Spectrum-Based Fault Localization for Diagnosing Concurrency Faults

Feyzullah Koca[1,2], Hasan Sözer[2], and Rui Abreu[3]

[1] TUBITAK BILGEM, Information Technologies Institute, Kocaeli, Turkey
[2] Department of Computer Science, Özyeğin University, İstanbul, Turkey
{feyzullah.koca,hasan.sozer}@ozyegin.edu.tr
[3] Faculty of Engineering, University of Porto, Portugal
rui@computer.org

Abstract. Concurrency faults are activated by specific thread interleavings at runtime. Traditional fault localization techniques and static analysis fall short to diagnose these faults efficiently. Existing dynamic fault-localization techniques focus on pinpointing data-access patterns that are subject to concurrency faults. In this paper, we propose a spectrum-based fault localization technique for localizing faulty code blocks instead. We systematically instrument the program to create versions that run in particular combinations of thread interleavings. We run tests on all these versions and utilize spectrum-based fault localization to correlate detected errors with concurrently executing code blocks. We have implemented a tool and applied our approach on several industrial case studies. Case studies show that our approach can effectively and efficiently localize concurrency faults.

Keywords: Debugging, multithreading, concurrency faults, thread safety, dynamic analysis, spectrum-based fault localization.

1 Introduction

Concurrency faults are activated by specific thread interleavings at runtime, which makes them hard to detect by testing since they do not deterministically lead to an error. Traditional fault localization techniques and static analysis fall short to detect these faults efficiently. Existing dynamic fault-localization techniques focus on pinpointing data-access patterns that are subject to concurrency faults [9,12,19,20,25]. Although these techniques are effective in capturing faulty data-access patterns, the corresponding code blocks should still be identified by the programmer to locate and fix the defect. More importantly, not all concurrency faults are related to data access and shared memory. There exist various type of shared resources other than memory. Concurrent access to these shared resources (e.g., file access) can also lead to errors.

We have applied spectrum-based fault localization [1, 3, 10, 11] for directly pinpointing code blocks, of which multi-threaded execution leads to concurrency errors. In our approach, we systematically instrument a multi-threaded subject

H. Yenigün, C. Yilmaz, and A. Ulrich (Eds.): ICTSS 2013, LNCS 8254, pp. 239–254, 2013.

program to force context switch within different code blocks. As such, each version turns out to be actually the same program that is executed with a different combination of thread interleavings. We run tests on all the generated versions and utilize spectrum-based fault localization to correlate detected errors with concurrently executing code blocks. The result is a ranking of code blocks with respect to the probability that their re-entrance causes the detected errors. These code blocks should be analyzed by the programmer to introduce thread-safety.

We have implemented a tool, dubbed SCURF, and applied our approach on several industrial case studies. Case studies show that our approach can effectively and efficiently localize concurrency faults. The contributions of this paper are threefold

- We introduce a novel approach for localizing concurrency faults by means of spectrum-based fault localization techniques;
- We developed a toolset, *SCURF* that provides automation for our approach;
- We discuss our experiences in applying our approach in several industrial software projects.

The remainder of this paper is organized as follows. Section 2 provides background on spectrum-based fault localization. Section 3 introduces a motivating example. We introduce our approach in Section 4. Industrial case studies and the evaluation of the approach are presented in Section 5. Related studies are summarized in Section 6. Finally, conclusions are provided in Section 7.

2 Background: Spectrum-Based Fault Localization

The process of pinpointing the fault(s) that led to the observed symptoms (failures/errors) is called fault localization. Depending on the amount of knowledge that is required about the system's internal component structure and behavior, the most predominant approaches can be classified as *i)* statistical approaches or *ii)* reasoning approaches (for an overview of approaches, see [2]). The former approach uses an abstraction of program traces, dynamically collected at runtime (also known as program spectra [8]), to produce a list of likely candidates to be at fault [3, 10, 11], whereas the latter combines a static *model* of the expected behavior with a set of observations to compute the diagnostic report [14]. In this paper, we use a statistical technique, in particular spectrum-based fault localization [3,10] due its effectiveness in locating faults, while entailing low time and space complexity [2].

