ReBucket: A Method for Clustering Duplicate Crash Reports Based on Call Stack Similarity

Yingnong Dang[§], Rongxin Wu[†], Hongyu Zhang[†], Dongmei Zhang[§] and Peter Nobel[§]

[§]Microsoft Research Asia, Beijing, China
Dang.Yingnong@microsoft.com, dongmeiz@microsoft.com

[†]Tsinghua University, Beijing, China
se.wu.rongxin@gmail.com, hongyu@tsinghua.edu.cn

[§]Microsoft Corporation, One Microsoft Way, Redmond, WA, USA
pnobel@microsoft.com

Abstract—**Software often crashes. Once a crash happens, a crash report could be sent to software developers for investigation upon user permission. To facilitate efficient handling of crashes, crash reports received by Microsoft's Windows Error Reporting (WER) system are organized into a set of "buckets". Each bucket contains duplicate crash reports that are deemed as manifestations of the same bug. The bucket information is important for prioritizing efforts to resolve crashing bugs. To improve the accuracy of bucketing, we propose ReBucket, a method for clustering crash reports based on call stack matching. ReBucket measures the similarities of call stacks in crash reports and then assigns the reports to appropriate buckets based on the similarity values. We evaluate ReBucket using crash data collected from five widely-used Microsoft products. The results show that ReBucket achieves better overall performance than the existing methods. In average, the F-measure obtained by ReBucket is about 0.88.**

Keywords: Crash reports, clustering, duplicate crash report detection, call stack trace, WER

I. INTRODUCTION

A software crash is one of the most severe manifestations of a defect (bug) in software, and is typically assigned a high priority to be fixed. To facilitate debugging, many crash reporting systems such as Windows Error Reporting [10], Apple crash report [2], and Mozilla crash report [20] have been deployed to automatically collect crash reports from users at the time of crash.

Crash reports, which may be considered "telemetry data", can include information such as the crashed module's name and call stack traces. Such information is useful to software developers trying to determine the cause of a crash [10, 23]. In some cases, a large number of crash reports may arrive daily. Many of these crash reports are actually caused by the same bug and are therefore duplicate reports. To help developers reduce debugging efforts, it is important to automatically organize duplicate crash reports into one group.

In the Microsoft Windows Error Reporting (WER) system, crash reports are organized according to "buckets". Ideally, each bucket contains crash reports that are caused by the same bug. A number of heuristics are used to generate the buckets [10]. Developers prioritize bug fixing efforts based on the number of crash reports received by each bucket (i.e., the number of hits). A bucket with a higher number of hits will be investigated with higher priority compared with a bucket with a lower number of hits. However, it is still not uncommon that crashes caused by one bug spread to multiple buckets (the "second bucket problem") [10]. Also, WER may generate many buckets that contain only one or a few number of crash reports (the "long tail" problem). The existence of the second bucket problem and the long tail problem reduces the effectiveness of effort prioritization and problem diagnosis.

To improve the accuracy of bucketing, in this paper, we propose ReBucket, a method for clustering crash reports based on call stack similarities. To measure the similarity between two call stacks, we propose a new similarity measure called the Position Dependent Model (PDM). PDM computes the similarity between two call stacks based on the number of functions on two call stacks, the distance of those functions to the top frame, and the offset distance between the matched functions. In ReBucket, a training process is also designed to tune the parameters required by PDM.

We evaluate ReBucket using crash data collected from five widely-used Microsoft products. The results show that the performance of ReBucket is promising. On average, the F-measure achieved by our method is about 0.88. ReBucket also achieves better overall results than the existing methods, including the existing WER bucketing method.

We believe the proposed ReBucket method can help prioritize debugging efforts and facilitate problem diagnosis. We have worked closely with a Microsoft product team and they helped review twenty crash report clusters we obtained by using ReBucket and confirmed that 70% of them are very meaningful clusters.

The contributions of this paper are as follows:

- We propose a new bucketing method, ReBucket, for clustering duplicate crash reports.
- In ReBucket, we propose a new metric (Position Dependent Model) for measuring similarity between two call stacks.
- We evaluate our approach on five widely-used Microsoft products.

The remainder of this paper is organized as follows: We introduce the background information in Section II. Section III describes the existing crash bucketing method used in Microsoft and its limitations. Section IV describes our proposed bucketing method ReBucket. Section V presents our experimental design and Section VI shows the experimental results. We discuss the application of ReBucket in practice in Section VII and threats to validity in Section VIII. Section IX surveys related work followed by Section X that concludes this paper.

II. BACKGROUND

A. The Windows Error Reporting System

Although development teams spend much resource and effort on software testing before releasing products, in reality, released software still contains bugs [26]. Some bugs manifest as crashes in the field.

Because of the wide deployment of Microsoft Windows systems and the large number of Windows applications including third-party applications, the volume of crash reports becomes overwhelming. To automatically collect crash information from the field, Microsoft deployed a distributed system called Windows Error Reporting (WER) [10]. When a crash happens on a client's Windows platform, the WER system collects the crash information (including application/module name, application/module version, and the call stack trace). The crash information is reported to a WER server after user permission is obtained. The server then checks the duplication of the crash report and classifies it into a bucket. Each bucket is a collection of crash reports that are likely caused by the same bug. A new bucket is created if the crash report is considered a new one. Finally, WER automatically generates bug reports for highly hit buckets and presents them to developers.

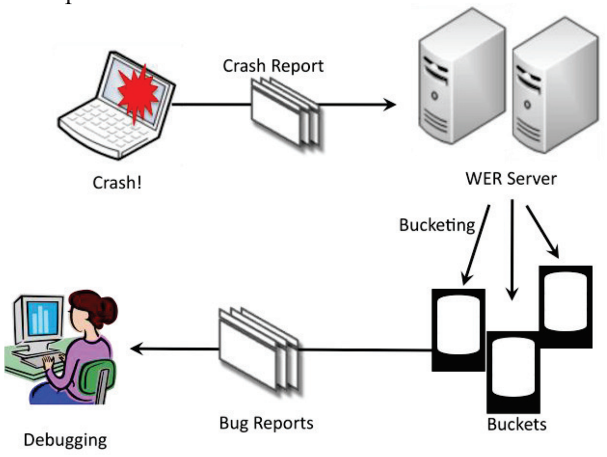


Figure 1. An overview of crash reporting system

WER has proven its value to Microsoft development teams [10]. During its ten years of operation, it has collected billions of crash reports. These crash reports have helped developers diagnose problems. For example, the Windows Vista team found and fixed over 5,000 bugs isolated by WER in the beta release of Vista alone.

B. Crash Call Stack

The call stack is part of crash information collected by WER at the client side. It plays an important role in understanding a crash. A call stack consists of a list of ordered frames, which are recorded at the time of crash. A frame in a call stack consists of a module name and a function name, representing a function call or procedure.

