basic blocks. After that, since we have no further clue to prioritize the statements within any block, the ranking of basic block will be assigned to the ranking of every statement of the basic block. Developers can examine the statements in the descending order of their rankings to seek faults.

1) Constructing the Suspicious Edge Profile

In our model, for each program module, we construct an individual CFG. We investigate how the edge frequencies correlate to the failures by comparing the edge profiles of passed executions with those of failed executions. Since the number of passed executions may vary significantly from the number of failed executions, it may not be systematic to compare them directly. In our model, we normalize them first before comparing them with each other. Thus, we calculate two mean edge profiles, one for passed executions and the other for failed executions, and use the notation $P^{\surd} = \{\theta^{\surd}(e_1), \theta^{\surd}(e_2), ..., \theta^{\surd}(e_m)\}$ and $P^{\times} = \{\theta^{\times}(e_1), \theta^{\times}(e_2), ..., \theta^{\times}(e_m)\}$ to denote them. The values $\theta^{\surd}(e_i)$ in $P^{\surd}$ and $\theta^{\times}(e_i)$ in $P^{\times}$ are computed by:[6]

$$\theta^{\surd}(e_i) = e_i \mapsto \frac{1}{u} \sum_{k=1}^{u} |\theta(e_i, t_k)|$$

[6] Here, we initialize the central tendency by the arithmetic mean so that we can compare it with *CP* more directly.

$$\theta^{\times}(e_i) = e_i \mapsto \frac{1}{v}\sum_{k=1}^{v}|\theta(e_i, t'_k)|$$

They represent the expected number of times that an edge is exercised in a passed and a failed execution, respectively. Another benefit is that using such mean values reduces the bias effect from individual executions. To investigate the net contribution of an edge to failures, we subtract the edge frequencies in $P^{\surd}$ from the corresponding edge frequencies in $P^{\times}$, since the former captures normal program behavior while the latter can be seen as a mixture of observed abnormal program behavior (failures) and unobserved abnormal program behavior (coincidental correctness).

Definition 1. *The* ***suspicious edge profile*** $\mathbf{P}^{\Delta}$ *is a frequency relation such that each element relates an edge e_i to its frequency in the mean edge profile of the failed executions minus its mean frequency in the mean edge profile of the passed executions. Thus,* $P^{\Delta} = \{\theta^{\Delta}(e_1), \theta^{\Delta}(e_2), ..., \theta^{\Delta}(e_m)\}$, *where each $\theta^{\Delta}(e_i)$ is given by:*

$$\theta^{\Delta}(e_i) = e_i \mapsto \left(|\theta^{\times}(e_i)| - \left|\theta^{\surd}(e_i)\right|\right)$$

The frequency of an edge e_i in P^{Δ} indicates the change in the mathematical expectation of the number of times that the edge e_i is executed in a failed execution from that in a passed execution. Thus, a larger $|\theta^{\Delta}(e_i)|$ means that the edge e_i has a greater frequency in the mean edge profile for failed executions than that for passed executions. A smaller $|\theta^{\Delta}(e_i)|$ means the opposite. In the next section, we present how different elements in P^{Δ} may affect one another.

2) Calculating the Fault Relevance Scores

From the suspicious edge profile, we obtain the difference in the expected edge frequency from passed executions to failed executions. Such information holds the clue to correlate a branch transition to failures. We further use it to estimate the fault relevance score of each basic block. To ease our reference, we use the notation $BR(b_j)$ to represent the fault relevance score of a basic block b_j.

Let us first discuss how a program execution transits from one basic block to another. After executing a basic block b_j, the execution may transfer control to one of b_j's successor basic blocks. Suppose b_k is a successor basic block of b_j. The infected program states in b_j may propagate to b_k. We thus let the fault relevance of b_k backwardly contribute to the fault relevance of b_j, to reflect the propagation. However, b_k may have a number of incoming edges so that the fault relevance of b_k may contribute to a number of predecessor basic blocks. We therefore use a fraction of the fault relevance score $BR(b_k)$ of b_k to contribute to the fault-relevant score of b_j. To determine the fraction, we compute the sum of frequencies of the incoming edges of b_k (i.e., the edges in $edges(*, b_k)$) in the suspicious edge profile (see Definition 1), and compute the ratio of the frequency of this particular $edge(b_j, b_k)$ over the sum of all frequencies. This ratio is given by:[7]

$$W\left(edge(b_j, b_k)\right) = \frac{\theta^{\Delta}\left(edge(b_j, b_k)\right)}{\sum_{e\in edges(*,b_k)}\left|\theta^{\Delta}(e)\right|}$$

The fraction of the fault-relevant score that b_k contributes to b_j is, therefore, the product of this ratio and the fault relevance score of b_k (that is, $BR(b_k)\cdot W(b_j, b_k)$).