Spectrum-based fault localization (SFL) is a dynamic program analysis technique. The basic idea of SFL is that comparing the program behavior over multiple test runs can indicate which program components may be likely to contribute to an observed program failure. In the following, we assume that a program $\mathcal{P}$ comprises a set of components $\mathcal{C}$ and is executed using a set of test cases $\mathcal{T}$ that either pass or fail, with $M = |\mathcal{C}|$ and $N = |\mathcal{T}|$, respectively. Program (component) activity is recorded in terms of program spectra [3, 10, 11]. These data are collected at run-time and typically consist of a number of counters or flags for

the different components of a program. Usually, the so-called *hit spectra* is used, indicating whether a component was involved in a (test) run or not.

$$
\begin{array}{c}
\qquad\qquad M \text{ components} \quad \text{error vector} \\
N \text{ spectra} \left[\begin{array}{cccc|c}
a_{11} & a_{12} & \dots & a_{1N} & e_1 \\
a_{21} & a_{22} & \dots & a_{2N} & e_2 \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
a_{M1} & a_{M2} & \dots & a_{MN} & e_N
\end{array}\right] \\
\qquad\qquad\quad s_1 \quad s_2 \quad \dots \quad s_N
\end{array}
$$

Fig. 1. The ingredients of fault diagnosis

Both spectra and pass/fail information is input to SFL. The combined information is expressed in terms of the $N \times (M + 1)$ *activity matrix* A. An element a_{ij} is equal to 1 if component j took part in the execution of test run i, and 0 otherwise. The rightmost column of A, the error vector e, represents the test outcome. The element $e_i = a_{i,m+1}$ is equal to 1 if run i *failed*, and 0 if run i *passed*. For $j \leq M$ and $i \leq N$, the row A_{i*} indicates whether a component was executed in run i, whereas the column O_{*j} indicates in which runs component j was involved.

In SFL one measures the similarity between the error vector e and the activity profile vector A_{*j} for each component j (see Figure 1). This similarity is quantified by a *similarity coefficient*, s_j. In this work, we employ the Ochiai similarity coefficient, which was previously identified as the best coefficient to be used for SFL [3].

$$s_j = \frac{n_{11}(j)}{\sqrt{(n_{11}(j) + n_{01}(j)) \cdot (n_{11}(j) + n_{10}(j))}} \tag{1}$$

where $n_{11}(j)$ is the number of failed runs in which part j is involved, $n_{10}(j)$ is the number of passed runs in which part j is involved, and $n_{01}(j)$ is the number of failed runs in which part j is not involved, i.e., formally and referring to Figure 1,

$$
\begin{aligned}
n_{01}(j) &= |\{i \mid a_{ij} = 0 \wedge e_i = 1\}| \\
n_{10}(j) &= |\{i \mid a_{ij} = 1 \wedge e_i = 0\}| \\
n_{11}(j) &= |\{i \mid a_{ij} = 1 \wedge e_i = 1\}|
\end{aligned}
$$

The Ochiai coefficient s_j associated with each component $C_j \in \mathcal{C}$ indicates the correlation between the executions of C_j and the observed incorrect program behavior. Applying the hypothesis that closely correlated components are more likely to be relevant to an observed misbehavior, s_j can be reinterpreted as "fault probability" and components can be listed (i.e., ranked) in order of likelihood to be at fault. Note that $n_{11}(j) + n_{10}(j)$ equals the number of runs in which part

j is involved, whereas $n_{11}(j) + n_{01}(j)$ equals the number of failed runs, which is the same for all j.

We adapted SFL for localizing concurrency faults. For this purpose, we modified the collected hit spectra and the analysis process. In the following, we present a motivating example, followed by our approach illustrated on this example.

3 Motivating Example

In this section, we present a running example for illustrating the problem and our solution. A C function, *addNumberToPhoneList*, is shown in Listing 1.1, which adds a name to a phone book. If the phone book already exists, the function just opens it (Line 12) and adds the name that is provided as an argument (Line 16). Otherwise, the function first creates the phone book (Line 6) and then adds the name into this newly created phone book.

Listing 1.1. The motivating example

```
1int addNumberToPhoneList(char *name, char *number) {
2  FILE* fp;
3  int retVal = -1;
4
5  if(FALSE == doesPhoneListExist) { /* Component 1 */
6    fp = createPhoneBook();
7    if(NULL != fp) { /* Component 2 */
8      doesPhoneListExist = TRUE;
9    }
10 }
11 else { /* Component 3 */
12   fp = openPhoneBook();
13 }
14
15 if(NULL != fp ) { /* Component 4 */
16   retVal = addPhoneBook(fp, name, number);
17   (void) closePhoneBook(fp);
18 }
19 return retVal;
20}
```

In a single-threaded environment, *addNumberToPhoneList* function works as expected. However, the function is subject to a concurrency fault when it is executed concurrently by multiple threads. To illustrate this problem, we have executed the function using three threads calling the *addNumberToPhoneList* function to add different entries. As a result, we would expect the phone book to contain three names. Most of the times, this was indeed the case. However, there were executions where the phone book had less than three entries. The reason is the concurrent execution of the function: a context switch can occur when a

thread is in *component 1* (lines 5-10)[1]. This component creates a phone book and sets the global variable *doesPhoneListExist* to *true* to eliminate the need for creating a phone book again. Concurrent execution of *component 1* leads to an error because every newly created phone book deletes (overrides) the old one.

