Context-Sensitive Delta Inference for Identifying Workload-Dependent Performance Bottlenecks

Xusheng Xiao[1] Shi Han[2] Dongmei Zhang[2] Tao Xie[1]
[1]Dept. of Computer Science, North Carolina State University, Raleigh, NC, USA
[2]Microsoft Research Asia, Beijing, China
[1]xxiao2@ncsu.edu, [2]{shihan, dongmeiz}@microsoft.com, [1]xie@csc.ncsu.edu

ABSTRACT

Software hangs can be caused by expensive operations in responsive actions (such as time-consuming operations in UI threads). Some of the expensive operations depend on the input workloads, referred to as workload-dependent performance bottlenecks (WDPBs). WDPBs are usually caused by workload-dependent loops (i.e., WDPB loops) that contain relatively expensive operations. Traditional performance testing and single-execution profiling may not reveal WDPBs due to incorrect assumptions of workloads. To address these issues, we propose the ΔInfer approach that predicts WDPB loops under large workloads via inferring iteration counts of WDPB loops using complexity models for the workload size. ΔInfer incorporates a novel concept named *context-sensitive delta inference* that consists of two parts: *temporal inference* for inferring the complexity models of different program locations, and *spatial inference* for identifying WDPB loops as WDPB candidates. We conducted evaluations on two popular open-source GUI applications, and identified impactful WDPBs that caused 10 performance bugs.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*; D.4.8 [**Software Engineering**]: Performance—*Modeling and prediction*

General Terms

Measurement, Performance

Keywords

Performance analysis, model prediction

1. INTRODUCTION

Performance problems exist widely in released software [18, 28]. As a type of widespread performance problems, software hangs cause unresponsiveness of software applications [33, 35]. A recent study of hang problems [33] shows that 27.04% of the 233 studied hang bugs are caused by time-consuming operations in responsive actions[1], such as expensive computations in the UI thread for GUI applications. Among the expensive operations that cause hang problems, some of these operations are constantly expensive (such as server initializations), whereas some of them depend on the input workloads. These problems are referred to as workload-dependent performance bottlenecks (WDPBs). WDPBs are usually caused by workload-dependent loops (referred to as WDPB loops)[2] that contain certain relatively expensive operations, such as temporary-object creation/destruction [37], file I/O, and UI updates.

To remove WDPBs, a typical solution is to move expensive operations out of the responsive actions, such as spawning separate threads to handle expensive operations in the background for GUI applications [9], or adding program logics to limit the size of workloads (e.g., allowing only up to a specific size k of workloads or processing only the first k items of a workload).

Although WDPBs can be identified with traditional approaches, such as performance testing and single-execution profiling (e.g., call-tree profiling [11, 12, 20] and stack sampling [23]), such traditional approaches are ineffective, suffering from two major issues: the insufficiency issue and the incompleteness issue. First, performance testing mainly relies on black-box random testing or manual input design, often insufficient to identify WDPBs that may not surface on small or even relatively large workloads [29]. A huge amount of existing legacy software lacks workload specifications, and specifications from performance engineers tend to be outdated over time. It often remains unclear to performance testers on how large is large enough for workloads to expose WDPBs (if any indeed exists in the application under test). A recent study [28] shows that 41 out of 109 studied performance bugs are due to wrong assumption of workloads. Second, by increasing the input workload, single-execution profiling may reveal the most expensive WDPBs, but it is often incomplete in capturing all the WDPBs that may cause performance problems when the workload size increases. For example, given a large workload, some WDPBs' cost may occupy more than 90% of the total cost, dominating the cost of other WDPBs. In other words, some important WDPBs can be overshadowed by other WDPBs.

To address these two issues suffered by traditional approaches, our research contributes a novel predictive approach, called ΔInfer. ΔInfer *predicts* occurrences of WDPB

[1]Actions that are expected to return instantly.
[2]A loop whose iteration count depends on the input workload.

loops within a GUI application[3] under large future workloads (that have not been generated or executed yet) in contrast to existing approaches on performance testing, which require the generated and executed workloads to directly expose these WDPB loops. In addition, to gain the prediction power, our approach infers complexity models of program locations within the application under analysis from profiles of *multiple* workloads instead of the profile from just a single workload, which existing approaches on single-execution profiling focus on. Our approach then infers complexity models for loops based on the complexity models of program locations at the loop bodies. Such complexity model for a loop captures the relationship between the iteration count of the loop and the workload size, and then is used to predict the iteration count of the loop given a workload.

To identify WDPB loops in GUI applications, our approach addresses two significant challenges: complex contexts and implicit loops. First, in GUI applications, developers usually write code as handlers for various UI events (e.g., button clicks or item selections). When an event is fired, the corresponding handlers would be invoked. Such event-driven nature causes a program location to be invoked in different contexts. Thus, a program location may exhibit quite different execution complexities under different calling contexts, posing challenges for a complexity model to accurately model its complexity. Second, among the most widely-used UI controls, multi-item UI controls (e.g., ListView or TreeView) [32] may fire events for each item, behaving like an *implicit loop* that invokes the handlers repetitively. Such implicit loops do not have explicit loop statements in the application, posing challenges for manual inspection or static analysis [13, 22, 38] to identify the WDPB loops.

To address the aforementioned challenges, our ΔInfer approach incorporates a novel general concept: *context-sensitive delta inference*, which consists of two major parts: *temporal inference* (inferring differences between executions) and *spatial inference* (inferring differences between program locations).

Temporal Inference. The temporal inference analyzes the *differences of execution counts* of a specific program location among its executions under different workloads to infer complexity models. ΔInfer employs least-squares regressions (such as linear and power-law regressions) [15] to infer a complexity model that uses the workload to predict a program location's execution count.

To address the challenge of complex contexts posed by GUI applications, our approach is context-sensitive: our approach infers complexity models from behaviors exhibited by the executions of a program location under the same calling context (from its caller up to the root function such as a main function or thread-start function), instead of executions of the program location under different calling contexts.

To improve the accuracy of the inferred complexity models, ΔInfer starts with training profiles of workloads selected from the representative usage, and iteratively selects new workloads to obtain new profiles based on the prediction accuracy of the inferred models in previous iterations. The iteration of the model inference and refinement continues until the model accuracy reaches a specified threshold.

[3]Among applications with WDPBs, our research focuses on identifying WDPBs in GUI applications, since responsiveness in GUI applications is a major source of performance problems [33, 35].