Table I shows an example of crash call stack. There are 7 frames in the call stack, with the most recently executed function at the top and the least recently executed at the bottom. The ordered sequence for each frame indicates the invoking relationship between frames. For example, the crashed function at level 0 (QuickSort) is the top frame, which is invoked by the function at level 1 (MyProgram::SearchRightPart).Often, one bug could cause different crash stack traces because of the different execution scenarios.

Table I. An example of crash call stack

Frame Level	Module	Function
0	SORT.DLL	QuickSort
1	FINDPOINT.DLL	MyProgram::SearchRightPart
2	FINDPOINT.DLL	MyProgram::DivideAndConquer
3	FINDPOINT.DLL	MyProgram::SearchLeftPart
4	FINDPOINT.DLL	MyProgram::DivideAndConquer
5	FINDPOINT.DLL	FindPoint
6	MYPROGRAM.DLL	MyMain

III. THE EXISTING BUCKETING METHOD IN WER

For widely used systems, developers could face a large number of crash reports sent from users all over the world. Many of these crash reports are actually caused by the same bug and are therefore duplicates. To help developers reduce debugging efforts, it is important to automatically organize duplicate crash reports into one group after receiving a large number of crash reports.

In WER, the process of organizing crash reports is named Bucketing [10]. For each bucket, WER counts the occurrence of crashes (i.e., the number of hits). Once the crash occurrence for a bucket exceeds a threshold, a bug report will be generated for this bucket and will be presented to developers. Developers can also prioritize the bug fixing efforts based on the number of crash reports received in each bucket. A bucket with a higher number of hits will be investigated with higher priority compared to a bucket with lower hits. Therefore, WER helps focus the effort on the bugs that have a bigger impact on the users.

WER has implemented more than 500 bucketing heuristics. Some of the top heuristics include "L7: include offset of crashing instruction in faulty module", "L4: include faulty module name", "C14: Identify known faulty device", and "C12: Known out-of-date program" [10].

Ideally, the bucketing algorithm should organize all the crash reports caused by the same bug into one unique bucket. However, in some scenarios, the WER bucketing heuristics may assign crash reports caused by the same bug to multiple buckets (the "second-bucket" problem [10]). It is reported that for products such as PowerPoint 2010, 30.6% reports are assigned to a second bucket other than the primary bucket for a bug [10]. In such cases, the accuracy of bucketing is hampered because the bucket contains crash reports that are actually caused by different bugs.

Our empirical studies also find that the existing WER bucketing method can result in the "long tail" problem. That is, it could produce a large number of small buckets (i.e., buckets that contains a small number of crash reports). Experiences have shown that a bug for a popular system can be always encountered by many users. Therefore, the "long tail" phenomenon indicates the existence of problematic buckets. WER may assign crash reports caused by the same bug to some "long tail" buckets due to the different manifestations of the bug in different hardware and software settings. Figure 2 illustrates the "long tail" problem for the MS Publisher project (release 12 RTM). Around 87.26% buckets contain only 20% of the hits, while 12.7% buckets contain 80% of the hits. The "long tail" phenomenon is a general case of the "one-hit wonders" problem identified by the WER designers [10]). The "one-hit wonders" are buckets that contain exactly one crash report. In [10], the authors reported that for products such as Outlook 2010, 10% buckets are "one-hit wonders". The existence of the "long tail" behavior and its special case "one-hit wonders" may cause misunderstanding of the severity of crashes and thus wrong prioritization.

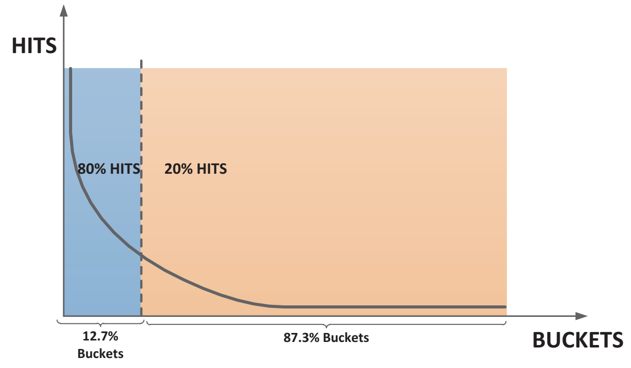


Figure 2.The long tail behavior of WER bucketing method

In the next section, we will present our proposed method that can improve the effectiveness of bucketing crash reports.

IV. THE PROPOSED METHOD: REBUCKET

Our proposed bucketing method is called ReBucket. Figure 3 shows the overall structure of ReBucket. For newly arrived crash reports, ReBucket first preprocesses them to extract the simplified call stacks. It then calculates the similarities among the call stacks using a proposed similarity measure called Position Dependent Model (PDM). Finally, it clusters the crash reports into corresponding buckets using the hierarchical clustering method. The parameters used in PDM can be learned from a trained model constructed by using the historical bucket data.

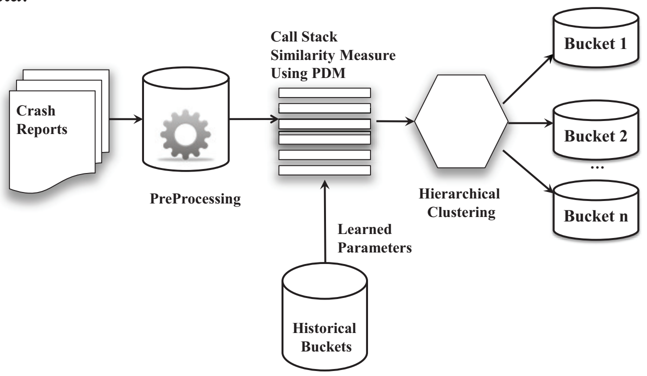


Figure 3. The overall structure of ReBucket

A. PreProcessing

Before calculating the similarities among call stacks and performing clustering, we first preprocess the crash reports by extracting the call stack data and removing the following functions from the call stack:

Immune Functions: Immune functions are functions that are considered to be "immune" from fault in the event of a software crash. Immune functions may include those functions that are simple enough, or have been used successfully for long enough time, therefore are unlikely to be buggy. By eliminating the immune functions, developers can concentrate on other areas within which a bug is more likely located. In our experiments, we use an immune function list provided by the Microsoft product team that maintains the subject systems.

Recursive Functions: It is common that recursive functions occur in call stacks. Typically, the recursive functions are uninformative and could affect the similarity measurement, especially when the number of the recursive functions is large. Therefore, we remove the recursive functions using the removal algorithm proposed by Brodie et al. [6].

B. Computing Similarity between Call Stacks

1) Analysis of Call Stacks

Before introducing the proposed method for computing the similarity between two call stacks, we first introduce two metrics that are used in our method:

Distance to the Top Frame: the position offset between the current frame and the top frame of a call stack.

Alignment Offset: the offset between the distances to the top frame for the two matched functions. In two call stacks, matched functions are the functions that appear in both call stacks.

