# Fault Localization in Embedded Software based on a Single Cyclic Trace

Azzeddine Amiar, Mickaël Delahaye, Yliès Falcone, Lydie du Bousquet
Université Grenoble Alpes, Laboratoire d'Informatique de Grenoble (LIG)
Email: FirstName.LastName@imag.fr

***Abstract*—Locating faults in embedded software, especially in microcontrollers, is still difficult. Quite recently, it became possible to recover execution traces from microcontrollers using specific hardware probes. However, the collected traces contain a huge volume of low-level data. Consequently, manual analysis is difficult and our industrial partners call for automatic and more effective fault-localization methods for embedded software.**

**This paper presents a new approach to automatically locate faults in embedded programs given a single faulty execution trace. Our approach exploits the cyclic nature of embedded programs and uses several adapted spectrum-based methods in order to find faults on a single execution, rather than a set of multiple failing and passing executions. Our approach is implemented in the tool CoMET and evaluated on several faulty programs. The evaluation shows that our single-trace fault-localization method using Ochiai [1] allows engineers to find a fault by inspecting less than 5% of the program in most cases, and it confirms the interest of automatic fault localization for microcontrollers.**

***Index Terms*—dynamic analysis; automatic fault localization; embedded systems**

## I. Introduction

A microcontroller is an integrated circuit that incorporates onto the same microchip the essential computer elements such as the processor, memory, peripherals and input/output interfaces [2]. Microcontrollers are embedded in various kinds of equipment such as cars, washing machines or toys. Surprisingly, even though microcontrollers are now quite affordable, the development of embedded software still weights heavily both on the final cost of the product and the time to market.

According to our industrial partners, the main costs are due to the validation step, and especially the fault diagnosis. In fact, in spite of the existence of several development environments for embedded applications, there are few tools dedicated to validation. Classical approaches used for software validation are difficult to use. For instance, specification-based verification tools (provers, model checkers) can not be used since applications are not formally specified. Moreover, because the actual code is very low-level and specific to the environment, static analysis is almost impossible. Note that dynamic validation is still difficult due to the limited observability of the execution of the embedded software. Indeed, engineers still use oscilloscopes to analyze embedded applications by interpreting electric signals. Consequently, validation and fault diagnosis are usually carried out manually, and are thus tedious and time-consuming tasks [3].

Last generation of microcontrollers include parts dedicated to trace collections. For example, ARM Cortex-M includes a section dedicated to trace collection, called *Embedded Trace Macrocell* (ETM) [4]. Using specialized probes, such as Keil *UlinkPro* [5] and STMicroelectronics *ST-Link* probes [6], it is possible to collect basic execution traces without input/output data. An execution trace consists in a sequence of *program counters* (PCs) that reflects the execution. However, even in a recording of a few seconds, the execution trace contains huge volume of low-level data. In addition, the analysis of a trace is particularly difficult because the association between PCs and the actual source code's statements to debug is often complicated. In fact, because of memory constraints or performance reasons, software code is heavily optimized before being loaded in microcontrollers.

Many embedded programs can be categorized as *cyclic programs*, as they rely on a main loop that iterates indefinitely. In the following, the instruction that defines this main loop is called the *loop header*. Usually, at each iteration of the main loop, the sensors are monitored and actions are taken in response to changes. For instance, at each cycle, an Anti-lock Braking System (ABS) reads speed sensors attached to the driving wheels and adjusts the braking power accordingly. Due to the cyclic nature of embedded programs, collected execution traces consist in long sequences of multiple repetitions of instructions.

In this context, our industrial partners would like to localize a fault in a program given a *single* trace that ends at the failure. They are interested into failure that stops the execution of an application (such as the so-called hard fault). The only available piece of information is one trace because of the due to difficulty of reproducing failures in general. To summarize, our context contains several interesting challenges: a single execution trace, a huge volume of data, and a fragile association between source code and execution trace. However, any improvement of the manual process of locating faults may considerably speed up the development process.

*Contributions:* In this paper, we propose a complete and novel approach to help locating faults in cyclic programs based on a *single* faulty execution trace. This approach is based on Automatic Fault-Localization (AFL) methods [7], [8], [1], [9] and takes advantage of the cyclic nature of traces. Our approach first automatically detects the cycles in the trace before using adapted AFL method to find the most suspicious statements. The effectiveness of our method is demonstrated by an experimental evaluation, made possible thanks to the development of a tool named CoMET [10]. This evaluation shows encouraging results. In fact, the proposed method allows to find the bug in several faulty programs by inspecting in most cases less than 5% of the program code, with the best suspiciousness ranking, namely Ochiai.

*Paper Organization:* Section II gives an overview of related work in fault localization. In Section III, the automatic fault-localization methods on which our approach is based are detailed. Section IV describes our fault-localization approach using a single microcontroller execution trace. Section V (resp. VI) presents an experimental evaluation of loop-header detection (resp. fault localization). Section VII discusses the threats to validity of this evaluation. Finally, Section VIII proposes some conclusions and perspectives.

## II. Related Work

The goal of automatic fault localization (AFL) is to ease the debugging step, this by pointing out to the engineer the lines of code that are the most likely responsible for the observed failure. A lot of techniques have been proposed to locate fault(s) in programs. This section discusses existing techniques. However, note that our very specific context has not been investigated earlier, mostly because our work is based on a single trace.

To localize the fault, some methods compare passing and failing executions. For instance, given a set of passing executions and a single failing execution, the nearest neighbor method [7] finds the passing execution most similar to the failing execution. Then, it computes statements that are executed by one but not the other, using the so-called union and intersection models. Other methods compares executions statistically [8], [1], [9], [11]. In particular, the Tarantula approach [8] computes a measure of association between executions of program statements and failures. The measure allow Tarantula to rank program statements according to their *suspiciousness* for consideration by the engineer. This method is based on the idea that program statements whose absence or presence are most strongly associated with failures are more likely to be faulty. As shown in [12], this method can be re-interpreted as a data mining procedure, because it uses an indicator which characterizes association rules between data. In this paper, we propose an adaptation of such methods to our (industrial) context described in the introduction.

Another way to help locating fault is program slicing, see for instance [13], [14], [15]. Program slicing points to a set of statements that may affect the values of variables at a given point in the program (e.g., at the failure). However, for the purpose of fault localization, the set contains statements that are associated with the failure and that may not cause the failure. Also, the set does not come with an order for suspect examination. Our approach however does rank statements given their suspiciousness.

Some techniques combine statistics and program slicing, e.g., [16], [17], [18], [19]. This kind of techniques requires a fine-grain association between the program source code, on which the slicing is usually done, and program execution traces. In our context, because of the various code-optimizations performed during compilation, this association does not exist, and as a result, such techniques are not directly usable.