The basic block b_j however may have a number of successors. Therefore, we sum up such fractions from all successors of b_j as the fault relevance contributions from the successors (if the execution can transit to the successors after executing b_j), as follows:

$$P(b_j) = \sum\nolimits_{e=edge(b_j,b_k)\in edges(b_j,*)}[BR(b_k)\cdot W(e)]$$

Sometimes, the execution of the statements in a basic block may simply crash, throw an unhandled exception, or invoke an exit function call. In our model, we do not construct edges to connect such statements to the standalone exit block. Because of the existence of such statements, a program execution may leave a basic block (as what we have described above), or cease any further branch transitions after exercising b_j (i.e., the program may have exited, crashed or the execution may leave the current module and execute statements of other program modules). We therefore distinguish whether or not the infected program states of a basic block may propagate to any of its successor basic blocks via an outgoing edge. We model the chance that the basic block propagates its infected program states to its successor basic blocks via an outgoing edge by the value of *block transition rate*.

Definition 2. *The* ***block transition rate*** $\boldsymbol{T(b_j)}$ for j = 1, 2, ..., *n, is the probability of the program control flow continues to transfer to other basic blocks in the same CFG after the basic block b_j has been executed. $T(b_j)$ is given by:*

$$T(b_j) = \frac{u\sum_{e\in edges(b_j,*)}\left|\theta^{\surd}(e)\right| + v\sum_{e\in edges(b_j,*)}\left|\theta^{\times}(e)\right|}{u\sum_{e\in edges(*,b_j)}\left|\theta^{\surd}(e)\right| + v\sum_{e\in edges(*,b_j)}\left|\theta^{\times}(e)\right|} \quad (7)$$

In equation (7), the denominator captures the total number of times (in both passed and failed executions) that the program execution enters the basic block b_j, from any incoming edge. The numerator captures the total number of times (in both passed and failed executions) that the program execution leaves b_j, from any outgoing edge. The value of such a defined $T(b_j)$ is in the range of [0, 1]. The higher the value of $T(b_j)$ is, the program execution has a higher probability to transfer control to other basic blocks on the same CFG after the basic block b_j has been executed.

$$BR(b_j) = I(b_j)\cdot\left[1 - T(b_j)\right] + T(b_j)\cdot P(b_j) \quad (8)$$

Equation (8) calculates the fault relevance score for a basic block b_j. The term $T(b_j)\cdot P(b_j)$ represents the amount of fault relevance contributed from b_j's successor basic blocks in case of successful block transition. The term $I(b_j)\cdot\left[1 - T(b_j)\right]$ represents the amount of fault relevance

[7] An exception is that the denominator in equation (10) may be zero. In that case, we simply use zero as the result.

of b_j when the execution cannot transit to another basic block after executing b_j, where $I(b_j)$ is given by:

$$I(b_j) = \sum_{e \in edges(*,b_j)} |\theta^{\Delta}(e)| \tag{9}$$

In equation (9), $I(b_j)$ captures the amount of fault relevance of b_j in case of no block transition. In this scenario, we cannot use the outgoing edges of b_j to estimate this score (since there is no transition to successor basic blocks). We therefore use the sum of the frequencies of incoming edges of b_j in the suspicious edge profile to estimate the fault relevance score $I(b_j)$. We recall that the frequency of an edge in the suspicious edge profile represents the increase in the execution frequency from a passed execution to a failed execution. Thus, to find the suspiciousness of such b_j, we use the frequency values captured in the suspicious edge profiles for the set of edges in $edges(*, b_j)$.

