Performance Debugging in the Large via Mining Millions of Stack Traces

Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang
Microsoft Research Asia
{shihan, yidang, songge, dongmeiz}@microsoft.com

Tao Xie
North Carolina State University
xie@csc.ncsu.edu

—Given limited resource and time before software release, development-site testing and debugging become more and more insufficient to ensure satisfactory software performance. As a counterpart for pioneered by the Microsoft Windows Error Reporting (WER) system focusing on crashing/hanging bugs, has emerged thanks to available infrastructure support to collect execution traces with performance issues from a huge number of users at the deployment sites. However, performance debugging against these numerous and complex traces remains a significant challenge for performance analysts. In this paper, to enable performance debugging in the large in practice, we propose a novel approach, called StackMine, that mines callstack traces to help performance analysts effectively discover highly impactful performance bugs (e.g., bugs impacting many users with long response delay). As a successful technology-transfer effort, since December 2010, StackMine has been applied in performance-debugging activities at a Microsoft team for performance analysis, especially for a large number of execution traces. Based on real-adoption experiences of StackMine in practice, we conducted an evaluation of StackMine on performance debugging in the large for Microsoft Windows 7. We also conducted another evaluation on a third-party application. The results highlight substantial benefits offered by StackMine in performance debugging in the large for large-scale software systems.

I. INTRODUCTION

A modern software system tends to include an increasingly large number of components and lines of code (LOC), and depend on an increasingly large number of system components. For example, among commercial software systems, more than 10 Microsoft products could amount to more than 600 million LOC [7].

To assure high quality of such modern software systems, testing and debugging remain the most commonly used techniques. However, given limited resource and time before software release, development-site testing and debugging become more and more insufficient to ensure high software quality. Development-site testing typically covers only a limited percentage of various usage scenarios and vast multitude of execution environments that could occur upon the deployed software systems. Development-site debugging typically relies on a limited number of failing runs, which could be insufficient for effective debugging. Furthermore, during development-site debugging with limited resource and time, developers have relatively little knowledge on which bugs should be given higher priority to fix based on their impact on users including their *impact scope*, e.g., which bugs impact a large number of users at deployment sites, and their *impact severity*, e.g., which bugs impact an individual user at a high severity level.

To address these issues, recent industrial solutions are developed and deployed for collecting and leveraging a high volume of deployment-site usage data to improve debugging with postmortem analysis. There emerges a new frontier of debugging practice: *debugging in the large*, with a prominent example as the Microsoft Windows Error Reporting (WER) system (a.k.a. Dr. Watson) [9]. Using a long time period in the post-release stage and a huge number of information sources from real-world users, WER allows developers to obtain distribution information of crashing/hanging bugs to guide their debugging prioritization. Since its operation in 1999, WER has accomplished huge success within Microsoft.

However, in spite of high importance, performance bugs are not handled by WER, which heavily focuses on crash/hang debugging. Performance is one of the key properties of software, primarily concerning responsiveness, throughput, and resource utilization. For example, to conduct performance debugging, along a call path associated with a thread blocked on a wait, performance analysts typically start with looking for a subsequence of function calls that account for a non-trivial portion of waiting time and then try to derive performance signatures from the subsequence. A performance signature is one or more functions in which a potential performance bug resides, often manifested as the location where the fix to the bug is applied. Optimizing code implementation within the performance signature could reduce time consumption on wait. For example, a subsequence of function calls (`InitComponents`, `GetHashCode`, `GetShortPathName`, `MmAccessFault`) comes from a trace associated with slow startup of a third-party application, accounting for approximately 0.3 second waiting time. There, the function `GetHashCode` is a performance signature, fixing which could resolve the performance issue of slow startup (this example is explained in more details in Section2)).

In contrast to performance debugging against one or a few execution traces, performance debugging in the large deals with execution traces with performance issues from a huge number of users from the deployment sites. Similar to WER, Microsoft has also provided infrastructure support for such purpose with mechanisms such as PerfTrack [25] based on the Event Tracing for Windows (ETW) [28] platform. In particular, PerfTrack resides inside Windows 7 for measuring system responsiveness to user actions on operating systems (OS). For example, when a user clicks on

a folder name, PerfTrack measures how long it takes for the user to receive an expected reaction from the system, and if the response time exceeds a pre-defined threshold, PerfTrack sends execution traces based on ETW such as callstack information collected in the preceding period back to Microsoft for performance debugging. Note that other OS platforms also provide similar mechanisms such as DTrace [26] for Solaris and several other Unix-like systems.

Such execution traces collected from real users provide substantial advantages as discussed earlier, including more traces with performance issues and more information on a performance issue's impact on users. For example, performance analysts now can start with looking for a subsequence of function calls that account for a non-trivial portion of time consumption *across all the collected traces from real users* to hunt for highly impactful performance bugs (e.g., bugs impacting many users with long response delay). For example, the earlier described subsequence of function calls accounts for a total amount of 8.28 seconds waiting time from 24 startups of the application. However, there exists no effective support for performance debugging in the large similar to WER to deal with the high volume of complex traces for performance debugging. For example, PerfTrack collected execution traces including more than 1 billion of callstacks for response delay of the Windows Explorer User Interface (UI), far beyond affordable investigation effort of performance analysts.

To fill the gap of existing performance debugging practices and the vision of performance debugging in the large similar to WER, in this paper, we propose StackMine, a novel approach to enable performance debugging in the large, and its supporting scalable system for postmortem performance debugging as the counterpart of WER in performance debugging. Our approach includes a novel costly-pattern clustering mechanism, consisting of two phases, to reduce investigation effort of performance analysts. In the first phase, we apply a costly-pattern mining algorithm (such as subsequence mining) on trace streams[1] that include callstack traces, and then apply clustering on the mined costly patterns based on a set of novel similarity metrics. We propose these metrics to reflect domain-specific characteristics of program-execution traces in contrast to other types of data.

In particular, our costly-pattern clustering mechanism addresses three significant challenges in performance debugging in the large.

First, in modern multi-tasking systems, the collected trace streams recorded not only performance-issue-exhibiting executions of an application, but also simultaneous normal executions of the same application, as well as executions of other applications. For a subsequence of function calls under investigation, its measured performance metric values such as waiting time would include contributions from both performance-issue-exhibiting executions (such as lock contention) and normal executions (such as waiting time on user inputs), with the latter as noise for compromising effectiveness of performance analysis. To address this challenge, we propose the Wait Graph model, which extracts from raw trace streams relevant traces with respect to performance issues, before we apply mining techniques on the traces.

Second, it is infeasible to enumerate all subsequences of function calls and then rank them based on their time consumption across trace streams, due to the combinatorial explosion. To address this challenge, we propose to conduct subsequence mining on traces to extract highly impactful subsequences of function calls. Based on overall performance metric values of the traces, performance analysts can set a meaningful performance-metric threshold for the subsequence mining algorithm to efficiently output only costly subsequence patterns of function calls with performance metric values beyond the threshold.