Another kind of fault-localization techniques, sometimes called delta debugging [20], [21], proposes to change the program state during its execution, to detect the origin of the fault. As they rely on experiments with the programs, those techniques are not usable in the embedded context.

## III. Background

In this section, we focus on two existing fault-localization methods: the nearest neighbor method [7] and the statistical method [8], [1]. Our approach is based on these two methods. Both of these methods take as input several passing test cases and at least one failing test case. A test case combines a test input and the verdict of an oracle. Then, using several techniques, they infer the lines of code that the engineer needs to inspect. In addition to the test cases, the *program spectrum* [22] is assumed to be available. The program spectrum is a set of data that provides a specific view on the dynamic behavior of the considered program. A typical program spectrum is the statement coverage. In this paper, only the statement coverage will be considered. In this section, we consider the program runs, and for each program run we have a spectrum and a verdict (passing or failing).

### A. Nearest neighbor method

Assume a given failing run $f$ and a set of passing runs $S$. To find events that are present in the failing run but absent from passing runs, a simple approach consists in computing the differences of spectrum between the failing run $f$ and the union of all the passing runs:

$$\mathsf{spectrum}(f) - \big(\bigcup_{s\in S}\mathsf{spectrum}(s)\big)\,.$$

This approach is called the *union model*. The *intersection model* expresses another idea that is complementary to the union model. It tries to find features that are absent in the failing run but present in passing runs:

$$\big(\bigcap_{s\in S}\mathsf{spectrum}(s)\big) - \mathsf{spectrum}(f)\,.$$

Proposed by Renieris and Reiss [7], the *nearest neighbor* approach first consists in finding the passing run that corresponds the most to the failing run, by comparing their spectra. The similarity between spectra, represented as bit vectors, is measured using the *Hamming distance*[1], which was originally conceived for detecting and correcting errors in digital communication [23]. Then, the *nearest neighbor* approach applies the union and intersection models on the selected passing runs and the failing run. This step produces

[1]The Hamming distance of two bit vectors $u$ and $v$ is defined as $a_{10}+a_{01}$ where $a_{10}$ (resp. $a_{01}$) is an indicator defined in Table I that gives the number of bits that hold true in $u$ and false in $v$ (resp. false in $u$ and true in $v$).

TABLE I
BINARY INDICATORS FOR TWO BIT VECTORS $u$ AND $v$ OF SIZE $n$

| $u \backslash v$ | 0 | 1 |
|---|---|---|
| 0 | $a_{00} = \sum_{1 \leq i \leq n} \bar{u}_i \cdot \bar{v}_i$ | $a_{01} = \sum_{1 \leq i \leq n} \bar{u}_i \cdot v_i$ |
| 1 | $a_{10} = \sum_{1 \leq i \leq n} u_i \cdot \bar{v}_i$ | $a_{11} = \sum_{1 \leq i \leq n} u_i \cdot v_i$ |

$$M \text{ runs} \left\{ \begin{bmatrix} \overbrace{x_{1,1} \quad x_{1,2} \quad \cdots \quad x_{1,N}}^{N \text{ elements}} \\ x_{2,1} \quad x_{2,2} \quad \cdots \quad x_{2,N} \\ \vdots \qquad \vdots \qquad \ddots \qquad \vdots \\ x_{M,1} \quad x_{M,2} \quad \cdots \quad x_{M,N} \end{bmatrix} \right. \qquad \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_M \end{bmatrix}$$

Spectrum matrix        Error detection vector

Fig. 1. The spectrum matrix and the error detection vector used in fault diagnosis

two sets of spectrum elements (typically covered statements). The first one, from the union model, contains elements present in the failing run but absent from the passing run. The other set comes from the intersection model and contains the elements absent from the failing run but present in the passing run. Those two sets contain elements that, by their presence or absence, are likely to have caused the observed failure. Finally the engineer analyzes the two sets to localize the fault.

## B. Suspiciousness ranking method

Another kind of spectrum-based fault localization, proposed by Jones, Harold and Stasko in [8], aims to identify spectrum elements (in most cases, statements) whose presence or absence in runs are strongly correlated with the failure of the runs. Such elements are likely the cause of the observed failure. For each spectrum element, the method computes a *suspiciousness score*, which measures the correlation of the execution of the element to the failure of the program.

*Spectrum matrix and error detection vector:* To obtain the suspiciousness score for each program statement, the *spectrum matrix* is built. For $M$ runs and $N$ program statements, the spectrum matrix is a $M \times N$-matrix, as shown in Fig. 1, such that:

- each row $i$ corresponds to a particular run;
- each column $j$ corresponds to a particular spectrum element (e.g., a program statement, a basic block);
- and each value $x_{i,j}$ of the matrix is a Boolean value indicating whether during the $i$-th run the spectrum element $j$ is collected ($x_{i,j} = 1$) or not ($x_{i,j} = 0$).

In some of the $M$ runs, an error is detected. Other runs complete without error. This information yields a bit vector $e$ of size $M$, named the error detection vector, where, if the $i$-th run in the matrix ($1 \leq i \leq M$), is a failing run, then $e_i = 1$, otherwise $e_i = 0$.

TABLE II
FORMULAS OF MAJOR SIMILARITY COEFFICIENTS FOR $(u, v)$

| **Name** | **Formula** |
|---|---|
| Tarantula | $\frac{\frac{a_{11}}{a_{11}+a_{01}}}{\frac{a_{11}}{a_{11}+a_{01}} + \frac{a_{10}}{a_{10}+a_{00}}}$ |
| Jaccard | $\frac{a_{11}}{a_{11} + a_{01} + a_{10}}$ |
| AMPLE | $\left\lvert \frac{a_{11}}{a_{11} + a_{01}} - \frac{a_{10}}{a_{10} + a_{00}} \right\rvert$ |
| Ochiai | $\frac{a_{11}}{\sqrt{(a_{11} + a_{01}) \times (a_{11} + a_{10})}}$ |
| $O^p$ | $a_{11} - \frac{a_{10}}{a_{10} + a_{00} + 1}$ |

The suspiciousness score of a particular spectrum element numbered $j$ is computed by comparing the vector $(x_{i,j})_{1 \leq i \leq M}$ (i.e, the $j$-th column of *spectrum matrix*) and the error detection vector $e$. This comparison uses a *similarity coefficient*, that is, a measure of similarity between two bit vectors of the same size.