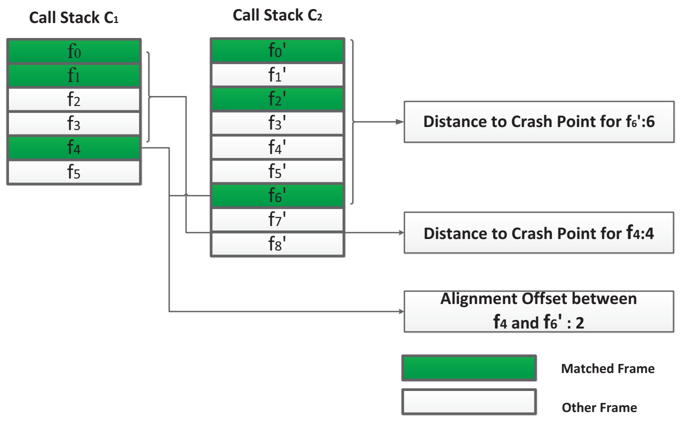

Call Stack C₁
Call Stack C₂
f0
f1
f2
f3
f4
f5
f0'
f1'
f2'
f3'
f4'
f5'
f6'
f7'
f8'
Distance to Crash Point for f6':6
Distance to Crash Point for f4:4
Alignment Offset between f4 and f6' : 2
Matched Frame
Other Frame

Figure 4. Illustration of call stack metrics

Figure 4 illustrates the above concepts. For both call stacks C_1 and C_2, the crash point is at the top frame. The C_1 functions f_0, f_1 and f_4 match the C_2 functions f_0', f_2' and f_6', respectively. For the f_4 function in C_1, its distance to the top frame is 4. The distance to the top frame for f_6' in call stack C_2 is 6. The alignment offset between the two functions is measured by the difference in their distances to the top frame. Since the distance to the top frame for f_4 and f_6' is 4 and 6 respectively, the alignment offset is 2.

2) The Position Dependent Model

Basically, if two call stacks belong to the same bug, they are more likely to be similar. In this subsection, we introduce the similarity measure used in ReBucket. Our similarity measure is called Position Dependent Model (PDM), which is based on the insights that:

- More weight should be put into frames whose position is closer to the top, since the frame that is blamed for the bug is likely to occur near the top of the call stack.
- The alignment offset between two matched functions in two similar call stacks is likely to be small.

Based on the above insights, the similarity between two call stacks C_1 and C_2 is defined as follows.

Let L be the set of all the common frame sequences between C_1 and C_2. Let L_i be one of the common frame sequences, where $S_{i,1}$,$S_{i,2}$,…$S_{i,k…}$ are the matched functions that both C_1 and C_2 contain.

$$L = \{L_1, L_s, L_3 \ldots\} ,\quad L_i = \{S_{i,1}, S_{i,2}, S_{i,3}, \ldots S_{i,k} \cdots\}$$

Let $Pos(C_q, S_{i,k})$ be the position of frame $S_{i,k}$ in the call stack C_q, l be the minimum of the number of frames in call stacks C_1 and C_2.

The similarity between the call stack C_1 and C_2 is defined as Equation 1.

$$\begin{cases} sim(C_1, C_2) = \dfrac{\max\limits_{L_i \in L}[Q(L_i)]}{\sum\limits_{j=0}^{l} e^{-cj}} \\ Q(L_i) = \sum\limits_{s_{i,k} \in L_i} e^{-c\min(Pos(C_1, s_{i,k}), Pos(C_2, s_{i,k}))} e^{-o\left|Pos(C_1, s_{i,k}) - Pos(C_2, s_{i,k})\right|} \end{cases} \quad (1)$$

, where c is a coefficient for the distance to the top frame, o is a coefficient for the alignment offset. The values of c and o can be set manually based on past experience. In Section IV.D, we also propose a learning-based method to automatically obtain the optimal coefficient values. The function $Q(L_i)$ is used to summarize the similarity values achieved by matched functions in the common frame sequence L_i. Its first exponential function considers the minimum distance to the top frame between a pair of matched functions. The second exponential function considers the minimum alignment offset between a pair of matched functions. The smaller the distance (or offset), the larger the returned value of the function $Q(L_i)$.

According to Equation (1), the call stack similarity metric is determined by the common frame sequence that can achieve the maximum value for the function $Q(L_i)$. It is inefficient to exhaustively search all the common frame sequences for the maximum value. Inspired by the solution to the Longest Common Subsequence problem [8], we apply a dynamic programming algorithm to solve Equation (1) as follows:

1) We define a similarity matrix *M[i,j],* which represents the similarity between two subsequences. The first subsequence is from the top frame to the i^{th} frame in C_1, and the second subsequence is from the top frame to the j^{th} frame in C_2.
2) According to the definition of similarity matrix *M[i,j]*, the calculation of $sim(C_1,C_2)$ can be transferred into the problem of calculating *M[m,n],* where *m* is the length of C_1 and *n* is the length of C_2, as shown in Equation (4).
3) The problem of calculating *M[i,j]* can be divided into several sub-problems, as shown in Equations (2) and (3) (the coefficients *c* and *o* are the same as the ones shown in Equation 1).The similarity matrix *M[i,j]* can be obtained by progressively calculating matrix elements.

$$M_{i,j} = \max\begin{cases} M_{i-1,j-1} + \mathrm{cost}(i,j) \\ M_{i-1,j} \\ M_{i,j-1} \end{cases} \quad (2)$$

$$\mathrm{cost}(i,j) = \begin{cases} e^{-c*\min(i,j)} e^{-o*abs(i-j)} & \text{if } ith \text{ } frame of\ C_1 = jth \text{ } frame of\ C_2 \\ 0 & otherwise \end{cases} \quad (3)$$

$$sim(C_1, C_2) = \frac{M_{m,n}}{\sum_{j=0}^{l} e^{-cj}} \quad (4)$$

where m is the length of C_1, and n is the length of C_2.

Using PDM, the similarity of any two call stacks can be evaluated. Such an evaluation can be used to determine the appropriateness of including two crash reports into a cluster.

C. Clustering

The clustering is based on the similarity of the call stacks computed using PDM. If the call stacks are sufficiently similar, the associated crash reports are grouped into the same bucket.

