Crash Graphs: An Aggregated View of Multiple Crashes to Improve Crash Triage

Sunghun Kim*
Hong Kong University of Science and Technology
Hong Kong
hunkim@cse.ust.hk

Thomas Zimmermann, Nachiappan Nagappan
Microsoft Research
Redmond, WA, USA
{tzimmer, nachin}@microsoft.com

Abstract—**Crash reporting systems play an important role in the overall reliability and dependability of the system helping in identifying and debugging crashes in software systems deployed in the field. In Microsoft for example, the Windows Error Reporting (WER) system receives crash data from users, classifies them, and presents crash information for developers to fix crashes. However, most crash reporting systems deal with crashes individually; they compare crashes individually to classify them, which may cause misclassification. Developers need to download multiple crash data files for debugging, which requires non-trivial effort. In this paper, we propose an approach based on crash graphs, which are an aggregated view of multiple crashes. Our experience with crash graphs indicates that it reduces misclassification and helps identify fixable crashes in advance.**

Keywords-component; crash; graph; triaging; network

I. INTRODUCTION

Crash reporting systems [1] such as Windows Error Reporting (WER) [13], Mozilla Crash Stats [15], and Apple CrashReporter [3] have been widely deployed in practice. The crash reporting systems help organizations determine the overall reliability of their software systems in the field.

The crash reporting systems collect crash related data, classify them, and present the information to developers to fix crashes [9, 11]. For example, WER receives crash stack traces from users in the field. Then, WER identifies crash root cause(s) using heuristics (analyzing the traces using machine learning/pattern analysis algorithms) and put similar crashes into one bucket. WER counts crashes per bucket to decide which buckets to fix first. If the number of crash reports in a bucket exceeds a threshold value, then WER automatically reports such crashes as bugs. Developers investigate these bug reports and fix the crashes using the collected crash data from users in the field.

This practice helps developers quickly identify important and frequent crashes and fix them. Crash data provided by users is useful for developers to identify the root causes of the crashes and as a result debugging crashes is easier.

However, most crash reporting systems including WER deal with crashes individually rather than aggregating them into a combined view [9]. This individual crash based technique is computationally efficient for data collection but difficult to analyze the vast repositories of data. For example, WER uses individual crash data to bucket crashes. When considering crash data one by one, WER may misclassify crashes.

When WER reports a crash as a bug, it provides multiple crash data files to developers. Then, to investigate and debug one crash bug, developers need to download multiple data files one by one, since crash bug reports include multiple crash data files. This process requires non-trivial effort. This is similar in spirit to how other crash collection systems (like Mozilla) work [11].

In this paper, we propose *Crash Graphs* which capture multiple crashes at once and provide an aggregated view of multiple crashes in the same bucket. These crash graphs are useful for developers to get high-level information about crashes in the same bucket. Crash graphs are also useful for crash classification since they include aggregated information of crashes.

We evaluate crash graphs using two Microsoft products, Microsoft Windows and Microsoft Exchange Server. First, we use crash graphs to detect duplicate crash-bug reports. Our crash graphs identify duplicate bug reports with 71.5% precision and 62.4% recall. Second, we predict fixable crashes using features from crash graphs. A machine-learning algorithm using crash graph features predicts fixable crashes with 72~80% precision, which is useful for automatic crash triaging.

A. Contributions

Our paper makes the following contributions:

- **Crash Graphs**: We propose an aggregated view of multiple crashes in the same bucket.
- **Experience**: We present several experiences with crash graphs, which experimentally show the usefulness of crash graphs for crash triage tasks, more specifically, predicting fixable crashes and detecting duplicate reports.

Overall, our experiences in practice reveal the crash graph approach, an aggregated view of crashes, is efficient for crash triaging.

B. Section Guide

In the remainder of the paper, we start by presenting the background of crash reporting systems in Section II. Section III presents our crash graph building algorithms and hypothesizes the usefulness of crash graphs. Our crash graph experience is presented in Section IV and its limitations are discussed in Section V. Section VI surveys related work and Section VII concludes.

II. BACKGROUND

In this section, we present the WER system, the common crash debugging process using WER, and WER challenges.

* Sunghun Kim was a visiting researcher with the Empirical Software Engineering Research Group (ESE), Microsoft Research in the summer of 2010 when this work was carried out.