*Similarity coefficients:* There exists a whole variety of similarity coefficients between bit vectors used in the automatic fault localization. Among the best known and most used coefficients, we choose to experiment with the following ones:

- Tarantula's, the original coefficient used in [8] to assist fault localization using a visualization technique;
- The Jaccard index, a well-known statistic measure used to compare sets and used in the Pinpoint framework [24];
- AMPLE, the coefficient used in the tool AMPLE (Analyzing Method Patterns to Locate Errors) to locate error in object-oriented software [25];
- Ochiai, a coefficient originally used in molecular biology, but used successfully in fault localization [1];
- $O^p$, a coefficient proposed in [9], designed to be optimal on programs respecting the ITE2 model, i.e., consisting of a sequence of two if-then-else constructs.

For two spectra $u$ and $v$ (as bit vectors), Table II indicates the formula of each of those coefficients, given the indicators defined in Table I.

Given a similarity coefficient $S$, the suspiciousness score $s_j$ of a particular spectrum element numbered $j$ is defined as:

$$s_j = S((x_{i,j})_{1 \leq i \leq N}, e).$$

For this purpose, the indicators have the following meaning:

- $a_{11}$ is the number of *failing* runs where the spectrum element $j$ is *recorded*;
- $a_{10}$ is the number of *passing* runs where the spectrum element $j$ is *recorded*;
- $a_{01}$ is the number of *failing* runs where the spectrum element $j$ is *not recorded*;
- $a_{00}$ is the number of *passing* runs where the spectrum element $j$ is *not recorded*.

## IV. Fault Localization Using A Single Trace

In this section, we present an approach to analyze a single failure execution trace (without data) to provide engineers with program counters that are most likely the suspects for a fault. This approach leverages the cyclic nature of embedded programs and is based on the fault-localization techniques presented in the previous section. This section first presents our hypothesis, then our pretreatment of traces in order to detect cycles in the considered execution trace, and finally the two adapted fault-localization methods.

### A. Hypotheses

It must be noted that our technical approach relies on the analogy between program runs and cycles. Programs runs are usually independent, especially if they are obtained by executing a test suite (usually) consisting of independent test cases. On the contrary, multiple cycles of same run could interact in many ways. In general independence between cycles cannot be guaranteed. However, the cyclic nature of embedded software can make this assumption valid for a lot of errors. In fact, many embedded programs interact with the microcontroller environment via hardware (e.g., sensors), and execute tasks without taking into account the results of previous tasks. All programs provided by our industrial partners are with independent cycles and fits into the model depicted in Fig. 2.

Moreover, since the abnormal behavior necessarily appears at the end of the trace, we assume that the last cycle in the trace corresponds to a failing run, the other cycles being considered as passing runs. Therefore, the cause of the failure and the failure occur in the last cycle.

### B. Pretreatment

Before applying any fault-localization technique on the faulty execution, an important pretreatment takes place in our approach. Indeed, fault localization relies on the division of the trace into cycles. This can be done through a preliminary treatment of each execution trace that consists in two steps: first identifying the particular program counter (PC) that corresponds to the loop-header, and second slicing the trace before each of the occurrence of that particular PC.

*1) Detecting the likely loop-header:* Identifying the program counter corresponding to the loop-header is not easy, because, first, finding the main loop in the source code can be difficult. Second, even if the loop-header is identified, the compilation of the program may render the association with program counters difficult. For instance, more than one program counter may be associated with the loop-header. Moreover, if the compiler heavily optimizes the program, the association between machine instructions and program statements may be lost. For those reasons, based on the `while`-model (see Fig. 2), we designed a way to detect automatically the program counter that is likely to correspond to the loop-header. This automatic detection relies on three measures for each program counter $k$ in the execution trace $\sigma$. First, for a

```
s1();
while(condition){
 s2();
}
s3();
```

Fig. 2. `While`-model

program counter $k$ in the trace $\sigma$, the number of occurrences $Na(k)$ is defined as follows:

$$Na(k) = \sum_{i=1}^{|\sigma|} x_i \text{ where } x_i = \begin{cases} 1 & \text{if } \sigma_i = k, \\ 0 & \text{otherwise.} \end{cases}$$

where $\sigma_i$ is the $i$-th element in the trace $\sigma$.
Then, the average of the distance between its *consecutive* occurrences, noted $Da(k)$, is defined as follows:

$$\begin{aligned} Da(k) &= \frac{(o_2 - o_1) + \cdots + (o_n - o_{n-1})}{n-1} \\ &= \frac{\sum_{i=1}^{Na(k)-1} o_{i+1} - o_i}{Na(k) - 1}, \end{aligned}$$

where $o_1, o_2, \ldots, o_n$ are the indexes of the occurrences of $k$ in the order that they appear the trace $\sigma$, i.e., on the one hand, $\sigma_{o_1} = k$, $\sigma_{o_2} = k$, $\ldots$, $\sigma_{o_n} = k$ and on the other hand $o_1 < o_2 < \cdots < o_n$.
Finally, the index of the first occurrence of a symbol $k$ in the trace, noted $\mathsf{first}(k)$, is defined such that:

$$\mathsf{first}(k) = o_1 = \min\{i \in [0..|\sigma|] \mid \sigma_i = k\}.$$

The indicators $Na(k), Da(k)$, and $\mathsf{first}(k)$ are combined to form a score that allows us to rank program counters according to their likelihood to correspond to the loop-header. It is important to note that the first ranked PC is selected as the loop-header and used to divide the trace into cycles. The loop-header score is noted $\mathsf{lhscore(k)}$ and is defined as:

$$\mathsf{lhscore}(k) = \frac{Na(k) \times Da(k)}{\mathsf{first}(k)}$$

This score relies on two basic observations. First, loop-headers are repeated a lot of times ($Na(k)$ is high). Second, the main loop of a cyclic program has in average longer iterations than any other loop of the program (i.e., $Da(k)$ is high). Additionally, $\mathsf{first}(k)$ has two roles. On the one hand, it allows us to quickly discard active error loops, which are loops that are used to mark an erroneous sink state. On the other hand, it is a tie breaker when multiple symbols of the main loop occur the same number of times and with the same average distance between occurrences. Such ties happen quite often, because the loop-header is often coded into multiple machine statements.