In our approach, we apply the Agglomerative Hierarchical clustering technique [11,12], which is a bottom-up clustering approach. At the beginning of the agglomerative hierarchical clustering, each call stack belongs to its own cluster. Then, the closest pair of clusters is selected and merged. To decide which pair of clusters should be merged, the distance metric between the clusters should be defined. In our approach, we adopt the maximum distance of all element pairs between two clusters as the cluster distance metric. In other words, the cluster distance metric depends on the maximum distance between the call stacks in each cluster. Formally, the distance between clusters is defined as Equations (5) and (6), where Cl_i and Cl_j are a pair of clusters, C_1 and C_2 are call stacks in Cl_i and $Cl_{j,}$, respectively.

$$distance(Cl_i, Cl_j) = \max_{C_1 \in Cl_i, C_2 \in Cl_j} dist(C_1, C_2) \quad (5)$$

$$dist(C_1, C_2) = 1 - sim(C_1, C_2) \quad (6)$$

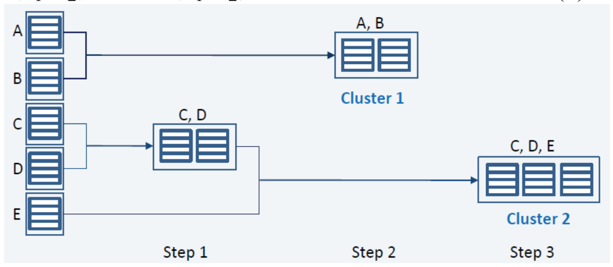


Figure 5. Illustration of the Agglomerative Hierarchical Clustering process

In general cases, the complexity of agglomerative hierarchical clustering is $O(n^3)$ if all the elements are merged as one cluster. In our case, we adopt a distance threshold *d* as a stopping criterion for the clustering process. The value of *d* can be set manually or learnt through a training process (which is discussed in Section IV.D). Once the maximum distance between a pair of clusters is above the distance threshold, the clustering process for this pair is stopped. Finally, the resulting clusters are the buckets that contain the similar crash reports. For example, in Figure 5, two buckets (resulted from Cluster 1 and Cluster 2) are produced.

D. *Learning Parameter Values via Training*

PDM uses two coefficients: *c* is a coefficient for the distance to the top frame, and *o* is a coefficient for the alignment offset. The distance threshold for clustering is also a parameter that should be tuned. The values of these parameters can be set manually. However, inappropriately set parameters may lead to significantly different similarity results. The parameter values can also vary from project to project. In our method, we propose a training process to learn these parameters' values. The training process is described as follows.

First, we construct a training dataset from historical bucket data and the corresponding bug data. From the historical data, we extract the duplicate crash reports that are caused by the same bugs and are confirmed by the developers. We also extract the same number of dissimilar crash reports from different buckets that are mapped to different bugs. As the number of dissimilar crash reports is typically large, the process of extracting dissimilar crash reports is performed by random sampling. Based on the obtained duplicate and dissimilar crash reports, we collect pairs of similar and dissimilar call stacks, and form the training dataset.

For the coefficients *c* (for the distance to the top frame), *o* (for the alignment offset), and *d* (the distance threshold for clustering), their values vary independently. Different combinations can result in different clustering performance. We propose a search-based algorithm, which exhaustively searches for the optimal combination.

```
DetermineOptimalParameters($D$: call stack pairs)
1. Assign the coefficient for the distance to top frame $c$ with a small initial value $c_0$
2. Assign the coefficient for the alignment offset $o$ with a small initial value $o_0$
3.   For each call stack pair $p$ in $D$
4.      Calculate the similarity of $p$ using Equation (1) with $c$ and $o$
5.   EndFor
6. Assign the distance threshold $d$ with a small initial value $d_0$
7. For each call stack pair $p$ in $D$
8.    If the similarity of $p$ is greater than $1-d$, Then
            label $p$ as similar
      Else
            label $p$ as dissimilar
9. EndFor
10. Compute F-measure for all the call stack pairs in $D$
11. Increase $d$ by a small step $s_1$
12. Repeat step 7-11 until $d$ reaches the maximum threshold $d_{\max}$
13. Increase $o$ by a small step $s_2$
14. Repeat step 3-13 until $o$ reaches the maximum threshold $o_{\max}$
15. Increase $c$ by a small step $s_3$
16. Repeat step 2-15 until $c$ reaches the maximum threshold $c_{\max}$
17. Select the value of $c_{optimal}$, $o_{optimal}$, $d_{optimal}$ that achieve the best F-measure
18. Return $c_{optimal}$, $o_{optimal}$, $d_{optimal}$
```

Figure 6. Determining the optimal parameter values

Figure 6 shows our training algorithm for determining the optimal parameter values. *D* contains the collected pairs of similar and dissimilar call stacks in the training dataset. According to our empirical experience, the range for *c* and *o* are bounded by (0, 2). In our training algorithm, we specify o_0=0, o_{max}=2, s_2=0.1 and c_0=0, c_{max}=2, s_3=0.1, which means that we increase the values of *c* and *o* by 0.1 at each time of iteration. For the distance threshold, we specify d_0=0, d_{max}=1, s_1=0.01, which means that we try similarity threshold from 0 to 1, with a step of 0.01. Finally, the optimal values that can achieve the best F-measure are selected.

V. EXPERIMENTAL DESIGN

In this section, we describe our experimental design for evaluating the proposed bucketing method ReBucket.

A. *Experimental Setup*

In our experiment, we have selected five products of Microsoft Corporation as our subjects. These systems are selected because of their popularity and the availability of the crash data.

- Microsoft Publisher, which is a desktop publishing application. In our experiment, we use the version 11.0.4920.
- Microsoft OneNote, which is for information gathering and multi-user collaboration. We use the version 11.0.4920.
- Microsoft PowerPoint, which is used to create slideshows for presentation purposes. We use the version 10.0.525.
- Microsoft Project, which is a project management tool. We use the version 10.0.2002.
- Microsoft Access, which is a relational database management system from Microsoft. We use the version 10.0.2511.

In the remainder of the paper, for the sake of confidentiality, we only refer to these products as A, B, C, D, E (the assignment of a product to a letter is random).We have collected crash reports for each product from the WER system. The average number of crash reports is 1198 and the average number of buckets is 75. The mappings between crash reports and bugs have already been examined and confirmed by Microsoft developers. Therefore, we have a high quality "ground truth" dataset for evaluating the performance of ReBucket. To learn the parameters required by PDM, we use the 20% first reported data for training and the remaining 80% data for testing.

B. *Research Questions*

To evaluate our approach, we design experiments to address the following research questions:

RQ1: How accurate are the buckets produced by ReBucket?

RQ2: Can ReBucket reduce the number of buckets ?

RQ1 evaluates the effectiveness of our ReBucket method for the five Microsoft products and compares it to the existing WER bucketing method. In Section III, we have described the "long tail" problem that is faced by the existing WER bucketing method. RQ2 evaluates if ReBucket can mitigate this problem.

