Hindawi
Mathematical Problems in Engineering
Volume 2021, Article ID 7038979, 17 pages
https://doi.org/10.1155/2021/7038979

Research Article

A Multiple-Fault Localization Method for Embedded Software with Applications in Engineering

Lu Kong, JinBo Wang, Shan Zhou, and MengRu Wang

Technology and Engineering Center for Space Utilization, Chinese Academy of Sciences, Beijing, China

Correspondence should be addressed to Lu Kong; konglu@csu.ac.cn

Received 6 July 2020; Revised 2 February 2021; Accepted 5 February 2021; Published 24 February 2021

Academic Editor: Francisco Chicano

Embedded software is increasingly being used with high reliability. However, the fault localization of embedded software is still largely dependent on the experience of engineers. Besides, faults in embedded software programs are not independent individuals; they are related to each other and affect each other, which may lead to more complex interaction behavior. These uncertainties render the traditional methods for single-fault localization with limited practical value. This paper has proposed a multiple-fault localization method to be applied to the embedded software, with emphasis on the cache-based program spectra-acquiring method and the hybrid clustering-based fault partition method. Through case studies on 108 groups of the subject program, it has been proved that the hybrid clustering-based fault partition method has significantly improved the effectiveness of multiple-fault localization in comparison with the traditional fault localization methods. Experiments on three embedded software programs in engineering have revealed that the cache-based program spectra-acquiring method saves nearly half of the running-time cost compared with the traditional spectrum-acquiring method based on real-time transmission. Therefore, the multiple-fault localization method proposed in this paper can be applied in embedded software debugging and testing in engineering.

1. Introduction

With the rapid development of the modern society, computer software has been integrated into all walks of human life, including embedded systems, operating systems, databases, games, etc. Among them, the embedded system is widely used in key fields, such as daily electronics, home applications, vehicle equipment, transportation systems, military equipment, and aerospace applications, due to its small size, high reliability, flexibility, and convenience [1]. The main functions in the embedded system designs are customized through the embedded software. With the strong support of hardware technology, embedded software is constantly developing toward complexity, openness, and large-scale. Compared with nonembedded software, embedded software has such outstanding characteristics as strong real-time performance, tight hardware coupling, and complicated interaction environment, and is more prone to errors during the development process. This is why stricter requirements are placed on the reliability of embedded software [2]. Once the undetected faults or potential faults occur, the embedded system can no longer perform the right functions, resulting in data loss, system crash, and even threat to the safety of human life [3–6].

Like most software, faults in embedded software come from the code programming. They can be divided into several categories, such as memory-related faults, initialization faults, calculation faults, input and output faults, control flow faults, data processing faults, etc. Among the kinds of faults, some tools can detect and locate memory-related faults and initialization faults, which are easier to avoid and solve. However, it is arduous to detect such failures as control flow faults, calculation faults, input and output faults, and data processing faults with tools due to the complexity of logical semantics.

In the past decades, researchers have made valuable achievements to explore fault localization methods. The general idea of fault localization is analyzing the program statement and its execution result; therefore, the testing data such as execution information is an intensely important

basis during fault localization. In embedded software, due to the expansion caused by code instrumentation, the memory space usage may be tighter, and the real-time performance is greatly affected. Besides, the embedded software runs independently of the development environment, making it more arduous to get the coverage data compared with the general software. In the traditional embedded software development, developers mostly debugged by assertions [7], setting breakpoints [8], and program logging [9]. However, these debugging methods need a high level of understanding of the logic, structure, and function of the software, making it time-consuming and with low efficiency. Besides, the effect of fault localization also depends on the mastery of prior knowledge, assumptions, debugging experience, and the construction of the test case set. Therefore, a more effective method is needed to solve the problem of fault localization in embedded software.

In this paper, we have put forward a fault localization method applied to embedded software, and this paper contributes to the following.

First, the negative impacts of multiple-faults in the software are analyzed. As is known, faults in programs are often not independent. They are related to each other and affect each other, which may lead to more complex behavior. For instance, faults in subroutines may infect callers through subroutine calls and may infect more through communications between processes. There are many mature methods in the existing literature to solve the problem of single-fault localization, but through the empirical analysis in Section 3, they are found to be not fully applicable to multi-fault software.

Second, a fault localization method for embedded software is proposed. Different from the fault localization methods for general software, the particularity of the embedded software application program is satisfied through the following: (1) An embedded software spectrum-acquiring method based on data caching is adopted to consume minimal resources and minimize the interference to the software itself. (2) A clustering combination based fault localization method is designed to improve the traditional fault localization method to be suitable for multiple fault localization.

Third, engineering practice has been carried out in embedded software with the proposed method. In most literature, only typical programs in the SIR (Software Infrastructure Repository) [10] are employed for experiment evaluation. However, the code size of these programs is relatively small, and the correlation between faults is relatively single; therefore, the verification of the fault localization method is not sufficient enough. This paper uses both open source software and real engineering software to verify the proposed methods, confirming the applicability and effectiveness of the method in practical applications.

The remaining part of the paper proceeds as follows. Section 2 gives the background of the software fault localization methods. Section 3 discusses some negative effects of traditional fault localization methods when locating the programs with multiple faults. Section 4 presents the multiple-fault localization method based on the clustering combination. Some empirical studies are carried out on both open-source software and real engineering software in Section 5. Finally, we also discuss the threats to validity and consider our future work.

2. Related Work

Researchers have proposed many fault localization methods that have excellent performance. Most of them come from different fields of computer science, such as neural networks, graphics theory, artificial intelligence, information theory, and automation theory. Wong et al. has classified fault localization methods into eight categories, including slice-based, spectrum-based, statistics-based, program-state-based, machine-learning-based, data-mining-based, model-based, and miscellaneous methods [11–13]. Among them, the program-spectrum-based fault localization (SFL) [14] is an effective method for software fault localization and has been widely studied and applied in engineering. A program spectrum is defined as the execution information about a program from certain perspectives, such as the execution information for conditional branches or loop-free intraprocedural paths [15]. Code coverage [16], or Executable Statement Hit Spectrum (ESHS) [17], is used to present the program entity, which has been covered during the testing. Using this information, the program entity related to the failure is easily identified, thus narrowing the searching scope for the fault code. Among most SFL methods, Tarantula, Jaccard, and Ochiai methods have achieved outstanding results, and they use a statistical method to calculate the suspiciousness score of the program entities and rank them in sequence [18, 19].

$$\text{Tarantula}(s) = \frac{\left(N_{ef}(s)/\left(N_{ef}(s) + N_{nf}(s)\right)\right)}{\left(N_{ef}(s)/\left(N_{ef}(s) + N_{nf}(s)\right)\right) + \left(N_{ep}(s)/\left(N_{ep}(s) + N_{np}(s)\right)\right)}, \tag{1}$$

$$\text{Ochiai}(s) = \frac{N_{ef}(s)}{\sqrt{N_f(s) * \left(N_{ef}(s) + N_{ep}(s)\right)}}, \tag{2}$$

$$\text{Jaccard}(s) = \frac{N_{ef}(s)}{N_f(s) + N_{ep}(s)}. \quad (3)$$

The term $N_f(s)$ represents the total number of the failed test cases. The term $N_{np}(s)$ represents the number of times when the statement S is not covered and the test case passes. The term $N_{ep}(s)$ represents the number of times when the statement S is covered and the test case passes. The term $N_{nf}(s)$ represents the number of times when the statement S is not covered and the test case fails. The term $N_{ef}(s)$ represents the number of times when the statement S is covered and the test case fails. The value range of equation (1) is between 0 (the lowest suspicion value) and 1 (the highest suspicion value). Program entities can be sorted according to the suspicion value in descending order and debugged until the fault is located.

The number of faults contained in the tested program is arduous to know in advance, and mostly there is more than one; therefore, a growing number of research studies are focusing on exploring effective multiple-fault localization methods in recent years [18, 20–26]. Jones et al. divide the program into several parts based on the execution of the testing cases and then assign different developers to locate the faults in parallel [27]. Abreu et al. proposed the BARINEL method, which uses the Bayesian model to sort candidate sets representing multiple-faults [28]. This method has a good performance both in single-fault and multiple-fault localizations. However, it needs developers to keep real-time interaction in code debugging to ensure that candidate set sorting can be modified continuously. Steimann et al. tried to use probability distribution to estimate the number of internal defects [29] and found that using an integer linear programming algorithm can significantly improve the parallelism of fault localization [30]. Ruizhi and Wong proposed an advanced fault localization method Mseer for multiple bugs in parallel, based on revised Kendall tau and K-medoids clustering methods. The Mseer proved to have more efficiency and accuracy compared to the other two methods by experimental results [31].

However, neither single-fault localization methods nor multiple-fault localization methods in literature have practical applications due to the higher performance costs or lower efficiency. Besides, because of the difference in research focus and experimental subjects, it is arduous to compare the above methods using a uniform evaluation.

3. Negative Impacts of Multiple-Faults in Software

The relationship between the software code and the fault is intensely complex. When a software program is divided into several modules, the relationships exist not only among modules but also among program slices or statements within a module. From the perspective of control flow and data flow analysis, the root of the associated faults is that the current state of the software is affected by the previous state. Although most of the software programs are developed based on high cohesion and low coupling, they still cannot achieve complete independence of modules. This is especially true in object-oriented software development [32]. Inheritance determines that there will be an inheritance or derivation among classes, which leads to the same inheritance and transitivity of faults. Faults at the bottom modules will be passed to the upper modules through interface calls, etc., affecting other related objects or modules. Therefore, the environment of software with multiple-faults is more complex, leading to some unexpected situations in traditional spectrum-based fault localization methods.