In a multi-threaded program, context switches may take place at any time depending on interrupts, operating devices, and the operating system scheduler. Although a program is not subject to an error in a single-threaded execution, multi-threaded execution can, in fact, lead to concurrency errors by enabling (uncontrolled) multiple entries to a component. This is a common issue we have observed in the industry when single-threaded legacy software is adopted within the context of multi-threaded software systems. The legacy software has not been developed with multi-threaded execution in mind, and testing does not always reveal the impact of re-entrance to the employed functions.

We propose and evaluate an approach to automatically detect errors and localize faults in multi-threaded execution of a program. We assume that the program itself is fault-free. As such, any detected error is due to a concurrency fault, caused by the multi-threaded execution of the program. Our approach is explained in the following section.

4 The Approach

Our approach is based on systematically instrumenting the program to trigger a context switch in different components. This enables us to test the same program in different thread interleavings, potentially triggering an error. We apply spectrum-based fault localization to reveal the particular thread interleavings of faulty components that lead to the detected errors. We have developed a toolset called *SCURF* to automate our approach[2], which is realized in three steps as described in the following.

In the first step, the program under test is instrumented to generate different versions each of which execute in different thread interleavings. At this step, the program code is instrumented also to collect spectra information at runtime. Second, each version is tested being subject to the same test suite. Program-spectra are collected for the number of re-entries to each component within a function. Third, the collected spectra are analyzed and correlated with the detected errors. All the components are ranked with respect to the probability that they are subject to a concurrency fault as the cause of the detected errors. These components should be further analyzed by the programmer and possibly considered for introducing thread-safety. In the following subsections, we explain the three steps of the approach in more detail.

[1] We refer to code blocks (encapsulated in while, for, if, else statement etc.) as *components* throughout the paper.

[2] SCURF currently supports C make file projects that are deployed on Linux-based operating systems only.

4.1 Step I. Code Instrumentation

In this step, the source code of the program is instrumented to collect program spectra at runtime and to control the scheduling of threads. The collected program spectra record the number of entries made to each component. To control the scheduling of threads, extra code is inserted at the beginning of each component that optionally[3] yields the current thread and forces a context switch. The instrumented code does not change the behavior of the program other than the thread interleavings.

SCURF inserts *sched yield* statements to force context switch at a component. This does not guarantee, in all cases, the scheduler to switch to another thread. We have utilized *usleep* statements in some Linux distributions instead. The code insertion for context switch is performed for each combination of components. That means that the instrumentation step generates $O(2^n)$ versions for a function with n components. However, in practice we have seen that the execution of $O(n)$ versions (context switch at one of the components each time) is usually enough to activate a concurrency fault.

4.2 Step II. Test Case Execution

In this step, the generated versions are executed being subject to the same test suite. We assume that a test oracle and test case(s) exist. For each test, a different thread interleaving occurs and program spectra are collected regarding the number of entries made to each component.

An example result, regarding the *addNumberToPhoneList* running example, for this step is presented in Table 1. Hereby, each row of this table represents a test run. The table is separated into three parts. The first part shows for each component, if a context switch is enforced or not. For example, for the first line, we know that context switch will be forced only in the first component which corresponds to the first *if* statement in the original code. The second part of the table shows the number of entries made to each component during the test run. For example, from the first row we can see that in test run 1, the first, the second and the fourth components were executed three times, whereas the third component was not executed at all. The third, and the final part/column shows the error vector, i.e., whether an error was detected during the corresponding test run or not.

The table is generated incrementally. First, only one component is influenced at a time (test runs 1 through 4 in the example). Similarity calculation is applied for only these set of runs to check if there are significant differences in rankings. We name this step as *level-1*. Depending on the available resources and significance of the results, SCURF can move forward with *level-2*, in which context switch is forced in two of the components at each test run (test runs 5 through 10 in the example). As such, SCURF can incrementally refine the rankings as much as necessary and as long as resources are available [26].

[3] The option to activate the inserted code block can be set *ON* or *OFF* differently for each version.