By applying equation (9) to set up an equation for each basic block, we obtain an equation set. The number of equations in this set is equal to the number of basic blocks of the program. By solving the equation set, we obtain the fault relevance score for each basic block. Our model works regardless of whether the CFG contain loops.[8]

3) *Sorting the Basic Blocks*

After obtaining the fault relevance score of each basic block, we produce a ranking list of the basic blocks (from all CFGs) in descending order of the fault relevance scores associated with them. All statements not in any basic block will be grouped under an additional block, which will be appended to the above ranking list. It will have a lower fault relevance score than any other block.

After we build up the rankings for basic blocks, we proceed to assign rankings to statements. The ranking of a statement is the sum of total number of statements in its belonging basic block and total number of statements in the basic blocks ranked before its belonging basic block [15].

IV. Empirical Evaluation

A. *Experimental Setup*

1) *Selection of Subject Programs*

To evaluate our technique, we use four UNIX programs as our subject programs. Their functionality can be found in the SIR website [10]. Some of them are real-world programs and have real-life scales. Each of them has many sequential versions. They have been adopted to evaluate fault localization techniques in previous work (such as [14][27][28]). Both the programs and the associated test suites we use are downloaded from SIR [10]. Table I shows their real-life program version numbers, number of executable statements, number of applicable faulty versions, and the number of test cases. Let us take the program flex as an example. The real-life versions used are in the range of flex-2.4.7 to flex-2.5.4. Each of them has 8571 to 10124 lines of executable statements. There are a total of 21 faulty versions finally used in our experiment. All these faulty versions share a test suite containing 567 test cases.

TABLE I. Statistics of Subject Programs.

Program	Real-Life Version Numbers	LOC	No. of Faulty Versions	No. of Test Cases
flex	2.4.7–2.5.4	8571–10124	21	567
grep	2.2–2.4.2	8053–9089	17	809
gzip	1.1.2–1.3	4081–5159	55	217
sed	1.18–3.02	4756–9289	17	370
Total			110	

Following the documentation of SIR and previous work [15][18][19], we exclude the faulty versions whose faults cannot be revealed by any test case. It is because that both our technique and the other peer techniques used in the experiment [1][15][18][19] require the existence of failed test cases. In addition, following the advice of the previous work [11], if a faulty version comes with more than 20% of all the test cases to be failed ones, we exclude it. Besides, the faulty versions not supported by our experiment environment (in which we use a Sun Studio C++ compiler) are also excluded. Finally, all remaining 110 faulty versions are selected in our experiment (listed out in Table I).

Following our previous study [19], we apply the whole test suite as the input to individual subject programs.

2) *Selection of Peer Techniques*

In our experiment, we select nine representative techniques to compare with. *Tarantula*, *Jaccard*, *Ochiai*, and *SBI* are techniques that work at the statement level. *CBI* and *SOBER* are techniques based on predicates. *DES-CBI* and *DES-SOBER* are based on conditions. Another technique *CP* is a propagation-based and uses edge profile information. Our approach, BlockRank, uses edge profile information and works at the statement level. Comparisons with the above techniques will evaluate the effectiveness of BlockRank and give further insights.

3) *Effectiveness Metrics*

CBI and *SOBER* generate ranking lists, which contain all predicate statements, and sort them in the descending order of their fault relevance scores. In previous studies, the metrics T-score [22] is used to evaluate the effectiveness of these techniques. The metrics, T-score, uses program dependence graph to calculate the distance between program statements. Starting from top-ranked predicate statements[9] generated by *CBI* or *SOBER*, T-score conducts a breadth-first search of all the statements for faults. The search terminates when it encounters any faulty statement, and the percentage of statements examined (out of all the statements) is returned as the effectiveness [22]. The same strategy is applied to *DES-CBI* and *DES-SOBER* [30].

On the other hand, *Tarantula*, *Jaccard*, *Ochiai*, *SBI*, *CP*, and our method BlockRank output a ranking list containing

[8] In the case of loops, we iteratively solve the fault relevance scores [28]. To address convergence issue, one may adopt an upper bound (such as the frequently used constant 200 [21]) as the maximum number of iterations.

[9] Since it is reported that the top-5 *t*-score strategy achieves the highest effectiveness for CBI and SOBER [19], we follow previous studies to choose the top-5 *t*-score results to evaluate them. It means that we pick the top five predicates in the ranked predicates list to start a breadth-first search.