In the following section, the negative impacts the traditional spectrum-based fault localization methods have on software with multiple-faults will be discussed according to the empirical analysis.

3.1. The Sample Program with Multiple-Faults. The sample program with multiple-faults [21] to illustrate the negative impacts the traditional spectrum-based fault localization methods have on software with multiple-faults is demonstrated in Figure 1, which has two faults on s7 and s20. There are ten test cases from t1 to t10 in the test suite. The execution trace of each test case is represented by the black dots. The testing results are given at the bottom of each test case, and the term F or P points out whether the test case is failed or passed executed. The suspiciousness score calculated by the Tarantula method is listed in the last column of the table.

3.2. Inspiring Our Work. The process of locating the first bug is as follows. As statements s14, s16, s18, s21, and s24 have the highest suspiciousness score, they are examined first in a logical order. But, there is no bug existing in any statement. Similarly, statements s11 and s13 have the second-highest suspiciousness score and they are examined next, but there is no bug existing either. Next, statements s1 to s6 and s15 are examined after s11 and s13. Finally, when statements s7 and s8 are examined, a bug is found in statement s7.

As seen from the above localization process, 15 nonfaulty statements were examined before locating the first fault, proving that the fault localization efficiency will be greatly reduced when using the traditional Tarantula method to solve the multiple-faults problems.

Besides, it is amazing that the nonfaulty code s24 has a higher suspiciousness score than the faulty code s20. According to the code analysis, the fault on s20 propagates to s24 along with the control flow of the true branch. However, the Tarantula method ignores the fault propagation with the control flow and the data flow among program blocks, making the suspiciousness score of s24 higher than that of s20.

We have also found that the suspiciousness score of s1 to s4 is higher than the s7 statement because they are executed in each test case. It can be inferred that in programs with

		Testcase										
		t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	
		(0,0,0)	(−2,−2,0)	(−8,−8,0)	(2,1,3)	(−1,−1,0)	(4,2,9)	(2,2,1)	(−5,−5,0)	(3,1,5)	(−4,−4,0)	The suspicious value
string buf[0*x*0f];	s1	●	●	●	●	●	●	●	●	●	●	0.5
read ("Input 3 numbers:", *x, y, z*);	s2	●	●	●	●	●	●	●	●	●	●	0.5
incove mid (*x, y, z,* buf);	s3	●	●	●	●	●	●	●	●	●	●	0.5
write (buf);	s4	●	●	●	●	●	●	●	●	●	●	0.5
function mid (int *x*, int *y*, int *z*, string msg) {												
msg="mid: *z*";	s5	●	●	●	●	●	●	●	●	●	●	0.5
if (*y*<*z*)	s6	●	●	●	●	●	●	●	●	●	●	0.5
if (*x*>=*y*) // bug1:x<y	s7		●	●	●	●	●		●	●	●	0.47
msg = "mid : *y*";	s8		●	●	●	●	●		●	●	●	0.47
else if (*x*<*z*)	s9											
msg = "mid : *x*";	s10											
else if (*x*>*y*)	s11	●						●				0.6
msg = "mid : *y*";	s12											
else if (*x*>*z*)	s13	●						●				0.6
msg = "mid : *x*";	s14							●				1
if (*x*!=*y*)	s15	●	●	●	●	●	●	●	●	●	●	0.5
if (*y*==*z*)	s16				●		●			●		1
msg = "mid : *y* or *z*";	s17											
else if (x==z)	s18				●		●			●		1
msg = "mid : *x* or *z*";	s19											
else if (*y*+=*z*) //bug2:*y*==*z*	s20	●	●	●		●		●	●		●	0.2
if (*x*==0)	s21							●				1
msg = "warning:uninitalized";	s22											
else	s23											
msg = "mid : *x, y* or *z*";	s24							●				1
else	s25											
msg = "mid: *x* or *y*";	s26	●	●	●		●			●		●	0
}												
		P	P	P	F	P	F	F	P	F	P	

Figure 1: A motivating example: multiple-fault localization process using the Tarantula method.

multiple-faults, the value of shared program entities such as the program entry will be greater than that of the faulty program entity, which makes the fault localization effect worse. We can also get the same conclusion through the equation of Tarantula. In a program with multiple-faults, the $N_{ef}(s)$ value of nonfaulty code may be greater than that of the faulty code, which reduces the suspiciousness score of the faulty code and makes the accuracy of fault localization worse.

Furthermore, we have observed that the faulty code s7 has been executed by test cases t2, t3, t5, t8, and t10, but all of the execution results do not fail. Thus, the suspiciousness score of the faulty code s7 is ranked in the third last place by the Tarantula method. Test cases t2, t3, t5, t8, and t10 are named coincidental correctness test cases [33]. According to the equation of the Tarantula method, there is an inverse relationship between $N_{ep}(s)$ and the final result of the equation. If there are a large number of coincidental correctness test cases, the value $N_{ep}(s)$ of the faulty code will increase while$N_{np}(s)$, $N_{nf}(s)$, and $N_{ef}(s)$ remain unchanged. In this condition, the denominator value of the equation increases, and the suspiciousness score of faulty code is reduced, affecting the ranking of the faulty code. It is inferred that with a greater increase of coincidental correctness test cases in software with multiple-faults, the traditional Tarantula method suffers a larger impact.

Overall, it is well supported from the fault localization process of the sample program that the traditional spectrum-based fault localization method is not fully applicable in software with multiple-faults, which is consistent with the research conclusions in [34]. Therefore, a more effective fault

localization method is required to solve the problem in software with multiple-faults.

4. The Proposed Approach

In this section, we propose a multiple-fault localization method applied to the embedded software. Similar to most studies, the proposed approach also rests on the following assumptions raised by [34]:

(i) The faulty code can be covered by both failing and passing test cases

(ii) At least one bug can be triggered by each failed test case, which leads to the fault

(iii) The prior probability distribution of faultiness is unknown

(iv) Developers can accurately judge whether the suspect statement is a defective statement during code review, and then effectively remove the defect

The framework of the proposed method is displayed in Figure 2, which can be divided into four phases: (1) cache-based spectrum acquiring; (2) spectrum matrix constructing and preprocessing; (3) hybrid clustering-based fault partition; (4) faults locating.

4.1. Cache-Based Spectrum Acquiring. Suppose the embedded software program $P = (s_1, s_2, \ldots, s_n)$ contains n program entities and $P_i\,(1 \le i \le n)$ refers to the $i\,\text{th}\,(1 \le i \le n)$ program entity of the program P. The test suite $T = (t_1, t_2, \ldots, t_m)$ corresponding to program P contains m test cases and $t_j\,(1 \le j \le m)$ is the $j\,\text{th}\,(1 \le j \le m)$ test case in test suite T.

Harrold et al. generalize the spectrum and propose various types of spectrum, such as the Complete-Path Spectra (CPS), Path-Count Spectra (PCS), and Branch-Count Spectra (BCS) [15], which is acquired by implanting probe functions at appropriate locations of the software program under test. The CPS spectra and the PCS spectra have played an important role in helping developers analyze information about the execution of the program and localize faults in general software programs. However, the memory resources in the embedded software are extremely limited; it would not be cost-effective to collect the traces required for the CPS spectra and the PCS spectra. The traditional way of instrumentation used in general software would inevitably bring a certain amount of code expansion and greatly affect the function and performance of the embedded software itself. Besides, quite a few embedded software has no extra output channels, making it arduous to transmit the spectrum data in real-time during running. To solve the above problem, we propose an embedded software spectrum-acquiring method based on data caching with the following steps:

Step 1 Add the start and end braces of each logic block, and generate the correspondence of line numbers before and after the above execution.

Step 2 Count the number of statements of the program and establish a statement array to record the number of times each statement is executed. Initialize the corresponding elements in the info array according to the analysis of the program.

Step 3 Perform lexical analysis of the program, and implant the instrumentation function.

Step 4 Run test cases on the program after instrumentation, and update the number of times the corresponding instance is executed in the array according to the result of the instrumentation function.

Step 5 Obtain the spectrum data of the original program before instrumentation according to the data in the array.

Step 6 After executing each test case, transmit the spectrum data to the host computer using the idle output channels.

The architecture of the embedded software program spectrum acquisition is demonstrated in Figure 3. The embedded software after instrumentation runs on the target board. During the execution of the test cases, the target board puts the instrumentation information to the message queue in real-time, and then sends the information to the host computer at the appropriate time.

4.2. Spectrum Matrix Constructing and Preprocessing. The instrumentation of program P is implemented and the executable file is generated after compilation. The test suite T is then loaded and executed, and the execution data of the program has to construct the program spectrum matrix demonstrated in Figure 4.

Matrix M is used to represent the coverage information of test suite T, where $M_{ij}\,(M_{ij} \ge 0)$represents the number of times the $i\,\text{th}\,(1 \le i \le n)$ program entity in program P is executed by the $j\,\text{th}\,(1 \le j \le m)$ test case. The testing results of the program P are represented by the matrix RE, where $\text{RE}_j = 1$ indicates that the testing result of the $j\,\text{th}\,(1 \le j \le m)$ test case in T is passed, and $\text{RE}_j = 0$ indicates that the execution result of the $j\,\text{th}\,(1 \le j \le m)$ test case in T is failed. $N_f(s)$, $N_{ep}(s)$, $N_{np}(s)$, $N_{nf}(s)$, and $N_{ef}(s)$ have the same meanings as illustrated in equations (1)–(3).