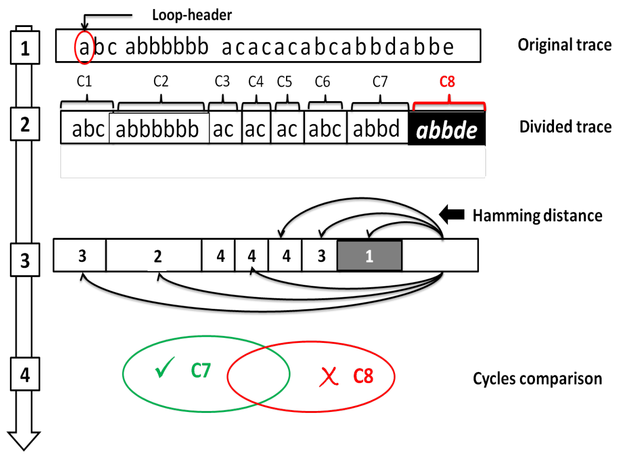

Fig. 3. Nearest neighbor fault-localization process

*2) Dividing the trace:* As defined in [10], given the likely loop-header $lh$, it is possible to find the range of indexes corresponding to each cycle:

$$C(\sigma, lh) = \left\{ \langle i, j \rangle \in [1..|\sigma|] \times [1..|\sigma|] \mid (1) \vee (2) \right\}$$

where:

(1) $j = |\sigma| \wedge \sigma_i = lh \wedge \forall k \in [1, |\sigma|] : \sigma_k \neq lh$,
(2) $i \leq j \wedge \sigma_i = \sigma_{j+1} = lh \wedge \forall k \in [i+1, j] : \sigma_k \neq lh$.

A cycle can be of two forms: it can be either a sequence of symbols starting on an $lh$-symbol and ending on a symbol preceding an $lh$-symbol without any $lh$-symbol in between, or, it starts on an $lh$-symbol and terminates on the last symbol of the trace without any $lh$-symbol in between.

### C. Nearest neighbor on a single trace

The nearest neighbor method relies on the comparison of spectra to identify the closest passing run to the failing one. In this section, we propose an adaptation of the classical method in the context of a single trace. As noted above, a run becomes a cycle. In other words, we consider the failing cycle $f'$ and a set of passing cycles $S'$. Therefore, we apply the formulas of the union and intersection models on the cycles of a given trace where we consider each cycle as a spectrum. The Hamming distance is used to identify the passing run closest to the failing one.

*1) Computing Hamming distances:* The passing run (cycle) that most corresponds to the failing run is found by measuring the similarity between each passing run and the failing run. The similarity between runs is measured using the Hamming distance. Recall that the Hamming distance consists in the difference between two binary vectors. Therefore, the smaller the distance is, the higher the similarity is.

For each cycle in trace, a binary vector is constructed. The Hamming distance is computed on the basic element of the trace, that is, in our context, the program counters (PCs).

Let $PCs = \{PC_1, PC_2, \ldots, PC_m\}$ be the set of different PCs in an execution trace, and let $C = \{C_1, C_2, \ldots, C_n\}$ be the set of different cycles in the trace. As shown in Table III,

TABLE III
CONSTRUCTION OF BINARY VECTORS

| | $C_1$ | $C_2$ | $\ldots$ | $C_n$ |
|---|---|---|---|---|
| $PC_1$ | $x_{1,1}$ | $x_{1,2}$ | $\ldots$ | $x_{1,n}$ |
| $PC_2$ | $x_{2,1}$ | $x_{2,2}$ | $\ldots$ | $x_{2,n}$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| $PC_m$ | $x_{m,1}$ | $x_{m,2}$ | $\ldots$ | $x_{m,n}$ |

$$M \text{ cycles} \left\{ \underset{\text{Cycle matrix}}{\overbrace{\begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,N} \\ \vdots & \vdots & \ddots & \vdots \\ x_{M-1,1} & x_{M-1,2} & \cdots & x_{M-1,N} \\ x_{M,1} & x_{M,2} & \cdots & x_{M,N} \end{bmatrix}}^{N \text{ program counters}}} \right. \quad \underset{\substack{\text{Error detection} \\ \text{vector}}}{\begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}}$$

Fig. 4. The cycles matrix and the error detection vector used in single-trace fault localization

for each cycle $C_j$ with $j \in [1..n]$, a binary vector of length $|PCs|$ is constructed, where, for all $i \in [1..m]$:

- $x_{i,j} = 1$ if the $PC_i$ appears in the cycle $C_j$;
- $x_{i,j} = 0$ if the $PC_i$ does not appear in the cycle $C_j$.

*2) Applying the union and intersection models:* Given this estimation, we can compute for each correct cycle the distance to the faulty one and select the closest match, that is, the cycle for which the Hamming distance is the lowest. Then both the union and intersection model are applied to find the suspect candidates.

Figure 3 illustrates the steps of our approach on an example. In this example, the failing run is the cycle $c_8$, and we have seven passing runs $\{C_1, C_2, C_3, C_4, C_5, C_6, C_7\}$. On the example, the cycle $c_7$ is the successful run cycle that has the highest similarity (or lowest distance) with $c_8$, in the example of Fig. 3. On these two cycles, the union and intersection models are applied to localize the fault.

### D. Suspiciousness ranking using a single trace

Adapting suspiciousness ranking should be straightforward when considering cycles as independent runs. However, the adaptation must consider the important size of the execution trace. One way to reduce the problem is to limit the number of cycles to analyze. Our suspiciousness-based method for a single faulty trace consists of two steps: a coarse filtration of cycles and the suspiciousness scoring.

*Coarse filtration:* Embedded software traces tend to present very distinct cycles corresponding to very distinct behaviors. Indeed, cyclic programming tends to distribute different tasks over different cycles. It is our belief that such disparity may hinder the search of the fault using suspiciousness ranking. Consequently, the selection of cycles to analyze is important To mitigate this phenomenon, our coarse filtration method first

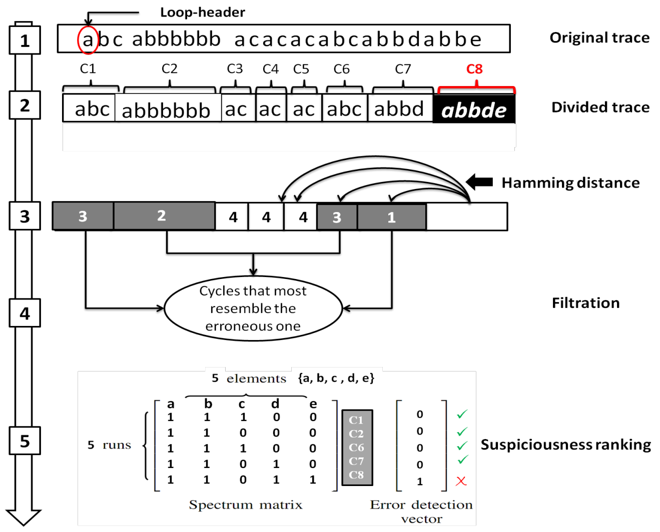

Fig. 5. Suspiciousness ranking using a single trace

determines, using the Hamming distance estimation discussed earlier, the cycles that most resemble the erroneous one.