Table 1. Spectra Collected During Test Runs of the Versions of the *addNumberToPhoneList* function

	Context Switch				Number of Entries				Error
	C1	C2	C3	C4	C1	C2	C3	C4	Vector
Run1	1	0	0	0	3	3	0	3	1
Run2	0	1	0	0	3	3	0	3	1
Run3	0	0	1	0	1	1	3	3	0
Run4	0	0	0	1	1	1	3	3	0
Run5	1	1	0	0	3	3	0	3	1
Run6	1	0	1	0	3	3	0	3	1
Run7	1	0	0	1	3	3	0	3	1
Run8	0	1	1	0	3	3	0	3	1
Run9	0	1	0	1	3	3	0	3	1
Run10	0	0	1	1	1	1	3	3	0
Run11	1	1	1	0	3	3	0	3	1
Run12	1	1	0	1	3	3	0	3	1
Run13	1	0	1	1	3	3	0	3	1
Run14	0	1	1	1	3	3	0	3	1
Run15	1	1	1	1	3	3	0	3	1

In the following subsection, we illustrate the third step of our approach at *level-1* for the running example.

4.3 Step III. Spectra Analysis

In this step, the collected spectra during test runs are analyzed to rank components with respect to the probability that they cause an error. Analysis is performed iteratively to refine the rankings incrementally until a significant result is achieved or as long as resources permit.

We use the Ochiai similarity metric [3] to correlate the detected errors with component entries at runtime. We have slightly modified this similarity metric to correlate errors with concurrent execution of components. The original metric considers whether a component is executed during a test run or not. In our case, we are interested for each test run, whether a component was executed multiple times by different threads or not. Therefore, we modified the metric such that $n_{11}(j)$ is the number of failed runs in which part j is executed multiple times concurrently, $n_{10}(j)$ is the number of passed runs in which part j is executed multiple times concurrently, and $n_{01}(j)$ is the number of failed runs in which part j is executed only once or not at all, i.e., formally,

$$n_{01}(j) = |\{i \mid a_{ij} \leq 1 \wedge e_i = 1\}|$$
$$n_{10}(j) = |\{i \mid a_{ij} > 1 \wedge e_i = 0\}|$$
$$n_{11}(j) = |\{i \mid a_{ij} > 1 \wedge e_i = 1\}|$$

For the example case, the component rankings are formed as shown in Table 2. Note that the calculations are based on the first group of test runs (*level-1*) only. As we can see from the output in Table 2, $s_0(j)$ is the highest for component 1 and component 2. That means, these components are most probably subject to a concurrency fault that leads to the detected errors. Either of these components or both of them needs to be thread-safe. Not only for this example, but also in our industrial case studies, we have seen that usually rankings at *level-1* already provide accurate diagnosis. In the following section, we present examples from such industrial case studies.

Table 2. Analysis Results for the First Group of Test Runs (*level-1*) of Versions of the *addNumberToPhoneList* function

	C1	C2	C3	C4	Error Vector
Run1	M	M	N	M	1
Run2	M	M	N	M	1
Run3	S	S	M	M	0
Run4	S	S	M	M	0
$n_{11}(j)$	2	2	0	2	
$n_{10}(j)$	0	0	2	2	
$n_{01}(j)$	0	0	2	0	
$s_0(j)$	1.0	1.0	0.0	0.707	

M: Multiple Exec. ($a_{ij} > 1$), **S**: Single Exec. ($a_{ij} = 1$) **N**: No Exec. ($a_{ij} = 0$)

After testing a function, SCURF continues to test other functions that are called by that function. For instance, the function *createPhoneBook* is called by the *addNumberToPhoneList* function (Line 6). Hence, after testing the *addNumberToPhoneList* function, if the user asks, SCURF proceeds with testing the *createPhoneBook* function. Similarly, the functions *openPhoneBook*, *addPhoneBook* and *closePhoneBook* will be tested as well. SCURF continues to follow the call hierarchy until the tested function does not call any other function or it makes calls to POSIX functions only, e.g., *strcpy*, *strcat* and *sprintf*.

5 Industrial Case Studies and Evaluation

SCURF has been applied in the context of different industrial software projects that have been developed within TUBITAK. TUBITAK[4] is a government institution, which was formed in 1964. Since then, it has been responsible for many large-scale software development projects for the Turkish government. At TUBITAK, we have observed that one of the common root causes of concurrency faults was the adoption of legacy software within the context of multi-threaded software systems. Usually, the previously implemented functions have been designed to be single-threaded, without multi-threaded execution in consideration.

[4] The Scientific and Technological Research Council of Turkey.

Due to indeterministic behavior, testing does not always reveal the impact of concurrent execution and re-entrance to these previously implemented functions. Moreover, the lack of knowledge/documentation regarding the legacy software makes it even harder to locate a fault manually.

