Find bugs faster using stack traces

Andrej Gavrilov and supervisor: prof. Alberto Sillitti

Faculty of Computer science, Free University of Bolzano, Bolzano, Italy,
andrej.gavrilov@stud-inf.unibz.it, alberto.sillitti@unibz.it

Abstract. Modern software systems are growing in size, complexity and cost. Producing bug-free software is becoming more and more complicated. Failures appearing on the client side are strongly affecting business process. Every minute of service downtime costs money and reputation. Thus, fixing crashing bugs in the shortest time is of the highest importance. Stack traces provide essential information about the crashing failure which can fasten finding and fixing the bug. This article contains an overview of what a stack trace is and how it is used in existing code analysis techniques, describing latest advancements in the field, identifying existing gaps, and proposing directions for future research.

Keywords: debugging, stack trace, code analysis, slicing, clustering

Research area: Debugging, code slicing

1 Problem

Debugging distributed applications is a difficult but important part of software system maintenance. Nowadays software systems have hundreds of thousands of clients distributed around all over the world. Every failure of running software (including crashes, hangs, incorrect results and other software anomalies) can have critical impact on customers and their business. Thus locating and fixing bugs appearing during production runs in a short time is a task with the highest priority [14]. However, the number of collected errors can be huge. Thus, manual fault analysis and correction becomes problematic.

Even when using automated techniques, locating the bug in a big system is hard and time consuming. Existing clustering techniques lack accuracy, and a large number of ungrouped or inaccurately grouped (one group containing stack traces from different bugs) stack traces greatly increase the debugging time [5]. Our goal is to increase the accuracy of clustering stack traces and the code slicing with minimal additional cost (time and effort). To lower the cost of the automated analysis we will use a static code analysis technique. First, it does not require to instrument the source code. Thus there is no additional overhead for the system in use. Second, there is no need in reproducing the whole production environment of the system, as would be required for the dynamic code analysis. Third, a static technique allows analyzing only part of the source code, reducing the search space for debugging and saving more time [15].

Our work has two major goals. First, we will fill in the existing gap in the field by producing the comparative analysis of existing stack trace grouping techniques showing their advantages and limitations, propose the way to improve their accuracy by using additional information from multiple stack traces and the source code. Second, we will explore a new direction in debugging by performing an analysis of how effective in the terms of accuracy is using multiple stack traces for slicing the source code. This will help to locate and remove crashing bugs faster minimizing losses caused by service downtime both for developers and users.

2 Stack trace

The stack traces are compact, contain valuable data about the bug and are easy to collect. Since, stack traces contain no private data they can be collected automatically without prompting or distracting the user [9]. The example of the stack trace generated by Java virtual machine looks like this:

```
Exception in thread "main" java.lang.NullPointerException
  at JExample.method3(JExample.java:27)
  at JExample.method2(JExample.java:15)
  at Jexample.main(JExample.java:8)
```

Many widespread programming languages like Java and C# already have the built in and well tested functionality of generating stack traces. Collecting and sending them to the server is a constant time operation with no overhead and no impact on normally running application. Moreover, no additional data is collected from the client's machine, preserving any confidential (or restricted by client's policies) data loss. In most of the modern programming languages stack traces are available out of the box (C#, Java) or through a custom library (C++, Pyhton, Haskell).

3 Related work

A. Hamou-Lhadj and T. C. Lethbridge [6] present a survey of different trace exploration tools. However the survey is old, published in 2004. Since that time a lot of new techniques and tools were presented. The paper deals more with available tools rather than existing methods of performing the analysis. Listed tools are mostly based on dynamic analysis and use of execution traces that are usually produced using code instrumentation. The current survey is focused on the analysis of stack traces only. Thus, it is creating a new view at existent techniques and tools.

To get a better insight into the field of our research we performed a systematic literature review (SLR) on existing stack trace analysis methods. The review covers 20 publications from the three biggest electronic databases focused on computer science: ACM Digital library, IEEExplore, and Google scholar. The summary of the SLR findings is presented below.