A. *Windows Error Reporting System*

Software crashes are manifestations of errors from actual field usage. Developers spend tremendous resources and efforts to fix crash bugs before releasing products. However, often, released programs include bugs/errors (due to various factors ranging from incorrect code to improper interaction with third party applications), and some of the bugs manifest as crashes in the field.

To collect crash information from the field, crash-reporting systems such as WER, Apple CrashReporter, and Mozilla crash stats have been proposed and deployed widely. Most of these systems have three modules: (1) collecting crash information from clients, (2) classifying crashes in the server side, and (3) presenting the crashes to developers to facilitate debugging.

Figure 1 shows the WER system overview.

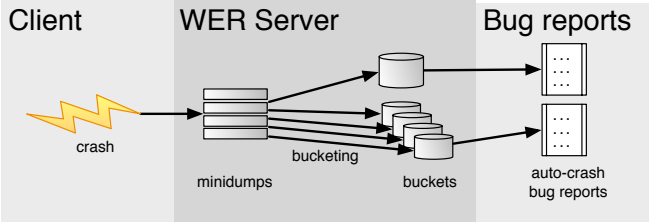

Figure 1. WER System overview

Collecting Crashes: Windows OSs such as Windows XP or Windows 7 include a WER client in the OS level. Some programs such as Microsoft Office have their own WER clients. When a crash occurs in the field, the WER client shows a popup screen and allows users the option to send crash information as shown in Figure 2. If users accept, the WER client collects crash related data such as stack traces, static variables and register values, and packs this information as a minidump file. This minidump file is sent to the WER server.

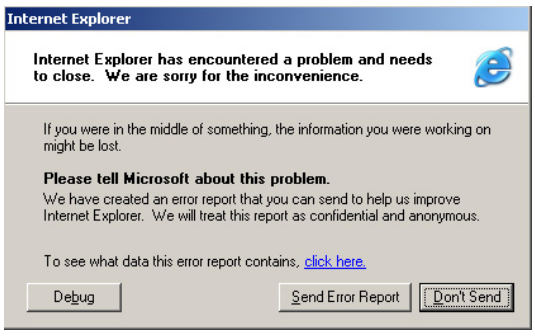

Figure 2. WER client popup screen

Classification: The WER server classifies the minidump files received from WER clients. First, WER server identifies the names of crashed modules by resolving *Windows Symbols* which are similar to debug symbols [9]. Then, using heuristics called bucketing algorithms [9], the WER server classifies crashes based on causes and collects similar crashes in the same bucket. This bucketing process is a core part of WER. Buckets are the basic unit of crash triaging. WER counts crash hits per bucket to determine crashes occurring most frequently. After crash hits of buckets exceed a predefined threshold value, WER automatically reports the bucket to developers.

Presenting Crashes: WER presents highly hit buckets and their crash data as bug reports, called *auto-crash bugs*. WER can identify crashed modules by resolving Windows Symbols and then mapping the owners of the modules. When WER reports auto-crash bugs, WER assigns the bugs to the module owners.

These bug reports include statistics such as hit counts, client distributions, and crashed software versions. The most important information in bug reports is minidump files which includes crash stack traces. Usually an auto-crash bug report includes more than one minidump file, since WER collects multiple minidump files from multiple crashes for each bucket.

B. *WER Common Debugging Process*

Once WER has automatically reported crash bugs and assigned them to developers, developers start debugging by reading the bug report and analyzing the statistics (frequency of crashes) associated with the bug report. Then developers download multiple minidump files and investigate the crashes using debugging tools such as windbg [2] and "!Analyzer" [9]. Often, stack trace information in minidump files is very useful to fix the corresponding crashes [6, 9].

C. *Challenges*

In this section, we briefly discuss challenges in crash reporting systems.

Second-bucket problem: In general, the WER bucketing algorithm, based on over 500 heuristics [9] (we do not discuss the WER process in detail here as it is not the goal of the paper and only briefly touch on it given reference [9] which discusses it in detail) works well and helps developers identify crash causes quickly. However, it is possible that the bucketing algorithm puts crashes caused by the same bug into different buckets; this is called the *second-bucket problem*. For MS Office products, about 30% of crashes have this second bucket problem [9].