Several functions from different code bases were tested to *i)* check if their multi-threaded execution leads to an error, and if so, *ii)* locate the components that are subject to a concurrency fault as the root cause of the error. In the following subsections, we report three such faults that are detected/diagnosed by SCURF and discuss our experiences with SCURF. Due to confidentiality of the projects (and also for brevity), we present modified and simplified code examples. Nevertheless, they are representative examples to illustrate relevant cases and discuss our experiences.

Example 1. One of the concurrency faults was detected in a function called *pipe*; see its implementation in Listing 1.2. This function also calls other functions. The first lines of the *pipe* function is used for initialization. Then, a name (label) is obtained for the pipe to be created (line 8). Two file descriptors are opened with different access modes (lines 10 and 12). These descriptors are used as the read and write end of the pipe.

Listing 1.2. The implementation of the *pipe* function

```
1int pipe(int fds[2]) {
2 int retVal = -1;
3 //...
4 if(FALSE == isPipeInitialized) {
5   funcRet = init();
6 }
7 if(0 == funcRet) {
8   funcRet = generateNewName(pipeName);
9   if(0 == funcRet) {
10    fds[0] = open(pipeName, O_CREAT | O_RDONLY, S_IRWXU);
11    if(0 <= fds[0]) {
12      fds[1] = open(pipeName, O_CREAT | O_WRONLY, S_IRWXU)↩
          ;
13      if(0 <= fds[1]) {
14        retVal = 0;
15      }
16      else {
17        close(fds[0]);
18        remove(pipeName);
19      }
20    }
21  }
22 }
23 return retVal;
24}
```

Table 3. Spectra collected for level-1 of the versions of the *pipe* function

	Context Switch						Number of Entries						Error
	C1	C2	C3	C4	C5	C6	C1	C2	C3	C4	C5	C6	Vector
Run1	1	0	0	0	0	0	2	1	1	1	1	0	1
Run2	0	1	0	0	0	0	1	2	2	2	2	0	0
Run3	0	0	1	0	0	0	1	2	2	2	2	0	0
Run4	0	0	0	1	0	0	1	2	2	2	2	0	0
Run5	0	0	0	0	1	0	1	2	2	2	2	0	0
Run6	0	0	0	0	0	1	1	2	2	2	2	0	0

During the testing phase of the *pipe* function at *level 1*, SCURF collected the spectra shown in Table 3. In this table, we can see that an error was detected during the first test run. Note that we assume the *pipe* function to be fault-free in a single-threaded environment. Therefore, the detected error must have been caused by a concurrency fault. SCURF stopped execution after *level-1*. As such, there are 6 test runs in total and in each test run, only one component is influenced to force a context switch. SCURF runs two threads concurrently to test the function. Therefore the number of entries for each component are either 0, 1, or 2. Multiple execution of looping components are still treated as a single execution if being iterated within the same thread. Based on the results listed in Table 3, SCURF calculated fault probabilities for each component as shown in Table 4.

Table 4. Analysis results for level-1 of versions for the *pipe* function

	C1	C2	C3	C4	C5	C6	Error Vector
Run1	M	S	S	S	S	N	1
Run2	S	M	M	M	M	N	0
Run3	S	M	M	M	M	N	0
Run4	S	M	M	M	M	N	0
Run5	S	M	M	M	M	N	0
Run6	S	M	M	M	M	N	0
$n_{11}(j)$	1	0	0	0	0	0	
$n_{10}(j)$	0	5	5	5	5	0	
$n_{01}(j)$	0	1	1	1	1	1	
$s_0(j)$	1.0	0.0	0.0	0.0	0.0	0.0	

According to the results in Table 4, it can be seen that the reason for concurrency violation is multiple execution of *component 1*, before any thread leaves that component. In this case, the cause of the concurrency fault is a call to another function, *init*. This function is supposed to run only once even though the *pipe* function can be called multiple times. Therefore, its multiple execution was intended to be prevented by a global variable named as *isPipeInitialized*. However, the access to this variable should be protected for thread-safe execution.

To remove the fault, *component 1* was protected by a lock mechanism. Although the solution is easy to implement, it is not always easy to locate such a concurrency fault manually. Automated error detection and fault diagnosis facilitated by SCURF helped to perform this task with almost no effort.

Example 2. It turns out that the function *generateNewName*, which is called by the *pipe* function (Line 8), is also subject to a concurrency fault as detected by SCURF. There are no issues regarding the sequential execution of the function, which is shown in Listing 1.3. The function serves as a name generator until it reaches a limit that is imposed by the system. Every call to this function is supposed to return a new name.