However, the values in matrix M of program P vary due to the influence of the test suite and the fault codes. To eliminate the influence on the accuracy of data processing introduced by the magnitude difference of data values, it is indispensable to carry out the data standardization to keep the values in uniform measures. The Z-score standardized method defined in (4) is used. The data in matrix M are converted into scores without units. In Equation (4), the data x_i equals $N_{ef}(s)$, μ points out the average value of the code coverage data on $S_i\,(1 \le i \le n)$ by the failed test case and σ points out the standard deviation of the code coverage data.

$$x^* = \frac{x_i - \mu}{\sigma}. \tag{4}$$

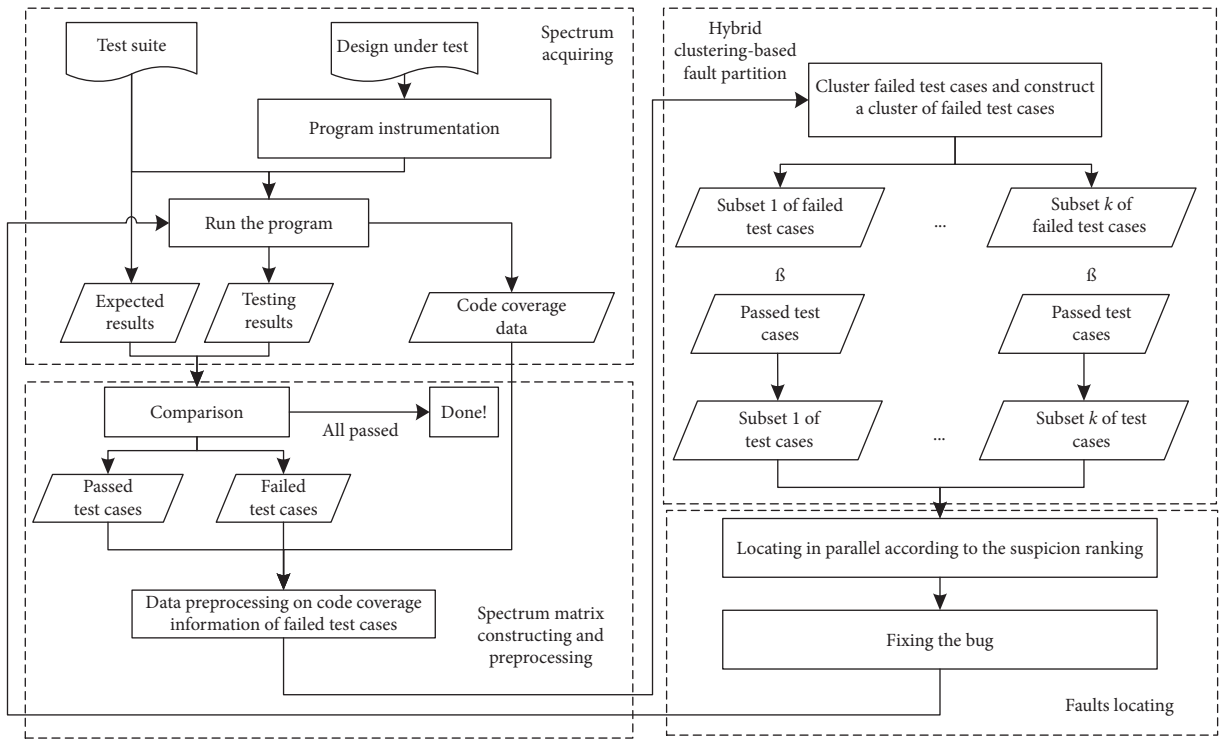

Figure 2: The framework of the proposed approach.

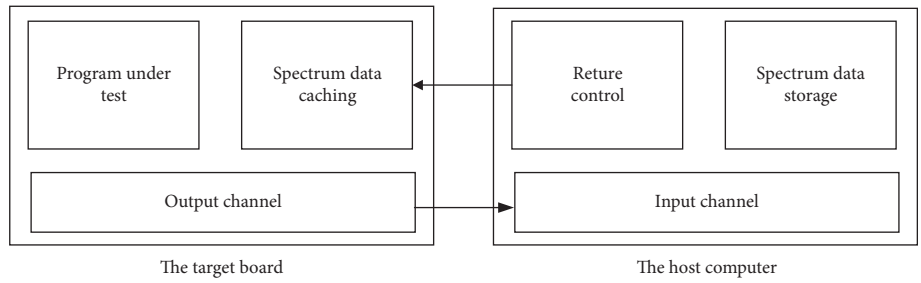

Figure 3: System architecture of the embedded software program spectrum acquisition.

$$T: \quad (t_1 \quad t_2 \quad t_3 \quad \dots \quad t_m)$$

$$P: \begin{pmatrix} s_1 \\ s_2 \\ s_3 \\ \dots \\ s_n \end{pmatrix} \quad M: \begin{pmatrix} M_{11} & M_{12} & M_{13} & \dots & M_{1m} \\ M_{21} & M_{22} & M_{23} & \dots & M_{2m} \\ M_{31} & M_{32} & M_{33} & \dots & M_{3m} \\ \dots & \dots & \dots & \dots & \dots \\ M_{n1} & M_{n2} & M_{n3} & \dots & M_{nm} \end{pmatrix} \quad N: \begin{pmatrix} N_{ep}(s_1) & N_{np}(s_1) & N_{nf}(s_1) & N_{ep}(s_1) \\ N_{ep}(s_2) & N_{np}(s_2) & N_{nf}(s_2) & N_{ep}(s_2) \\ N_{ep}(s_3) & N_{np}(s_3) & N_{nf}(s_3) & N_{ep}(s_3) \\ \dots & \dots & \dots & \dots \\ N_{ep}(s_n) & N_{np}(s_n) & N_{nf}(s_n) & N_{ep}(s_n) \end{pmatrix}$$

$$RE: \quad (RE_1 \quad RE_2 \quad RE_3 \quad \dots \quad RE_m)$$

Figure 4: The proposed spectrum matrix.

We present an example of a program spectrum, explaining the method of data standardization. Assuming the coverage data of the failed test case t_x is $t_x = (x_1, x_2, \dots, x_n)$, where x_i represents the number of times the statement S_i $(1 \le i \le n)$ is covered by the test case t_x. The spectrum of the first failed test case t1 is as given in Table 1.

The number in the first column represents the test case number. The number in the first line represents the statement line number. The number at the intersection of the line

Table 1: The program spectrum before standardization.

Test case	s1	s2	s3	s4	s5	s6	s7	s8	s9	s10	s11	s12	s13	s14	s15
t1	1	1	1	1	1	10	9	0	0	0	0	4	1	0	1

and column represents the number of times the statement was executed by the test case. As seen from the example, the number of times the statement was executed by the test case varies greatly. As an example, statement s6 was covered by test case t1 10 times, while statement s7 was covered by test case t1 9 times. After the calculation, the average value μ of the coverage data on $S_i\,(1 \le i \le 15)$ by the failed test case t1 is 2, and the standard deviation of the coverage data σ is 3.2071. Due to equation 4, the program spectrum data calculated after standardization is demonstrated in Table 2.

4.3. Hybrid Clustering-Based Fault Partition. Due to the previous research, the execution paths of the failed test cases have high similarity [35]. Failed test cases can be partitioned based on the similarity of execution paths, converting the multiple-fault localization into multiple single fault localization processes. Data clustering, which aims to group objects into subsets that have the meaning of the context, is an effective method to deal with the problems of multiple-faults [36]. The *K*-means method is one of the simple and commonly used clustering methods that group the given dataset into k clusters. The benefit of this method is simple and fast, which is relatively scalable and efficient for processing large datasets. Suppose the term N represents the number of objects in datasets, and the term K represents the number of clusters. The *K*-means method often ends with a local optimum when $K << N$ in most datasets. Therefore, the clustering effect is remarkable when the difference among the clusters is obvious. However, the traditional *K*-means method is sensitive to the first value, and the selection of the first clustering center has a great influence on the clustering results.

In this work, a hybrid clustering-based fault localization (HCFL) method is employed to reduce the influence of the traditional *K*-means method and improve the clustering efficiency and accuracy of the traditional *K*-means method. The HCFL method improves the K-means method in the selection of the first clustering center by incorporating the distance-based clustering methods and density-based clustering methods [37, 38]. The HCFL method is resolved in two stages. The first k cluster centers are decided in Step 1 to Step 6 and the traditional *K*-means method is executed based on the first k cluster centers in Step 7 to Step 9.

Input: 1. The failed test case set $T_a = (a_1, a_2, \ldots, a_n)$ and $T_b = (b_1, b_2, \ldots, b_n)$, where a_n and b_n represent the poststandardization code coverage data of the same program entity $S_i\,(1 \le i \le n)$ under the two test case sets, respectively. T_a and T_b, respectively; Input 2. The number of k clusters.

Output: k clusters of failed test cases.

Step 1 Calculate distances between any data on the same program entity $S_i\,(1 \le i \le n)$ in the set T_a and T_b:

$$d(T_a, T_b) = \sqrt{\sum_{i=1}^{n} (a_i - b_i)^2}. \tag{5}$$

Step 2 Calculate the average distance $\mathrm{AVE}\,d(T_a, T_b)$ between data objects in the set T_a and T_b.

$$\mathrm{AVE}\,d(T_a, T_b) = \frac{\sum d(T_a, T_b)}{C_n^2}. \tag{6}$$

The term C_n^2 is the number of couples of failed test cases in the set T_a and T_b.