This second bucket problem yields duplicate bug reports. Since WER regards crashes in different buckets as different types of crashes, they become different bug reports, i.e. duplicated bug reports. These duplicated bug reports significantly consume developers' resources. Developers often realize the existence of duplicates only after putting significant efforts to investigate the reports and the corresponding crashes [16, 17].

[2] http://www.microsoft.com/whdc/devtools/debugging/default.mspx

Manual inspection of multiple minidump files: Multiple minidump files provided in auto-crash bug reports constitute useful information for debugging crashes. Though the cause of all crashes (in the same bucket) is the same, it is possible that information in different minidump files in the same bucket differ. These variations are very useful for developers to understand the context of the crashes and to identify the root cause (bug) for the crashes. For this reason, WER usually provides 10 to 20 minidump files per crash.

However, in the current practice, developers need to download multiple minidump files one by one and analyze them. This is a labor-intensive task. In addition, it is possible that one minidump file by itself does not have enough information to identify the root cause of the crash.

We propose Crash Graphs to address these challenges.

III. CRASH GRAPH

Crash graphs combine all crash traces in one bucket and provide an aggregated view of all crashes in a bucket. Since crashes in the same bucket share the main cause, crash graphs provide a high-level information of all crash traces in detail. In addition, crash graphs can show trace variations, which help developers understand the context of the crashes and identify the bugs. In section III.A, we present the crash graph construction technique and in section III.B, our hypotheses evaluating the usefulness of crash graphs.

A. Graph Construction

We construct crash graphs from crash traces in minidump files, and the frames (functions) in crash traces are the first class element of crash graphs; they become nodes in the graphs. Their call relations become edges in the graphs.

To construct a crash graph from multiple crash traces, the first step is to decompose each crash trace to two-frame elements. From a crash trace A→B→C→D shown in Figure 3, we get three two-frame elements, A→B, B→C, and C→D by decomposing the crash trace. In the same manner, we decompose all crash traces in the same bucket.

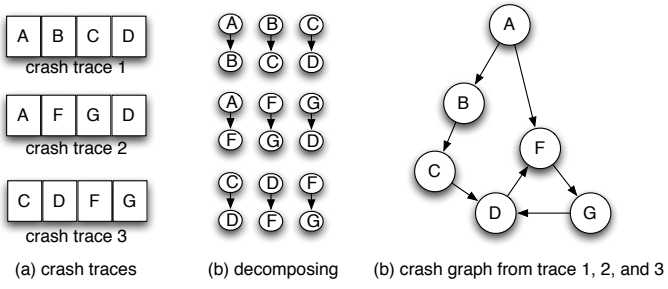

Figure 3. A crash graph example from multiple crash traces.

The frames in a two-frame element become two nodes in the graph. Then, we add an edge between these two nodes in the graph. We continue this process for all decomposed elements. This graph construction technique is inspired by bug tossing graphs [10], which decompose tossing (reassignment) relations and construct graphs using decomposed elements.

Since some frames may appear more than one time, nodes and edges in a crash graph can be weighted. For example, frame A appears 2 times in traces in Figure 3. Therefore weight of node A in the crash graph is 2. Similarly, since edge C→D appears 2 times, the edge weight is 2.

In this paper, we use both weighted and un-weighted crash graphs for bug triaging tasks.

B. Hypotheses

Crash graphs combine all crashes in the same bucket and show the entire view of all crashes. We investigate if crash graphs can be used to detect the second bucket problems and predict fixable crashes in advance.

Specifically, we hypothesize the following:

Hypothesis 1: *Crash graphs can detect duplicate crash reports (the second bucket problem) with high accuracy.* Currently, WER compares individual crash traces to bucket them. Since crash graphs combine all traces together, combined crashes are more efficient in detecting the second bucket problem and thus in identifying duplicated auto-crash bug reports.

Hypothesis 2: *Crash graphs can predict if a given crash will be fixed.* Since crash graphs capture properties of all crashes in one bucket, graph features can be useful to predict fixable crashes.

IV. EXPERIMENTS

This section presents our crash graph experience of applying the graphs for crash triage tasks.

A. Duplicate Detection

WER uses bucketing algorithms based on over 500 heuristics to identify causes of crashes and to classify the crashes into buckets based on their causes [9]. Each bucket is a basic unit for crash triage, i.e. prioritizing crashes based on hit counts in each bucket.