Spatial Inference. The spatial inference analyzes *differences of complexity models across program locations* to identify workload-dependent loops as WDPB candidates. If a complexity model for the workload is used to describe a program location's execution count (i.e., $count = f(workload)$), it can be observed that workload-dependent loop raises the complexity model of the program locations inside the loop body to a higher order (such as *constant* to *linear*), and results in a complexity transition. Thus, the *order differences* between complexity models of *different program locations*, i.e., complexity transitions, can be used to effectively identify workload-dependent loops. To identify complexity transitions as workload-dependent loops, the spatial inference abstracts orders from the inferred complexity models and compares the orders for a loop's entry point and program locations inside the loop body. If we denote program locations as methods, then we compare the orders of caller-callee pairs, since callees inside a loop body would exhibit different complexity orders.

To address the challenge of implicit loops posed by GUI applications, ΔInfer uses complexity transitions from certain UI library calls to the application code to identify implicit loops. All the complexity transitions inferred by ΔInfer are considered as WDPB candidates. Based on the complexity models and the average cost per execution obtained from the profiles, ΔInfer predicts costs of the complexity transitions on large workloads to identify WDPBs.

This paper makes the following major contributions:

- A predictive approach for performance analysis that predicts WDPBs on large workloads based on the concept of context-sensitive delta inference.
- A technique of temporal inference that iteratively infers and refines context-sensitive complexity models based on the deltas of executions on different workloads.
- A technique of spatial inference that abstracts orders from complexity models of locations and identifies the order deltas of different locations to infer complexity transitions. The predicted costs of such complexity transitions on large workloads are used to identify WDPBs.
- Evaluations on two popular open source GUI applications, the 7-Zip file manager [1] and Notepad++ [7]. The results show that our approach effectively identifies impactful WDPBs that cause 10 performance bugs.

2. PROBLEM FORMULATION

In this section, we formalize the problem of identifying complexity transitions. For a given application A, we use the term *location*, l, to denote a program location (e.g., a basic block in a method or a method itself) of A, and *cost*, y, to denote a location's performance (e.g., execution count or time). To formulate our context-sensitive analysis and complexity transitions, we first define the *call graph* G for A and the *calling context* c of a location l in A.

Definition 1. *A* ***call graph*** *is a directed graph* $G(E, V)$*, where each vertex* $v \in V$ *denotes a unique method, and each edge* $e(a, b) \in E$ *denotes a calling relationship from* a *to* b*.*

Without losing the generality, we use the term *belonging method* to denote a method where l is in when l represents a basic block, or a method represented by l. l's belonging method corresponds to a vertex v in G.

DEFINITION 2. *A **calling context**, c, of a location l is a call path from the root of the call graph (usually a main function or thread-start function) to the parent vertex (caller) of vertex v corresponding to l's belonging method.*

To simplify description, we denote a location l under a calling context c as l_c in the rest of the paper. Using the calling context, we then define the call-tree profiling [11] used in our approach.

DEFINITION 3. *An **execution profile**, P, obtained by executing an application A on a given input, is a call-tree profile that records the execution counts of each location l_c in A.*

An input to A can have a set of parameters that characterize the input from different aspects. Based on the scenarios of A, we identify workload parameters $(W_1, \ldots, W_d)$ that could potentially influence performance, such as the number of lines or the number of characters for the text input to a text editor. For k workloads, we have a vector of values for each workload parameter W_d $(< w_{d,1}, \ldots, w_{d,k} >)$ to denote the values of W_i for these k workloads. After executing the application A on k workloads to obtain k profiles, we have a vector of counters for each location l_c $(< y_{l_c,1}, \ldots, y_{l_c,k} >)$. Based on these vectors, we then define the *k-profile graph* as below.

DEFINITION 4. *A **k-profile graph** is an annotated call graph, $G(E, V)$, where a location l with its corresponding vertex is annotated with a vector of counters for l on k workloads for each of its calling context c.*

For each location l with its corresponding vertex in the k-profile graph, our approach infers **complexity models** using regression learning.

DEFINITION 5. *Given a workload parameter W, a **complexity model** of a location l under the calling context c is a function $f_{l,c}(W)$ that predicts l's execution counts in terms of values of W under the calling context c.*

Based on the definition of a complexity model, we denote the exponent of the highest order term of the complexity model as the *order* of the complexity model, denoted as $O(f_{l,c}(W))$. We next define a *complexity transition.*

DEFINITION 6. *Given a workload parameter W, a **complexity transition** is a pair (n, M), such that*

1. *n is a vertex (method) in the k-profile graph and M is a subset of children vertices (callees) of n;*
2. *$f_{n,c}(W)$ is the complexity model of n under the calling context c, and $f_{l_i,c_i}(W)$ is the complexity model of the location l_i, where l_i is a location in M and the calling context c_i is c concatenated with n.*
3. *$O(f_{l_i,c_i}(W))$ is at least 1 **more than** $O(f_{n,c}(W))$;*
4. *$\forall l_i, l_j \in M, i \neq j$, $O(f_{l_i,c_i}(W)) = O(f_{l_j,c_j}(W))$.*

The definition ensures that a complexity transition captures the workload-dependent loops whose iteration bounds have the same order inside the method n under the calling context c. To simplify our description in the rest of the paper, we use methods as the locations.

3. EXAMPLES

In this section, we use an example to illustrate how ΔInfer identifies WDPBs. Figure 1 shows two WDPBs found in the 7-Zip file manager [1] written in C++. The first WDPB (PB_1) is caused by the method CPanel::OnRefreshStatusBar, which contains a non-trivial UI update operation (Line 4). PB_1 is invoked when a selection-change event is fired. The second WDPB (PB_2) is caused by the method GetSelectedItemsIndices, which contains a workload-dependent loop L (Lines 11-12). Let us assume that the number of files in the current folder is n; if a user clicks a file after selecting all files, the selection-change event will be fired for each file. Such repeated firing will cause PB_1 to be invoked n times (refreshing the status bar n times), and PB_2 to be executed n^2 times. Thus, when n is large, PB_1 and PB_2 would be very expensive and cause the 7-Zip file manager [1] to hang.

```
1  void CPanel::OnRefreshStatusBar() { // PB_1
2    ...
3    GetOperatedItemIndices(indices);
4    _statusBar.SetText(...); // UI operation
5    ... }
6  void CPanel::GetOperatedItemIndices(CRecordVector<UInt32>
         &indices) const {
7    GetSelectedItemsIndices(indices);
8     ... }
9  void GetSelectedItemsIndices(CRecordVector<UInt32> &indices) {
10   indices.Clear();
11   for (int i = 0; i < _selectedStatusVector.Size(); i++) // PB_2
12     if (_selectedStatusVector[i]) indices.Add(i);
13   ... }
```

Figure 1: Two WDPBs found in the 7-Zip file manager [1]

With ΔInfer, developers can perform the selection-change action to obtain the profiles on multiple workloads, and use the prediction results of ΔInfer to identify WDPBs on large workloads. To simplify the description, here we simply assume that the selected workload values are 50, 100, and 200 files, and we obtain the profiles P_{50}, P_{100}, and P_{200}.