Step 3 Suppose the distance between T_a and T_b is $\mathrm{withinAVE}\,d(T_a, T_b)$, then T_b is considered as a neighboring point of T_a. Calculate the set of all neighboring points T_a.

$$\mathrm{Den}(T_a) = \sum_{b=1}^{n} F\left(\mathrm{AVE}\,d(T_a, T_b) - d(T_a, T_b)\right). \tag{7}$$

The term $F(z)$ is a function according to

$$f(z) = \begin{cases} 1, & z < 0, \\ 0, & z \ge 0, \end{cases} \tag{8}$$

Step 4 Count and arrange the number of neighboring points of all failed test cases, and select the one with the largest number of neighboring points as the first clustering center. Add the first clustering center TC1 to collection TC, and delete it from TR.

Step 5 Select the test case with the furthest distance from TC as the second clustering center TC2. Add it to collection TC, and delete it from TR.

Step 6 Select the test case with the furthest distance from both TC1 and TC2 as the third clustering center. Repeat Step 6 until the first k clustering centers are contained in the collection TC.

$$\mathrm{TC}_k = \max\left(\min\left(d(\mathrm{TC}_1, \mathrm{Tr}), d(\mathrm{TC}_2, \mathrm{Tr}), \ldots, d(\mathrm{TC}_{k-1}, \mathrm{Tr})\right)\right), \quad (\mathrm{Tr} \in \mathrm{TR}). \tag{9}$$

Table 2: The program spectrum after standardization.

Test case	s1	s2	s3	s4	s5	s6	s7	s8	s9	s10	s11	s12	s13	s14	s15
t1	−0.32	−0.32	−0.32	−0.32	−0.32	2.49	2.18	−0.62	−0.62	−0.62	−0.62	0.62	−0.31	−0.62	−0.31

Step 7 Calculate the distances from the remaining test cases to the first clustering centers, and assign the remaining n-k test cases to the nearest cluster.

Step 8 Recalculate the clustering centers of each cluster.

Step 9 Repeat Step 7 and Step 8 until the Sum Squared Error (SEE) value of all clusters are unchanged.

$$\mathrm{SEE} = \sum_{i=1}^{k} \sum_{j=1}^{m} \left[\left(x_{1ij} - \overline{x}_{1i}\right)^2 + \left(x_{2ij} - \overline{x}_{2i}\right)^2 + \cdots + \left(x_{nij} - \overline{x}_{ni}\right)^2 \right]. \tag{10}$$

The terms $x_{1ij}, x_{2ij}, \ldots, x_{nij}$ represent the coverage data of the $j\,\mathrm{th}\,(1 \le j \le m)$ failed test cases in the $i\,\mathrm{th}\,(1 \le i \le n)$ cluster for program P. $\overline{x}_{1i}, \overline{x}_{2i}, \ldots, \overline{x}_{ni}$ represent the center of the $i\,\mathrm{th}\,(1 \le i \le n)$ cluster.

We present a running example of the hybrid clustering method. Suppose Table 3 is a spectrum of a sample program.

According to Step 1, the distance between every two test cases is calculated using equation (5), which is listed in Table 4. The cross-point of the table is the distance between the two test cases in the row and the column. As an example, the distance between t5 and t2 is 2.828, and the distance between t8 and t2 is 2.646. Based on the distances between each couple of test cases, the average distance described in Step 2 can be calculated using equation (6), which is 2.742450.

Then in Step 3, the density of each test case is calculated according to equation (7). As an example, for test case t1, the test cases where the distance from t1 is less than the average distance are t3, t4, and t5, that is, the density value of t1 is 3. The density values of all test cases are listed in Table 5.

According to Step 4, the test case t3 is selected as the first clustering center TC1 because of the highest density. And, the test case t5 is selected as the second clustering center TC2 because it has the largest distance (3.000) from the test case t3.

In Step 5 and Step 6, calculate the distance between the first clustering center t3 and the remaining test case, and the distances between the second clustering center t5 and the remaining test case, which are named $d(t3, tx)(x \in \{1, 2, 4, 5, 6, 7, 8, 9, 10\})$ and $d(t5, tx)(x \in \{1, 2, 3, 4, 6, 7, 8, 9, 10\})$, respectively. The third clustering center TC3 is selected according to equation (9). Suppose the clustering number k is 3. All three clustering centers are demonstrated in Table 6.

Based on the three initial clustering centers, the partition results calculated according to Step 7 to Step 9 are in Figure 5. The three initial cluster centers, t3, t5, and t8 are distributed in the final three clusters after one iteration, which is more convenient for future clustering.

Several studies [39, 40] have revealed that the number of clusters is an important factor in the K-means method, which is also applicable to the proposed method. Assume that the number of clusters is less than the number of faults, there may be a cluster containing two or more faults. If engineers stop debugging when locating the first fault, it needs to be re-executed to debug more faults in the program, causing too many iterations. Assume that the number of clusters is more than the actual number of faults, then two or more clusters may contain the same fault. In the parallel debugging mode [27], multiple engineers debug a program simultaneously for multiple faults. After each engineer has found and fixed a fault, the program is retested. If the program still exhibits failures, the debug process is repeated. In this way, the waste of debugging costs caused by the same fault in two subsets is minimized and the executing and debugging interactions are reduced greatly. Therefore, in the HCFL method, the cluster number is suggested to be greater than or equal to the fault number.

4.4. Faults Locating. In this phase, each subset of failed test cases is merged with passed test cases to obtain k test case subsets. Test cases that may fail due to the same fault code are partitioned so that the multiple-fault localization process is decomposed into multiple parallel single-fault localization processes. For each test group, calculate the program entity suspicion using equations in Section 2 and check the code in the descending order of the suspicion value until all faults are located. After a round of testing is completed, more than one bug is often discovered and modified. Then, the fault localization method is to be executed continually until all faults are discovered and modified.

5. Case Studies

This section evaluates the proposed fault localization method, including effectiveness and performing costs. However, the application of the fault localization method in embedded software cannot be queried in the existing literature, so it is virtually impossible to select a general embedded software for cross-comparison experiments of different localization methods. Considering that the embedded software is a special kind of software, the fault localization method applied to embedded software should have general applicability except for the acquisition of the program spectrum. Therefore, the evaluation experiments consist of two parts. First, conduct the cross-comparison experiments on open-source software to evaluate the effectiveness of the HCFL method. Second, apply the proposed method to the real embedded software to evaluate the operating cost.

5.1. Experiment 1

5.1.1. Subject Program. The subject programs Flex, with their accompanying test suites obtained from the SIR library, were adopted for demonstration. Twelve versions of the

Table 3: The spectrum of a sample program.

Test case	s1	s2	s3	s4	s5	s6	s7	s8	s9	s10	s11	s12	s13	s14	s15
t1	1	1	0	1	1	0	0	1	1	1	0	1	1	0	1
t2	1	0	1	0	1	0	0	0	0	1	1	0	1	0	0
t3	1	0	0	0	1	0	1	0	1	0	1	1	1	1	1
t4	1	0	0	1	1	1	1	0	1	1	0	1	0	0	1
t5	0	1	0	1	1	1	0	0	0	1	0	1	1	1	0
t6	0	1	0	0	0	0	1	1	1	0	1	1	0	1	0
t7	0	1	0	0	1	0	0	0	0	0	1	1	1	0	0
t8	1	0	1	0	0	1	1	0	1	1	0	0	0	0	1
t9	1	1	0	0	0	0	1	0	0	0	0	0	0	1	0
t10	0	0	0	0	0	1	0	0	1	0	1	1	0	1	0

Table 4: The distances between each pair of test cases.

Test case	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10
t1	0.000	2.828	2.646	2.236	2.449	3.000	2.646	3.000	3.162	3.317
t2	2.828	0.000	2.646	3.000	2.828	3.317	2.236	2.646	2.828	3.000
t3	2.646	2.646	0.000	2.449	3.000	2.449	2.449	2.828	2.646	2.449
t4	2.236	3.000	2.449	0.000	2.646	3.162	3.162	2.000	3.000	2.828
t5	2.449	2.828	3.000	2.646	0.000	3.000	2.236	3.317	2.828	2.646
t6	3.000	3.317	2.449	3.162	3.000	0.000	2.449	3.162	2.236	2.000
t7	2.646	2.236	2.449	3.162	2.236	2.449	0.000	3.464	2.646	2.449
t8	3.000	2.646	2.828	2.000	3.317	3.162	3.464	0.000	2.646	2.828
t9	3.162	2.828	2.646	3.000	2.828	2.236	2.646	2.646	0.000	2.646
t10	3.317	3.000	2.449	2.828	2.646	2.000	2.449	2.828	2.646	0.000

Table 5: The density of each test case.

Test case	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10
Density	4	3	7	4	4	4	7	3	5	5

Table 6: The three clustering centers of the sample program.

TC1 (t3)	1	0	0	0	1	0	1	0	1	0	1	1	1	1	1
TC2 (t5)	0	1	0	1	1	1	0	0	0	1	0	1	1	1	0
TC3 (t8)	1	0	1	0	0	1	1	0	1	1	0	0	0	0	1

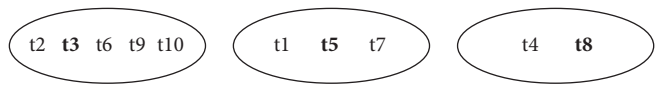

Figure 5: The partition results of the sample program. (a) C1. (b) C2. (c) C3.

subject program with different numbers of faults were obtained by artificial fault activation or injection, as demonstrated in Table 7. The following conditions should be met when injecting a fault.

(i) The injected faults must be realistic, which often occur when programming;

(ii) The injected faults must conform to the grammar rules;

(iii) The injected faults can be tested. Otherwise, it may bring a certain difficulty to the accurate measurement of the fault localization effect.