Third, partly due to trace collection from various deployment sites with different execution environments (a unique nature of performance debugging in the large), multiple mined costly patterns of function calls with slight differences could be related to the same performance bug, compromising effectiveness of performance analysis for two main reasons. First, analysts would waste time to investigate multiple variant patterns corresponding to the same performance bug. Second, the impact computation used to hunt for performance bugs would be inaccurate, since the contributions of the same performance bug are dispersed among multiple patterns. For example, a series of function calls for Windows 32-bit on Windows 64-bit (wow64) simulation would appear in trace streams on a 64-bit system, but would not appear in the corresponding ones on a 32-bit system. As another example, a performance-issue-exhibiting subsequence of function calls in the kernel mode might be invoked by some user-mode functions in some trace streams, and by different user-mode functions in some other trace streams. To address this challenge, we propose a clustering mechanism to group multiple mined costly patterns of function calls with slight differences.

StackMine is the outcome of two-year effort of continuous development and improvement at the Software Analytics group of Microsoft Research Asia in collaboration with Microsoft product teams. As a successful technology-transfer effort, since December 2010, StackMine has been applied in performance-debugging activities at a Microsoft team for performance analysis, especially for a large number of execution traces. During this period, performance analysts in the team have applied StackMine to analyze hundreds of millions of callstacks. StackMine has been shown to reduce more than 90% of the human investigation effort for identifying highly impactful bugs. The substantial benefits of StackMine have been reflected by feedback given by the performance analysis team: "We believe that the StackMine tool is highly valuable and much more efficient for mass trace streams (100+ trace streams) analysis. For 1000 trace streams, we believe the tool saves us 4-6 weeks of time to create new performance signatures, which is quite a significant productivity boost."

This paper makes the following main contributions:

[1] A trace stream records a stream of system level events to capture the program execution behaviors during a time period of bad performance.

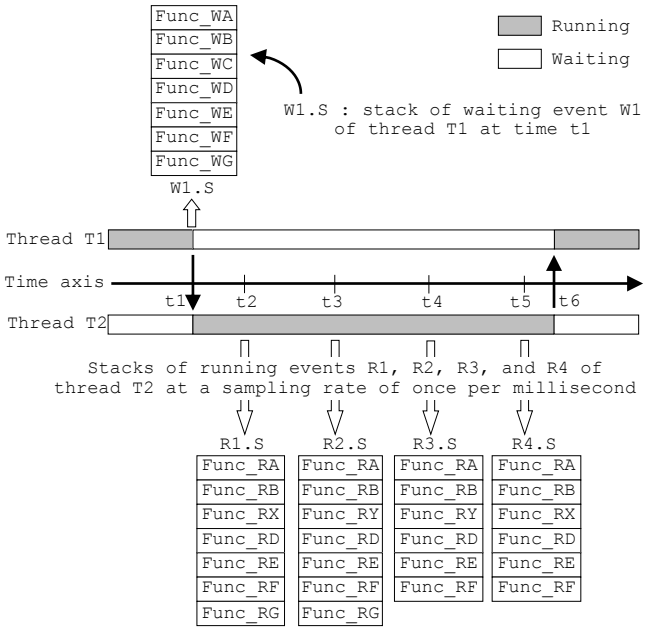

Figure 1. A time window of program execution

- The first formulation and real-world deployment of performance debugging in the large as a data mining problem on callstacks;
- The Wait Graph representation abstracted from callstacks to capture program behaviors essential to performance analysis;
- A clustering mechanism for reducing costly-pattern mining results based on domain-specific characteristics of program-execution traces;
- Industrial experiences on using StackMine in performance debugging in the large for Microsoft Windows 7. For example, a Microsoft performance analyst applied StackMine on 921 real-world trace streams for response delays in the Windows Explorer UI. The analysts reviewed the top 400 pattern clusters produced by StackMine, created 93 performance signatures, and identified 12 highly impactful hidden performance bugs of Windows Explorer. These bugs had been hidden for at least more than one year since the release of Windows 7, and some of them can even be traced back to earlier versions of Windows.
- Third-party-application experiences on using StackMine in performance debugging in the large for a third-party application. With the assistance of StackMine, we discovered 6 highly impactful performance bugs of this application. Among these 6 bugs, 5 of them have been confirmed with third-party sources and 1 of them is yet to be confirmed.

II. Problem Formulation

In this section, we first use an example to informally explain the intuitions behind formulating performance debugging in the large as a data mining problem. We then provide a more formal description of the problem.

A. Performance-Bug Hunting from Callstacks

Figure 1 shows an illustrative scenario within a small time window of a program's execution such as startup of program Foo. Along the time axis, thread T1 was running until it was blocked at time t1 because it requested a lock held by another thread T2. Then a context switch from T1 to T2 occurred on the CPU, and then T2 entered its running state while T1 entered its waiting state. After T2 released the lock requested by T1, another context switch occurred to enable T1 to resume running at time t6. After thread T1 finished its execution, the corresponding startup time of Foo exceeded a pre-defined threshold, satisfying the triggering condition for trace collection.

Then a low-overhead mechanism for trace collection such as ETW can collect two types of events with callstacks shown in the figure. W1 is a waiting event of T1, with callstack snapshot W1.S (with an upper function invoking a lower function) on which T1 was blocked and switched out of the CPU. We name this type of callstack as a *waiting stack*. R1, R2, R3, and R4 are running events of T2, which are sampled with callstack snapshots R1.S, R2.S, R3.S, and R4.S, respectively. Typically the sampling is at a constant rate for every CPU, e.g., once per millisecond. We name this type of callstack as a *running stack*. From a thread's point of view, its running stacks capture how it behaves when using the CPU resource, while its waiting stacks capture why it is temporarily blocked and switched out of the CPU. The waiting time associated with a waiting stack shows for how long the thread is blocked, while the sampling rate and sample size determine how much CPU time is associated with a running stack, such as 4 ms period of CPU consumption for T2 in Figure 1.

Based on the experience of performance analysts from Microsoft, a majority of existing software performance bugs fall into two categories: CPU consumption and wait.

CPU consumption bug. As shown in Figure 1, the CPU usage pattern of thread T2 can be reflected by running stacks R1.S, R2.S, R3.S, and R4.S. We notice that a subsequence pattern from these running stacks, <Func_RA, Func_RB, Func_RD, Func_RE, Func_RF>, is executed throughout the 4 ms period of CPU consumption. We name such non-consecutive subsequence pattern from callstacks as *callstack pattern*. If a large amount of CPU consumption from many trace streams is observed on this callstack pattern, it is suspicious to be a highly impactful performance bug. The aggregated performance metric values on this callstack pattern can be used to reflect its impact and to rank the bug along with other bugs.