With these profiles as input, ΔInfer infers complexity models by using regression learning to fit the execution counts of methods to workload sizes. In Figure 1, CPanel::OnRefreshStatusBar is associated with a linear complexity model, and any method call inside the loop (Lines 11-12) is associated with a quadratic complexity model. By inferring complexity transitions from lower order to higher order (e.g., *constant to linear* and *linear to quadratic*), the complexity transition from some UI library call (not shown in Figure 1) to CPanel::OnRefreshStatusBar() helps identify the implicit loop, and the complexity transition from GetSelectedItemsIndices to _selectedStatusVector.Size helps identify the loop L. These workload-dependent loops are considered as WDPB candidates.

To predict whether PB_1 and PB_2 would cause performance problems on large workloads, ΔInfer predicts the execution counts of PB_1 and PB_2 when the workloads become 10 or 100 times larger (i.e., 500 or 5000). With the predicted execution counts, ΔInfer then uses their average costs (e.g., execution time) per execution to compute their estimated costs on these large workloads. Although PB_2 belongs to the callees of PB_1, the complexity model of PB_2 has a higher order than PB_1's model. Thus, ΔInfer separates the predicted costs of PB_2 from PB_1.

With the predicted costs on large workloads, ΔInfer ranks PB_1 and PB_2 as the top 2 complexity transitions (others are not illustrated here due to space limit), and their combined costs are 10 times of P_{100}'s cost when the workload is 5000. Such costs significantly degrade the performance and cause the file manager to hang. Recall that PB_1 contains a UI-update operation and has a non-trivial cost per exe-

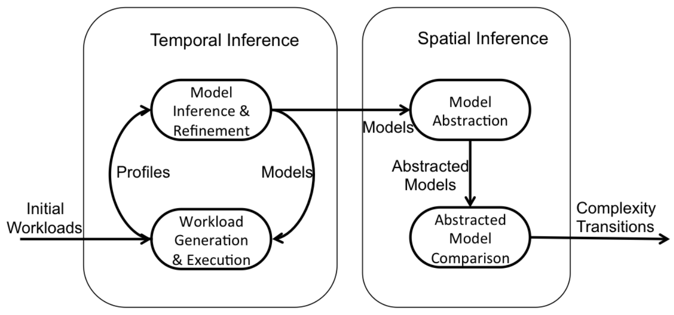


Figure 2: Overview of ΔInfer

cution, and the cost of PB_2 grows much faster than PB_1 due to PB_2's n^2 complexity model, even though the cost per execution for PB_2 is not that large.

4. APPROACH OVERVIEW

In this section, we present the overview of ΔInfer. As shown in Figure 2, ΔInfer consists of two major parts: temporal inference and spatial inference.

The temporal inference accepts profiles of different workloads as input and infers context-sensitive complexity models. ΔInfer starts by applying regression learning on an initial set of profiles to infer complexity models, and iteratively selects new workloads to refine the inferred models. To validate the inferred models after each iteration, ΔInfer uses a random-validation strategy. ΔInfer repeats the model inference and refinement until the model accuracy reaches a pre-specified threshold.

The spatial inference accepts the inferred complexity models from temporal inference and infers complexity transitions as WDPB candidates. In particular, to make the complexity models comparable, ΔInfer abstracts orders from complexity models, and compares the orders of caller-callee pairs to infer complexity transitions as WDPB candidates. ΔInfer then predicts costs of the complexity transitions on large workloads to identify WDPBs.

5. TEMPORAL INFERENCE

This section describes how the technique of temporal inference infers and refines complexity models.

5.1 Workload Generation and Execution

Our approach focuses on detecting scenario-specific WDPBs for an application under analysis. Each scenario is assumed to use a specific configuration of the GUI, such as switching the GUI to the “Wrap Line” mode for a text editor. Based on the chosen scenarios, performance analysts select appropriate performance metrics (such as execution time or energy cost), and characterize the input as performance-relevant workload parameters [26], such as *# lines* in a document as the input to a text editor. When we generate workloads based on a parameter W (referred to as the focused workload parameter), we vary workloads only on W, while the values of other parameters remain the same. For example, when we vary the focused workload parameter *# lines* to generate different workloads, we keep constant the other parameters such as *# of characters in a line.* Doing so can help us avoid the difficulties on inferring models for multiple parameters.

To select an initial set of workloads, performance analysts are expected to define the representative value range (RVR) for the focused workload parameter. For example, the RVR for *# lines* in a document can be $[1, 1280]$. Often the time, performance analysts and developers are well aware of RVRs and can agree on RVRs with a certain variance, but it is difficult to know a triggering workload value for an unknown performance bottleneck. Within the RVR, performance analysts can select an initial value and vary the initial value via arithmetic progression or geometric progression to obtain sorted inputs [38], or can choose the values randomly. For the least-squares regression used by our approach, a guideline for selecting the initial values is to avoid selecting a very small workload, such as 1 or 2 files, as the initial workload for a file manager. We empirically find that such small workloads produce noise in the inferred models, consistent with the finding by Goldsmith et al. [19].

To obtain execution profiles, we instrument the application and execute the application on the chosen workloads. The profiles used by our approach are call-tree profiles, which measure execution counts of program locations in the instrumented application, and distribute the total execution counts of a location for each of its calling contexts [11]. In this paper, since we focus on GUI applications whose responsive action is in the UI thread, our approach uses the execution profiles of the UI thread as input.

5.2 Least-Squares Regression

Given the counter vector of a location l under a calling context c ($< y_{l_c,1}, y_{l_c,2}, \ldots, y_{l_c,k} >$) and the value vector of a workload parameter W ($< w_1, w_2, \ldots, w_k >$), our approach uses least-squares regressions [15], including linear and power law regressions, to infer a complexity model using a set of data points $(w_i, y_{l_c,i})$.