In most cases, WER bucketing algorithms work reasonably well. However, due to non-deterministic properties of crashes, those caused by the same bugs may also produce slightly or (sometimes) significantly different crash traces. As a result, the bucketing algorithms put such crashes into different buckets. This second bucket problem leads to duplicated auto-crash bug reports as discussed in Section II.C.

We apply our crash graphs to automatically detect duplicate auto-crash bug reports. First, we propose a crash graph similarity measure to detect duplicates in Section IV.A.1 and show the experimental results using Windows OS bug reports in Section IV.A.2. Section IV.A.3 discusses the results and the role of crash graphs in detecting duplicates.

1) Measure

Since a crash graph represents multiple crashes in one bucket, our approach is to compare two crash graphs from two bug reports to determine if they are duplicates.

Graph similarity measures have been widely proposed and used in various domains including face recognition [19], text mining [14], and social network analysis [12].

In this paper, we use the following graph subset similarity measure to compare two crash graphs G_1 and G_2:

$$Sim(G_1, G_2) = \frac{|E_1 \cap E_2|}{min(|E_1|, |E_2|)}$$

where E is the set of edges in G.

Basically, this equation measures if a smaller graph is a subset of the bigger graph. In this equation, we ignore the node or edge weights.

2) Experiments

We measure the similarity of two given bug reports using the equation shown in Section IV.A.1. If the similarity is above a threshold, we assume the two bug reports are duplicates.

To evaluate the similarity measure using crash graphs, we use auto crash-bug reports from the Windows OS project. These bug reports include duplicates due to the second bucket problem. These duplicates are manually marked by Windows OS developers which allows us to examine the efficiency of our method. In total, we use '*n*' (anonymized for confidentiality) bug reports from Windows OS projects. Among them, 13.3% of the reports are duplicates. We apply the crash graph construction algorithm and the similarity measure, and check if our approach can detect these manually marked duplicates.

Since our approach compares similarity of two given crash graphs (bug reports), we conduct pair-wise comparisons for all bug reports. As shown in Table I, there are *n*(n-1)/2* pairs for *n* bug reports. Among these pairs, only 0.32% are duplicated pairs.

To evaluate the performance of duplicate detection, we use recall and precision measures [1].

The **recall** for a given similarity threshold denotes:

$$\text{Recall (similarity)} = \frac{|MD \cap PD_{similarity}|}{|MD|}$$

where $|MD|$ is the number of manually marked duplicates, $PD_{similarity}$ is predicted duplicates based on the given threshold value, and $|MD \cap PD_{similarity}|$ is the number of correctly predicted duplicates.

The **precision** for a given similarity threshold value is:

$$\text{Precision (similarity)} = \frac{|MD \cap PD_{similarity}|}{|PD_{similarity}|}.$$

In general, identifying only 0.32% of the entire population is a challenging problem. The precision of existing approaches for detection of duplicate bug reports is around 40~60% [16, 17]; recall is typically very low or not measured.

TABLE I. WINDOWS OS BUG REPORTS FOR DUPLICATED BUG DETECTION

Name	*Value*
# of bug reports	n
# of duplicated bugs	13.3%
# total bug pair	n*(n-1)/2
# of duplicated bug pair	0.32%
# of non-duplicated bug	99.68%

Note that sophisticated bucketing algorithms are already applied and missed the duplicates used in our experiment data.

Table II shows the precision and recall of detecting duplicates using the crash graph similarity measure. The precision and recall vary based on the similarity threshold values. For example, when the similarity threshold is set to 0.95, the recall and precision is around 60%. On setting the threshold to 0.98, the precision is over 70% while recall remains around 60%.

TABLE II. DUPLICATE DETECTION PRECISION AND RECALL FOR SELECTED SIMILARITY THRESHOLD VALUES.

Similarity Threshold	*Precision*	*Recall*
1	70.3	58.8
0.99	71.5	62.4
0.98	71.0	63.6
0.97	68.4	64.2
0.96	65.0	64.2
0.95	61.6	64.2

The low recall is due to non-deterministic behaviors of bugs and crashes. It is possible that the same bugs can manifest completely different crash traces.

Even if one graph is completely a sub-graph of another (similarity is 1), the precision will not reach 100%. This is due to manual duplication marking, since the developer may neglect or forget to mark duplicates in the bug reporting system. In this case, we cannot decide if our prediction is correct, so we conservatively assume it is a wrong detection. Thus, our precision will not reach 100%, even if we only consider the exact sub-graphs (similarity is 1).