Besides the WER Bucketing method, there are also some other methods for measuring call stack similarities and grouping crash reports. In our experiments, we compare ReBucket with the following two related methods:

- **Prefix Match [19]**: This method is based on string matching. Its basic assumption is that two call stacks caused by the same problem share the common frames closer to the top of both stacks. In this algorithm, the longest common prefix is computed as the number of consecutive frames starting from the top of the stack. Since the Prefix Match method is only for measuring the similarity between two call stacks, we apply the hierarchical clustering algorithm to bucket the crash reports based on the similarity values computed by Prefix Match. We then compare the results of ReBucket with those of Prefix Match.
- **Crash Graph [14]:** A crash graph is an aggregated graphical view of multiple crashes in the same bucket. Duplicate crash reports are identified based on graph similarity measures and their crash graphs are merged. When comparing the similarity between two crash graphs, a similarity threshold is needed. As the Crash Graph method does not specify a training process for selecting the similarity threshold value, in our experiment, we choose the similarity threshold that can achieve the best performance for the training dataset.

C. *Evaluation Metrics*

To evaluate the performance of our method, we adopt metrics Purity [1, 24], Inverse Purity [1], and F-measure [1, 24]. These metrics are based on the precision and recall concepts inherited from Information Retrieval, and are widely used for cluster evaluations.

We denote C as the set of clusters to be evaluated, L as the set of categories (actual clusters), C_j as the jth cluster, and L_i as the ith category. Then the Precision and Recall of L_i corresponding with C_j are defined as follows:

$$\text{Precision}(L_i, C_j) = \frac{|L_i \cap C_j|}{|C_j|}$$

$$\text{Recall}(L_i, C_j) = \frac{|L_i \cap C_j|}{|L_i|}$$

Based on the above the definitions, the metrics Purity, Inverse Purity and F-measure are defined as follows. Here, we denote N as the total number of clustered elements.

Purity is computed by taking the weighted average of maximal precision values for each cluster:

$$\text{Purity} = \sum_j \frac{|C_j|}{N} \max_i \left\{ \text{Precision}(L_i, C_j) \right\}$$

The value of Purity is from 0 to 1, the higher the better. Purity penalizes the noise (wrongly grouped items) in a cluster, but it does not reward grouping items from the same category together (i.e., if every cluster contains only one item, the Purity will be 1).

Inverse Purity is computed by taking the weighted average of maximal recall values for each category:

$$\text{InversePurity} = \sum_i \frac{|L_i|}{N} \max_j \left\{ \text{Recall}(L_i, C_j) \right\}$$

The value of Inverse Purity is from 0 to 1, the higher the better. Inverse Purity rewards grouping items together, but it does not penalize noisy items from different categories (i.e., if only one cluster is identified, the Inverse Purity will be 1). Tradeoffs are often made between Purity and Inverse Purity: increasing one at the cost of reducing the other.

F-measure combines Purity and Inverse Purity. In our evaluation we use Van Rijsbergen's F-measure [15, 21, 24], which computes the weighted average of maximal F-measure values for each category:

$$\text{F-measure} = \sum_i \frac{|L_i|}{N} \max_j \left\{ F(L_i, C_j) \right\}$$

$$F(L_i, C_j) = \frac{2 * \text{Precision}(L_i, C_j) * \text{Recall}(L_i, C_j)}{\text{Precision}(L_i, C_j) + \text{Recall}(L_i, C_j)}$$

The value of F-measure is from 0 to 1, the higher the better. It is a more robust metric that measures the overall quality of a clustering algorithm.

VI. RESULTS

This section presents our experimental results by addressing the research questions.

A. *Experimental Results for the Research Questions*

RQ1: *How accurate are the buckets produced by ReBucket?*

Table II shows the results achieved by ReBucket for all subject systems. In general, ReBucket can achieve high Purity (ranging from 0.828 to 0.969), high Inverse Purity (ranging from 0.828 to 0.970), and high F-measure (ranging from 0.792 to 0.952). In average, the Purity is 0.925, Inverse Purity is 0.907, and the F-measure is about 0.876.

For all participating products, ReBucket achieves slightly worse Purity but much better Inverse Purity than the WER bucketing method. In terms of F-measure, for all products except Product A, ReBucket achieves better F-measure than the WER method (the improvement ranging from 3% to 56%). For Product A, WER only performs slightly better than ReBucket in terms of F-measure (less than 1%).

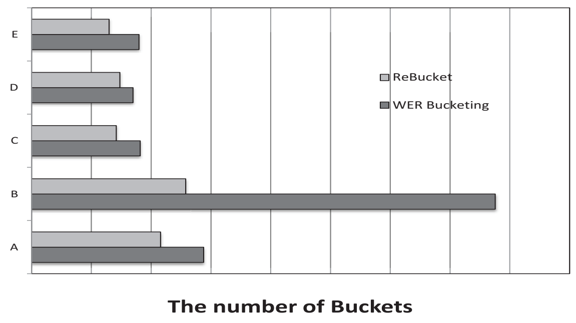
As shown in the Table II, in terms of F-measure, ReBucket performs better than the Prefix Match method in all of the products. Although the Prefix Match method can achieve high precision (Purity values are close to 1 for all products), its recall value (Inverse Purity) is low.

Compared with the Crash Graph method, ReBucket achieves significantly better F-measures for three out of five products (Products A, B and E). The improvement ranges from 0.12 to 0.56. For example, for Product A, ReBucket achieves F-measure 0.792, while Crash Graph achieves 0.676. For the rest of the two products (Products C and D), Crash Graph performs just slightly better than ReBucket.

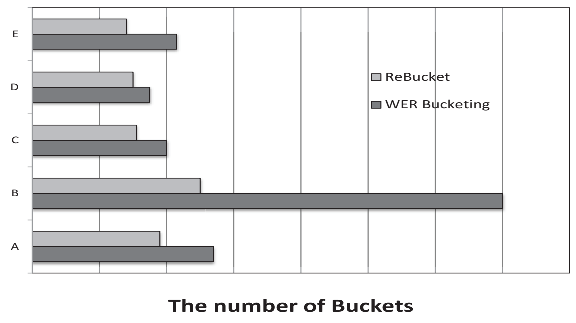
The results show that in general, ReBucket can achieve better overall performance than the existing methods.