Linear Regression. Linear regression infers a complexity model $y = A + Bw$ and predicts $y_{l_c,i}$ as $\hat{y_{l_c,i}} = A + Bw_i$. The difference $y_{l_c,i} - \hat{y_{l_c,i}}$ is called the residual of the fit at $(w_i, y_{l_c,i})$. Linear regression finds parameters A and B to minimize the sum of squared residuals, $Q(A, B)$, where $Q(A, B) = \sum_{i=1}^{k}(y_{l_c,i} - (A + Bw_i))^2$.

Power-law Regression. Power-law regression infers a complexity model $y = Aw^B$ and predicts $y_{l_c,i}$ as $\hat{y_{l_c,i}} = Aw_i^B$. Power-law regression finds parameters A and B to minimize the sum of squared residuals, $Q(A, B)$, where $Q(A, B) = \sum_{i=1}^{k}(y_{l_c,i} - (Aw_i^B))^2$.

To measure how good the models fit the data points, our approach computes the correlation coefficient R^2. For the linear regression, the R^2 is defined as below:

$$R^2 = \frac{(\sum_{i=1}^{k} wy - k\bar{w}\bar{y})^2}{(\sum_{i=1}^{k} w^2 - k\bar{w}^2)(\sum_{i=1}^{k} y^2 - k\bar{y}^2)}$$

By replacing w with $ln(w)$ and y with $ln(y)$, we can transform a power-law regression model to a linear regression model: $ln(y) = ln(Aln(w)^B) = ln(A) + Bln(w)$. Thus, R^2 is applicable to power-law regressions by replacing w with $ln(w)$ and y with $ln(y)$.

Using both linear and power-law regressions, regressions can fit a set of data points $(w_i, y_{l_c,i})$ of a location l to two complexity models. Our approach selects the complexity model with better R^2 as the complexity model for l.

5.3 Model Validation

The model validation provides the relative prediction error of the inferred models. While the correlation coefficient R^2 measures how the models fit the given data points, the relative prediction error measures how good the prediction accuracy of the models is. For each iteration of model validation, the model validation compares the predicted values of the inferred models to the values of the corresponding locations in all validation profiles, and computes the average relative error.

In our current approach, we use a random-validation strategy for the model validation. We randomly select a set of workload values from a validation value range (VVR) predetermined based on a guideline (described below), and obtain a set of corresponding validation profiles for determining the accuracy of the models. To prevent the models from overfitting the values within the RVR and better validate the prediction accuracy of the inferred models, VVR must include RVR and the value range should be larger than the RVR. For example, if the RVR is $[1, 500]$, we may set the VVR as $[1, 1000]$. We suggest that the range of VVR should be at least 2 times larger than the RVR. However, a very large validation range would require more iterations to refine the models, and large workload values cause long processing time in obtaining validation profiles. Thus, a very large validation range is not cost-effective for the iterative model inference and refinement.

The advantage of the random-validation strategy is that the validation workloads are distributed across the validation range, preventing models from over-fitting a specific range of values within the whole validation range. Note that the randomly selected workload values cannot be the same as the workload values used to infer the models.

5.4 Model Inference and Refinement

Model inference and refinement accept as input a set of profiles on initial workloads and a set of validation profiles, iteratively select new workloads to improve the inferred models, and output the inferred complexity models. These inferred complexity models are then associated with the corresponding vertices in the k-profile graph.

The number of initial workloads and the number of new workloads selected for each iteration can be configured by performance analysts. The configuration mainly depends on the value range of the workload parameter and the time taken to obtain a profile. Adding the new workloads in subsequent iterations improves the average relative error of the inferred models. We terminate the iterations if the average prediction error falls below a predefined prediction-error threshold (e.g., 5%). In certain cases, after adding new profiles, the improvement on the accuracies of the inferred models may be marginal or even negative, and the number of the inferred models whose R^2 is above a predefined R^2 threshold (referred to as $threshold_{R^2}$) may not change. We terminate the iterations for such cases to prevent infinite iterations.

Algorithm 1 shows the details of our iterative algorithm for model inference and refinement. The main part of the algorithm is the iteration cycle of inferring and refining complexity models (Lines 3-30), where the guard condition of the loop at Line 3 ensures that the algorithm terminates after the predefined maximum number of iterations has been reached. Lines 1-2 initialize the model count pc and the prediction error e_{pre}. pc records the number of models whose R^2 are above $threshold_{R^2}$ (e.g., 0.9) in the previous iteration, and e_{pre} records the average prediction error in the previous iteration. Within the iteration cycle, $ModelInfer$ first infers complexity models (Lines 4-6) based on the set of profiles. $ModelInfer$ obtains a k-profile graph by aligning the set of profiles (Line 4) and retrieves the workload values from the profiles (Line 5). $ModelInfer$ then applies regression learning with the workload values and the k-profile graph as input to infer the complexity models (Line 6). Here regression learning returns only the inferred models whose R^2 is above $threshold_{R^2}$. In the next step, $ModelInfer$ computes the errors e_M for each validation profile (Lines 8-17) and the average error e_{total} for all validation profiles (Line 18). Based on the computed errors, $ModelInfer$ terminates the iteration and returns the current models (Line 31) if one of the following two conditions is satisfied: (1) the improvement of the average error is less than the threshold of model improvement $threshold_{imp}$ (Line 19); (2) the average error is less than the threshold of prediction error $threshold_e$ and $M.Count$ is the same as the previous iteration pc (Line 22). If neither of the condition is satisfied, $ModelInfer$ selects a new workload for obtaining a new profile (Lines 25-26), and updates the average error e_{pre} and model count pc for the next iteration. After the iteration terminates, these complexity models are then associated with the corresponding vertices in the k-profile graph.