Listing 1.3. The implementation of the *generateNewName* function

```
1int generateNewName(char *fileName) {
2 //...
3 while((0 != fileNameList[index])
4       && (index < MAX_NUM_OF_FILES)) { /* Component 1 */
5  index++;
6 }
7 if(MAX_NUM_OF_FILES != index) { /* Component 2 */
8  fileNameList[index] = index + 1;
9  //...
10  retVal = 0;
11 }
12 else { /* Component 3 */
13  retVal = ERANGE;
14 }
15 return retVal;
16}
```

SCURF detected a concurrency error for this function during the *level-1* tests. *Component 2* was associated with the detected error. When we check the code in Listing 1.3, we can figure out that concurrent access to the global variable named as *fileNameList* leads to an error because of uncontrolled access both in *component 1* and *component 2*. As such, both of these components must be protected together. SCURF was of valuable help to detect the error and locate the fault for this function. Nevertheless, manual analysis was necessary to successfully remove the concurrency fault concerning both *component 1* and *component 2*.

Example 3. The third case we present is regarding a concurrency fault in a function called *syncResources*. This function also makes calls to other functions but all these functions are thread-safe. However, there is a concurrency fault due to the implementation of the *syncResources* function itself. The function reads from a buffer of a device and transfers the data to another stream to be synchronized with the file system. Every call of this function synchronizes the buffers and flushes them to a permanent storage space. The function has 4

components. This case is particularly interesting because SCURF was able to diagnose the fault only after the test runs at *level-2*. The collected spectra can be seen in Table 5. A concurrency error was not triggered when only one component is influenced at a time to trigger a context switch. At *level-2*, two components were influenced at each test run to trigger an error. For instance, to trigger the error detected in test run 7, the execution of both components 1 and 4 were influenced. To trigger the error detected in test run 9, on the other hand, the execution of both components 2 and 4 were influenced. SCURF blamed three components for the detected errors. We figured out that an uncontrolled access to a global variable in these components caused the errors.

Table 5. Spectra collected during the first and second group of test runs (level-2) of the versions of the *syncResources* function

	Context Switch				Number of Entries				Error
	C1	C2	C3	C4	C1	C2	C3	C4	Vector
Run1	1	0	0	0	2	2	2	2	0
Run2	0	1	0	0	2	2	2	2	0
Run3	0	0	1	0	1	1	1	1	0
Run4	0	0	0	1	1	1	1	1	0
Run5	1	1	0	0	2	2	2	2	0
Run6	1	0	1	0	2	2	2	2	0
Run7	1	0	0	1	2	2	2	1	1
Run8	0	1	1	0	2	2	2	2	0
Run9	0	1	0	1	2	2	2	1	1
Run10	0	0	1	1	1	1	1	1	0

5.1 Performance and Scalability

One might claim that it could be impractical to instrument the code for all possible thread interleavings. This leads to 2^n versions for a function with n components. However, in practice we have seen that test runs at *level-1* are usually enough to diagnose a concurrency fault. At this level, only one component is influenced to trigger a context switch at each test run. As a result, n versions are enough for a function with n components. Only in the third case, SCURF needed to make use of test runs at *level-2*. Based on these observations we have implemented an incremental approach, inspired by approximation algorithms [26]. As such, SCURF can proceed until an error is detected or refine the rankings as much as necessary and as long as resources are available [26]. Tests on different versions can also be performed in parallel to improve scalability.

We performed tests on a Pentium 4 - 3.0 GHz HT Single core 32-bit desktop computer running openSUSE 12.1. In our first study, we executed functions of different size (with respect to the number of components) in 2 threads concurrently. For each of these tests, we measured the time it takes to localize a concurrency fault. For functions that have 6, 16 and 26 components, an error was triggered in 494 ms., 645 ms. and 720 ms., respectively.

In our second study, we performed measurements for different number of threads. The functions that have 6, 16 and 26 components are executed with 6, 12 and 24 threads, respectively. SCURF detected and diagnosed an error within 499 ms., 509 ms. and 547 ms., listed in the order of the corresponding tests.

Our approach is incomparable with respect to stress testing. We have applied stress testing on the *generateNewName* function (Listing 1.3). Even if the function was concurrently being executed in 24 different threads, the concurrency error was still not triggered after 100,000 tests. The error was triggered by SCURF within milliseconds.

5.2 Assumptions and Threats to Validity

SCURF requires that a test oracle and test case(s) are available for testing the functions of the subject system. Also, the original program should not be subject to an error when executed in a single-threaded manner. Otherwise, not all the detected errors can be associated with concurrency issues.

We instrument the program code to force context switches at different components. Inevitably, the effects of our instrumentation are dependent on the platform and the operating system. We perform our tests by assigning the same priority to all the child threads used for test runs, and a higher priority to the parent thread where these threads are created and joined. As such, all the child threads are created before any other terminates. Also, we use FIFO scheduling to eliminate the context switch because of time quantum.