3.1 Study of usefulness of stack traces

Stack traces are an important part of the error report and contain valuable information about the bug. Analyzed studies reveal the following reasons of why stack traces are important:

- Software developers rate a stack trace as one of the most important components they want to see in an error report [2, 16]
- Stack traces help to better understand the cause of the bug [11]
- Multiple stack traces contain more information about the bug [3]
- Error reports containing stack traces get fixed faster [12]
- There is a high probability to obtain a bug in one of the stack frames, probability is higher for the top frames [5, 12]

3.2 Grouping stack traces

Grouping stack traces is done in order to triage (assigning a report to a certain developer) and prioritize the debug effort. Properly done grouping saves a lot of time for a developer, since all the information about the bug is stored in one place. Most used stack traces grouping techniques:

- Statistical debugging based. It requires analyzing each stack trace prior to grouping [10]
- Heuristics based. Used in Windows Error Reporting system [9]
- Levenshtein distance - the number of changes needed to transform one stack trace to another [1, 5]
- Graph intersection [8]
- Position dependent model. More weight is put for top frames where the probability of obtaining the bug is higher. [4]

3.3 Stack trace analysis

The code analysis using stack traces consists of two major parts:

- Code slicing - using stack trace information to reduce the amount of the source code to analyze. Currently, the resulting slice when using stack traces is up to 8% smaller [15].
- Fault localization - locating buggy lines of code. Both existing approaches are combining dynamic information from stack traces with static (backward data-flow) code analysis [7, 13].

4 Current work

After analyzing the latest advancements in the field and limitations of existing techniques following research questions to achieve the first major goal of the research were formulated:

1. Which grouping techniques are more accurate in clustering stack traces?
2. What are the advantages and the limitations of every technique?
3. Does the additional information about the source code provide better results?
4. What kind of additional information brings the biggest increase in clustering accuracy?

The first reasearch question is addressing the absence of a comparative analysis of different stack trace analysis and grouping techniques. Existent studies use different techniques on different samples of data. Therefore a comparison of the results of different studies is very complicated.

Position dependent model presented in [4] is a big step forward in grouping stack traces. Previously widely used Levenshtein distance was not accurate enough. As it was shown by A. Schroter et al. [12] the dissimilarity obtained in the first frame is more significant than the one in the tenth frame. Thus, there is a bigger probability to obtain a bug in the top frames of the stack trace. By applying this knowledge better grouping accuracy is achieved.

The weighting approach could be also applied for other grouping techniques. A specific computational algorithm may depend on the way of representing the stack trace. Usually, stack trace is represented as a directed graph with caller-callee relations or as a string where every symbol is a method signature from the stack trace (Method1→Method2→. . . →MethodN→Exception). Thus different string clusterization techniques can be also applied to the stack traces.

Q-grams clusterization algorithm is one of the commonly used for strings. It uses tuples of N (usually 2 or 3) sequential symbols that can be easily mapped to caller-callee pairs (or sequences) in a stack trace. Combining this algorithm with weights could be potentially efficient way to group stack traces.

Another string clusterization technique, that already uses weights, is agglomerative information bottleneck. It has additional advantage that total number of resulting groups (that is equal to number of bugs) is not needed to be known in advance. This is convenient because usually the overall number of present bugs is not known. However, the drawback of the information bottleneck technique is the high cost of computation. If the number and length of stack traces is big, the time needed for computing similarities can become an issue.

For now there are no studies or experiments on how good these and other techniques are (or could be) in grouping stack traces. Neither is there a reasoning why these techniques should not work for stack traces, what are their the advantages and limitations.

To answer these questions we are preparing an experiment. The implementations of different clustering techniques will be run on the same data set. And the results of each run will be evaluated using the same metrics. We have selected both most used until now and also not so common but promising (having potencial) clustering techniques based on following stack trace similarities:

– Crash point
– Levenshtein distance

- Q-grams
- Agglomerative information bottleneck
- Position dependent model

Every selected technique will be applied to the same data set. We have prepared a testing data set containing more then 250 stack traces of more then 100 bugs from the open source bug repositories. The test set contains stack traces of both single and multiple (duplicated) bugs. If two or more bugs were marked in the batabase as duplicates, their stack traces should be grouped together. The relations between bugs will be used to evaluate resulting clusters produced by different clustering techniques.

As a result of the experiment we will have a comparative description of different clustering approaches will be created. The description will contain the evaluation of resulting clusters (precision, recall, F-measure) and a list of advantages and limitations for every tested approach.