Table II. Evaluation Results of ReBucket

Product	Method	Purity	Inverse Purity	F-measure
A	WER Bucketing	0.892	0.840	0.798
	ReBucket	0.828	0.925	0.792
	Prefix Match	0.977	0.412	0.500
	Crash Graph	0.731	0.900	0.676
B	WER Bucketing	0.992	0.466	0.556
	ReBucket	0.969	0.828	0.869
	Prefix Match	1	0.302	0.415
	Crash Graph	0.357	0.957	0.308
C	WER Bucketing	0.995	0.907	0.923
	ReBucket	0.969	0.970	0.952
	Prefix Match	0.992	0.329	0.452
	Crash Graph	0.987	0.959	0.960
D	WER Bucketing	0.918	0.859	0.835
	ReBucket	0.907	0.916	0.861
	Prefix Match	1	0.520	0.651
	Crash Graph	0.932	0.920	0.898
E	WER Bucketing	0.983	0.729	0.791
	ReBucket	0.954	0.897	0.906
	Prefix Match	1	0.407	0.539
	Crash Graph	0.533	0.963	0.493



(a)The comparisons on the number of buckets



(b)The comparisons on the number of buckets that contains 80% crash reports

Figure 7. The impact of ReBucket on bucket numbers

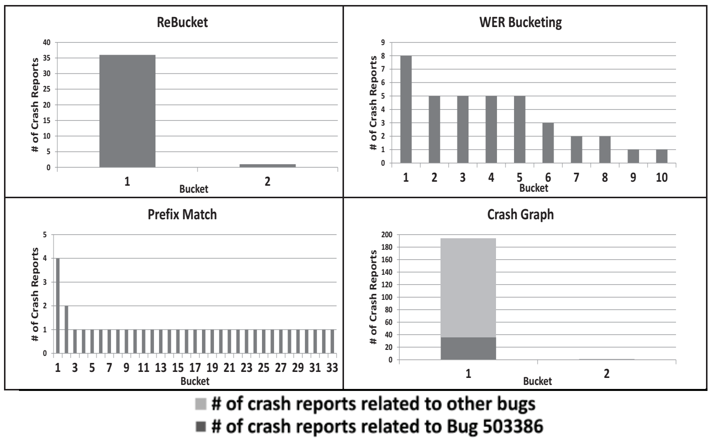


Figure 8. The buckets for Bug 503386

RQ2: *Can ReBucket reduce the number of buckets?*

Figure 7(a) shows the number of buckets produces by ReBucket and WER Bucketing methods. For the sake of confidentiality, the axis labels that indicate the number of buckets are removed. We can see that after applying ReBucket, the number of buckets is reduced. For example, for Product A, ReBucket reduces the number of buckets generated by the WER bucketing method by 25%.

We also find that after applying ReBucket, the number of head buckets (i.e., buckets that contain most of the crash reports) is also reduced. For example, for Product A, using the original WER bucketing method, 80% crash reports are contained in 54 buckets, while using ReBucket 80% crash reports are contained in 38 buckets. Figure 7(b) shows the number of buckets that contains 80% crash reports, for all studied products.

Note that Table II also shows that ReBucket and WER Bucketing methods can achieve similar Purity values, therefore, the reduction of bucket number does not sacrifice precisions. These results indicate that using ReBucket, the "long tail" problem described in Section III is mitigated. The debugging effort can be reduced since developers need to examine less number of buckets.

To illustrate the differences in the number of buckets generated by all compared methods, we select the bug 503386 in Microsoft Access as an example. This bug has 37 associated crash reports. As shown in the Figure 8, ReBucket clusters all the crashes related to bug 503386 into two buckets. One bucket contains 36 duplicate crash reports. One crash report was misclassified into a single bucket. These two buckets have high precision in that they do not contain crash reports for other bugs. The existing WER Bucketing approach generates 10 buckets, which all contain a small number of crash reports (less than or equal to 8). This is a scenario of "long tail" phenomenon, which may cause difficulties in debugging. The Prefix Match method generates even more buckets (33 in total). The above methods all have high precision in that their buckets do not contain crash reports for other bugs. Although the Crash Graph method also produces two buckets for bug 503386, there are 158 unrelated crashes in the first cluster, which are misclassified and make it difficult for developers diagnosing the bug.

B. *Discussions of the Results*

1) Why does ReBucket Work?

In this section, we discuss why ReBucket outperforms other methods in bucketing crash reports.

The heuristics of the WER bucketing approach [10] are mainly based on crash information such as program name, program version, module name, module version, and function offset where a crash occurs. However, some of the heuristics could assign the crashes caused by the same bug into different buckets because the crash reports contain different crash information. For example, the crash reports 6683165 and 3720507 are both caused by bug 503386 and should be placed into the same bucket. Although these two crash reports contain the same Application ID, Module ID and Version ID information, they have different crashing function offsets. Therefore, WER classified them into two different buckets.

The Prefix Match method is based on a call stack similarity metric, which is computed as the number of consecutive frames starting from the top of the stacks. As shown in Table II, the Prefix Match method achieves the best Purity, which indicates that there is little noise in each cluster. However, this similarity metric has limitations when the shared frames are not consecutive. This would result in assigning the duplicate crashes into different buckets. Table II shows that the Prefix Match method produces the largest number of buckets, compared to the other methods.

The Crash Graph method uses a similarity metric based on a graph model. One of the main problems in Crash Graph is that, if the crash graph for a bucket contains a large amount of edges, it is likely to introduce noise. Figure 9 shows one such example, which depicts the crash graphs for Bucket 880522 and Bucket 1016065. Each function is assigned an ID. According to the Crash Graph method, the similarity between these two graphs is high (0.89), so these two buckets should be merged into one bucket. However, the ground truth is that Bucket 880522 and Bucket 1016065 belong to different bugs.

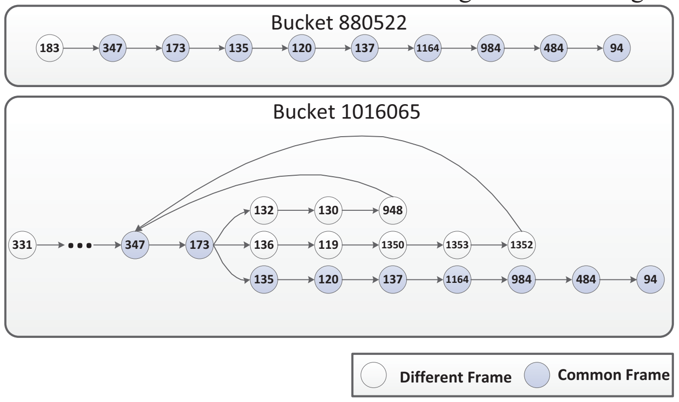


Figure 9. An unsuccessful example in Crash Graph

Unlike the related methods, ReBucket introduces less noise into the buckets and reduces the total number of buckets. There are four reasons. First, ReBucket removes the immune functions, reducing the impact of uninformative functions on similarity measurement. Second, related methods such as the Crash Graph method do not consider the positions of the frames, while ReBucket considers both the position of the frames and the alignment offset between the matched frames. As a result, the unsuccessful example in Figure 9 can be avoided using ReBucket. Third, ReBucket adopts the Agglomerative Hierarchical clustering method, and takes the maximum distance between the call stacks in each cluster as the clustering distance. Only when a crash is a duplication of all the crashes in a bucket, this crash can be merged into this bucket. Fourth, ReBucket introduces a training process to tune parameters for different projects. We have verified that, without a training process, if we use fixed values for the parameters for all projects (such as setting *c* and *o* to 0), the performance of ReBucket would drop dramatically.

2). The Misclassified Cases by ReBucket

Although our evaluation results show that ReBucket can achieve better overall results than the existing methods, ReBucket could still misclassify some crash reports.