The running time is dependent on the test cases and the algorithmic complexity of the function being tested. Hence, our performance measures can not be interpreted as absolute measures. They only reflect relative measures for a particular case/function.

6 Related Work

There is a large body of related work on analysis and detection of concurrency problems. The first attempt to address this problem focused on detecting race conditions[5]. Static analysis techniques addressing this issue include those based on type systems [6], model checking [16], and general program analysis [17]. There were also dynamic analysis techniques proposed like RecPlay [23] and Eraser [24] However, these techniques were subject to a significant number of false positives. In our approach, we cope with this issue by exploiting a probability score to rank the components instead of providing a binary decision.

More recent dynamic analysis techniques such as CCI [9] and Bugaboo [13] rely on predicate-based fault localization of concurrent programs. In particular, CCI samples shared-memory accesses during program executions and computes

[5] A race condition occurs when multiple threads perform unsynchronized access (with at least one of the threads writing) to a shared memory location.

likelihood scores for those memory accesses. Similarly, DefUse [25] samples def-use pairs between two threads. It finds the def-use pairs that are in failed executions and not in passed executions. Recon [12] compares memory accesses with the five previous memory accesses to compute the likelihood scores regarding the faulty memory accesses. CTrigger [18] profiles the program execution to identify thread interleavings correlated to atomicity violation bugs. Aspect oriented techniques have been used [5] for weaving assert statements that verify sequential access. The main distinction of our approach from these studies is that we do not employ passive monitoring. We instrument the code to force the application to run in different combinations of thread interleavings. This improves the diagnostic accuracy. As a complementary approach for improving the diagnostic accuracy, one can utilize test frameworks such as MultithreadedTC [22] to generate test cases that deterministically exercise specific interleavings of threads in an application.

PCT [4] is proposed as a randomized scheduler for finding concurrency bugs. This scheduler quantifies the probability of missing bugs. The quantification is based on so-called *depth* of the bug, which is defined as the minimum number of scheduling constraints that are sufficient to find the bug. Bugs that have higher depths are revealed in fewer schedules, making them harder to detect and diagnose. Experimental results show that in practice, many bugs (e.g., ordering errors, atomicity violations, and deadlocks) have small depths [4]. This result is also consistent with our observations. SCURF is able to diagnose most of the concurrency bugs at *level-1* already, by just influencing the execution of one component at a time.

Yet another approach for detecting concurrency errors is by detecting violations of the atomic property. It has been suggested that atomicity is a property that could be checked to detect concurrent errors at a more abstract, higher-level. The main limitation of atomicity violation detectors is the need for the user to annotate the source code, incurring a considerable overhead during the development phase of the software [7].

Similar to our approach, Falcon [19] and recently introduced UNICORN [21] also utilize spectrum-based fault localization for localizing concurrency faults. Conversely to our approach, both Falcon and UNICORN rank data access patterns (e.g., Read1, Write2, Read1) instead of statements/code blocks. However, not all concurrency problems are about data access. This was also the case for our motivating example.

Chess [15] is a concurrent unit testing tool that can provide fine-grained diagnosis regarding concurrency bugs. In our approach, we can detect such bugs at the component level. As an advantage over Chess, SCURF does not require additional scaffolding or test code to facilitate concurrency testing. An existing test suite prepared for functional unit testing can be used as is. Moreover, tests can be run in different processes in parallel.

7 Conclusion and Future Work

Concurrency faults are hard to diagnose. We have observed that adoption of legacy software in the context of multi-threaded software systems is one of the common root causes of these faults. It becomes even harder to manually locate concurrency faults when legacy software is involved. Therefore, we proposed a 3-step automated approach to diagnose these faults. We first instrument the code to force the program to run in different combinations of thread interleavings. At runtime, we collect information regarding the number of entries to each code block for each test. Then, we employ spectrum-based fault localization to correlate the detected errors with code blocks. Our tool, called SCURF, has been applied in the context of several industrial software systems. We have seen that our approach can accurately localize concurrency faults. We also obtained promising results with respect to performance and scalability.

As future work, we plan to experiment with various diagnostic algorithms that exploit the information regarding the number of times components get executed. Another interesting work is to use static analysis to determine where to enforce context switches, as such reducing the number of tests needed.

Acknowledgement. We thank the anonymous reviewers for their feedback to improve this paper. We also thank software developers and managers at TUBITAK BILGEM for sharing their code base with us and supporting our analysis.