*Suspiciousness scoring:* Computing the suspiciousness of a spectrum element relies on comparing the error detection vector with the vector of presence of the given spectrum element over the cycles. In case of an execution trace, the spectrum element is a program counter (see Fig. 4).

Figure 5 summarizes the steps needed to rank program counters in a faulty execution trace: from the cycle detection to the actual scoring. One final step is needed to point out the fault in the program, that is, to associate the program counters with the source code statement from which it originates.

## V. Experimental Evaluation of Loop-Header Localization

In this section, we independently evaluate our approach to automatically localize the loop-header. This evaluation consists in comparing our automatic detection of the loop-header on several programs with an oracle.

*Programs and Traces:* We use execution traces that come from 14 cyclic programs, denoted $P_i$, where $i$ is the number of the program. Table IV indicates for each program the number of lines in the source code, and the number of lines in the collected execution trace. Each program contains a main `while (true)`-loop, which is repeatedly executed. In order to ensure that the loop-header localization is not biased regarding fault localization, we chose programs that differ from the faulty programs used in Section VI. Note however that our fault-localization approach relies on the loop-header localization, and that the fault localization fails if the loop-header is not correctly identified.

*Oracle:* The oracle analyzes the execution trace and the source code for each program used in the experimental evaluation to identify the loop-header. The oracle in our experimental evaluation is an engineer.

TABLE IV
Evaluation Results for the Loop-Header Localization

| Program | #Lines | # Lines in trace | lhscore Rank of lh | lhscore Verdict |
|---|---|---|---|---|
| $P_1$ | 154 | 24747 | 1 | ● |
| $P_2$ | 162 | 614391 | 1 | ● |
| $P_3$ | 358 | 11593 | 1 | ● |
| $P_4$ | 254 | 5110 | 1 | ● |
| $P_5$ | 222 | 5789 | 1 | ● |
| $P_6$ | 61 | 2681 | 1 | ● |
| $P_7$ | 126 | 2000 | 1 | ● |
| $P_8$ | 107 | 11797 | 1 | ● |
| $P_9$ | 87 | 6701 | 1 | ● |
| $P_{10}$ | 62 | 2740 | 1 | ● |
| $P_{11}$ | 68 | 924 | 1 | ● |
| $P_{12}$ | 61 | 5471 | 1 | ● |
| $P_{13}$ | 31 | 1702 | 1 | ● |
| $P_{14}$ | 47 | 802 | 1 | ● |
| **Successes** | | | 14 | |
| **Failures** | | | 0 | |

*Experiments:* For each program used in the experimental evaluation we observe the program counter selected as loop-header by our approach and we compare it with the one defined by the oracle. If they are the same, then the localization of the loop-header is a success. Otherwise, if they are different, we analyze the ranking proposed by our approach to determinate the rank of the loop-header in the list of candidates. Our approach provides a margin of error in the loop-header localization. Therefore, if the rank of the loop-header is lesser than or equal to 3 [2], the localization of the loop-header is considered a success. However, if the rank of the loop-header is greater than three, then the localization of the loop-header is a failure. According to our experimental results shown in Table IV, we observe that our automatic loop-header localization approach succeeds in 100% of cases. Note that the loop-header is always ranked first.

## VI. Experimental Evaluation of Fault Localization

In this section, we present the evaluation of the proposed automatic fault-localization approaches for a single trace, namely the adapted nearest neighbor (Section IV-C) and the adapted suspiciousness ranking (Section IV-D). Our evaluation essentially consists in applying our approach to known erroneous programs and to measure the quality of the diagnosis emitted by our fault-localization approaches. The evaluation is performed using our tool named CoMET.

### A. CoMET

CoMET is a tool written in Java in about 12,000 LOC that implements several approaches. As illustrated in Fig. 6, it

[2]Our experiments show that when the loop-header is not in the three first statements, then it is much further in the program. In this (rare) case, a manual analysis is more relevant.

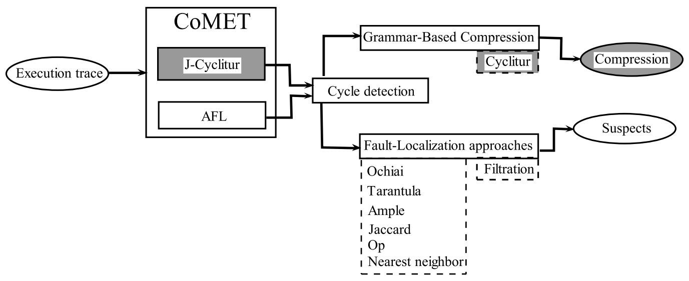

Fig. 6. CoMET workflow

takes as input an execution trace file. As presented in [10], our tool assists in the trace analysis, by compressing the execution trace. To generate a compression, CoMET uses Cyclitur algorithm [10], which is inspired from Sequitur [26]. Note that our tool uses a grammar-based compression to identify the repeated sequences in the execution trace. The generated compression provides a comprehensive view of the execution trace. Then, the compression allows to identify specific cycles and specific sequences of cycles, e.g., the most repeated cycles in the trace.

Moreover, the AFL approaches were implemented and integrated in CoMET. First CoMET uses our compression approach to divide the trace into cycles. Then, it applies an AFL method chosen by the user, either the adapted nearest neighbor or the adapted suspiciousness ranking in which case the user also needs to specify the similarity coefficient to use (among Tarantula's, Jaccard, AMPLE, Ochiai and $O^p$) and the coarse filtration parameter as a percentage (cf. Section IV-D). By default, the Ochiai coefficient is used (as experiments show that it provides better results), and the percentage of the coarse filtration is set to 30%. Finally, CoMET outputs the results. For the nearest neighbor approach, the results consist of two sets: one containing potential missing statements and one containing potentially superfluous statements. For the suspiciousness ranking method, the statements are given in decreasing order of suspiciousness.

## B. Programs and Errors

The traces used to evaluate our approach come from 13 embedded programs, which are provided by STMicroelectronics and EASii IC (see also Table V). Each of these 13 programs contains a fault that is commonly found in embedded software development. Also each program allows the user to interact with the microcontroller, by using the LCD-Screen or the 4 microcontroller buttons. When a button is pressed, the microcontroller executes a specific processing. If the processing is finished without error, the LED corresponding to the pressed button is turned-on, and a message is displayed on the LCD-screen.