For each experimental group of the software, the traditional Jaccard, Ochiai, and Tarantula methods, as well as the proposed HCFL method, were used to locate the fault, respectively. The program spectrums were collected with the aid of the GCOV (GNU call-coverage profiler) tool.

5.1.2. Evaluation Criteria for Effectiveness. The traditional way of evaluating fault localization accuracy is by calculating the percentage of statements in a program that has to be examined until the first faulty statement is reached [41–43]. This evaluation criterion has been widely used and verified

Table 7: Subject programs of experiment 1.

Group	1	2	3	4	5	6	8	9	A	B	C	D
LOC	3453	3453	3453	3453	3453	3453	4008	4008	4035	4034	4035	4035

while evaluating the effect of the single-fault localization, but it is not fully applicable in programs with multiple-faults. The multiple-fault localization method usually targets multiple or all bugs. In our experiments, the following criteria were adopted to evaluate the effectiveness of the proposed method.

(1) The Average Number of Statements Examined. The term AVE − S represents the total number of the code lines to be examined to locate all faults in program P. The term Count_s means the total number of statements that are to be examined according to the list of suspicion values, and the term N represents the number of testing rounds. If $(\text{AVE} - \text{S})_x < (\text{AVE} - \text{S})_y$, we define method X to be more efficient than method Y.

$$\text{AVE} - \text{S} = \frac{\sum_{i=1}^{N} \text{Count}_s}{N}. \tag{11}$$

Assume that a slice of nonfaulty statements has the same suspiciousness as the faulty statement. If the statement examined first is exactly the faulty statement, we define this condition as the best case. If the bug is not found until the last statement has been checked, we define this condition as the worst case. It is inferred that in the worst case, we have to examine all the nonfaulty statements with the same suspiciousness as the faulty statement. If we examine some nonfaulty statements but not as many as the worst case, we define it as the average case. Therefore, it is proposed to calculate the average number of statements examined in all the three cases.

(2) The Average Expense Value. The average expense value means the average number of statements examined AVE − S as a percentage of the total executable lines of code (LOC). The smaller the value of expense is, the better the multiple-fault localization method performance will be.

$$\text{expense} = \frac{\text{AVE} - \text{S}}{\text{LOC}} * 100\%. \tag{12}$$

(3) The P Value. In statistics, when the data conform to the normal distribution and the homogeneity of the variance, the parameter tests, such as the u-test and the t-test, are commonly used. However, when the data do not conform to the normal distribution or the unevenness of the variance, the nonparametric test, such as the Wilcoxon signed-rank test method, is required [44]. To prove that the HCFL method is more effective, the difference between the number of statements that need to be examined using these methods is computed. We have proposed a one-tailed assumption that other methods require more statements to be examined than HCFL. The P value reflects the significance level between the two groups of results, and $P < 0.05$ is indicative of a significant difference between the fault localization effectiveness of these two strategies.

As a whole, it can be considered that the fault localization effectiveness of method A is better than that of method B when the following two conditions are met:

(i) The average number of statements examined by method A is less than method B, or the average expense value of method A is less than method B

(ii) The P value of methods A and method B is less than 0.05

6. Results and Analysis

Tables 8 and 9 demonstrate the AVE − S values that need to be examined when locating the first bug and all bugs using the Jaccard, Ochiai, Tarantula, and HCFL methods in the best, worst, and average cases. For instance, in experiment Group1, the average number of statements that need to be examined was 5 when locating the first bug by the Jaccard and Ochiai methods in the best case, while the AVE-S value was 1 by the Tarantula method and our HCFL method.

Referring to the data in Table 8, the AVE-S value by the HCFL method is smaller than that by Jaccard and Ochiai methods when locating the first bug in the program. However, we have found that in all the 12 experiment groups in the best case, the AVE-S values using the Tarantula method are all 1, which are better than the HCFL method. This can be analyzed using the equation of Tarantula. In the best case, all the faulty statements are covered by failed test cases and were not covered by passing test cases. Therefore, the values of $N_{ep}(s)$ and $N_{np}(s)$ are 0, and the suspiciousness scores of all faulty codes are 1 as calculated by the Tarantula method. In the best conditions, the statement that is debugged first is the faulty code.

In Table 9, the AVE-S value by the HCFL method is very much smaller than that by the Jaccard and Ochiai methods when locating all bugs in the program. However, the AVE-S values in Group 2, Group 5, Group 8, Group 9, and Group C by the Tarantula method are 3, which are better than the HCFL method. This is due to the fact that the suspiciousness scores of the three faulty statements are 1, and are checked in the first three places. Anyway, in the worst and average cases, the AVE-S values of the HCFL method are much smaller than the Tarantula method, proving that the efficiency of the HCFL method is generally better than that of the Jaccard, Ochiai, and Tarantula methods.

The performance comparison among the Jaccard, Ochiai, Tarantula, and HCFL methods can be revealed intuitively by the expense value in Figures 6 and 7. The bar chart in Figure 6 demonstrates the average value of expense when locating the first bug, named Expense-first, in the best, worst, and average cases. Similarly, the bar chart in Figure 7 demonstrates the average value of expense when locating all the bugs in the three cases, named Expense-All. The red bar represents the expense value when using the HCFL method,

Table 8: Comparison of AVE-S value when locating the first bug in three cases.

Methods and cases		1	2	3	4	5	6	8	9	A	B	C	D
Jaccard	Best	5	5	985	5	985	69	1	1	1397	1	1	1397
	Worst	363	142	987	363	987	108	130	130	1400	15	15	1400
	Average	184	73	986	184	986	88	65	65	1398	8	8	1398
Ochiai	Best	5	5	958	5	958	69	1	1	1336	1	1	1336
	Worst	363	142	960	363	960	108	130	130	1339	15	15	1339
	Average	184	73	959	184	959	88	65	65	1337	8	8	1337
Tarantula	Best	1	1	1	1	1	1	1	1	1	1	1	1
	Worst	652	171	746	651	746	195	196	196	394	549	577	394
	Average	326	86	373	326	373	98	98	98	197	275	289	197
HCFL	Best	1	1	8.52	1	8.6	1	1	1	1	1	1	1
	Worst	15	69	36	15	36	40	130	130	54.32	15	15	47.4
	Average	8	46	22	8	22	20	65.06	65.06	27.16	8	8	24.2

Table 9: Comparison of AVE-S value when locating all bugs in three cases.

Methods and cases		1	2	3	4	5	6	8	9	A	B	C	D
Jaccard	Best	1762	327	1647	1762	1819	1116	1881	1881	1844	1761	1754	1837
	Worst	1889	343	1686	1888	1946	1137	1929	1929	1849	1764	1757	1842
	Average	1825	335	1666	1825	1882	1126	1905	1905	1846	1762	1755	1839
Ochiai	Best	1745	723	1617	1745	1781	1018	1866	1866	1718	1634	1627	1711
	Worst	1872	744	1656	1871	1908	1039	1914	1914	1723	1637	1630	1716
	Average	1808	733	1636	1808	1844	1028	1890	1890	1720	1635	1628	1713
Tarantula	Best	1007	502	3	1006	3	840	3	3	497	1133	3	497
	Worst	1365	518	746	1364	746	856	588	588	502	1506	577	502
	Average	1186	510	373	1185	373	848	588	588	499	1319	289	499
HCFL	Best	205	176	83.83	204	87.55	197	545.81	545.74	336.97	253.07	166.22	328.75
	Worst	597	320	154	596	258.4	261	723.06	723.06	352.83	654.99	178.36	346.16
	Average	401	243	117.93	400	172.45	228	634.06	634.06	344.38	453.22	204.91	337.33

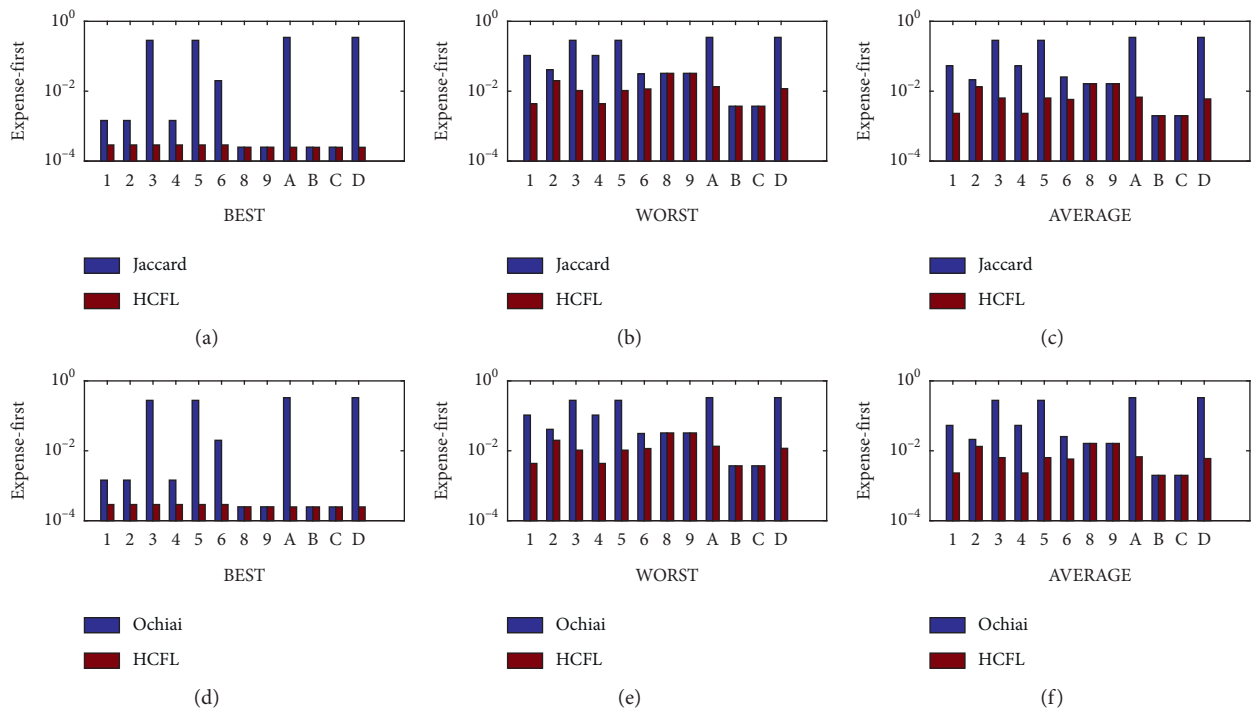

Figure 6: Continued.