Algorithm 1 $ModelInfer$

```
Require: $P$ as a set of profiles, $VP$ as a set of validation profiles
Ensure: $M$ as complexity models
 1: $pc = -1$ // previous model count
 2: $e_{pre} = -1$ // previous error
 3: for $ite = 0; ite < max; ite = ite + 1$ do
 4:   $kG = AlignProfiles(P)$
 5:   $x = GetWorkloads(P)$
 6:   $M = RegressionLearning(x, kG)$
 7:   $e_M = \{\}$
 8:   for all $vp$ in $VP$ do
 9:     $w = GetWorkload(vp)$
10:     $e_{vp} = 0.0$
11:     for all $m$ in $M$ do
12:       $r_w = Predict(w, m)$
13:       $a_m = GetActual(vp, m)$
14:       $e_{vp} = e_{vp} + \frac{Abs(a_m - r_w)}{a_m}$
15:     end for
16:     $e_M = e_M.Add(\frac{e_{vp}}{M.Count})$
17:   end for
18:   $e_{total} = \frac{Sum(e_M)}{VP.Count}$
19:   if $e_{pre} - e_{total} < threshold_{imp}$ then
20:     break // improvement is below the threshold
21:   end if
22:   if $e_{total} < threshold_e$ AND $pc == M.Count$ then
23:     break // accuracy is acceptable
24:   else
25:     $np = NewWorkload(P, VP, e_M)$
26:     $P = P.Add(np)$
27:     $pc = M.Count$
28:     $e_{pre} = e_{total}$
29:   end if
30: end for
31: return $M$
```

Aligning Profiles. Given k execution profiles on k workloads, our approach aligns the locations in the profiles using calling contexts, and represents the execution counts of each location l under each calling contex c in k profiles as a vector: $(y_{l_c,1}, y_{l_c,2}, \ldots, y_{l_c,k})$. Our approach then builds the k-profile graph by associating the vectors with each location.

Selecting New Workloads. Our workload-augmentation mechanism ($NewWorkload$ at Line 25) is based on the assumption that a new workload at the area with the highest prediction error improves most the prediction errors of the models. $NewWorkload$ first finds the validation profile p_e that has the highest prediction error based on e_M, and then identifies a profile in P whose workload value w_c is closest to the workload value w_e of p_e. $NewWorkload$ returns the center of w_c and w_e as the new workload value. If the new workload value exceeds the RVR, our approach doubles the ranges of RVR and VVR, and selects a new validation profile in the extended range not overlapping with the original VVR. By doing so, our approach may adaptively evolve the ranges of RVR and VVR to improve the model accuracies.

6. SPATIAL INFERENCE

This section describes how the technique of spatial inference infers complexity transitions by comparing abstracted models and predicts costs of complexity transitions on large workloads.

6.1 Abstraction of Model

For each vertex associated with a complexity model in the k-profile graph, model abstraction extracts the exponent of the highest order term from the complexity model as the order of the complexity model:

- For a complexity model inferred by linear regressions ($y = A + Bw$), the abstracted order of the model is 1 if B is larger than 0; otherwise, the abstracted order of the model is 0.
- For a complexity model whose orders are decimal values, the abstracted order of the model is the closest integer.
- For a complexity model whose R^2 is below $threshold_{R^2}$, the abstracted order of the model is 0.

These orders are used as the abstracted models for each complexity model in the k-profile graph, making the complexity models comparable for each caller-callee pair.

6.2 Inference of Complexity Transitions

Our algorithm, $TransInfer$, accepts as input a k-profile graph G with vertices associated with abstracted models, and outputs a set of inferred complexity transitions T. The details are shown in Algorithm 2.

$TransInfer$ starts by creating an empty set of T, and retrieving a vertex v from $G.vertices$ (Lines 1-2). $TransInfer$ next iterates over each child vertex $child$ of v (Line 3). For each calling context c of v (Line 4), $TransInfer$ computes the children context cc by appending v to c (Line 5). $TransInfer$ checks whether the order of the complexity model of $child$ under the calling context cc is at least 1 more than the complexity model of v under the calling context c (Line 6). If the condition at Line 6 is not satisfied, $TransInfer$ continues to check the next child vertex (back to Line 3). If the condition at Line 6 is satisfied, $TransInfer$ further checks whether there exist complexity transitions whose complexity model has the same order as the complexity model of $child$ (Line 10), and appends $child$ to the transition if the condition at Line 10 is satisfied (Line 12). If $TransInfer$ does not find an existing transition whose complexity model matches $child.model$, $TransInfer$ creates a new complexity transition from v to $child$ under the calling context c (Line 16). After all children vertices of v are checked, $TransInfer$ continues to check the next vertex v (back to Line 2). The algorithm continues until all the vertices are checked and outputs the set of complexity transitions T (Line 19), which captures workload-dependent loops, including implicit loops.

Algorithm 2 $TransInfer$

```
Require: $G$ as a $k$-profile graph with vertices associated
    with abstracted models
Ensure: $T$ as complexity transitions
 1: $T = \{\}$ // empty set
 2: for all $v$ in $G.vertices$ do
 3:   for all $child$ in $v.Children$ do
 4:     for all $c$ in $v.Contexts$ do
 5:       $cc = c.append(v)$ // children context
 6:       if $child.model(cc).O >= v.model(c).O + 1$ then
 7:         $trans = T.GetTransitions(v, c)$
 8:         $found = false$
 9:         for all $tran$ in $trans$ do
10:           if $tran.model.O == child.model(cc).O$
              then
11:             $found = true$
12:             $tran.AddVertex(child)$
13:           end if
14:         end for
15:         if $!found$ then
16:           $T.AddTransition(v, c, child)$
17:         end if
18:       end if
19:     end for
20:   end for
21: end for
22: return $T$
```

6.3 Cost Prediction of Complexity Transitions

To predict costs of complexity transitions (n, M) on large workloads, our approach computes the average costs per execution avg_{l_c} for each location l_c in M and the predicted execution count $pred_{l_c}$ for each location l_c. To obtain avg_{l_c} for a location l_c, our approach finds l_c on each profile p given for inferring the complexity models, computes the cost per execution $avg_{l_c,p}$ count for each profile p, and then computes avg_{l_c} by computing the average of $avg_{l_c,p}$ for each p. Given a workload w, our approach predicts the execution count $pred_{l_c}$ of a location l_c using its complexity model, and then computes the cost of l_c by multiplying $pred_{l_c}$ with avg_{l_c}. By summing up the costs of each location l_c in M, our approach obtains the predicted cost for (n, M).

7. EVALUATIONS

To show the effectiveness of ΔInfer, we conducted evaluations on popular open source GUI applications (Notepad++ [7] and 7-Zip [1]). In our evaluations, we seek to answer the following research questions:

- **RQ1**: How effectively does ΔInfer identify WDPBs?
- **RQ2**: How effectively does the iterative model refinement improve the accuracy of the inferred complexity models?