Figure 4 shows the precision-recall curve for various threshold values. Overall, recall and precision are around 60%. By sacrificing recall, precision can be increased to over 70%. Our experimental results indicate crash graphs can detect duplicated bug reports with reasonable accuracy. Note that previous approaches [16, 17] yield around 40-60% precision.

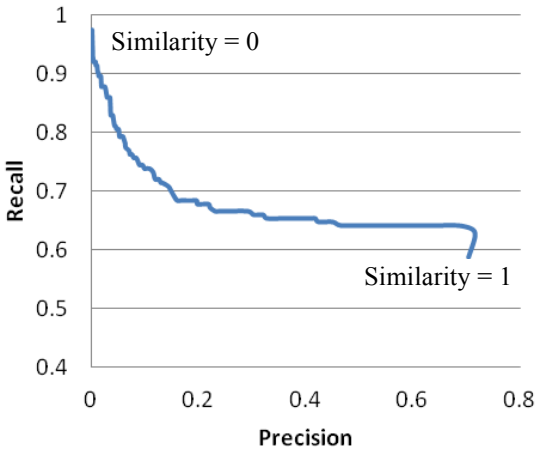

Figure 4. Precision-recall curve for duplicate detection.

3) *Discussion*

In this section, we discuss why crash graphs can efficiently detect duplicated reports missed by sophisticated bucketing algorithms. One simple explanation is that the crash graph is an aggregation of all crashes in a bucket. Comparing all crashes using crash graphs is more efficient than comparing crashes one by one.

The current WER bucketing algorithms compare and classify crashes one by one. When a client sends a new crash, WER creates a new bucket and makes the first crash as the representative for the bucket. If there is another new crash, WER compares the crash with representative crashes in each bucket to decide if the new crash belongs to any of the existing buckets.

This comparison is computationally efficient, but may cause the second bucket problem. As shown in Figure 5, suppose Trace 1 is the representative crash for Bucket 1. Later, Trace 2 and 3 are collected from clients. Suppose the trace similarity threshold is 90% – if the similarity of a new crash and Trace 1 is over 90%, the new crash will be put in Bucket 1. Suppose the trace similarity between Trace 1 and 2, and Trace 1 and 3 are over 90%. Then, Trace 2 and 3 will be put in Bucket 1.

However, since WER only measures similarity with the representative crash, it is possible that trace X is similar to one of the other crashes in the bucket, but not similar enough with the representative crash to be put in Bucket 1. This may result in a second bucket problem as shown in Figure 5.

However, since crash graphs compare all traces, they are more effective for avoiding the second bucket problem.

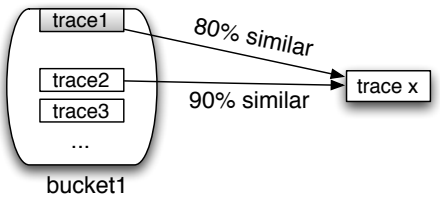

Figure 5. Bucketing algorithms which compare crash traces one by one.

Another reason of the second bucket problem is partial crash traces. It might be possible that only partial crash traces are sent to the WER server. In this case, measuring partial trace similarity may cause the second bucket problem.

For example, suppose we have two traces in Bucket 1 as shown in Figure 6. Suppose a client sent a new crash trace to the WER server. Unfortunately, it is a partial trace, or due to the missing symbol information, we can figure out only partial frame names. If we just compare similarities between two crash traces, the new trace will be put in a new bucket, since it is not similar to Trace 1 or Trace 2 in Bucket 1.

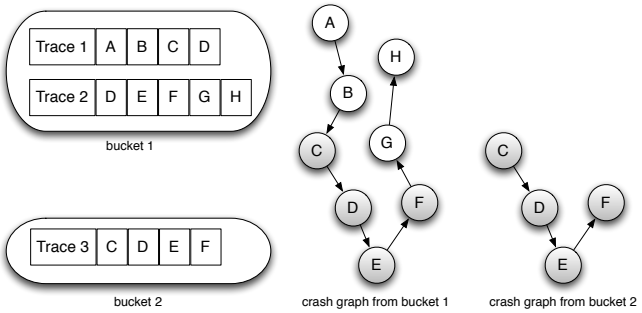

Figure 6. Second bucket problem due to partial crash trace. Crash graphs do not suffer from this issue.