Wait bug. As shown in Figure 1, the wait reason of thread T1 at time t1 can be reflected by waiting stack W1.S. If a large number of waiting events from many trace streams are observed on the same waiting stack, or on waiting stacks containing a common callstack pattern reflecting the same waiting reason, e.g., <Func_WA, Func_WC, Func_WD, Func_WF>, it is quite suspicious to be a highly impactful performance bug. The content of the callstack pattern (the full waiting stack W1.S can also be a callstack pattern) can be used to both represent the bug and help analysts narrow

down the investigation scope. The aggregated performance metric values on the callstack pattern, i.e., the number of occurrences and total waiting time, can be used to reflect its impact and to rank the bug along with other bugs.

B. Problem Definition

Before we formalize the problem, we define some concepts used in the problem definition.

Definition 1. A callstack S is a sequence of function calls $S =< f_i >: f_0 f_1 f_2 \dots f_{M-1}$, where f_i is a function call for $0 \le i < M$.

Definition 2. An event e is of running event type r or waiting event type w with fields defined as in TABLE 1:

TABLE 1. DEFINITIONS OF THE FIELDS IN AN EVENT

Field	Type	Description
$e.S$	Callstack	Running stack if $e.type == r$ Waiting stack if $e.type == w$
$e.T$	Int64	Timestamp of event e
$e.C$	Int64	Cost as CPU consumption in ms if $e.type == r$ Cost as waiting time in ms if $e.type == w$
$e.TID$	Int32	Thread ID

Definition 3. A trace stream TS is a sequence of events $TS =< e_i >: e_0 e_1 e_2 \dots e_{L-1}$, where e_i is an event for $0 \le i < L$.

Definition 4. A callstack pattern CP for a set of trace streams $\{TS_j\}$ is a non-consecutive subsequence of a callstack $e_i.S$ of TS_j, i.e., $CP \sqsubseteq e_i.S$.

Definition 5. The cost of a callstack pattern CP for a set of trace streams $\{TS_j\}$ is defined as

$$Cost(CP) = \sum_{\substack{0 \le j < |\{TS_j\}|, e_i \in TS_j \\ s.t.\ CP \sqsubseteq e_i.S}} e_i.C$$

Basic-Problem Definition:
Inputs:
A set of trace streams $\{TS_j\}$.
A number λ as the threshold of waiting time or CPU consumption time.
Outputs: All callstack patterns $< CP_j >$ for $\{TS_j\}$ where $\forall CP_j, Cost(CP_j) \ge \lambda$, sorted by their costs in the descending order.

To make the problem manageable, we reduce the basic problem to a costly-subsequence-pattern mining problem where the outputs are costly callstack patterns whose costs are equal to or higher than λ. Furthermore, because multiple mined costly callstack patterns with slight differences could be related to the same performance bug, we apply clustering on the mined costly subsequences to produce the final output as a set of clusters of costly callstack patterns $\{CCP_i\}$, where a cluster of callstack patterns CCP_i is a set of callstack patterns CP_j, denoted as $CCP_i = \{CP_j\}$.

We next define four typical performance metrics for a callstack pattern cluster CCP as below:

(1) Total cost,

$$Cost_{total}(CCP) = \sum_{\substack{0 \le j < |\{TS_j\}|, e_i \in TS_j \\ s.t.\ \exists CP \in CCP, CP \sqsubseteq e_i.S}} e_i.C$$

(2) Number of trace streams,

$$Num_{trace\_stream}(CCP) = \left|\{TS_j | \exists e \in TS_j, \exists CP \in CCP, s.t.\ CP \sqsubseteq e.S\}\right|$$

(3) Number of events,

$$Num_{event}(CCP) = \left|\{e_i | 0 \le j < |\{TS_j\}|, e_i \in TS_j, \exists CP \in CCP, s.t.\ CP \sqsubseteq e_i.S\}\right|$$

(4) Average event cost,

$$Cost_{Average}(CCP) = Cost_{total}(CCP) / Num_{event}(CCP)$$

III. APPROACH OVERVIEW

For the described problem, we propose our StackMine approach that includes the costly-pattern clustering mechanism to address significant challenges in performance debugging in the large. In particular, given a (large) set of trace streams, StackMine includes three steps to reduce the investigation scope as callstack pattern clusters for performance analysts to investigate.

AOI extraction: For different subjects under analysis, the area of interests (AOIs) within the trace streams might be different. For example, when analyzing response delay of the Windows Explorer UI, a large proportion of the given trace streams can be irrelevant to the delay in the UI thread. The AOI extraction step is responsible for extracting relevant events and callstacks based on dependencies among thread executions due to resource sharing.

Costly-maximal-pattern mining: From callstacks in the extracted AOIs, this step mines callstack patterns out of waiting stacks and running stacks, respectively, with maximal subsequence pattern mining.

Callstack pattern clustering: To group variations of one performance bug that are in the form of a set of similar callstack patterns, we propose a similarity model for callstack patterns and conduct hierarchical clustering on callstack patterns with this similarity model.

Recall that each callstack pattern cluster is associated with a set of performance metrics as presented earlier. Analysts can inspect the pattern clusters ranked based on one of these metrics. We next illustrate the technical details of these three steps.

IV. AOI EXTRACTION

The step of AOI extraction addresses two major issues related to effectiveness and efficiency, respectively, faced when applying performance analysis directly on all events from the trace streams.

First, a trace collection platform typically collects trace streams that record not only the problematic time period with respect to performance, but also some normal time periods around the problematic period. In modern multi-tasking system environments, even within the problematic period including performance-issue-exhibiting executions (such as

Inputs: trace stream TS and symptom $< T, t_0, t_1 >$, denoting execution of thread T within $TimeSpan(t_0, t_1)$ with bad performance
Output: wait graph $WG =< V, E >$

```
1.  V := φ; E := φ; V' := φ;
2.  foreach e ∈ TS where e.TID == T.TID
3.      if TimeSpan(e.T, e.T + e.C) ⊆ TimeSpan(t0, t1)
4.          V.AddNode (e);
5.  ΔV := V\V';
6.  foreach e ∈ ΔV
7.      V'.AddNode(e);
8.      if e.type == w
9.          TID' := e.GetReadierThreadID();
10.         foreach e' ∈ TS where e'.TID == TID'
11.             if (e'.T + e'.C) ∈ TimeSpan(e.T, e.T + e.C)
12.                 V.AddNode(e');
13.                 E.AddEdge(e,e');
14. if V ≠ V'
15.     goto 5;
16. return < V, E >;
```

Figure 2. Pseudo-algorithm for Wait Graph construction

lock contention), there would be normal executions (such as waiting time on user inputs) recorded together. Performance metric values of such normal executions need to be excluded since they would be noise for compromising effectiveness of performance analysis. Second, all events within collected trace streams are typically too many and too complicated for even well-designed and engineered mining algorithms to handle.