References

1. Abreu, R., Zoeteweij, P., Golsteijn, R., van Gemund, A.: A practical evaluation of spectrum-based fault localization. Journal of Systems and Software 82(11), 1780–1792 (2009)
2. Abreu, R.: Spectrum-based Fault Localization in Embedded Software. Ph.D. thesis, Delft University of Technology (2009)
3. Abreu, R., Zoeteweij, P., van Gemund, A.J.C.: On the accuracy of spectrum-based fault localization. In: Proceedings of Testing: Academic and Industrial Conference Practice and Research Techniques, Windsor, UK, pp. 89–98 (2007)
4. Burckhardt, S., Kothari, P., Musuvathi, M., Nagarakatte, S.: A randomized scheduler with probabilistic guarantees of finding bugs. ACM SIGPLAN Notices 45(3), 167–178 (2010)
5. Dobbelsteen, J., Golsteijn, R., van de Laar, P.: An infrastructure for traceability to increase insight in complex embedded systems. Tech. Rep. PR-TN 2006/00506, Philips Electronics (2006)
6. Flanagan, C., Freund, S.N.: Type-based race detection for java. ACM SIGPLAN Notices 35(5), 219–232 (2000)
7. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. ACM SIGPLAN Notices 38(5), 338–349 (2003)
8. Harrold, M., Rothermel, G., Wu, R., Yi, L.: An empirical investigation of program spectra. ACM SIGPLAN Notices 33(7) (1998)

9. Jin, G., Thakur, A., Liblit, B., Lu, S.: Instrumentation and sampling strategies for cooperative concurrency bug isolation. ACM SIGPLAN Notices 45(10), 241–255 (2010)
10. Jones, J.A., Harrold, M.J.: Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of the International Conference on Automated Software Engineering, Long Beach, California, USA, pp. 273–282 (2005)
11. Liblit, B., Naik, M., Zheng, A.X., Aiken, A., Jordan, M.I.: Scalable statistical bug isolation. In: Proceedings of the Conference on Programming Language Design and Implementation, Chicago, Illinois, USA, pp. 15–26 (2005)
12. Lucia, B., Wood, B., Ceze, L.: Isolating and understanding concurrency errors using reconstructed execution fragments. ACM SIGPLAN Notices 47(6), 378–388 (2011)
13. Lucia, B., Ceze, L.: Finding concurrency bugs with context-aware communication graphs. In: Proceedings of the International Symposium on Microarchitecture, New York, NY, USA, pp. 553–563 (2009)
14. Mayer, W., Stumptner, M.: Evaluating models for model-based debugging. In: Proceedings of the International Conference on Automated Software Engineering, L'Aquila, Italy, pp. 128–137 (2008)
15. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: Proceedings of the USENIX Conference on Operating Systems Design and Implementation, pp. 267–280 (2008)
16. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: Proceedings of the Conference on Programming Language Design and Implementation, New York, NY, USA, pp. 446–455 (2007)
17. Naik, M., Aiken, A.: Conditional must not aliasing for static race detection. In: Proceedings of the Symposium on Principles of Programming Languages, New York, NY, USA, pp. 327–338 (2007)
18. Park, S., Lu, S., Zhou, Y.: CTrigger: exposing atomicity violation bugs from their hiding places. ACM SIGPLAN Notices 44(3), 25–36 (2009)
19. Park, S., Vuduc, R., Harrold, M.: Falcon: fault localization in concurrent programs. In: Proceedings of the International Conference on Software Engineering, pp. 245–254 (2010)
20. Park, S., Harrold, M.J., Vuduc, R.: Griffin: grouping suspicious memory-access patterns to improve understanding of concurrency bugs. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis. ACM (2013)
21. Park, S., Vuduc, R., Harrold, M.J.: A unified approach for localizing non-deadlock concurrency bugs. In: International Conference on Software Testing, Verification and Validation, Montreal, QC, pp. 51–60 (2012)
22. Pugh, W., Ayewah, N.: Unit testing concurrent software. In: Proceedings of the International Conference on Automated Software Engineering, pp. 513–516 (2007)
23. Ronsse, M., De Bosschere, K.: Recplay: a fully integrated practical record/replay system. ACM Transactions on Computer Systems 17(2), 133–152 (1999)
24. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: a dynamic data race detector for multithreaded programs. ACM Transactions on Computer Systems 15(4), 391–411 (1997)
25. Shi, Y., Park, S., Yin, Z., Lu, S., Zhou, Y., Chen, W., Zheng, W.: Do I use the wrong definition?: Defuse: definition-use invariants for detecting concurrency and sequential bugs. ACM SIGPLAN Notices 45(10), 160–174 (2010)
26. Vazirani, V.: Approximation Algorithms. Springer (2003)