5 Future work

5.1 Adding additional information

About 40% of all registered bugs are fixed in methods that are not included in the stack trace [12]. Unfortunately, data contained in the stack trace is not enough to assist developers in this specific case. However, if combined with the information extracted from the source code it still can point to the location of the bug. Source code can provide the full information about caller-callee method relations in the application. We are going to use the directed caller-callee graph to detect hidden relations between the stack traces and measure the increase in accuracy this information brings to different stack trace clustering techniques. Using a caller-calle graph requires an access to the source code and an additional tooling to extract the relations between methods. But the benefit of this approach is potentially high and can be very useful in both grouping error reports and localizing the bug.

5.2 Code slicing

Further improvement of the accuracy of the source code clicing should consider using groups of stack traces, representing the same bug. For now, all existing approaches use separate stack traces as an input for the analysis. Using grouped stack traces implying the same bug should provide additional information to the analysis system. Thus, it may lead to increased accuracy and better performance. Moreover, there would be no need to separately analyze every stack trace, belonging to the same group. This approach can potentially save even more time and effort.

References

1. Bartz K. Stokes J., Platt J., Kivett R., Grant D., Calinoiu S., Loihle G.: Finding Similar Failures Using Callstack Similarity. Proc. of the Third conference on Tackling computer systems problems with machine learning techniques (2008)
2. Bettenburg N., Weiß C., Just S., Premraj R., Schröter A., Zimmermann T.: Quality of Bug Reports in Eclipse. Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange (2007)
3. Bettenburg N., Premraj R., Zimmermann T., and Kim S.: Duplicate Bug Reports Considered Harmful . . . Really? ICSM 2008, IEEE International Conference (2008)
4. Dang Y., Wu R., Zhang H., Zhang D., and Nobel P.: ReBucket: A Method for Clustering Duplicate Crash Reports Based on Call Stack Similarity. Software Engineering (ICSE), 34th International Conference (2012)
5. Dhaliwal T., Khomh F., and Zou Y.: Classifying Field Crash Reports for Fixing Bugs: A Case Study of Mozilla Firefox. 27th IEEE International Conference on Software Maintenance (ICSM) (2011)
6. Hamou-Lhadj A. and Lethbridge T.: A Survey of Trace Exploration Tools and Techniques. Proc. of the 2008 conference of the Center of Advanced Studies on Collaborative research (2004)
7. Jiang S., Zhang H., Wang Q., and Zhang Y.: A Debugging Approach for Java Runtime Exceptions Based on Program Slicing and stack traces. 10th International Conference on Quality Software (2010)
8. Kim S., Zimmermann T., and Nagappan N.: Crash Graphs: An Aggregated View of Multiple Crashes to Improve Crash Triage. Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference (2011)
9. Kinshumann K., Glerum K., Greenberg S., Aul G., Orgovan V., Nichols G., Grant D., Loihle G., and Hunt G.: Debugging in the (Very) Large: Ten Years of Implementation and Experience. Communications of the ACM (2011)
10. Liu C. and Han J.: Failure Proximity: A Fault Localization-Based Approach. Proc. of the 14th ACM SIGSOFT international symposium on Foundations of software engineering (2006)
11. Parnin C. and Orso A.: Are Automated Debugging Techniques Actually Helping Programmers? ISSTA '11 Proceedings of the 2011 International Symposium on Software Testing and Analysis (2011)
12. Schroter A., Bettenburg N., and Premray R.: Do Stack Traces Help Developers Fix Bugs. 7th IEEE Working Conference on Mining Software Repositories (MSR) (2010)
13. Sinha S., Shah H., Gorg C., Jiang S., Kim M., and Harrold M.: Fault Localization and Repair for Java Runtime Exceptions. Proc. of the 18th international symposium on Software testing and analysis (2009)
14. Yuan D., Mai H., Xiong W., Tan L., Zhou Y., and Pasupathy S.: SherLog: Error Diagnosis by Connecting Clues from Run-time Logs. Proc. of the 15th edition of ASPLOS on Architectural support for programming languages and operating systems (2010)
15. Zhang H., Jiang S., and Jin R.: An Improved Static Program Slicing Algorithm Using Stack Trace. IEEE International Conference on Software Engineering and Service Sciences - ICSESS (2011)
16. Zimmermann T., Premraj R., Bettenburg N., Just S., Schroter A., and Weiss C.: What Makes a Good Bug Report? IEEE Transactions on Software Engineering, Vol. 36, No. 5 (2010)