To address these issues, based on more than 1-year's interactions and collaborations with Microsoft performance analysts, from performance-analysis industrial practices, we identified two effective AOI extraction techniques (scope-based extraction and content-based extraction) and built tool support in StackMine to enable such effective and efficient AOI extraction. Note that in practice, performance analysts often use a mixture of these two techniques in this step.

Scope-based extraction. Recording a trace stream TS is typically triggered by some performance symptom, e.g., delayed finish of a feature or delayed handling of a message. Based on the recorded information of the trace stream, such symptom can be automatically identified in the form of a triple $< T, t_0, t_1 >$, denoting the execution of a thread T within a time period $TimeSpan(t_0, t_1)$ of bad performance. With respect to the identified symptom, our technique identifies a scope to include the symptom's relevant events and callstacks within the trace stream.

In particular, we propose the Wait Graph model to extract such scope for a symptom. A wait graph, denoted as $WG =< V, E >$, consists of a vertex set V and an edge set E. Each vertex $v \in V$ represents a running or waiting event of a thread. Each directed edge $x \in E$ always starts from a vertex for a waiting event $estart$ and ends at a vertex for a running or waiting event $eend$, denoting that (1) the time span of $estart$ has overlapping with the time span of $eend$, and (2) $eend$'s thread makes $estart$'s thread ready, where $eend.TID$ could be acquired by calling $estart.GetReadierThreadID()$. Figure 2 shows an algorithm for constructing a wait graph. From the wait graph for a symptom, we identify the relevant scope as all the waiting events and running events from the vertices in the graph. Despite the similar shape, the Wait Graph substantially differs from the Wait-For graph [27]. The Wait Graph (1) uses finer-grained entities (running/waiting events instead of processes/threads) as nodes, (2) uses directed edges to express their timing dependencies instead of resource dependencies, and (3) is used to reduce the investigation scope instead of detecting deadlocks. Note that the key idea of our technique can be applied in a more aggressive way to further reduce the investigation scope by identifying the relevant scope as the waiting events and running events from only (1) the UI threads or (2) the critical paths of the Wait Graph, as often done by Microsoft performance analysts.

Content-based extraction. Performance analysts may often initiate performance analyses with certain hypotheses or focuses in mind. For example, Microsoft performance analysts initiated a real-world analysis when they suspected that some locks in the win32k.sys module caused poor performance, and initiated another real-world analysis when the analysts wanted to discover hard-fault-related bugs for causing slow Windows logon. In these situations, our technique allows analysts to conveniently extract required events, e.g., including at least one function for win32k.sys lock access or including one function for hard-fault handling in its callstack for the preceding two examples, respectively.

V. CALLSTACK PATTERN MINING

After the step of AOI extraction, our proposed mining algorithm mines costly callstack patterns $\{CP\}$ from two types of callstacks (within the extracted AOI), i.e., waiting stacks and running stacks, separately.

Our algorithm is a novel adaptation of a classic algorithm for mining frequent maximal [2] subsequences, where the support of a pattern is the number of supporting entities in the input database, such as transactions and sequences. Both a costly callstack pattern CP (targeted by our algorithm) and a frequent subsequence pattern SP (targeted by the classic algorithm) obey the *Apriori* [2] property: $\forall CP' \sqsubseteq CP, Cost(CP') \geq Cost(CP)$ whereas a nonempty subsequence of a frequent SP pattern must also be frequent. In particular, we adapt the classic BIDE algorithm [23] with two major modifications. First, we modify the support of a subsequence pattern CP from its occurrences to $Cost(CP)$. Second, for maximal pattern checking, we modify the condition in the forward/backward extension check of a super-sequence pattern CP' from $Cost(CP') == Cost(CP)$ (originally used to mine closed patterns) to $Cost(CP') \geq \lambda$.

For the condition $Cost(CP) \geq \lambda$, an analyst can follow some guideline to determine an appropriate λ threshold value for the analysis task at hand. For example, assume that an analyst intends to conduct performance analysis on one

[2] When a frequent (costly) pattern is *maximal*, none of its super patterns can be frequent (costly). For our problem, we choose maximal patterns since doing so helps produce desirable results for inspection: reducing the pattern set while preserving call path information.

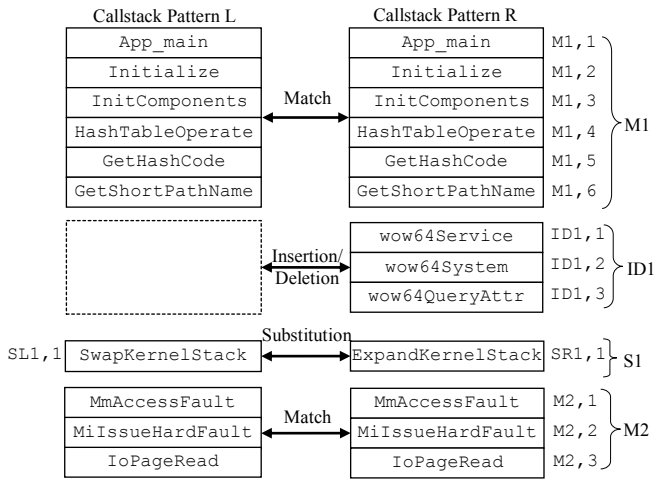

Figure 3. An illustration of callstack similarity model

performance bug triggered in over 10% of the collected 1000 trace streams with an impact of 0.3 second response delay on average. Then the analyst can set $\lambda = 30$ seconds, since 30 seconds = 10% * 0.3 second/trace stream * 1000 trace streams.

VI. Callstack Pattern Clustering

Given the mined costly callstack patterns, our step of clustering groups similar patterns into clusters with hierarchical clustering (a popular clustering technique), offering two main benefits. First, when exploring together callstack patterns of different variations, analysts can more easily recognize the common part and the variant part of the callstack patterns. Doing so not only helps better determine whether and where there is a hidden performance bug, but also helps find a high-covering fix (likely falling into the common part) that could fix performance issues across various similar patterns. Second, using performance metrics of a cluster can help produce better prioritization of results for investigation since without clustering, performance metric values of a high-impact bug can be spread across multiple callstack patterns of different variations, causing the bug not to emerge with a high rank.

A key component of any clustering technique is the underlying similarity model used to measure how similar two entities are. Based on years of performance-analysis experiences of Microsoft performance analysts, we construct a novel similarity model (sharing the key concept with the edit distance model [17]) that takes into account characteristics of programs and their behaviors. In particular, we calculate similarity of two callstack patterns with two major steps: align the two patterns (Section A) and then calculate the similarity of the two patterns (Section B) based on the weighted similarity between the aligned segments of the two patterns (Section C).