In the following we denote program number $i$ by P$i$. The programs P1, P2, and P12 exploit the call stack. When button B1 is pressed an element is pushed onto the stack, and if button B2 is pressed, an element is pulled from the stack. The bug in $P1$ consists in not checking if the stack is empty before pulling an element. Then, if the user presses B2 more times than B1 the execution crashes. P2 does not take into account the stack overflow. P2 uses a recursive function to fill the stack; however, when the function is called with a big enough argument, the stack overflows and it generates a memory interrupt.

TABLE V
INFORMATION ABOUT PROGRAMS AND TRACES.

| Program | Size (MB) | # Files | Trace size (MB) | # Lines in trace |
|---|---|---|---|---|
| P1 | 14.4 | 165 | 95.2 | 1048579 |
| P2 | 14.4 | 165 | 86.9 | 1,045,869 |
| P3 | 143 | 151 | 25.1 | 280,049 |
| P4 | 218 | 152 | 21.1 | 237,062 |
| P5 | 143 | 151 | 18.7 | 207,914 |
| P6 | 140 | 151 | 21.7 | 240,829 |
| P7 | 207 | 158 | 94.3 | 104,8577 |
| P8 | 139 | 151 | 21.8 | 235,788 |
| P9 | 218 | 152 | 22.4 | 241,404 |
| P10 | 143 | 151 | 25.2 | 280,298 |
| P11 | 207 | 158 | 95.6 | 1,048,576 |
| P12 | 14.4 | 165 | 92.2 | 1,048,573 |
| P13 | 50 | 164 | 84.7 | 1,047,568 |

Programs P3 and P5 calculate the minimum $m$ and the maximum $M$ of two integers $a$ and $b$, and check if the inequality $M \geq m$ holds. Note that $a$ (resp. $b$) is the number of times button B1 (resp. B2) is pressed. In P3, the maximum function always returns the first argument, that is a fault. Using P3, if the user presses B1 one time and B2 two times, the minimum is 1 and the maximum is also 1 instead of 2. In P5, the fault is in the minimum function and consists in multiplying the value of the second argument by ten before comparing the two values. Using P3, if the user presses B1 two time and B2 one time, the maximum is 2 and the minimum is also 2 instead of 1.

The bug in P4 consists in using a memory address as argument instead of an integer variable, i.e., let be `value_check (int x)` a function, and `v` an integer variable, the fault is the use of `value_check(&v)` instead of `value_check(v)`.

Program P6 contains a common mistake in C programs about string comparison. To compare two strings `s1` and `s2`, some sort of string comparison function like `strcmp(s1,s2)` should be used. In $P6$, the comparison between the strings `s1` and `s2` is done using the statement `s1 == s2`. Such statement compares memory addresses instead of comparing the string values. This fault leads to an interrupt at some point in the execution.

The `lcd_check` and `led_check` functions in P7 and P11 respectively, generate an interrupt after $n$ execution times of the main loop.

The function `value_check(float v)` in P8 checks if `v` $> 0.3$. At some point of the execution of the program, the value of `v` is truncated, and, later on, the value of `v` is passed to `value_check`, i.e. if `v` was 0.4, the cast sets it to 0 and the call to `value_check` fails.

The oversight of a `break` statement in a `case` is the bug in P9. If for the `case 1`, which executes "`v = v * 3`", the `break` is forgotten, the program will execute also the instruction "`v = v * 2`" of `case 2`, thus the value of $v$ is multiplied by six instead of three.

A `for`-loop in the program P10 contains one too many "`;`" as illustrated in the following example: "`for (i = 0; i < 10; i++);`". In this case any instruction intended for the `for`-loop will be executed only once when `i` is 10.

A *watchdog* is a mechanism that triggers a system reset if the main program neglects to check in regularly with the watchdog, because of some error. Using P13, if the user presses B3 button the regular check-ins with the *watchdog* are disabled and a system reset is triggered, which generates a hardware interrupt. This failure was reported to us as being known to be hard to debug due to the delay between the execution of the faulty statement and the hardware interrupt.

One at a time, each program is downloaded on a STM32F107 EVAL-C microcontroller board and executed. Then, the execution trace is retrieved using a Keil UlinkPro probe [5], and saved in *CSV* format. The trace file contains, for each instruction, its *index*, which is an ID, the *time* when it was executed, its corresponding *assembly instruction* and the *program counter (PC)*.

### C. Results

A preliminary step to our (adapted) fault-localization approaches is the loop-header localization. The experiments with the 13 faulty execution traces agree with the independent evaluation in Section V. Indeed, on each of the 13 traces, the proposed loop-header localization effectively detects the correct loop-header in the trace.

*1) Evaluation Method:* To evaluate our two approaches, we applied the six following *adapted* AFL methods to each of the 13 execution traces:

- the nearest neighbor,
- the suspiciousness ranking with the Tarantula coefficient,
- the ranking with the Jaccard index,
- the ranking with the AMPLE coefficient,
- the ranking with the Ochiai coefficient, and
- the ranking with the $O^p$ coefficient.

It is important to note that the original versions of the mentioned AFL methods does not work in our context where only one faulty trace is available.

The *expense* of locating a fault $E$ depends on the number of inspected statements $I$ and the total number of statements $T$ in the program. The *expense* is defined as follows:

$$E = \frac{I}{T}.$$

The expense of locating a fault with the nearest neighbor method depends on the number of statements in the output and the order in which they are inspected. We choose to inspect the statement in the reverse order of the trace, which is common in manual debugging. For locating a fault with suspiciousness ranking, the number of statements to inspect is simply the number of correct statements with a higher rank than the faulty statement. In case of a tie in the ranking, the strategy used for the nearest neighbor method is applied; statements are inspected in the reverse order of the trace.

*2) Analysis:* Figure 7 shows the results of the experimental evaluation, where the axis of abscissa represents the programs from P1 to P13, and the axis of ordinates represents the expense for the fault localization. For each program we calculate the expense using the Nearest Neighbor, Tarantula, Jaccard, AMPLE, Ochiai and $O^p$ methods.

First, consider the spikes where the expense is very high. To localize the fault, it was required to analyze the traces of P3 and P4 in details (expense above 75%), except with the nearest neighbor method and the $O^p$ suspiciousness ranking where the expense is less than 5%. The main reason behind this bad fault localization is that P3 and P4 propagate a state between cycles and that most suspiciousness rankings seems ill-adapted in this case. For P2 and P12, $O^p$ ranks very badly the fault, and as a result the engineer needs to inspect the whole trace and the source code (100%). In those execution traces, almost all statements obtain similar suspiciousness scores with $O^p$. A possible reason is that the programs are too different from the model ITE2 for which $O^p$ is optimal. For P13, the nearest neighbor and Ample also require a detailed analysis of the trace and the source code, with expenses above 80%.