For example, if the top *k* frames in two call stacks are the same, ReBucket is likely to consider these two call stacks similar. However, if the real buggy frame is close to the middle or bottom of the call stacks, it is possible that the two crashes are actually caused by different bugs. One such example is shown in Figure 10, which is an example taken from Microsoft Visio project. Two crashes (crash 378686808 and 375508594) share the top eleven frames and thus have a high similarity (0.927), ReBucket wrongly clusters these two reports into the same bucket. But actually the buggy frames (f11 and f'11) contain two different bugs (bug 640606 and 644158). However, our empirical studies show that such case is rare (in less than 5% of call stacks).

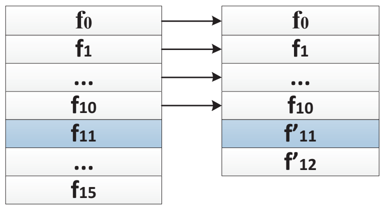


Figure 10. An unsuccessful example in ReBucket

The other possible reason that may cause wrong bucketing is that ReBucket uses parameters learned from historical data for clustering. Bias may be introduced from the sampling of the training data.

VII. Applying ReBucket In Practice

In this section, we discuss several issues when applying the ReBucket method in practice.

Since a bucket can contain many duplicate crash reports, manual examination of bucket information could be a time-consuming activity. In practice, to further help developers examine the bucketing results produced by ReBucket, we visualize the bucketing results in a graphical manner. For call stacks that belong to the same bucket, we construct the call graph for each call stack and merge all the call graphs together (based on the common nodes and edges). For example, Figure 11 illustrates a merged call graph for a bucket, where all the call stacks in a bucket share the same *i+1* frames in the bottom but differ in top frames. The directed edges indicate the invoking relationship between two frames and the weight for each edge (such as w_0) indicates the number of times the relationship appears in the crashes. Such graphs can help developers understand the behavior of crashes in a bucket. Another benefit of such graphical representation is that developers can easily detect patterns associated with certain types of bugs. For example, developers have found that the pattern shown in Figure 11 actually indicates the existence of Heap-related bugs.

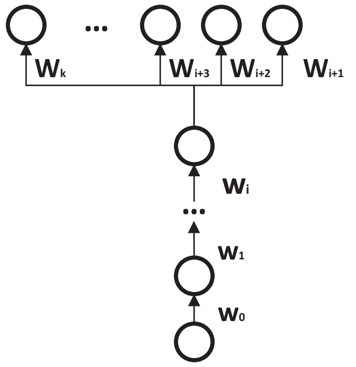


Figure 11. A merged call graph for a bucket

Computing cost is a practical concern when applying our method. For the five subject systems we evaluated, the computing cost ranges from 15 minutes to 2 hours. To reduce the time for computation, the clustering process can be accelerated by distributing computing. This is because most of the computing time is spent on the calculation of call stack similarity and the calculation of similarity for every call stack pair is independent with each other.

ReBucket requires historical data in order to train the PDM model to obtain the optimal parameters. For new projects that do not have sufficient historical data, the training process could be an iterative process, in that the parameters can be set with initial values, and then later tuned when sufficient data is accumulated.

To evaluate the effectiveness of ReBucket in practice, we have closely worked with a Microsoft product team and they have helped review 20 buckets produced by ReBucket. The developers confirmed that 70% of buckets are very meaningful clusters, because these buckets helped them understand the problems and debug. As an example, ReBucket successfully helped developers identify crash reports that were associated with a known bug (http://support.microsoft.com/kb/837320).

VIII. THREATS TO VALIDITY

We identify the following threats to validity:

- **Subject selection bias**: We use only Microsoft product data in our experiments. Projects of other companies may have different crash properties and the same experiments on open source projects may yield different results.
- **Data accuracy**: We assume that the crash data used in our evaluation is of high quality. Although the mappings between a crash report and a bucket are examined and confirmed by developers, it is possible that some data might be incorrect due to occasional human errors. However, although we cannot ensure the 100% accuracy for the mapping, we are confident with the overall quality of the data.
- **Popular systems**: Our bucketing method is based on the assumption that the software has a large customer base and can receive a large number of crash reports reported by the users. If the number of crash reports is small, the performance of ReBucket could be affected.

IX. RELATED WORK

In recent years, many studies have been dedicated to the analysis of crashes of real-world, large-scale software systems. These work studied the construction of crash reporting system [10], the replication of crash [3], the cause of crash [9], the prediction of crash-prone modules [13], and statistical debugging [7,17]. Our work focuses on bucketing crash reports.

Recently, some researchers have proposed methods for detecting duplicate crash reports. We have already discussed and compared the Prefix Match and Crash Graph methods in previous sections. Liu and Han [16] proposed R-Proximity, which regards two failing traces as similar if the same fault locations are suggested by fault localization methods. Bartz et al. [5] proposed a method for finding similar crash reports based on call stack similarity. They used a tuned call stack edit distance with five different edit operation costs. Their method requires to set many penalty parameters. For example, their full model requires to tune 11 parameters. We also found that the automatically learning of the optimal values of these parameters incurs heavy computational cost and is thus difficult to implement.

Lohman et al. [18] proposed an architecture for quickly detecting the recurrences of crashes. They also proposed a call stack matching metric. Similar to our work, they also considered the distance to the top frame and alignment offset. However, our method differs in the way the similarity metrics are formulated and in the way the parameters are tuned.

Modani et al. [19] also proposed to automatically identify known crash problems by comparing call stack similarities. They proposed two similarity metrics, top-k indexing and inverted index. The top-k indexing method is based on the intuition that two crash call stacks are likely caused by the same bug if both stacks share the same top k frames. The inverted index method is based on the assumption that two call stacks sharing more common functions have higher similarity. Their evaluation results show that their similarity methods are comparable to the Prefix Match method. Our proposed method is also based on similarity measures between call stacks. We take more information into consideration, such as the distance to the top frame, and an offset distance between the common functions. Our evaluation results show that our method outperforms the Prefix Match method.

Our work is also related to the detection of duplicate bug reports. The bug reports recorded by bug tracking systems (such as Bugzilla) contain detailed bug descriptions and can be treated as textual documents. Therefore, researchers have applied information retrieval techniques to compare the textual similarity between two bug reports and determine if they are duplicates [22]. Wang et al. [25] also combined the text similarity and execution trace similarity to detect the duplicate bug reports. The accuracies of their methods are around 40%~60%. DebugAdvisor [4] is a recommendation system that can detect similar bugs for a new bug and automatically recommend relevant information (such as experts and source files) useful for debugging. It can extract call stack from bug reports (if any) and use the call stack similarity as one feature for estimating the similarity of bug reports. Our method targets at clustering duplicate crash reports based on call stack information that is collected by a crash reporting system. We do not assume the availability of detailed bug descriptions.