A. Alignment

Similar to the edit distance model, we use dynamic-programming search to derive an optimal alignment with respect to minimizing the total cost of three kinds of editing operations: (1) match; (2) insertion/deletion; (3) substitution. For example, given a pair of callstack patterns L and R, we get the optimal alignment in the form of a series of operation segments, as shown in Figure 3.

When computing operation cost, we set the cost of a match operation as 0, the cost of an insertion/deletion operation as 1.0, and the cost of a substitution operation between function f_1 and f_2 as $Sub(f_1, f_2)$, where $Sub(f_1, f_2)$ is a cost function defined based on their function names (we split a name into words by treating upper-case characters as word boundaries):

$$Sub(f_1, f_2) = 1 - \frac{2 * \#words\ common\ in\ both\ names}{\#words\ in\ f_1.name + \#words\ in\ f_2.name}$$

Intuitively, f_1 and f_2 are more similar when $Sub(f_1, f_2)$ is smaller. We define the preceding cost function based on substantial observations from practice that two different callstack patterns of the same performance bug tend to share the similar name structure and words across their corresponding functions in a substitution segment, with an example as segment S1 in Figure 3.

More formally, we define a segment of k consecutive match operations as $M_i =< M_{i,j} >: M_{i,1}M_{i,2} \dots M_{i,k}$, such as M_1 and M_2 shown in Figure 3. Each $M_{i,j}$ represents a function that appears in both callstack patterns at the corresponding position j. We assign $M_{i,0}$ and $M_{i,k+1}$ with the value ϕ for the sake of simplicity, denoted as $M_{i,0} = M_{i,k+1} = \phi$.

We define a segment of k consecutive insertion/deletion operations as $ID_i =< ID_{i,j} >: ID_{i,1}ID_{i,2} \dots ID_{i,k}$, such as ID_1 shown in Figure 3. Each $ID_{i,j}$ represents a function that appears in one callstack pattern at the corresponding position j. $ID_{i,0} = ID_{i,k+1} = \phi$.

We define a segment of k consecutive substitution operations from the left callstack pattern to the right one as $S_i =< SL_{i,j}SR_{i,j} >: (SL_{i,1}SR_{i,1})(SL_{i,2}SR_{i,2}) \dots (SL_{i,k}SR_{i,k})$ such as S_1 shown in Figure 3. Each pair of $SL_{i,j}$ and $SR_{i,j}$ represents a pair of functions (from the two patterns) that are different at the corresponding position j. $SL_{i,0} = SL_{i,k+1} = \phi$, $SR_{i,0} = SR_{i,k+1} = \phi$.

Given a pair of callstack patterns, we first derive an alignment of the callstack patterns as a set of operation segments as $A = \{M_i\} \cup \{ID_i\} \cup \{S_i\}$. Then, based on this alignment A, we calculate the similarity of the two callstack patterns denoted as $Sim(A)$, with details illustrated next.

B. Similarity Calculation

Given an alignment A of a pair of callstack patterns, we use the edit distance model to calculate their similarity as the ratio of the length of the match-operation segments over the length of all the segments. However, using only the lengths (i.e., the number of function calls) of aligned segments to calculate similarity often cannot reflect desirable similarity in terms of program behaviors. To address the issue, we define a set of weight functions W on segments, and then define the similarity $Sim(A)$ as

$$Sim(A) = \frac{\sum_{M_i \in A} W(M_i)}{\sum_{M_i \in A} W(M_i) + \sum_{ID_i \in A} W(ID_i) + \sum_{S_i \in A} W(S_i)}$$

We denote the jth function in a segment X as X_j, where $X \in A$: $\{M_i\} \cup \{ID_i\} \cup \{S_i\}$. Once we define a weight function $\omega(X_j)$ for each function X_j, we can define the weight functions W for the three types of segments as

$$W(M_i) = \sum_{j=1}^{|M_i|} \omega(M_{i,j}) \qquad W(ID_i) = \sum_{j=1}^{|ID_i|} \omega(ID_{i,j})$$

$$W(S_i) = \sum_{j=1}^{|S_i|} Sub(SL_{i,j}, SR_{i,j}) * \frac{\omega(SL_{i,j}) + \omega(SR_{i,j})}{2}$$

We define function weight $\omega(X_j)$ in the next sub-section.

C. Function-Weight Calculation

We define the weight of function X_j as

$$\omega(X_j) = Uni(X_j) * \frac{FBi(X_{j-1}, X_j) + BBi(X_j, X_{j+1})}{2}$$

in which $Uni(X_j)$ (denoting unigram information) represents the weight for reflecting how unlikely X_j is a common-purpose function, e.g., the `App_main` function. Note that an editing operation on a common-purpose function has less contribution to the overall pattern similarity than a non-common-purpose function.

$FBi(X_{j-1}, X_j)$ and $BBi(X_j, X_{j+1})$ (denoting forward bigram information and backward bigram information, respectively) represent the weights for reflecting how unlikely X_j is in a constant/dominant call path, given its caller and callee functions as context. Note that a callstack segment X has less contribution to the overall pattern similarity when the segment X is in a constant/dominant call path.

For functions f and f_1, f_2, we calculate $Uni(f)$, $FBi(f_1, f_2)$, and $BBi(f_1, f_2)$ based on statistics derived from our trace-stream database (beyond just the trace streams under investigation) as

$$Uni(f) = 1 - \frac{\#callstacks\ including\ f}{\#callstacks\ in\ database}$$

$$FBi(f_1, f_2) = 1 - \frac{\#occurrence\ of\ f_1\ with\ f_2\ as\ its\ callee}{\#occurrence\ of\ f_1\ with\ a\ callee}$$

$$BBi(f_1, f_2) = 1 - \frac{\#occurrence\ of\ f_2\ with\ f_1\ as\ its\ caller}{\#occurrence\ of\ f_2\ with\ a\ caller}$$

Specially, we define $FBi(\phi, f) = BBi(f, \phi) = 1$. Intuitively, f tends to be a common-purpose function when $Uni(f)$ is small. $f_1 f_2$ tends to be a part of a constant/dominant call path when $FBi(f_1, f_2)$ or $BBi(f_1, f_2)$ is small.

VII. Evaluations

We conducted two evaluations on StackMine with two of the most popular real-world software products - Microsoft Windows 7 and a third-party application, respectively. In the evaluation with Windows 7, we intended to answer three research questions with results produced when Microsoft performance analysts used StackMine in real-world industrial settings.

- *Q1*. How much does StackMine improve practices of the performance debugging in the large?
- *Q2*. How well do the performance signatures derived with the assistance of StackMine capture performance bottlenecks caused by performance bugs?
- *Q3*. How does StackMine perform compared to alternative techniques?