However, when we construct a crash graph from Bucket 1, and compare similarity using the crash graph, Trace 3 is a complete sub-graph of the crash graph from Bucket 1. Our crash graph approach identifies them as the same crash.

B. *Predicting Fixable Crashes*

In this section, we investigate if crash graphs are useful to predict fixable crashes, since we believe crash graphs can capture crash properties such as fixability. Some crashes will not be fixed for various reasons. For example, a crash can occur due to third party software bugs or specific hardware issues. Often, identifying the fixability of a given crash requires manual effort. If we can predict fixable crashes in advance with reasonable accuracy, it helps developers triage crashes.

We first collect auto-crash bug repots from Windows 7 and Exchange 14. From each bug report, we construct a crash graph and extract (machine learning) features from each graph. We use the features to train a model to predict if a given auto-crash bug report is fixable or not.

1) *Subjects and Features*

For our experiments, we use auto-crash bugs from Windows 7 and Exchange 14 obtained from field crashes. From each auto-crash bug report, we construct weighted crash graphs as explained in Section III.A. From crash graphs, we extract the following machine learning features:

Simple graph complexity: We first compute the graph complexity and density [18]. Although graph complexity is extracted for graphs in general, it is possible the complexity of a crash graph may capture the properties of the crash. We use common graph complexity measures such as the

node/edge count and max in/out of nodes. Since we are using weighted crash graphs for these experiments, we also use weight related measures such as max/min weights of nodes and edges, and min/max out weight sums. Table III shows and explains selected features.

Distance-based complexity: Besides the simple graph complexity, we extract distance-based complexity measures based on the shortest distance between all pairs of crash graph nodes using the Floyd-Warshalls algorithm [7]. The initial distance between two connected nodes is set to 1. Then, we compute distance-based complexities such as eccentricity, density and radius. For example, the eccentricity of a node *v* is the greatest distance between *v* and any other node. We aggregate all eccentricities with minimum (=radius), maximum (=diameter) and average. Table III describes selected features, while detailed measures are described in [21].

Bug metadata: Bug metadata is widely used to classify bug reports [2, 10]. In our experiment, we extract features from auto-crash bug reports such as hit count, milestone, severity and priority. The hit count is very important to prioritize crashes to fix. The milestone (version) is also a good feature candidate, since developers care more/less about some milestones or releases. In addition, severity and priority are used as features.

We compare our crash graph feature based prediction performance to a baseline approach which uses the bug metadata features.

TABLE III. SELECTED MACHINE LEARNING FEATURES TO PREDICT FIXABLE BUGS

Group	Features	Explanation
Bug meta data	Hit Count	Crash hit count
	Milestone	Milestone of crashed program
	Severity	Severity of the crash
	Priority	Priority of the crash
Crash graph features	Node/edge count	Count of nodes and edges
	Max in/out	The number of incoming/outgoing edges of nodes
	In/out ratio	Edge in/out ratio
	Eccentricity	Average distances between nodes
	Density	Ratio of the number of edges and the number of possible edges
	Diameter	Max length (longest shortest path)
	Radius	Node radius

2) Experiments

Our approach is to train a machine learner using features described in Section IV.B.1. We use decision tree [1] as our machine learner, which is widely used to triage bug reports and predict software defects.

From our subjects Windows 7 and Exchange 14, we construct a corpus by extracting features and labels ("fixed" or "won't fix"). For the evaluation, we use random splits: to train a machine learner, we randomly select 2/3 of instances and use them as a training set; the remaining 1/3 is used as a testing set. To avoid label population bias in the training set, we make sure that the instances in the training set have 50% fixed bugs and 50% of won't fix bugs by randomly removing some instances in the training set. To also avoid sampling bias, we run this experiment 100 times and compute the average performance.