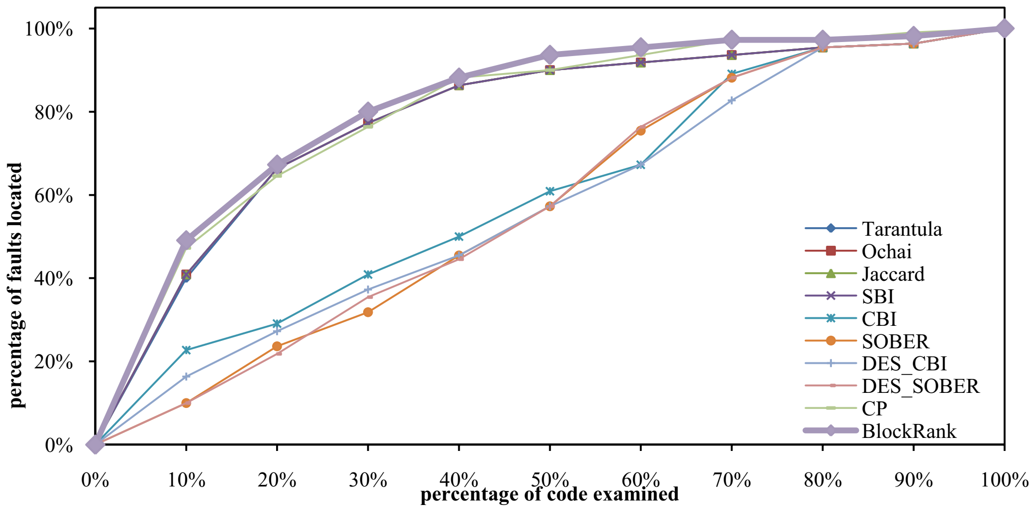


Fig. 2. Overall effectiveness comparison.

all statements, and the T-score metrics cannot be applied directly. To evaluate their effectiveness, we simply check all the statements in the ranking list in ascending order of their rankings until a faulty statement is encountered. The percentage of statements examined (out of all the statements) is returned as the effectiveness of that technique. Note that, statements in a tie case, which means statements of identical rankings, are examined as a whole.

B. *Results and Analysis*

In this section, we report the results of the techniques. The data indicated as *Tarantula*, *Jaccard*, *Ochiai*, *SBI*, *CBI*, *SOBER*, *DES-CBI*, *DES-SOBER*, and *CP* are worked out using the techniques described in their original papers [15], [1], [1], [26], [18], [19], [30], [30], and [28], respectively. The data indicated as BlockRank is our technique.

We first directly compare the overall effectiveness of the techniques. Fig. 2 gives an overview of the effectiveness results on the 110 faulty versions. Let us take the curve of BlockRank as an example. The *x*-coordinate represents the percentage of code examined; the *y*-coordinate represents the percentage of faults located by BlockRank within the given code examining effort specified by the *x*-coordinate. The curves of the other techniques can be similarly interpreted. All the ten curves start from the point (0%, 0%) and finally reach the point (100%, 100%). Apparently, it reflects the fact that no fault can be located when not examining any code (0%), while all the faults can be located when all code (100%) has been examined.

Let us focus on some checkpoints to compare the four techniques. When 10% of the code is examined, BlockRank can locate faults in 49% of the faulty versions. On the other hand, *CP*, which is the best among the other techniques with respect to this checkpoint, can locate 47% of the faults. The other techniques, as well as the other checkpoints, can be similarly explained. Since the curve of BlockRank is always (except on the 90% checkpoint) above all the other curves, it performs better than the other techniques.