StackMine was motivated by the needs of Microsoft Windows teams. Our evaluation serves as the first reported experience of performance debugging in the large for such large-scale software products as Windows. After research and incubation for more than one year, with close collaboration between Microsoft researchers and performance analysts, StackMine has been adopted by one team of Windows performance analysts, and is becoming part of the standard Windows performance analysis workflow. Our evaluation focuses on real-world experiences of Microsoft performance analysts when applying StackMine on a large performance analysis task: response-delay analysis of the Windows 7 Explorer (in short as Windows Explorer) User Interface (UI).

Due to confidentiality, we are not able to disclose some low-level details of the first evaluation's results such as detailed descriptions of the investigated bugs. To allow the community to build upon our research and results, with the help of researchers from North Carolina State University, our approach was applied on one popular third-party application. The details of the evaluation results can be found on our project website [29]. Note that in our setting, we take the role of third-party performance analysts (other than the developers of the application), lacking deep knowledge of the application's code base while achieving substantial success with the assistance of StackMine.

A. Windows 7 Study

In December 2010, as a continued effort to improve the performance of Windows, performance analysts from one performance analysis team for Microsoft Windows planned to conduct an investigation to hunt for the hidden performance bugs that caused common impact on Windows Explorer UI response. The investigation was against a large set of ETW event trace streams collected through the PerfTrack mechanism. This initial set included over 6,000 such trace streams collected by satisfying a triggering condition of Windows Explorer UI response delay. The performance analysts focused on 921 trace streams by first randomly sampling 1,000 trace streams and then excluding 79 irrelevant ones (e.g., those not including any key function calls related to the Windows Explorer UI). Among the 921 trace streams, there were 181 million callstacks in total, among which 140 million were waiting stacks and 41 million were running stacks. Each typical trace-stream file can include hundreds of thousands of events and callstacks with hundreds of megabytes in binary format.

1) Q1. Overall Improvement of Practices: Given the 921 trace streams, StackMine took about 10 hours of automatic analysis to output a ranked list of 1,215 pattern clusters. In particular, the AOI extraction phase reduced the 140 million callstacks to 689 thousand callstacks using both scope-based (in particular the critical-path model) and content-based

techniques. The maximal-callstack-pattern mining produces 2,239 costly patterns. The callstack-pattern clustering produces the final ranked list of the 1,215 pattern clusters.

One analyst took 1 day (8 hours) to go through the top 400 clusters, in the descending order of the $Cost_{average}$ metric values as defined in Section B, and derived 93 performance signatures from these clusters. These 93 signatures covered 58.26% of the response delay time (of the Windows Explorer UI) captured in the 921 trace streams.

With additional deep investigation on both trace streams and source code based on the 93 signatures, the analyst successfully diagnosed and filed 12 highly impactful hidden performance bugs of Windows Explorer. These bugs had been hidden for at least more than one year since the release of Windows 7, and some of them can even be traced back to earlier versions of Windows. In terms of their performance impact, one of the bugs caused significant response delay observed in 32% of the 921 trace streams, and another one caused significant response delay observed in 11% of the 921 trace streams with on average 1.6 seconds of UI response delay once triggered.

Ideally we would like to measure the effort of an analyst (without the assistance of StackMine) to derive performance signatures for discovering these highly impactful performance bugs, in order to measure effort reduction achieved by StackMine. However, we could not attain such statistics because the performance analysis team would not afford to invest analysts to manually investigate these 921 trace streams before we introduced StackMine into their practices. Therefore, we make a rough estimation as below based on past experiences of the performance analysis team. On average, it takes about 10~30 minutes for an experienced performance analyst to derive signatures from a single trace stream. Therefore, it would take 20~60 days to derive signatures if each and every single trace were analyzed manually. In contrast, with StackMine, deriving signatures can be finished in 1 day.

The substantial benefits of StackMine have been reflected by feedback given by the performance analysis team: "We believe that the StackMine tool is highly valuable and much more efficient for mass trace streams (100+ trace streams) analysis. For 1000 trace streams, we believe the tool saves us 4-6 weeks of time to create new signatures, which is quite a significant productivity boost."

2) Q2. Performance Bottleneck Coverage: The study results against Q1 show that StackMine substantially reduces the effort of the performance analyst on deriving performance signatures. However, it is an open question on whether the effort reduction compromises the effectiveness, i.e., missing other highly impactful hidden performance bugs (beyond these 12 bugs) that could be found by more-expensive investigation without the assistance of StackMine.

To address this evaluation issue, we propose and measure the performance bottleneck coverage of a set of performance signatures as below:

$$\frac{Total\ CPU\ time\ and\ wait\ time\ of\ the\ performance\ signatures}{Total\ response\ delay\ time\ of\ the\ collected\ trace\ streams}$$

The coverage denotes the proportion of the bad-performance time period that analysts can explain and take action on with the performance signatures. The higher coverage the performance signatures achieve, the lower possibility highly impactful performance bugs remain not captured by the performance signatures. The reason is that shorter (not-covered) bad-performance time period causes the remaining performance bugs (if any) to be less highly impactful (i.e., causing less-significant delay).

Recall that the analyst identified 93 performance signatures from the pattern clusters. These 93 signatures achieved 58.26% performance bottleneck coverage: nearly 60% of the total response delay captured in the 921 trace streams can be explained and improved with these 93 performance signatures. The performance analysis team indicated to us that the achieved coverage was quite satisfactory. Among these signatures, two signatures led the analyst to discover two highly impactful performance bugs with high impact as described in Section 1), respectively.

3) Q3. Comparison with Alternative Techniques: We identified three alternative techniques along with StackMine that rank trace streams for the performance analyst to investigate.

- Random: this technique ranks trace streams in a random order, serving as the baseline technique;
- Greedy-Total: this technique ranks trace streams in the descending order of total UI response delay time within a trace stream;
- Greedy-Max: this technique ranks trace streams in the descending order of the maximum UI response delay time within a trace stream;
- StackMine: this technique first defines a set of performance signatures (93 signatures in 1)) from the top pattern clusters (400 clusters in 1)). After the signatures are sorted by their performance bottleneck coverage, the technique then selects one trace stream containing the top 1 signature to investigate. Then it selects a new trace stream containing the next not-yet-investigated signature, and so on.

Figure 4 shows the number of trace streams (y-axis) required to achieve a certain performance bottleneck coverage (x-axis) for the Random, Greedy-Total, Greedy-Max, and StackMine techniques, respectively. For example,

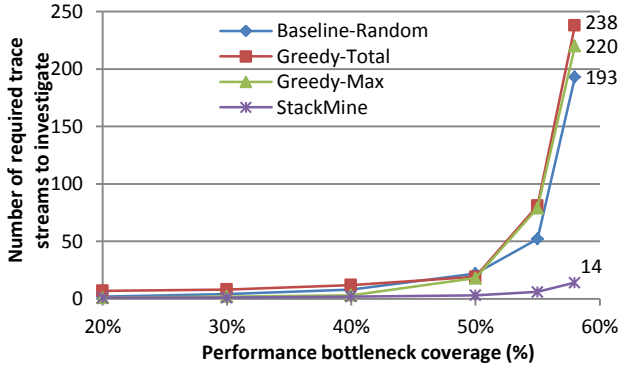

Figure 4. Comparison with alternative techniques

these four techniques require the analyst to investigate 193, 238, 220, and 14 trace streams, respectively, to achieve 58.26% coverage. On average, StackMine requires only 7.2%, 5.8%, and 6.3% of trace streams required by the Random, Greedy-Total, and Greedy-Max techniques, respectively, to achieve a certain performance bottleneck coverage. The results highlight the substantial advantages of StackMine over other alternative techniques.