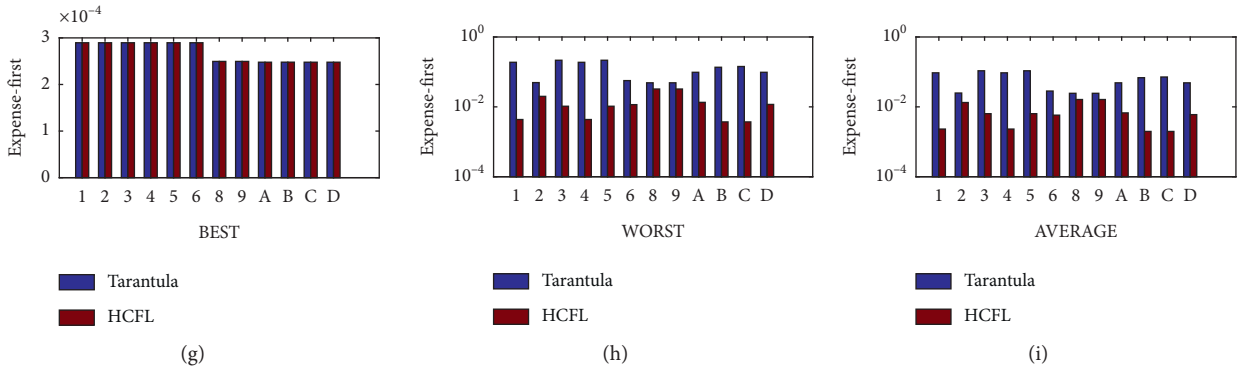

Figure 6: Comparison of the expense value when locating the first bug.

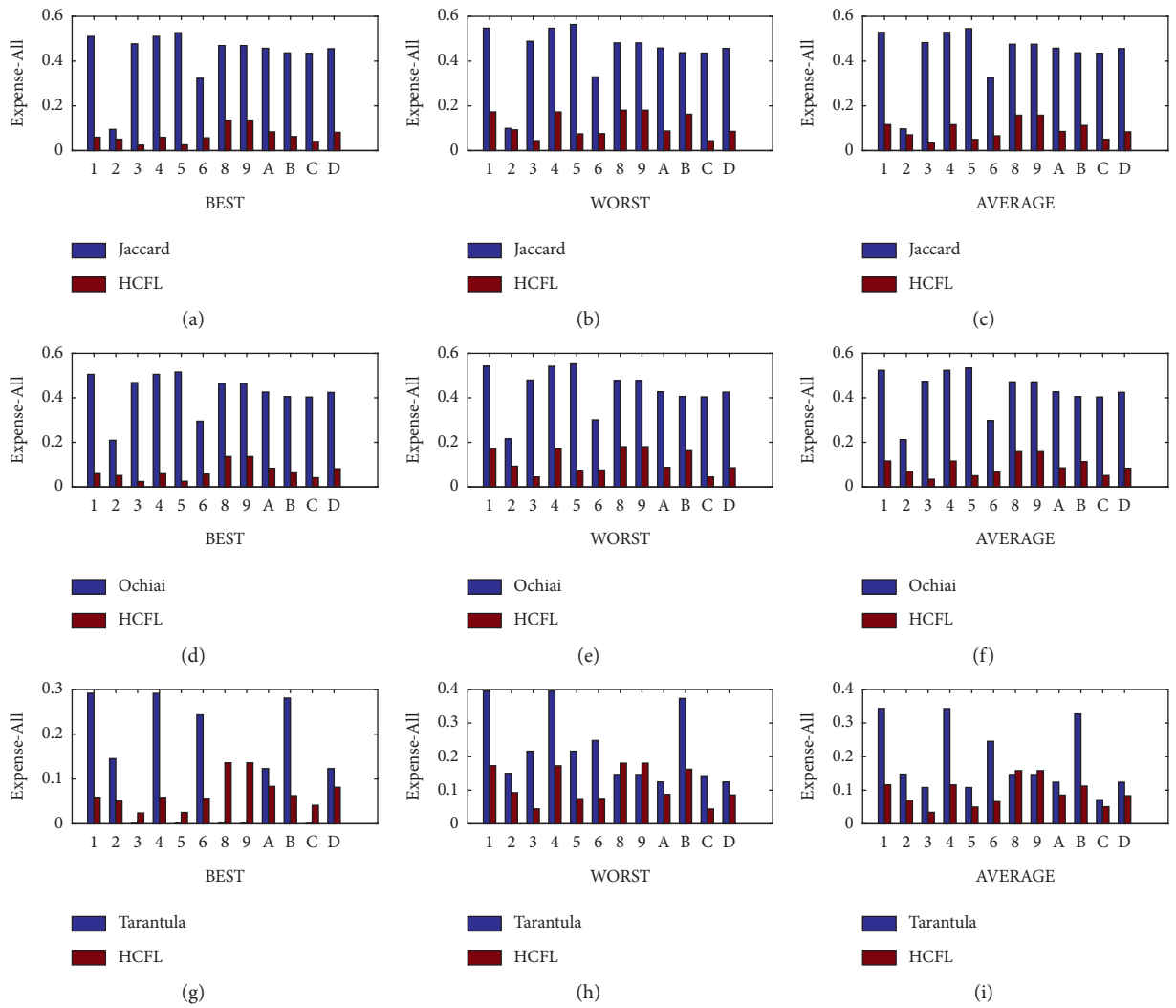

Figure 7: Comparison of the expense value when locating all bugs.

while the blue bar in each line represents the expense value when using the Jaccard, Ochiai, and Tarantula methods. It appears that the shorter the bar chart, the better the fault localization efficiency. The comparison results have inferred that the fault localization efficiency of the HCFL method is better than that of the Jaccard and Ochiai methods, but slightly lower than that of the Tarantula method when locating the first bug; however, when locating all the bugs, the efficiency of the HCFL method is higher than that of the Jaccard, Ochiai, and Tarantula methods in most of the

groups, especially in the worst and average cases. Compared with the data in the best cases, the comparison results in the worst and the average cases can better reflect the process of software fault localization. Therefore, from the comprehensive comparison results of Figures 6 and 7, the superiority of HCFL in fault localization efficiency is demonstrated.

Tables 10 and 11 revealed the P value of HCFL versus Jaccard, HCFL versus Ochiai, and HCFL versus Tarantula when locating the first bug and locating all the bugs, respectively. We could see in Table 10 that the P value is 1 in some groups, such as Group 8 and Group 9 in the best case, Group B, and Group C in all the three cases. Recalling the values in Table 8, this is because when locating for the first bug, the average number of statements examined by the three methods is the same, that is, the first bug had been located when debugging the first statement. In other groups, the P value is less than 0.05. In Table 11, the P values in all experimental groups are less than 0.001, revealing that the fault localization effectiveness of the HCFL method is extremely different from the Jaccard, Ochiai, and Tarantula methods.

Overall, the results of Experiment 1 indicate that the average number of statements examined by the HCFL method as well as the average expense value of the HCFL method when locating all the bugs is much less than the Jaccard, Ochiai, and Tarantula methods in most of the cases. The P values of HCFL versus Jaccard, HCFL versus Ochiai, and HCFL versus Tarantula are much less than 0.05 in most of the cases. We believe that the fault localization effectiveness of the HCFL method is much better than that of the Jaccard, Ochiai, and Tarantula methods when locating the multiple faults.

6.1. Experiment 2

6.1.1. Subject Program. In this experiment, three embedded software programs with different scales and different operating platforms in engineering projects were selected as demonstrated in Table 12. The Main_control program was used to carry out the central control function of an application system. Therefore, the logic of the program was relatively complex, with many branches and judgment statements. The Main_control was implemented in C++ language and executed on TMS320C6000, and the number of instrumentation points of the Main_control was 2489. Data_commu was a program of an application payload implementing the data interaction function, which was also written in C++ language and worked on TMS320F2812. The number of instrumentation points of Data_commu was 737. The program Data_process was the software part of a SoPC (System on Program Chip), which cooperated with the programmable logic part to complete the function of command receiving and command parsing. Data_process was implemented in C language and executed on PowerPC405 core of Xilinx FPGA, with 237 instrumentation points.

6.1.2. Evaluation Criteria for the Operating Cost. The software instrumentation to obtain the program spectrum may result in a decrease in software performance. In this experiment, the running time of each software program under the following three testing scenarios is statistically compared:

(i) The run-time of the software program before instrumentation

(ii) The run-time of the software program using the spectrum-acquiring method is based on real-time transmission

(iii) The run-time of the software program using the cache-based spectrum-acquiring method

In the second testing scenario, output channels are needed for transmitting the spectrum data in real-time. According to the characteristics of each software program, we used the SPI (Serial Peripheral Interface) port to transmit the program spectrum of the Main_control program, and the serial port for spectrum data transmission for the Data_commu and Data_process program.