To give a better understanding to the statistics of overall effectiveness for these techniques, we next use Table II to list the mean values of the effectiveness of locating faults (in the row “mean”) for the 110 faulty versions, as well as their standard deviations (in the row “stdev”). At the same time, since some previous studies (such as [19]) suggest that the top 20% code examining range is more important than other ranges, we also show the results of the studied techniques at the 5%, 10%, 15%, and 20% checkpoints (that is, with 5% intervals for code examining effort). For the upper four rows, a larger number means better effectiveness, while for the lower two rows, the smaller the better. We use a bold font to highlight the best technique for each test. For example, when up to 5% of the code is examined, BlockRank and *CP* can locate faults in 39% of all faulty versions, thus outperforming the others. BlockRank is the unique winner for 10%, 15%, and 20% of average code examining effort. It shows that BlockRank gives the best results among the studied techniques from these viewpoints. Such an observation consolidates our impression from Fig. 2. On the other hand, BlockRank achieves the smallest standard deviation among all techniques. It means that BlockRank has a more stable effectiveness than the other techniques in locating faults from the studied subjects.

C. *Threats to Validity*

Internal validity is related to the risk of having confounding factors to affect the experimental results. A key factor is whether our experiment platform is correctly built. In our implementation, the algorithms of related techniques strictly adhere to those published in the literature. SIR reports the distribution of passed and failed test cases [10]. It confirms that the fault matrix of the faulty versions of subject programs can be platform dependent.

We use four UNIX programs, as well as their associated test suites in the experiment. All these programs are real-world programs of real-life sizes, and they have been used by other researchers to verify their debugging research [14] [27][28]. A further comparison of our approach with other techniques, using other subjects, with other programming languages (other than C), or with real faults may further strengthen the external validity of our empirical study. Besides, although our technique ideally works in a manner of locating one fault each time before moving to another, the use of real-life multi-fault programs in evaluation may consolidate the empirical results.

We use different measures to evaluate the results of different techniques. Statement-level techniques (such as *Tarantula*) produce the ranking lists of statements, while predicate/condition-based techniques (such as *CBI*) give the ranking lists of predicate statements. We follow [19] and [26] to use T-score to evaluate the former, and another metric to evaluate the latter. Previous studies have reported that T-score may have limitations (see [8], for example), but we are not aware of other representative metrics that have been extensively used to evaluate those techniques.

V. Related Work

Comparing program executions of a faulty program over passed test cases and failed test cases is a frequently used fault localization strategy. Related fault localization techniques can be classified into different groups according to their working levels. *Tarantula* [15], *CBI* [18], and *SOBER* [19] have been discussed in Section I. In the sequel, we review other techniques.

There are statement-level techniques, which contrast program execution spectra of statements in passed and failed execution, and employ heuristics to estimate the extent that a statement is related to faults. Naish et al. [20] give a summary to this kind of technique. *CBI* [18] and *SOBER* [19] find suspicious predicates. Zhang et al. [29] propose to use non-parametric hypothesis testing to improve such predicate-based techniques. *DES-CBI* [30] and *DES-SOBER* [30] work on a finer-level program entity, namely conditions, to locate faults. *CP* [28] is another technique that captures the propagation of infected program states among basic blocks in order to locate faults. In Sections I and II, we have introduced these techniques.

Delta Debugging [8] isolates failed inputs, produces cause effect chains, and locates the root causes of failures. It considers a program execution (of a failed test case) as a sequence of program states, each inducing the next one, until the failure is reached. *Predicate switching* [27] is another technique to locate a fault by checking the execution state. It switches a predicate's decision at execution time to alter the original control flow of a failed execution, aiming at producing a correct output. If correct output is found, the technique then further searches from the switched predicate to locate a fault through backward or forward slicing (or both). *Delta Debugging* uses additional data-flow information rather than the above coverage-based techniques. Consequently, we do not compare with it in this paper.

TABLE II. Statistics Of Overall Effectiveness

	Tarantula	***Jaccard***	***Ochiai***	***SBI***	***CBI***	***SOBER***	***DES-CBI***	***DES-SOBER***	***CP***	**BlockRank**
5%	30%	30%	30%	30%	12%	1%	7%	5%	**39%**	**39%**
10%	40%	41%	41%	41%	23%	10%	16%	10%	47%	**49%**
15%	55%	55%	55%	55%	27%	16%	17%	16%	58%	**59%**
20%	66%	66%	66%	66%	29%	24%	27%	22%	65%	**67%**
mean	20%	20%	20%	20%	41%	43%	43%	43%	18%	**17%**
stdev	24%	24%	24%	24%	28%	24%	27%	24%	21%	**20%**