Table 1: Scenarios for the evaluations

ID	Scenario	W. Param
(**S1**)	open a folder	# files
(**S2**)	rename a file	# files
(**S3**)	select all items and then click the first item	# files
(**S4**)	create a folder	# files
(**S5**)	delete a file	# files
(**S6**)	open a file	# lines
(**S7**)	enter a character and save the file	# lines
(**S8**)	go to the last line	# lines
(**S9**)	find a word not present in the file	# char
(**S10**)	cut and paste the first character	# lines

- **RQ3**: How effectively does context-sensitive analysis improve the precision in identifying complexity transitions?

7.1 Subjects and Evaluation Setup

Subject Applications. Since our current implementation supports only Windows applications, we use two popular Windows applications from SourceForge [10][4] as the evaluation subjects: 7-Zip [1] and Notepad++ [7]. 7-Zip is a file archiver with high compression ratio, supporting archive file formats of 7z, zip, and so on. Our evaluations focused on the file manager of 7-Zip (7-Zip FM), a GUI tool that enables users to easily navigate and manipulate files for archiving. This application was rated by 83% of 30,081 users as recommended.[5] The version of 7-Zip used for our evaluations is 9.20, which consists of 86 files and 7,280 LOC. Notepad++ is a text editor and source-code editor for Windows, supporting tabbed editing and several programming languages (e.g., C/C++, Java, and C#). This application was rated by 94% of 14,950 users as recommended. The version of the Notepad++ used for our evaluations is 5.9.0, which consists of 396 files and 155,300 LOC.

These GUI applications represent different types of widely used GUI applications. Their GUIs consist of various types of GUI controls, such as buttons, list views, and text editors. We believe that the characteristics of these applications' WDPBs and performance bugs would be representative for many other GUI applications.

Evaluation Setup. As we do not have the developer knowledge of these subject applications, we choose scenarios that would manipulate inputs, so that the performance of the applications will vary based on workloads. Based on the scenarios, we characterize some inputs as performance-relevant workload parameters. The details of the scenarios and focused workload parameters are shown in Table 1. Column "ID" shows the scenario ID, Column "Scenario" shows the actions performed in each scenario, and Column "W. Param" shows the workload parameters that we use to vary the focused workload values. **S1**-**S5** are scenarios for the 7-Zip file manager, and **S6**-**S10** are scenarios for Notepad++.

For Notepad++, we configure it to use the "Word Wrap" mode, so that we can test its functionality of wrapping words and other functionalities at the same time for each scenario. We executed the instrumented subject applications on each workload, performed the interactions described in each scenario, and collected the call-tree profiles after the executions.

Thresholds. In our evaluations, we configure the maximum iteration count max to be 20, the threshold of R^2 for regression learning $threshold_{R^2}$ to be 0.9, the threshold of error improvement $threshold_{imp}$ to be 2%, and the threshold of prediction error $threshold_e$ to be 5%.

Workload Selection. For **S1**-**S8** and **S10**, we set the RVR as $[1, 1280]$ for choosing workload values and the VVR as $[1, 2560]$ for choosing random validation values. For **S9**, the value of the workload parameter *# char* is relatively large in practice, and thus we set the RVR as $[1, 20480]$ and the VVR as $[1, 40960]$. For each scenario, we select 3 values from the RVR to obtain initial training profiles, and randomly select 5 values from the VVR to obtain validation profiles. To observe how initial workloads may affect our algorithm of model inference and refinement, we design two contrast groups for each subject application by selecting different initial workload values. For the 7-Zip file manager, we use $\{20, 40, 80\}$ for **S1**-**S3** and $\{100, 200, 400\}$ for **S4**-**S5**; for Notepad++, we use $\{20, 40, 80\}$ for **S6**-**S7**, $\{100, 200, 400\}$ for **S8** and **S10**, and $\{1000, 2000, 4000\}$ for **S9**. When we choose workload values for the parameter *# lines*, we keep the number of characters on each line to be 200.

We applied ΔInfer to infer complexity models using the training profiles as input and iteratively select new workloads to obtain new profiles for improving the accuracies of the inferred models.

7.2 RQ1: WDPB Identification

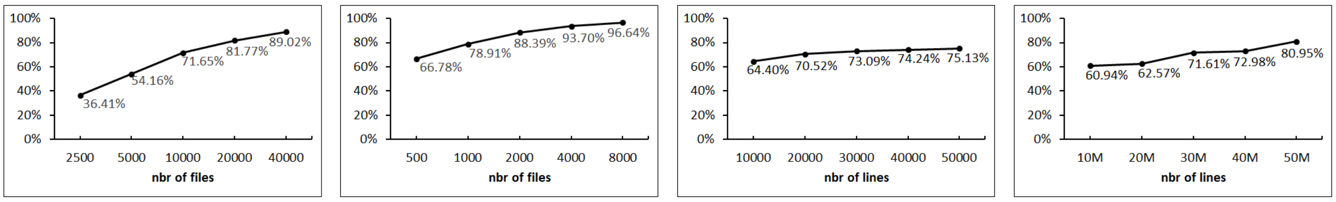
To answer RQ1, we first rank the complexity transitions using the predicted costs, and manually inspect the complexity transitions to confirm whether they will cause performance problems on large workloads. We then report the performance bugs caused by the identified WDPBs and get the confirmation from developers. Although it is difficult to measure the false negatives of ΔInfer, we propose and measure the cost coverage (i.e., execution time) of the identified WDPBs as the workloads increase. If the identified WDPBs achieve high cost coverage, the probability of ΔInfer missing impactful WDPBs would be low.

7-Zip File Manager. Due to space limit, we describe only two representative WDPBs in this paper. More details of the WDPBs can be found on our project website [2]. The first WDPB is RefreshListCtrl, a common WDPB for the Scenarios **S1**, **S2**, **S3**, and **S5**. In these scenarios, when the user opens/creates a folder or renames/deletes a file, the method RefreshListCtrl is invoked to refresh the list-view control. In RefreshListCtrl, the linear-workload-dependent loop is captured by the complexity transition (RefreshListCtrl, {GetItemRelPath, ...}). Inside the loop, GetItemRelPath computes the path prefix of a file using customized string-concatenation operations, which create temporary objects and destroy them after the concatenation. This computation results in intensive creations/destructions of temporary objects.

Another WDPB is a method that invokes ListView_SortItems. ListView_SortItems is a UI library call that repetitively invokes CompareItems to sort the files, behaving like an implicit loop whose complexity model is $nlog(n)$ in theory. ΔInfer identifies this implicit loop with a complexity transition from a constant order to a power-law order. Due to the inefficient implementation of retrieving file properties for comparison, this implicit loop causes a WDPB on large workloads.