To measure the model performance, we use standard measures including precision, recall and F-measure [1, 20]. Applying a machine learner to our problem can result in four possible outcomes: the learner predicts (1) a fixable crash as fixable (f → f); (2) a fixable crash as won't fix (f → w); (3) a won't fix crash as fixable (w → f); and (4) a won't fix crash as won't fix (w → w). These outcomes can be then used to evaluate the classification with the following three measures:

Precision: the number of crashes correctly classified as fixable ($N_{f\to f}$) over the number of all methods classified as fixable.

$$\text{Precision}\quad P(fix) = \frac{N_{f\to f}}{N_{f\to f} + N_{w\to f}}$$

Recall: the number of crashes correctly classified as fixable ($N_{f\to f}$) over the total number of fixable crashes.

$$\text{Recall}\quad R(fix) = \frac{N_{f\to f}}{N_{f\to f} + N_{f\to w}}$$

F-measure: a composite measure of precision and recall.

$$\text{F-measure (fix)} = \frac{2P(fix)R(fix)}{P(fix) + R(fix)}$$

TABLE IV. FIXABLE CRASH PREDICTION RESULTS

Subjects/Features		Precision	Recall	F-measure
Exchange 14	Bug meta data	80	57.2	66.3
	Crash graph	79.5	69.6	74.5
	All features	80	70.6	74.7
Windows 7	Bug meta data	69.9	66.1	68.6
	Crash graph	72.1	60.3	65.0
	All features	71.8	61.2	65.4

Table IV shows the average recall, precision and F-measure. For Exchange 14, F-measure of fixable crashes is around 75% using only crash graph features. However, F-measure using only bug meta-data is around 66%. When we use all features, the F-measure is around 75%. These results indicate that crash graph features are informative to predict fixable crashes.

For Windows 7, F-measure for bug meta-data features is around 69%, while F-measure for crash graph features is 65%. The F-measure using crash graph features is slightly lower. One possible explanation is that the crash fix process of Windows 7 is hit-count oriented. If a crash has a higher hit count, it is likely to be fixed. The hit count is one of bug meta-data features.

However, even without using the hit count, crash graph features can predict fixable crashes almost as well as when using the hit count. This indicates crash graph features are informative to predict fixable crashes when hit count is not available or not reliable. Overall, these results show that crash graph features are informative to predict fixable crashes in advance.

V. THREATS TO VALIDITY

We identify the following threats to validity:

Subject selection bias: We use only industrial project data for our experiments. Open source projects may have different crash properties and the same experiments on open source projects may yield different results. However, we could not find any open source project which had a crash reporting system with bucketing algorithms and auto-crash bug reporting features.

Data selection bias: In our experiments, we use partial auto-crash bug reports. Since Microsoft does not store all crash traces, only partial auto crash bug reports were available for our study. However, despite this fact given the wide deployment of Windows and Exchange the stored data traces are substantially large.

VI. RELATED WORK

Glerum et al. present ten years of debugging experience-using WER including designing WER, bucket algorithms, common debugging practice, and their challenges [9]. WER uses the server-client model to collect crash minidump from clients, and their bucketing algorithms classify crash information using over 500 heuristics such as crashed point and trace similarity. WER has significantly improved crash-debugging process by permitting developers to identify crashes quickly and providing useful information for debugging. Our crash graph approach is on top of WER, the classified buckets, and auto-crash bug reports.

However, WER may misclassify some crashes, which causes the second bucket problem and yields duplicated auto-crash bug reports. Our crash graph approach can efficiently detect duplicated reports by comparing the whole crashes rather than comparing them one by one as discussed in Section IV.A.3.

Research has also focused on identifying the causes of crashes. Ganapathi et al. [8] analyzed and collected Windows XP kernel crash data for a sample population and found out that OS crashes are predominantly caused by poorly-written device driver code.

Bartz et al. propose a stack trace similarity measure based on callstack edit distance with tuned edit penalties [5]. They show that their approach is superior to previous measure such as the Euclidian distance for detecting similar crashes. However, since their approach is based on crash trace-to-crash similarity measures, they have inevitable limitations discussed in Section IV.A.3 including the partial trace issue.

Arnold et al. proposed combining execution traces to facilitate program understanding [4]. Their trace combining approach is similar to our crash graph. However, they combine traces from execution traces rather than multiple crash traces. In addition, their goal is program understanding, and our goal is efficient crash triaging.

Wang et al. and Runeson et al. propose techniques to detect duplicated bug reports using text similarity or execution trace similarity [16, 17]. Usually the accuracy of text similarity based duplicate detection is around 40~50%. Wang et al. generate artificial execution traces and use them to detect duplicates. However, these traces are not collected in the field. Still the accuracy is around 40~60%, since they also compare traces one by one, while our crash graph approach compares the entire crashes using the sub graph similarity measure.