X. CONCLUSIONS

A widely-used, large scale software system could receive a large number of crash reports, among which many are duplicates. In Microsoft, the Windows Error Reporting system (WER) implemented a bucketing method to automatically group duplicate crash reports into buckets. However, the performance of WER bucketing method can be improved as it may classify crash reports caused by the same bugs into multiple buckets.

In this paper, we have proposed a novel bucketing method called ReBucket. Our method is based on the Position Dependent Model (PDM), which is a similarity measure for call stacks. PDM computes the similarity between two call stacks based on the number of functions on two call stacks, the distance of those functions to the top frame, and the offset distance between the matched functions. We have also designed a training process for ReBucket to tune the parameters required by PDM.

We have evaluated ReBucket using crash data collected from five widely-used Microsoft products. The results show that ReBucket achieves better overall performance than the related methods. In average, the F-measure achieved by our method is about 0.88. A Microsoft product team has also confirmed the usefulness of 20 sampled buckets produced by ReBucket.

We believe the proposed ReBucket method can help developers prioritize debugging efforts and facilitate problem diagnostic. In the future, we will investigate if the proposed bucketing method can be applied to projects of other organizations, as well as open source projects.

ACKNOWLEDGEMENT

We thank Song Ge, Song Huang, and Xiaohui Hou who helped on the prototype system development. We thank our Microsoft partners who gave us data access, evaluated our results and gave us comments, in particular, Meredith McClurg, Ben Canning, Ben Ross, Mike Hollinshead, and Amanda Kauffman. We thank our colleagues who discussed with us on this topic and gave us comments, including Shi Han, Thomas Moscibroda, and Ryan Kivett. We thank all Microsoft developers who tried our deployed prototype system and gave us comments. The Tsinghua co-authors also thank the support of the NSFC grant 61073006 and Tsinghua University research grant 2010THZ0.

REFERENCES

[1] E. Amigo, J. Gonzalo, J. Artiles, and F. Verdejo, A comparison of extrinsic clustering evaluation metrics based on formal constraints. *Inf. Retr.* 12, 4, August 2009, 461-486.

[2] Apple, “Technical Note TN2123:CrashReporter”, 2010, available at : http://developer.apple.com/library/mac/#technotes/tn2004/tn2123.html.

[3] S. Artzi, S. Kim, and M. D. Ernst, ReCrash: Making software failures reproducible by preserving object states, In *Proc.ECOOP 2008*, Paphos, Cyprus, July 2008. pp. 542-565.

[4] B. Ashok, J. Joy, H. Liang, S. Rajamani, G. Srinivasa, and V. Vangala, DebugAdvisor: A Recommender System for Debugging, in *Proc. ESEC/FSE'09*, Amsterdam, The Netherlands, August 2009. pp. 373-382.

[5] K. Bartz, J. W. Stokes, J. C. Platt, R. Kivett, D. Grant, S. Calinoiu, and G. Loihle, "Finding similar failures using callstack similarity," in *Proceedings of the Third conference on Tackling computer systems problems with machine learning techniques*. San Diego, California: USENIX Association, 2008.

[6] M. Brodie, S. Ma, L. Rachevsky, and J. Champlin. Automated Problem Determination Using Call-Stack Matching. *Journal of Network and Systems Management*, Vol. 13, No. 2, June 2005.

[7] T. Chilimbi, B. Liblit, K. Mehra, A. Nori, K. Vaswani. “Holmes: effective Statistical Debugging via Efficient Path Profiling”. In *Proceedings of ICSE 2009*, pages 34-44, ACM Press, 2009.

[8] T. Cormen, C. Leiserson, R. Rivest and C. Stein, *Introduction to Algorithms (2nd ed.)*, MIT Press and McGraw-Hill. pp. 350–355, 2001.

[9] A. Ganapathi, V. Ganapathi, and D. Patterson, "Windows XP kernel crash analysis," in *Proceedings of the 20th conference on Large Installation System Administration*. Washington, DC, USENIX Association, 2006, pp. 12-12.

[10] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt, "Debugging in the (very) large: ten years of implementation and experience," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. Big Sky, Montana, USA: ACM, 2009, pp. 103-116.

[11] J. Han and M. Kamber, *Data Mining: Concept and Techniques*, Elsevier, 2006.

[12] T. Hastie, R. Tibshirani, J. Friedman, *The Elements of Statistical Learning (2nd ed.)*. New York: Springer, 2009.

[13] D. Kim, X. Wang, S. Kim, A. Zeller, S. C. Cheung, and S. Park., "Which Crashes Should I Fix First?: Predicting Top Crashes at an Early Stage to Prioritize Debugging Efforts," *IEEE Trans. Softw. Eng*., 2011.

[14] S. Kim, T. Zimmermann, N. Nagappan, Crash graphs: An aggregated view of multiple crashes to improve crash triage, *Proc. 41st International Conference on Dependable Systems & Networks (DSN),* Hong Kong, June 2011, 486 – 493.

[15] B. Larsen and C. Aone, Fast and effective text mining using linear-time document clustering. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining* (KDD '99), San Diego, CA, 1999, pp. 16–22.

[16] C. Liu and J. Han, Failure Proximity: A Fault Localization-Based Approach, in *Proc FSE'06,* Portland, OR, 2006, pp. 286-295.

[17] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, M. Jordan. “Scalable statistical bug isolation”. In *Proceedings of PLDI 2005*, pp. 15-26.

[18] G. Lohman, J. Champlin, and P. Sohn. 2005. Quickly Finding Known Software Problems via Automated Symptom Matching. In *Proceedings of the Second International Conference on Automatic Computing* (ICAC '05). IEEE Computer Society, Washington, DC, USA, 101-110.

[19] N. Modani, R. Gupta, G. Lohman, T. Syeda-Mahmood, and L. Mignet, Automatically identifying known software problems. In *ICDE Workshops*, pages 433–441, 2007.

[20] Mozilla, "Crash Stats", 2010, http://crash-stats.mozilla.com.

[21] C.J. Van Rijsbergen. (1974). Foundation of evaluation. *Journal of Documentation*, 30(4), 365–373.

[22] P. Runeson, M. Alexandersson, and O. Nyholm, Detection of Duplicate Defect Reports Using Natural Language Processing, in *Proc. ICSE 2007*, Minneapolis,USA, May 2007. pp. 499-510.

[23] A. Schröter, N. Bettenburg, and R. Premraj, “Do stack traces help developers fix bugs?”, in *Proc. MSR 2010*, Cape Town, South Africa, May 2010. pp. 118–121.

[24] M. Steinbach, G. Karypis, and V. Kumar, A comparison of document clustering techniques, in *Proc KDD 2000*, Boston, MA, pp. 109–110.

[25] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proc. ICSE'08,* Leipzig, Germany, 2008, pp. 461-470.

[26] A. Zeller. *Why does my program fail? A guide to automated debugging*. Morgan Kaufmann, May 2005.