Le and Soffa [17] make use of path profile information to investigate the correlation of multiple faults in a faulty program. Jones et al. [16] further use *Tarantula* to explore how to cluster test cases to debug a faulty program in parallel. Baudry et al. [5] observe that some groups of statements (collectively known as dynamic basic blocks) are always executed by the same set of test cases. They use a bacteriologic approach to find a subset of the test set to maximize the number of dynamic basic blocks to further optimize *Tarantula*. Liblit et al. [18] further adapt *CBI* to handle compound Boolean expressions, and show that these expressions exhibit complex behaviors from the statistical fault localization perspective. Zhang et al. [30] study the issues on the granularity of program entity. They observe that the short-circuit rule in a program may significantly affect the effectiveness of fault localization techniques. The experimental result in Section IV appears to confirm this observation for our subject programs. Nonetheless, more statistical analysis on the data is required.

Clause and Orso [7] propose a technique to support debugging of field failures. It can help recording, reproducing and minimizing failing executions and is useful for developers to debug field failures in house. Csallner and Smaragdakis [9] present a hybrid analysis tool for bug finding. Using dynamic analysis, static analysis and automatic test cases generation based on the analysis results to locate faults; their paper reports a promising result. It will be interesting to know how our technique can be integrated with their analysis tool.

Gupta et al. [12] propose to narrow down slices using a forward dynamic slicing approach. Zhang et al. [27] integrate the forward and the standard dynamic slicing approaches for debugging. Since these techniques are based on slicing, we do not compare them with the coverage-based techniques in this paper. Wong et al. [24] propose heuristics to synthesize a code-coverage-based method for use in fault localization. The impact of such heuristics to our technique is interesting but is beyond the scope of the present paper.

Chen et al. [6] apply PageRank to construct explicit links to connect suspicious program entities. Their approach cannot handle those faults that are not obviously suspicious, while the present paper aims to find such kind of fault effectively. To address coincidental correctness, Wang et al. [23] develop fault patterns for identifying suspicious execution fragments for fault localization.

VI. Conclusion

In this paper, we have proposed BlockRank, a new statistical fault localization technique to address the issues of coincidental correctness and execution crash. Its main idea is to use the popularity of incoming basic blocks as the basis to recursively correlate observed failures to the incoming basic block, and handle severe errors that crash the program. An empirical study has shown that the technique is more effective than existing techniques.

Future work includes the investigation of other versatile program execution information (such as path profile) for link analysis to support fault localization, and the extension of our model to concurrent programs.

References

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of the Testing: Academic and Industrial Conference: Practice And Research Techniques* (*TAICPART-MUTATION 2007*), pages 89–98. IEEE Computer Society, Los Alamitos, CA, 2007.

[2] P. Arumuga Nainar, T. Chen, J. Rosin, and B. Liblit. Statistical debugging using compound Boolean predicates. In *Proceedings of the 2007 ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 2007*), pages 5–15. ACM, New York, NY, 2007.

[3] T. Ball and S. Horwitz. Slicing programs with arbitrary control-flow. In *Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging* (*AADEBUG 1993*), volume 749 of Lecture Notes in Computer Science, pages 206–222. Springer, London, UK, 1993.

[4] T. Ball, P. Mataga, and M. Sagiv. Edge profiling versus path profiling: the showdown. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (*POPL 1998*), pages 134–148. ACM, New York, NY, 1998.

[5] B. Baudry, F. Fleurey, and Y. Le Traon. Improving test suites for efficient fault localization. In *Proceedings of the 28th International Conference on Software Engineering* (*ICSE 2006*), pages 82–91. ACM, New York, NY, 2006.

[6] I.-X. Chen, C.-Z. Yang, T.-K. Lu, and H. Jaygarl. Implicit social network model for predicting and tracking the location of faults. In *Proceedings of the 32nd Annual International Computer Software and Applications Conference* (*COMPSAC 2008*), pages 136–143. IEEE Computer Society, Los Alamitos, CA, 2008.

[7] J. Clause and A. Orso. A technique for enabling and supporting debugging of field failures. In *Proceedings of the 29th International Conference on Software Engineering* (*ICSE 2007*), pages 261–270. IEEE Computer Society, Los Alamitos, CA, 2007.