Taking program Main_control as an example, the statistical method of the program run-time was as follows. The counter value of the timer Timer0 of the DSP was used to calculate the program run-time, with a timing accuracy of 0.02 microseconds. The register value of Timer0 was set to the upper limit at the beginning of the program to ensure that no overflow occurred during the statistical period. The control register was set to start the timing at the beginning of the program, and the register value of Timer 0 was read at the end of the program. The run-time of the program T_{runtime} was calculated as follows:

$$T_{\text{runtime}} = \left(\text{Timer}\,0_{\text{upper}} - \text{Timer}\,0_{\text{end}}\right) * T_{\text{accuracy}}, \quad (13)$$

$\text{Timer}\,0_{\text{upper}}$ means the upper limit value of the Timer 0 register, and $\text{Timer}\,0_{\text{end}}$ means the register value of Timer 0 read at the end of the program.

6.1.3. Results and Analysis. The maximum, the minimum, and the mean run-time were recorded as in Table 13 when, respectively, carrying out all the test cases in each test suite for program Main_control, Data_commu, and Data_process.

From the data in Table 13 and Figure 8, it can be concluded that no matter which program-spectrum acquisition method is used, the code instrumentation for acquiring the program-spectrum increases the run-time of the program. Among them, the average run-time when using the method based on the real-time transmission is more than twice that of run-time without instrumentation, while using the cache-based program spectrum acquisition method proposed in this paper, the run-time of the program is slightly more than twice that of without instrumentation.

As the basis of the fault localization method, the acquisition of program-spectrum is the bottleneck that affects the performance of the whole fault localization method for software, especially for embedded software. Using the cache-based program-spectrum acquisition method, the acquisition time of the program-spectrum for each test case is

Table 10: The P value when locating the first bug.

Methods and cases		1	2	3	4	5	6	8	9	A	B	C	D
HCFL versus Jaccard	Best	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	1	1	<0.001	1	1	<0.001
	Worst	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.013	0.013	<0.001	1	1	<0.001
	Average	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.013	0.013	<0.001	1	1	<0.001
HCFL versus Ochiai	Best	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	1	1	<0.001	1	1	<0.001
	Worst	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.013	0.013	<0.001	1	1	<0.001
	Average	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.013	0.013	<0.001	1	1	<0.001
HCFL versus Tarantula	Best	1	1	<0.001	1	<0.001	1	1	1	1	1	1	1
	Worst	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
	Average	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001

Table 11: The P value when locating all bugs.

Methods and cases		1	2	3	4	5	6	8	9	A	B	C	D
HCFL versus Jaccard	Best	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
	Worst	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
	Average	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
HCFL versus Ochiai	Best	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
	Worst	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
	Average	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
HCFL versus Tarantula	Best	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
	Worst	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
	Average	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001

Table 12: Subject programs of experiment 2.

Program	Executing platform	LOC	Number of instrumentation points	Test suite
Main_control	TMS320C6000	19547	2489	584
Data_commu	TMS320F2812	9230	737	486
Data_process	PowerPC405	1686	237	372

Table 13: Comparison of run-time by the three methods.

Program	Before instrumentation (sec)			Real-time transmission (sec)			Cache-based transmission (sec)		
	Max	Mean	Min	Max	Mean	Min	Max	Mean	Min
Main_control	2.886431	2.0999285	1.313426	4.820339	3.394385	1.968431	3.175074	2.387156	1.599238
Data_commu	2.105153	1.345316	0.585479	5.473399	3.439275	1.405151	2.294617	1.590078	0.885538
Data_process	1.280055	0.965821	0.127602	2.606832	1.956633	0.268435	1.327394	1.001160	0.130907

shortened by half, which improves the efficiency of the fault localization method.

7. Threats to Validity

7.1. Threats to Internal Validity. The internal validity threat involves the causal relationship between the independent and dependent variables provided by the experiment. The specific implementation of the clustering method and test script code in Section 4 may have some defects, which may affect the experimental results. To ensure the correctness of the specific implementation, the manually written code was strictly reviewed and sufficiently tested.

7.2. Threats to External Validity. The primary external threat of the experiment results lies in the selection of the object program, which has limits in software scale and number of faults. Therefore, we carried out empirical experiments on the application of embedded software and strengthened the confidence of the actual engineering application.

Besides, the quality of the test cases also has a certain impact on the software fault localization. A positive test case can expose as many faults as possible in the software. Efficient test cases should cover as many statements, conditions, decision conditions, and combinations of conditions as possible. The objects and test cases in Experiment 1 are the open-source widely used in related research and have certain

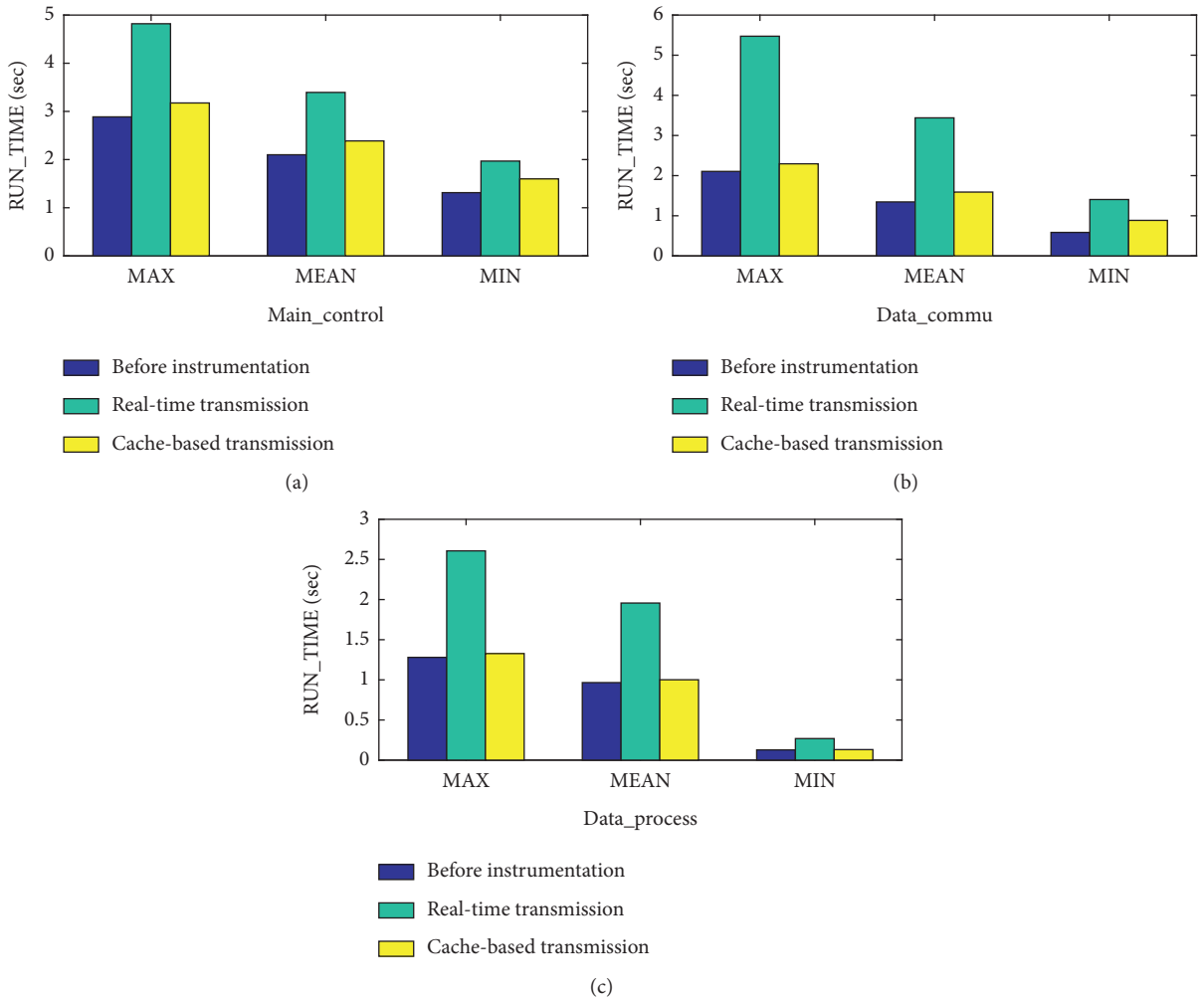

Figure 8: Run-time comparison between real-time transmission and cache-based transmission.

representativeness [45–48]. In Experiment 2, the Equivalence Partitioning analysis, Boundary Value analysis, Decision Table analysis, and other methods were used in test case design to ensure the sufficiency of test cases. The final versions of the three programs in Experiment 2 had gone through multiple rounds of testing and were successfully applied to products. Therefore, the sufficiency of the test suite set can be guaranteed, and software bugs can be covered by the test suite.

8. Conclusion and Future Work

With the increase in the size and complexity of embedded software, the traditional human-based fault localization method is no longer applicable; thus, an efficient fault localization method is urgently needed to help engineers for debugging. This paper proposes a special fault localization method, especially for embedded software, and improvements have been made in the following two aspects compared with previous researches. Firstly, adaptive modification has been made in acquiring program-spectrum schemes according to the characteristics of embedded software, which greatly promotes the efficiency of the acquisition of program-spectrum. Secondly, the traditional *K*-means clustering method is improved by using a hybrid clustering method based on density and distance to find out the initial clustering center, which improves the efficiency of clustering. Several experimental results demonstrate that the proposed fault localization method can accurately locate multiple-bugs in embedded software, which saves the debugging time for engineers.