Notepad++. We describe two representative WDPBs for Notepad++. The first WDPB is WrapLines that appears in every scenario. In these scenarios, when the user opens a file or modifies the content of the file (**S7** and **S10**), the method

[4]The largest open source applications and software directory
[5]All the rating data was collected on Dec. 23, 2011.



(a) 7-Zip FM (S1) (b) 7-Zip FM (S3) (c) Notepad++-(S6) (d) Notepad++-(S9)

Figure 3: Cost coverages of identified WDPBs as workloads increase

WrapLines is invoked to recompute the word-wrapping data structures by invoking WrapOneLine. WrapOneLine computes the layout of a line, creating temporary objects and dynamically allocating memory chunks for the computation results; such computation is quite expensive when the workload is large.

The second WDPB is Document::FindText for **S9**. ΔInfer identifies that Document::FindText contains a linear-workload-dependent loop and performs string comparison to find the matching word in the document. Although the cost per iteration is not high, the workload value in terms of *# chars* is easy to become huge in practice. For example, searching a 10MB file for matching a character not present in the document would cause the loop in Document::FindText to execute 10M times. Thus, word search is usually considered expensive; and many editors or viewers (such as Adobe PDF Viewer) spawn a separate thread to do so.

Cost Coverage. Figure 3 shows the cost coverage of the representative WDPBs on larger workloads. Due to space limit, we choose four scenarios to show the cost coverage of representative WDPBs. For 7-Zip, we choose **S1** to show the coverage of the WDPB caused by RefreshListCtrl, and **S3** for the WDPB caused by RefreshStatusBar. For Notepad++, we choose **S6** for the WDPB caused by WrapLines and **S9** for the WDPB caused by Document::FindText. The results show that the identified WDPBs account for more than 75% of all the scenarios when the workloads increase, indicating that the probability of missing impactful WDPBs is low. Moreover, as shown in Figure 3(b), the cost coverage increases so quickly that it reaches more than 90% on the workload of 8000 files. The reason is that the WDPB of **S3** has the quadratic complexity model.

Bug Confirmation. We reported the detected bugs to the project's forum, and the responses of the developers of 7-Zip are quite encouraging. They confirmed the bugs caused by RefreshListCtrl in **S1**, **S2**, **S3**, and **S5** (4 out of 5 bugs reported), and plan to fix the bugs in the next version [5]. Although the bug in **S1** was reported by others in the developers' bug database (*#3193577*), our approach further identifies the WDPBs as the root cause of the bug. Capturing the bug caused by OnRefreshStatusBar in **S4** demonstrates the advantages of using predictive models in our approach. Such bug is difficult for traditional performance testing to detect without knowing the triggering workload. This newly detected bug has been dormant since it was introduced in version 4.25 beta (released on 2005-08-01), and still remains in the latest version 9.22 (released on 2011-4-18). After we reported the bug at the forum [6], the developers confirmed the bug and plan to fix it in the next version.

For Notepad++, the performance problem of wrapping words in Scenario **S6** is confirmed as bug *#2909745* in the project's forum. Moreover, our approach further identifies that WrapLines causes performance bugs in Scenarios **S7**, **S8**, and **S10** after a document is modified. We are waiting for their responses for these not-yet-confirmed bugs. For the performance bug of opening a file, the developers confirmed that Notepad++ did not have good performance on large files, and stated that a patch could be applied to improve the performance [4]. We reported the new bug of finding a word in **S9**, and are still waiting for the response.

Table 2: Results of model inference and refinement

ID	# Ite.	# W.	E. I.	E. E.	M. I.	M. E.
(**S1**)	4	6	35.95	0.62	752	657
(**S2**)	4	6	62.31	0.47	1341	1283
(**S3**)	4	6	29.68	0.85	1234	1223
(**S4**)	4	6	4.71	0.20	1299	1267
(**S5**)	3	5	5.94	0.18	1630	1282
(**S6**)	6	8	536.74	8.62	742	1441
(**S7**)	5	7	455.00	7.69	789	1329
(**S8**)	7	9	17.12	5.51	448	1505
(**S9**)	4	6	138.36	1.83	1287	205
(**S10**)	7	9	7.38	1.86	324	296

7.3 RQ2: Model Inference and Refinement

To answer RQ2, we measure the improvement of model accuracies after the iterations of model refinement terminate, and the prediction accuracies of execution counts on large workloads.

Model Refinement. To show the improvement of model accuracies, we measure the average relative error of the inferred models with the initial workloads, measure the average relative error of the inferred models after the iterations of model refinement terminate, and compare the errors. Table 2 shows the results of RQ2. Column "ID" shows the ID of scenarios. Column "# Ite." shows the number of iterations used to refine the complexity models. Column "# W." shows the number of workloads used to infer the complexity models when the iterations terminate. Column "E. I." shows the average relative errors of the inferred complexity models on the initial workloads, and Column "E. E." shows the average relative errors of the inferred complexity models when the iterations terminate. Column "M. I." shows the number of the inferred complexity models on the initial workloads, and Column "M. E." shows the number of the inferred complexity models when the iterations terminate.

On average, it takes about 5 iterations (using 7 workloads) for ΔInfer to terminate, and improves the average relative error to 2.78%. From the results, we can see that the accuracies of the inferred models are significantly improved. For example, the average relative error of **S6** is improved from 536.74% to 8.62%. The results of Columns "M. I." and "M. E." indicate that certain locations that are incorrectly inferred as workload-dependent can be filtered out after iterative refinements. Different initial workloads may result in different initial accuracies. But with our workload-augmentation mechanism, these differences are not obvious

Table 3: Results of cost prediction

ID	10 (%)	20 (%)	50 (%)
(**S1**)	3.18	4.45	6.16
(**S2**)	2.98	4.07	5.55
(**S3**)	*1.40	*1.60	*1.86
(**S4**)	1.65	2.29	3.08
(**S5**)	1.58	2.19	2.95
(**Ave(7-Zip)**)	*2.35	*3.25	*4.44
(**S6**)	18.51	26.38	47.24
(**S7**)	16.84	22.56	36.28
(**S8**)	16.80	24.45	35.23
(**S9**)	11.15	15.97	39.09
(**S10**)	10.79	15.50	24.63
(**Ave(Notepad++)**)	14.82	20.97	36.49

after a few iterations, indicating that our approach is insensitive to the potential variance of intial workloads. In summary, the results show that our algorithm requires a reasonable number of iterations to achieve substantial improvement of model accuracy.