Then, globally, we observe that the nearest neighbor and the $O^p$ suspiciousness ranking require significantly more manual inspection to localize faults than the other suspiciousness rankings. In particular, on P1, P2, P5, P6, P8 and P11, while the fault localization expense with nearest neighbor rises up to 35%, the expenses with Tarantula, Jaccard, Ample and Ochiai stay under 5%. We also observe that in 75% of cases, at most, we analyze strictly less than 20% of the executed statements. Note that compared to the other techniques, the Ochiai method allows a fault localization with a less expensive analysis.

### D. Evaluation of Coarse filtration

Figure 8 shows the result of the filtration evaluation. The graph represents the expense of locating the fault in program $P6$, using the Ochiai coefficient. When our fault-localization approach uses 10% of cycles of the trace, the engineer needs to analyze 5% of the source-code to localize the fault. The execution trace generated by the program $P6$ contains 323 different cycles. Thus, by using 32 cycles, the engineer analyzes 5% of the source code before he localizes the fault. However, by using 30% to 100% of cycles, the expense of locating the fault remains stable (less than 4% of the source code to analyze). It is important to note that the experimental evaluation concerns the 13 programs presented in this section. The fault localization in the other 12 programs requires the same expense to locate the fault since at least 10% of cycles are used.

## VII. Threats to Validity and Discussion

*Threats to Validity:* The first threat to validity of our evaluation might be its limited size. We experimented only

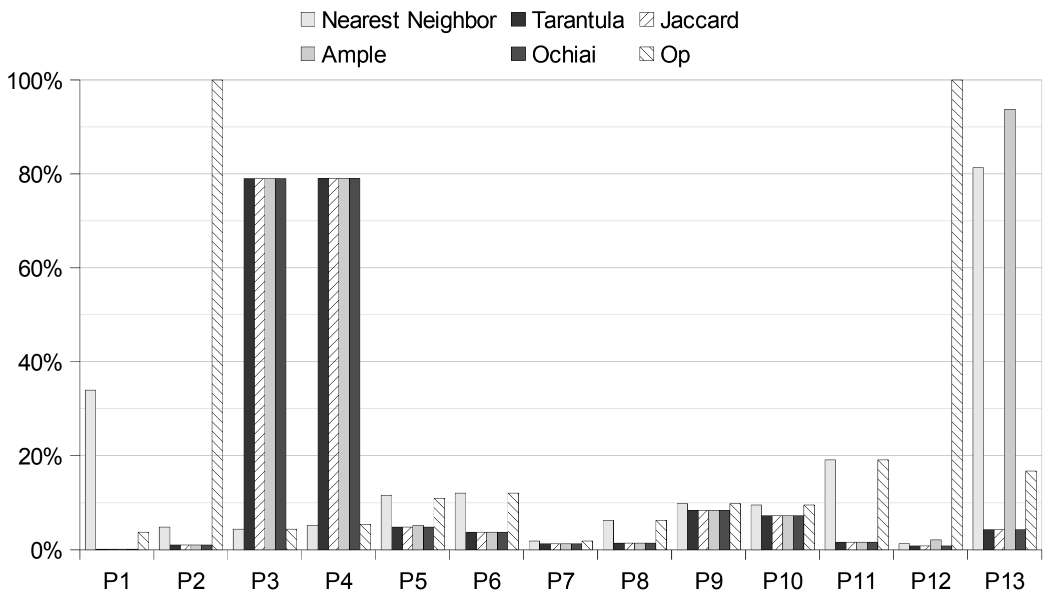

Fig. 7. Expenses when localizing faults for the 13 programs and the six methods

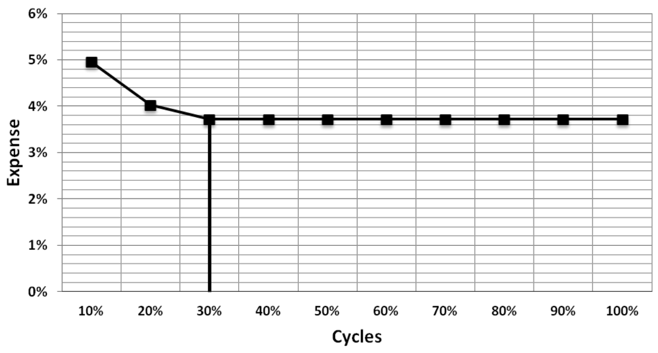

Fig. 8. Fault Localization using cycle filtration and Ochiai in $P6$

on thirteen faulty programs. However, each faulty program was chosen either by our industrial partners or because they are representative of common faults in the development of programs for microcontrollers. Consequently, we believe our benchmark is reasonable for our purposes. Another potential issue concerning the evaluation is the size of the considered programs. However, automatic fault localization arithmetically performs better, in terms of expense, with bigger programs. Indeed, as noted in [27], because of the separation of concerns in larger programs, fault localization techniques are able to quickly discard all chunks of the code that are completely unrelated to the fault.

The second threat to validity of our evaluation is that the proposed methods are useful to detect a single fault in a program and were not designed to detect multiple faults. However, a previous study [8] with Tarantula suggested that a single-fault localization technique may be used efficiently to detect multiple faults.

The third threat to validity comes from the loop-header localization, which is the basis of fault localization. An incorrect loop-header could lead to irrelevant and misleading results. However, in our experiments, such a case did not occur and a manual loop-header detection is always possible.

*Discussion:* A last point we would like to address is the translation of program counters into actual program statements. As noted earlier, embedded machine code is heavily optimized. Consequently, the actual association between machine code, on which the failure occurs, and the source code, which should be debugged, is very limited. The task of finding the statement of source code corresponding to some machine instruction is consequently in itself very hard. Automated or manual fault localization needs this task to be done precisely. However, given a trace in terms of program counters, manual approaches and automatic approaches differ greatly in the number of machine instruction to translate back into source statements. On the one hand, standard manual fault localization approaches would likely have to translate a lot of machine instructions to source statements, browsing the trace from the failure back to the error. On the other hand, our approach will limit this translations to a few instructions by finding first the most likely faulty machine instructions before actually translating them to source statements. Those elements lead us to believe that automatic fault localization will greatly speedup debugging in the embedded context and in any other context where the association between the collected trace and the source code is not a given.

## VIII. Conclusion and Perspectives

*1) Conclusion:* Debugging embedded software remains a difficult task. In fact, embedded software consists of low-level code with a tight integration in a unique environment in terms of sensors, outputs, etc. This makes both static and dynamic analyses very hard. Recent microcontrollers are able to record execution traces. However, the size of the collected traces makes the manual analysis tedious. In this paper, we propose a complete approach to help locating a fault in an embedded program based on a single failing execution trace.