[8] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering* (*ICSE 2005*), pages 342–351. ACM, New York, NY, 2005.

[9] C. Csallner and Y. Smaragdakis. DSD-crasher: a hybrid analysis tool for bug finding. In *Proceedings of the 2006 ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 2006*), pages 245–254. ACM, New York, NY, 2006.

[10] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. *Empirical Software Engineering*, 10 (4): 405–435, 2005.

[11] S. G. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Control*, 12 (3): 185–210, 2004.

[12] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering* (*ASE 2005*), pages 263–272. ACM, New York, NY, 2005.

[13] R. M. Hierons. Avoiding coincidental correctness in boundary value analysis. *ACM Transactions on Software Engineering and Methodology*, 15 (3): 227–241, 2006.

[14] D. Jeffrey, N. Gupta, and R. Gupta. Fault localization using value replacement. In *Proceedings of the 2008 ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 2008*), pages 167–178. ACM, New York, NY, 2008.

[15] J. A. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering* (*ASE 2005*), pages 273–282. ACM, New York, NY, 2005.

[16] J. A. Jones, M. J. Harrold, and J. F. Bowring. Debugging in parallel. In *Proceedings of the 2007 ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 2007*), pages 16–26. ACM, New York, NY, 2007.

[17] W. Le and M. L. Soffa. Path-based fault correlations. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*SIGSOFT 2010/FSE-18*), pages 307–316. ACM, New York, NY, 2010.

[18] 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* (*PLDI 2005*), pages 15–26. ACM, New York, NY, 2005.

[19] C. Liu, L. Fei, X. Yan, S. P. Midkiff, and J. Han. Statistical debugging: a hypothesis testing-based approach. *IEEE Transactions on Software Engineering*, 32 (10): 831–848, 2006.

[20] L. Naish, H. J. Lee, and K. Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on Software Engineering and Methodology*, to appear. Available at http://www.cs.mu.oz.au /~lee/papers/model/paper.pdf.

[21] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: bringing order to the web. Stanford InfoLab Publication 1999-66. Stanford University, Palo Alto, CA, 1999. Available at http://dbpubs.stanford.edu/pub/1999-66.

[22] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering* (*ASE 2003*), pages 30–39. IEEE Computer Society, Los Alamitos, CA, 2003.

[23] X. Wang, S. C. Cheung, W. K. Chan, and Z. Zhang. Taming coincidental correctness: coverage refinement with context patterns to improve fault localization. In *Proceedings of the 31st International Conference on Software Engineering* (*ICSE 2009*), pages 45–55. IEEE Computer Society, Los Alamitos, CA, 2009.

[24] W. E. Wong, V. Debroy, and B. Choi. A family of code coverage-based heuristics for effective fault localization. *Journal of Systems and Software*, 83 (2): 188–208, 2010.

[25] W. E. Wong, Y. Qi, L. Zhao, and K.-Y. Cai. Effective fault localization using code coverage. In *Proceedings of the 31st Annual International Computer Software and Applications Conference* (*COMPSAC 2007*), volume 1, pages 449–456. IEEE Computer Society, Los Alamitos, CA, 2007.

[26] Y. Yu, J. A. Jones, and M. J. Harrold. An empirical study of the effects of test-suite reduction on fault localization. In *Proceedings of the 30th International Conference on Software Engineering* (*ICSE 2008*), pages 201–210. ACM, New York, NY, 2008.

[27] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *Proceedings of the 28th International Conference on Software Engineering* (*ICSE 2006*), pages 272–281. ACM, New York, NY, 2006.

[28] 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 and the ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*ESEC 2009/FSE-17*), pages 43–52. ACM, New York, NY, 2009.

[29] Z. Zhang, W. K. Chan, T. H. Tse, Y. T. Yu, and P. Hu. Non-parametric statistical fault localization. *Journal of Systems and Software*, 84 (6): 885–905, 2011.

[30] Z. Zhang, B. Jiang, W. K. Chan, and T. H. Tse. Debugging through evaluation sequences: a controlled experimental study. In *Proceedings of the 32nd Annual International Computer Software and Applications Conference* (*COMPSAC 2008*), pages 128–135. IEEE Computer Society, Los Alamitos, CA, 2008.