B. Study Results on a Third-Party Application

To allow the community to build upon our research and results, with the help of researchers from North Carolina State University, our approach was applied on one popular third-party application[3]. Such case reflects situations where users of our approach take the role of third-party performance analysts (other than the developers of the application), lacking deep knowledge of the code base of the application.

In particular, StackMine was applied to analyze the startup performance of the third-party application, based on 54 trace streams with the following characteristics.

- Running a specific same version of the application;
- Containing the start point of a startup process, indicated by the process start event of the application;
- Containing the end point of the startup process, indicated by the first occurrence of function call "`user32.dll!PeekMessage`" after the start point of the startup process. This first attempt for handling a user's input tends to reflect the end point of the startup process.
- Spending > 1 second for the identified startup process.

These 54 trace streams included 59 occurrences of slow startups (each > 1 second). The average time consumption of a startup was 7.0 seconds and the slowest one took 52.8 seconds. These trace streams included 33 million callstacks, while 20 million were waiting stacks and 13 million were running stacks. In this study, we focused on wait-related bugs. Discovering CPU-consumption-related bugs can be conducted similarly.

1) Results of Applying StackMine: We next illustrate the results from each phase in details. In the phase of AOI extraction, waiting stacks were extracted from the UI thread using the Wait Graph model. Then, AOI extraction produced 41 thousand callstacks with average length of 36 function calls for each callstack. The phase of maximal-callstack-pattern mining produced 371 costly patterns. The phase of pattern clustering produced 251 pattern clusters. The entire automatic processing took less than 10 minutes.

Finally, these 251 pattern clusters were ranked according to the four different metrics as defined in Section B, and the top 60 pattern clusters in each prioritized list were examined. Based on the top pattern clusters, it took about 1 hour to manually derive 22 performance signatures, which achieved 66.69% performance bottleneck coverage of the collected 54 trace streams. In contrast, without the assistance of StackMine, it would have to take 9 to 27 hours to manually derive these performance signatures, based on empirical estimation.

Among the signatures, 16 of them with 55.80% performance bottleneck coverage in total corresponded to three major factors that were widely discussed on the Internet for slow startup of the application: (1) on-demand scanning conducted by anti-virus software before the application loads some external components, (2) loading of a non-trivial number of plugins, (3) loading of more-than-necessary contents by the prefetching functionality. The three main factors can be considered as known common issues, and developers of the application can make improvements accordingly.

Based on the remaining 6 signatures, 6 performance bugs were discovered with 10.89% performance bottleneck coverage. Among these 6 bugs, 5 of them have been confirmed with third-party sources and 1 of them is yet to be confirmed.

In summary, StackMine achieved substantial effort reduction with respect to research question Q1. For Q2, these 22 signatures achieved satisfying performance bottleneck coverage of 66.69%. Due to space limit, in this evaluation, we could not include detailed comparison with alternative techniques for Q3, which would have similar conclusions to what have been shown in the Windows study.

2) Representative Performance Bug: We next describe one representative performance bug that has been identified with StackMine. When ranked based on average waiting time, this bug emerged in the 24th cluster, with an average waiting time of 382 milliseconds over 36 trace streams. Note that among the higher-ranked 23 clusters, 4 of them were related to our 6 newly discovered bugs while the other 19 were related to known issues due to common factors. Figure 5 shows the common part of the patterns from this 24th cluster. It is divided into three segments, and between the segments there are variant function-call paths corresponding to different situations. Without clustering, the performance metric value of the same logic but along different call paths would be split into different patterns and thus such splitting would prohibit the bug from emerging from a pattern with a high performance metric value.

Reading the common part of the pattern cluster could help learn the following logic. In Segment 1, the application would load a set of components during the initialization stage. A hash table was used and the `GetShortPathName` Windows API function was invoked in `GetHashCode`. In Segment 2, the Windows kernel would perform some operations to support `GetShortPathName`. In Segment 3, hard faults happened and disk I/O was conducted, causing additional time consumption.

Based on the pattern cluster, further investigations exposed two major findings. On one hand, Segment 1 provided us with sufficient information to quickly locate the problem in the source code, e.g., the invocation of the `GetShortPathName` API function. The application used a hash table to manage the components that it loaded, and the

[3] Readers who would like to learn specifics of the third-party application could contact the last author.

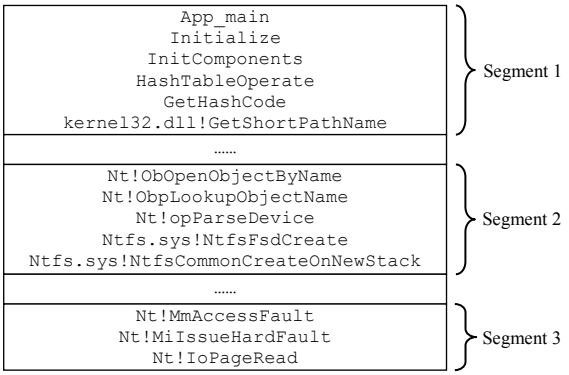

Figure 5. Callstack pattern of a slow startup bug of the third-party application

path name of a file was used as the key in the hash table. However, Windows has a historical issue about short path names. To ensure that both the long path name and short path name of a single component would map to the same entry in the hash table, the application adopted the short name and therefore the `GetShortPathName` API function was always invoked at the first time of loading a component.

On the other hand, the investigation on the trace streams exposed that the disk I/O in Segment 3 was for reading $MFT of the file system, and the readings on $MFT would introduce severe lock contention with many other applications and OS components; thus, the readings could be blocked for a long period of time.

It was observed that all the key functions involved in this performance bug had been captured and connected by a call context in the pattern cluster. Based on the call context, it was natural and efficient for analysts to figure out the performance bug. Beyond that, analysts can focus on this call context and figure out the corresponding optimization solution accordingly. It has been shown that the usage of long path names was dominant on recent Windows versions. Therefore, an optimization solution is to adopt the long path name as the key of the hash table. There are simple and reliable ways to detect whether a path name is a short name, and the `GetLongPathName` API function needs to be invoked only when a path name is detected as a short name. This optimization solution is expected to significantly reduce the chance to read $MFT of the file system, thus improving the performance.