In future work, more experiments on large-scale software programs with much more faults will be performed, strengthening the accuracy of the clustering method to localize multiple-faults at a lower expense. Besides the study of fault localization methods in these specific testing fields, applications of fault localization methods on other execution platforms of embedded software should also be concerned. The scene features of these new platforms will be analyzed to make full use of the existing research results of fault localization, proposing high-quality solutions.

Data Availability

The data that support the findings of this study are available from the corresponding author upon reasonable request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

Thanks are due to Dr. HanYu Pei and LinZhi Huang of Beihang University for their help with the experiments and their useful suggestions. This work was financially supported by a National Pre-Research Program of China (No. 61400020404).

References

[1] T. Henzinger, C. Kirsch, and J. Rushby, "Embedded software," in *Proceedings of the First International Workshop, EMSOFT 2001*, Tahoe City, CA, USA, October 2001.

[2] V. Narayanan and Y. Xie, "Reliability concerns in embedded system designs," *Computer*, vol. 39, no. 1, pp. 118–120, 2006.

[3] J. C. Munson and T. M. Khoshgoftaar, "The detection of fault-prone programs," *IEEE Transactions on Software Engineering*, vol. 18, no. 5, pp. 423–433, 1992.

[4] G. J. Pai and J. B. Dugan, "Empirical analysis of software fault content and fault proneness using bayesian methods," *IEEE Transactions on Software Engineering*, vol. 33, no. 10, pp. 675–686, 2007.

[5] C. S. Wright and T. A. Zia, "A quantitative analysis into the economics of correcting software bugs," *Computational Intelligence in Security for Information Systems*, vol. 6694, pp. 198–205, 2011.

[6] W. E. Wong, V. Debroy, A. Surampudi, H. J. Kim, and M. F. Siok, "Recent catastrophic accidents: investigating how software was responsible," in *Proceedings of the 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement, SSIRI 2010*, Singapore, June 2010.

[7] D. S. Rosenblum, "A practical approach to programming with assertions," *IEEE Transactions on Software Engineering*, vol. 21, no. 1, pp. 19–31, 1995.

[8] J. Hennessy, "Symbolic debugging of optimized code," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 323–344, 1982.

[9] J. C. Edwards, "Method, system, and program for logging statements to monitor the execution of a program," 2003.

[10] The software infrastructure repository, [Online]. Available: http://sir.unl.edu/portal/index.html, Accessed on Aug. 2016.

[11] W. E. Wong, R. Gao, Y. Li, A. Rui, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[12] R. Gupta, M. J. Harrold, and M. L. Soffa, "Program slicing-based regression testing techniques," *Software Testing, Verification, and Reliability*, vol. 6, no. 2, pp. 83–111, 1996.

[13] L. Guo, A. Roychoudhury, and T. Wang, "Accurately choosing execution runs for software fault localization," in *Proceedings of the 15th International Conference, Compiler Construction*, Vienna, Austria, March 2006.

[14] R. Abreu, P. Zoeteweij, and A. J. C. V. Gemund, "On the accuracy of spectrum-based fault localization," in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques-Mutation, 2007*, Windsor, UK, September 2007.

[15] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi, "An empirical investigation of the relationship between spectra differences and regression faults," *Software Testing, Verification and Reliability*, vol. 10, no. 3, pp. 171–194, 2000.

[16] W. E. Wong, Yu Qi, L. Zhao, and K. Y. Cai, "Effective fault localization using code coverage," in *Proceedings of the 2007 31st Annual International Computer Software and Applications Conference*, Beijing, China, July 2007.

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

[18] X. Xue and A. S. Namin, "How significant is the effect of fault interactions on coverage-based fault localizations?" in *Proceedings of the 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, Baltimore, MD, USA, October 2013.

[19] 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*, Long Beach, CA, USA, November 2005.

[20] Z. Wei and B. Han, "Multiple-bug oriented fault localization: a parameter-based combination approach," in *Proceedings of the 2013 IEEE Seventh International Conference on Software Security & Reliability-companion*, Gaithersburg, MD, USA, June 2013.

[21] G. Cheng, Z. Zheng, Y. Zhang, Z. Zhang, and Y. Xue, "Factorising the multiple fault localization problem: adapting single-fault localizer to multiple-fault programs," in *Proceedings of the 2012 19th Asia-Pacific Software Engineering Conference*, December 2012.

[22] D. Jeffrey, N. Gupta, and R. Gupta, "Fault localization using value replacement," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008*, pp. 167–178, Seattle, WA, USA, July 2008.

[23] D. Jeffrey, N. Gupta, and R. Gupta, "Effective and efficient localization of multiple faults using value replacement," in *Proceedings of the 2009 IEEE International Conference on Software Maintenance*, pp. 221–230, Edmonton, Canada, September 2009.

[24] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken, "Statistical debugging: simultaneous identification of multiple bugs," in *Proceedings of the Twenty-Third International Conference (ICML 2006)*, pp. 26–29, Pittsburgh, PA, USA, June 2006.

[25] V. Debroy and W. E. Wong, "Insights on fault interference for programs with multiple bugs," in *Proceedings of the 2009 20th International Symposium on Software Reliability Engineering*, pp. 165–174, Mysuru, Karnataka, India, November 2009.

[26] N. DiGiuseppe and J. A. Jones, "On the Influence of Multiple faults on coverage-based fault localization," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pp. 210–220, Toronto, Canada, July 2011.

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

[28] R. Abreu, P. Zoeteweij, and A. J. C. V. Gemund, "Spectrum-based multiple fault localization," in *Proceedings of the 2009 IEEE/ACM International Conference on Ase ACM*, Auckland, New Zealand, November 2009.

[29] S. Friedrich and M. Frenkel, "Improving coverage-based localization of multiple faults using algorithms from integer linear programming," in *Proceedings of the*https://ieeexplore.ieee.org/xpl/conhome/6403947/proceeding, Dallas, TX, USA, November 2012.

[30] S. Friedrich and M. Frenkel, "More debugging in parallel," in *Proceedings of the 2014 IEEE 25th International Symposium on*

Software Reliability Engineering, Naples, Italy, November 2014.

[31] R. Gao and W. E. Wong, "MSeer-an advanced technique for locating multiple bugs in parallel," *IEEE Transactions on Software Engineering*, vol. 45, no. 3, pp. 301–318, 2019.

[32] T. Nejmeddine, "Object-oriented system decomposition quality," in *Proceedings of the 2002 7th IEEE International Symposium on High Assurance Systems Engineering*, Tokyo, Japan, October 2002.

[33] W. Masri and R. A. Assi, "Prevalence of coincidental correctness and mitigation of its impact on fault localization," *ACM Transactions on Software Engineering and Methodology*, vol. 23, no. 1, pp. 1–28, 2014.

[34] F. Steinmann, M. Frenkel, and R. Abreu, "Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pp. 314–324, Lugano, Switzerland, July 2013.

[35] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "An evaluation of similarity coefficients for software fault localization," in *Proceedings of the 2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, pp. 39–46, Riverside, CA, USA, December 2006.

[36] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: a review," *Acm Computing Surveys*, vol. 31, no. 3, pp. 264–323, 1999.

[37] A. Rodriguez and A. Laio, "Clustering by fast search and find of density peaks," *Science*, vol. 344, no. 6191, pp. 1492–1496, 2014.

[38] C. Xiong, H. Zhen, L. Ke, and L. Xuan, "An improved K-means text clustering algorithm by optimizing initial cluster centers," in *Proceedings of the 2016 7th International Conference on Cloud Computing and Big Data (CCBD)*, November 2016.

[39] M. R. Rezaee, B. P. F. Lelieveldt, and J. H. C. Reiber, "A new cluster validity index for the fuzzy c-mean," *Pattern Recognition Letters*, vol. 19, no. 3-4, pp. 237–246, 1998.

[40] S. S. Khan and A. Ahmad, "Cluster center initialization algorithm for K -means clustering," *Pattern Recognition Letters*, vol. 25, no. 11, pp. 1293–1302, 2004.

[41] X. Ju, S. Jiang, X. Chen, X. Wang, Y. Zhang, and H. Cao, "HSFal: effective fault localization using hybrid spectrum of full slices and execution slices," *Journal of Systems and Software*, vol. 90, pp. 3–17, 2014.

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

[43] N. Digiuseppe and J. A. Jones, "Fault density, fault types, and spectra-based fault localization," *Empirical Software Engineering*, vol. 20, no. 4, pp. 928–967, 2015.

[44] S. M. Taheri and G. Hesamian, "A generalization of the Wilcoxon signed-rank test and its applications," *Statistical Papers*, vol. 54, no. 2, pp. 457–470, 2013.

[45] Y. Gao, Z. Zhang, Z. Long, G. Cheng, and Z. Zheng, "A theoretical study: the impact of cloning failed test cases on the effectiveness of fault localization," in *Proceedings of the 2013 13th International Conference on Quality Software*, September 2013.

[46] M. Khatibsyarbini, M. A. Isa, H. N. A. Hamed, H. N. A. Hamed, and M. D. Mohamed Suffian, "Test case prioritization using firefly algorithm for software testing," *IEEE Access*, vol. 7, pp. 132360–132373, 2019.

[47] G. Liang, D. Lo, L. Jiang, and H. Zhang, "Diversity maximization speedup for fault localization," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, Essen, Germany, September 2012.

[48] F. Keller, L. Grunske, H. Simon, A. Filieri, A. Van Hoorn, and D. Lo, "A critical evaluation of spectrum-based fault localization techniques on a large-scale software system," in *Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability, and Security*, July 2017.