In this approach, we take advantage of the cyclic nature of most embedded programs to adapt well known fault-localization techniques. The method first detects cycles in the execution trace automatically. Then, this work adapts known fault localization techniques to the context of a single failing execution trace. Those techniques take multiple passing and failing executions of the program as input. Our adapted techniques consider cycles as program runs. Finally, the paper presents an adapted nearest neighbor method and an adapted suspicious ranking method compatible with any similarity coefficient (e.g., Ochiai or Tarantula's).

The presented approaches are implemented in a tool named CoMET and evaluated on several faulty programs. The evaluation shows promising results, in particular for the Ochiai suspiciousness ranking. Indeed, by using this particular method, the user is able to find the faulty statement by inspecting in most cases less than 5% of the program.

*2) Perspectives:* In the future, we will apply our methods to bigger programs and different faults. In particular, it would be interesting to study programs where an erroneous state propagates through multiple cycles before a failure occurs. This kind of faults raises an interesting challenge in the context of fault localization based on a single erroneous trace. Indeed, we need to identify the cycle that we can trust to be correct in order to detect the fault. We believe that identifying cycles and comparing them will allow us to help find such faults rapidly. Moreover, it must be noted that embedded software is not the only type of systems with a cyclic and long running behavior. For instance, most server-side applications satisfy these same characteristics. That is why we are also interested in applying and adapting our approach to other cyclic systems.

## Acknowledgment

This work has been funded by the French-government Single Inter-Ministry Fund (FUI) through the IO32 project (Instrumentation and Tools for 32-bit Microcontrollers). The authors would like to thank our project partners STMicroelectronics, AIM and EASii IC for their help in providing programs and improving the relevance of our approach in an industrial context.

Laboratoire d'Informatique de Grenoble is partner of the LabEx PERSYVAL-Lab (ANR–11-LABX-0025).

## References

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "An evaluation of similarity coefficients for software fault localization," in *12th Pacific Rim International Symposium on Dependable Computing (PRDC)*. IEEE Computer Society, 2006, pp. 39–46.

[2] J. S. Parab, V. G. Shelake, R. K. Kamat, and G. M. Naik, *Exploring C for Microcontrollers: A Hands on Approach*. Springer, 2007.

[3] A. Rohani and H. Zarandi, "An analysis of fault effects and propagations in AVR microcontroller ATmega103(L)," in *International Conference on Availability, Reliability and Security (ARES)*, Mar. 2009, pp. 166 –172.

[4] Trace macrocells (ETM). ARM. [Online]. Available: http://www.arm.com/products/system-ip/debug-trace/trace-macrocells-etm/index.php

[5] Keil UlinkPro. [Online]. Available: http://www.keil.com/ulinkpro/

[6] ST Microelectronics. [Online]. Available: http://www.st.com/internet/evalboard/product/219866.jsp

[7] M. Renieris and S. P. Reiss, "Fault localization with nearest neighbor queries," in *18th IEEE International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 2003, pp. 30–39.

[8] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *24th International Conference on Software Engineering (ICSE)*. New York, NY, USA: ACM, 2002, pp. 467–477.

[9] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, pp. 11:1–11:32, Aug. 2011.

[10] A. Amiar, M. Delahaye, Y. Falcone, and L. du Bousquet, "Compressing microcontroller execution traces to assist system analysis," in *IESS*, ser. IFIP Advances in Information and Communication Technology, G. Schirner, M. Götz, A. Rettberg, M. C. Zanella, and F. J. Rammig, Eds., vol. 403. Springer, 2013, pp. 139–150.

[11] W. E. Wong, Y. Shi, Y. Qi, and R. Golden, "Using an RBF neural network to locate program bugs," in *19th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE CS, 2008, pp. 27–36.

[12] T. Denmat, M. Ducassé, and O. Ridoux, "Data mining and cross-checking of execution traces. a re-interpretation of Jones, Harrold and Stasko test information visualization," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, T. Ellman and A. Zisman, Eds. ACM Press, November 2005.

[13] F. Tip, "A survey of program slicing techniques," *J. Prog. Lang.*, vol. 3, no. 3, 1995.

[14] H. Agrawal, J. Horgan, S. London, and W. Wong, "Fault localization using execution slices and dataflow tests," in *Sixth International Symposium on Software Reliability Engineering (ISSRE)*, 1995, pp. 143–151.

[15] S. Kusumoto, A. Nishimatsu, K. Nishie, and K. Inoue, "Experimental evaluation of program slicing for fault localization," *Empirical Softw. Engg.*, vol. 7, no. 1, pp. 49–76, Mar. 2002.

[16] X. Zhang, N. Gupta, and R. Gupta, "Pruning dynamic slices with confidence," *SIGPLAN Not.*, vol. 41, no. 6, pp. 169–180, Jun. 2006.

[17] R. Gore and P. F. Reynolds Jr., "Causal program slicing," in *ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation (PADS)*. IEEE CS, 2009, pp. 19–26.

[18] G. K. Baah, A. Podgurski, and M. J. Harrold, "Mitigating the confounding effects of program dependences for effective fault localization," in *19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE)*. ACM, 2011, pp. 146–156.

[19] R. Yu, L. Zhao, L. Wang, and X. Yin, "Statistical fault localization via semi-dynamic program slicing," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, 2011, pp. 695–700.

[20] H. Cleve and A. Zeller, "Locating causes of program failures," in *27th International Conference on Software Engineering (ICSE)*. ACM, 2005, pp. 342–351.

[21] A. Zeller, "Isolating cause-effect chains from computer programs," in *10th ACM SIGSOFT symposium on Foundations of Software Engineering (FSE-10)*. ACM, 2002, pp. 1–10.

[22] T. Reps, T. Ball, M. Das, and J. Larus, "The use of program profiling for software maintenance with applications to the year 2000 problem," in *ACM Software Engineering Notes*. Springer-Verlag, 1997, pp. 432–449.

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

[24] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: problem determination in large dynamic internet services," in *International Conference on Dependable Systems and Networks (DSN)*, 2002, pp. 595–604.

[25] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight defect localization for Java," in *19th European Conference on Object-Oriented Programming*. Springer-Verlag, 2005, pp. 528–550.

[26] C. G. Nevill-Manning and I. H. Witten, "Identifying hierarchical structure in sequences: A linear-time algorithm," *Journal of Artificial Intellgence Research (JAIR)*, vol. 7, pp. 67–82, 1997.

[27] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *20th IEEE/ACM international Conference on Automated Software Engineering (ASE)*. ACM, 2005, pp. 273–282.