VIII. Related Work

Previous work on performance debugging typically focused on one or a few full or sampled trace streams of a software system, in contrast to a large number of trace streams focused by StackMine for performance debugging in the large.

For example, Ammons et al. [5] proposed an approach that includes a search tool built upon a simple profile interface to help analysts explore summaries of profile measurements to search for performance bottlenecks within a few trace streams. Their approach heavily relies on manual effort to navigate through traces.

To analyze a few sampled trace streams collected from deployment sites of modern enterprise-class multi-tier server applications, the IBM Whole-system Analysis of Idle Time (WAIT) approach [3] helps analysts diagnose idle time (indicating a lack of forward motion), which corresponds to wait bugs handled by StackMine (which also handles CPU consumption bugs). Their approach heavily relies on an extensible set of manually specified declarative rules to abstract traces to states of observed idleness.

Srinivas and Srinivasan [20] proposed to use thresholding and filtering to summarize performance problems on a component basis, by identifying a small set of interesting function calls in manually specified components of interest. The IBM Jinsight tool [19] allows analysts to explore traces at different dimensions with visualization support. In contrast to these approaches, StackMine does not require manual specifications of components of interest or heavy manual effort to explore traces.

There were various previous approaches on applying frequent pattern mining or clustering on execution traces [4][8][13][24] or source code [1][6][10][12][16][18][21][22]. However, these approaches typically focus on mining API specifications, rather than performance debugging, calling for novel techniques in StackMine. There were a number of previous debugging approaches [9][11][14][15] based on real-world usage data but none of them focused on performance debugging.

IX. Conclusion

To enable *performance debugging in the large*, we have proposed the StackMine approach that conducts mining and clustering on callstack traces from trace streams related to performance issues encountered by real-world users. StackMine helps performance analysts effectively discover highly impactful performance bugs. Since December 2010, StackMine has been applied in performance-debugging activities at a Microsoft team for performance analysis, especially for a large number of execution traces. Our evaluations on two large-scale real-world software products (Microsoft Windows 7 and a third-party application) demonstrated StackMine's substantial benefits in performance debugging in the large.

Exemplified by WER and StackMine, we envision and advocate a game-changing paradigm for software quality assurance in the large based on usage data collected from the real world, in order to cope with increasingly large and complex modern software systems, such as ultra-large-scale systems.

Acknowledgment

The authors would like to thank the developers and analysts from the Microsoft product teams for the collaboration throughout the StackMine project. The authors would like to also thank the researchers from Microsoft Research for the discussions on the Wait Graph definition.

REFERENCES

[1] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining API patterns as partial orders from source code: from usage scenarios to specifications," *Proc. ESEC/FSE*, 2007, pp. 25–34.

[2] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," *Proc. VLDB*, 1994, pp. 487-499.

[3] E. Altman, M. Arnold, S. Fink, and N. Mitchell, "Performance analysis of idle programs," *Proc. OOPSLA*, 2010, pp. 739-753.

[4] G. Ammons, R. Bodik, and J.R. Larus, "Mining specifications," *Proc. POPL*, 2002, pp. 4–16.

[5] G. Ammons, J.-D. Choi, M. Gupta, and N. Swamy, "Finding and removing performance bottlenecks in large systems," *Proc. ECOOP*, 2004, pp. 170-194.

[6] R.-Y. Chang, A. Podgurski, and J. Yang, "Finding what's not there: a new approach to revealing neglected conditions in software," *Proc. ISSTA*, 2007, pp. 163–173.

[7] Y. Dang, S. Ge, R. Huang, and D. Zhang, "Code clone detection experience at Microsoft," *Proc. IWSC*, 2011, pp. 63-64.

[8] W. Dickinson, D. Leon, and A. Podgurski, "Finding failures by cluster analysis of execution profiles," *Proc. ICSE*, 2001, pp. 339-348.

[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," *Proc. SOSP,* 2009, pp. 103-116.

[10] Z. Li and Y. Zhou, "PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software codes," *Proc. ESEC/FSE*, 2005, pp. 306–315.

[11] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug isolation via remote program sampling," *Proc. PLDI*, 2003, pp. 141-154.

[12] E. Linstead, S. K. Bajracharya, T. C. Ngo, P. Rigor, C. V. Lopes, and P. Baldi, "Sourcerer: mining and searching internet-scale software repositories," *Data Min. Knowl. Discov.,* vol.18, no.2, Apr. 2009, pp. 300-336.

[13] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," *Proc. ICSE*, 2008, pp. 501-510.

[14] A. Orso, D. Liang, M. J. Harrold, and R. J. Lipton, "Gamma system: continuous evolution of software after deployment," *Proc. ISSTA*, 2002, pp. 65-69.

[15] A. A. Porter, C. Yilmaz, A. M. Memon, D. C. Schmidt, and B. Natarajan, "Skoll: A process and infrastructure for distributed continuous quality assurance," *IEEE Trans. Software Eng.,* vol.33, no.8, Aug. 2007, pp. 510-525.

[16] M.K. Ramanathan, A. Grama, and S. Jagannathan, "Path-sensitive inference of function precedence protocols," *Proc. ICSE*, 2007, pp. 240–250.

[17] E. S. Ristad and P. N. Yianilos, "Learning string-edit distance," *IEEE Trans. PAMI*, vol.20, no.5, May 1998, pp.522-532.

[18] T. Schäfer, J. Jonas, and M. Mezini, "Mining framework usage changes from instantiation code," *Proc. ICSE*, 2008, pp. 471-480.

[19] G. Sevitsky, W. De Pauw, and R. Konuru, "An information exploration tool for performance analysis of Java programs," *Proc. TOOLS Euro*, 2001, pp. 85-101.

[20] K. Srinivas and H. Srinivasan, "Summarizing application performance from a components perspective," *Proc. ESEC/FSE*, 2005, pp. 136–145.

[21] S. Thummalapenta and T. Xie, "Alattin: mining alternative patterns for detecting neglected conditions," *Proc. ASE*, 2009, pp. 283–294.

[22] S. Thummalapenta and T. Xie, "Mining exception-handling rules as sequence association rules," *Proc. ICSE*, 2009, pp. 496–506.

[23] J. Wang and J. Han, "BIDE: Efficient mining of frequent closed sequences," *Proc. ICDE*, 2004, pp. 79-90.

[24] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: mining temporal API rules from imperfect traces," *Proc. ICSE*, 2006, pp. 282–291.

[25] http://channel9.msdn.com/Blogs/Charles/Inside-Windows-7-Reliability-Performance-and-PerfTrack

[26] http://en.wikipedia.org/wiki/DTrace

[27] http://en.wikipedia.org/wiki/Wait-for_graph

[28] http://msdn.microsoft.com/en-us/library/windows/desktop/bb968803(v=vs.85).aspx

[29] http://research.csc.ncsu.edu/ase/projects/perfanalysis/