VII. CONCLUSIONS

Crash reporting systems are common these days in most widely deployed software systems. Yet, there has been little research on how these crashes are analyzed to fix the problems. In this paper, we propose the use of crash graph, an aggregated form of multiple crashes, and show its efficacy. Crash graphs are more efficient to identify duplicate auto-bug reports than comparing individual stack traces. In addition, we show that machine-learning features of crash graphs are informative to predict fixable crashes.

Crash reporting systems have become more important to identify crashes and provide useful debugging information for developers. Our experience indicates it is important to use an aggregated form of crashes such as crash graphs for classifying or triaging crashes rather than using or comparing individual crashes.

As part of our future work, we have started a deployment to actual engineers at Microsoft to determine the engineering efficacy and utility of crash graphs. We constructed crash graphs for fixed crashes from Microsoft Exchange 14. Then we presented the crash graphs to the corresponding developer who fixed the crash. So far we have received very promising and enthusiastic support for our work, for example we received the following feedback from developers:

"... the graph would be showing me what a single minidump could not…" – Developer 1

"Usually developers can guess 50-80% of crash causes by reading call traces. This graph can help developers see all traces together." – Developer 2

We plan to investigate further along this line to deploy crash graphs widely across Microsoft. In particular, we plan to ask many developers quantitative and qualitative questions if the crash graphs would be useful to fix these kinds of crashes, solicit suggestions for visualization, and perform an empirical user study on the efficacy of crash graphs. We also hope to help the developer community outside of Microsoft to adopt these crash analysis processes.

VIII. ACKNOWLEDGEMENTS

Our thanks to Windows 7 and Exchange 14 developers for their valuable feedback and comments on our study. We thank Brendan Murphy for his help with data collection and his discussions on this work. We also thank Ray Buse and Caitlin Sadowski for discussion on this work.

IX. REFERENCES

[1] E. Alpaydin, *Introduction to Machine Learning*: The MIT Press, 2004.

[2] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?," in *Proceedings of the 28th international conference on Software engineering*. Shanghai, China: ACM, 2006, pp. 361-370.

[3] Apple, "Technical Note TN2123: CrashReporter," 2010.

[4] D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz, "Stack Trace Analysis for Large Scale Debugging," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007*, 2007.

[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, pp. 1-1.

[6] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Extracting structural information from bug reports," in *2008 international working conference on Mining software repositories*. Leipzig, Germany: ACM, 2008, pp. 27-30.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed: The MIT Press, 2001.

[8] 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.

[9] 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.

[10] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. Amsterdam, The Netherlands: ACM, 2009, pp. 111-120.

[11] 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.

[12] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?," in *Proceedings of the 19th international conference on World wide web*. Raleigh, North Carolina, USA: ACM, 2010, pp. 591-600.

[13] Microsoft, "Windows Error Reporting: Getting Started," 2010, http://www.microsoft.com/whdc/winlogo/maintain/StartWER.mspx.

[14] D. Molla, "Learning of graph-based question answering rules," in *Proceedings of TextGraphs: the First Workshop on Graph Based Methods for Natural Language Processing on the First Workshop on Graph Based Methods for Natural Language Processing*: Association for Computational Linguistics, 2006, pp. 37-44.

[15] Mozilla, "Crash Stats," 2010, crash-stats.mozilla.com.

[16] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of Duplicate Defect Reports Using Natural Language Processing," in *Proceedings of the 29th international conference on Software Engineering*: IEEE Computer Society, 2007, pp. 499-510.

[17] 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 *Proceedings of the 30th international conference on Software engineering*. Leipzig, Germany: ACM, 2008, pp. 461-470.

[18] D. B. West, *Introduction to Graph Theory*, 2nd ed: Prentice Hall, 2001.

[19] L. Wiskott, J.-M. Fellous, N. Krüger, and C. v. d. Malsburg, "Face Recognition by Elastic Bunch Graph Matching," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 19, no. 7, pp. 775-779, 1997.

[20] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques (Second Edition)*: Morgan Kaufmann, 2005.

[21] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proceedings of the 30th international conference on Software engineering*. Leipzig, Germany: ACM, 2008, pp. 531-540.