Cost Prediction. To show prediction accuracies on large workloads, we select the workload values that are 10, 20, and 50 times the upper bound of the RVR, obtain the profiles of these workloads, and compare the predicted execution counts and the actual execution counts to compute average relative errors. Table 3 shows the results of RQ2. Column "ID" shows the ID of scenarios. Columns "10", "20", and "50" show the average relative errors when the workloads are 10, 20, and 50 times the upper bound of the RVR. For example, for **S1**, we use the workload values $\{12800, 25600, 64000\}$.

On average, when the workload is 50 times of the upper bound of the RVR, the prediction error for the 7-Zip file manager (marked with *) is just 4.44% (excluding **S3**), and the prediction error for Notepad++ is acceptable (36.49%). For **S3** (marked with *), due to the quadratic workload-dependent loop, our profiler reaches its profiling limitations when the workload value exceeds 10,000. Thus, we use workload values $\{4000, 6000, 8000\}$ to estimate the prediction errors. The reason why Notepad++ has a relatively high prediction error is that the developers of Notepad++ optimize the message processing during idle time, causing certain workload-dependent loops to exhibit a bit different complexities under the same contexts. Such result shows that ΔInfer is robust even under such complex situations.

7.4 RQ3: Context-Sensitive Analysis

Existing approaches [16, 19] that infer complexity models using profiles of *multiple* workloads are context-insensitive. To show effectiveness of context-sensitive analysis and answer RQ3, we compare the number of complexity transitions identified by using context-sensitive analysis and context-insensitive analysis. We first apply ΔInfer to infer complexity transitions using the inferred complexity models. We then apply regression learnings on profiles collected to infer complexity models without calling context, and use these complexity models to infer another set of complexity transitions. We compare these two sets of complexity transitions to show the effectiveness of our context-sensitive analysis.

Table 4 shows the results of RQ3. Column "ID" shows the ID of scenarios. Column "ΔInfer" shows the number of complexity transitions inferred by ΔInfer, and Column "# L." shows the number of the inferred complexity transitions that are workload-dependent loops. Column "InSen." shows the number of complexity transitions inferred by the context-insensitive analysis, referred to as *ContextIns*. Column "# Miss." shows the number of complexity transitions inferred by ΔInfer but not ContextIns.

Table 4: Comparison to context-insensitive analysis

ID	ΔInfer	# L.	InSen.	# Miss.
(**S1**)	11	10	521	6
(**S2**)	21	19	579	12
(**S3**)	17	16	486	10
(**S4**)	21	19	640	12
(**S5**)	22	20	546	12
(**S6**)	10	10	509	3
(**S7**)	29	14	877	6
(**S8**)	10	10	526	5
(**S9**)	20	20	131	0
(**S10**)	12	11	861	3

The results show that ContextIns identifies much more (32.8 times on average) complexity transitions than ΔInfer, and more than 90% of them do not help identify workload-dependent loops, producing many false positives. Such result is mainly due to the complex contexts posed by the event-driven nature of GUI applications. Based on manual inspection, 86.1% of the complexity transitions inferred by ΔInfer are workload-dependent loops (including implicit loops), and 13.9% of them are false positives. Most of the false positives (15 in **S7**) are complexity transitions inside an internal string-allocator function of basic_string. The others involve top-level message handlers, such as WindowProcedure. These false positives can be reduced by excluding low-level and top-level system libraries in the analysis. Moreover, the results of Column "# Miss." show that ContextIns misses about 39.9% of the complexity transitions identified by ΔInfer. Based on manual inspection, some of these missing complexity transitions are real WDPBs that cause performance problems. Thus, ContextIns also produces false negatives. We also find that ΔInfer does not miss any WDPB detected by ContextIns. In summary, ΔInfer outperforms ContextIns in terms of greatly reducing false positives and false negatives.

8. THREATS TO VALIDITY

The threats to external validity include the representativeness of the subject applications and the chosen scenarios and focused workload parameters for our evaluations. The current subjects include GUI applications of two kinds of daily used productivity tools: file archivers and text editors. These two applications are popular open source applications from the SourceForge repository, and their bug databases and forums are actively maintained. This threat can be further reduced by more studies on more kinds of GUI applications. The threat to internal validity includes the human factor for determining whether an identified transition is a WDPB. To reduce this threat, we first searched their bug databases to see whether there are reported performance bugs for the WDPBs. We also posted the identified WDPBs on the forums of the subject applications, and got most of them confirmed by the developers.

9. DISCUSSION AND FUTURE WORK

Generalization to Other Types of Applications. Our current approach focuses on detecting WDPBs for GUI applications. However, our approach is applicable to any type of application that requires operations in responsive actions not to block subsequent operations, such as message-queue systems, event-driven servers, and chained filters. Our approach is also applicable to identify energy bugs [31] in mobile applications by using profiles of energy costs. Moreover, our approach can be used to assist tasks of performance analytics, such as predicting whether the execution time of complexity transitions exceeds the response requirement on a given workload or estimating a lower bound on the workload that causes the execution time of complexity transitions to violate the response requirement. We plan to investigate such applications in our future work.

Workload Parameters. Our current approach infers models by varying workload values on one focused workload parameter. To identify performance bottlenecks that require combinations of parameters [25], our approach can be used to infer multiple models by varying different parameters separately. Based on the order differences of the inferred models for different workload parameters, we can know that the execution count increases more quickly on which workload parameter; such information is sufficient for our approach to identify complexity transitions. Multi-dimensional models for multiple workload parameters are often in complex forms or even have no analytic form, making it difficult to uncover such models directly. Existing research [36] requires user-provided information for interdependencies of parameters to infer the models. By adapting the divide-and-conquer strategy to infer one model for one focused parameter, our approach reduces computational complexity without requiring information for the interdependencies of parameters, and yet preserves the effectiveness.

Value-Dependent Performance Bottlenecks. Some performance bottlenecks may depend on specific values of the input, instead of input workloads. Existing approaches [14] have explored the research of this direction, but have limitations due to the lack of runtime information. Moreover, there are other performance bottlenecks that may be triggered by combinations of configuration values as investigated by Hoffmann et al. [25]. These approaches can be leveraged to infer configurations for the scenarios, and our approach can be applied to automatically predict WDPBs afterwards.

Scalability of Scenario-Based Profiling. Our approach instruments an application under analysis and collects profiles for each scenario to detect scenario-specific WDPBs. Scenario-based instrumentation is widely adopted for testing/debugging, and the state of the practice is observed in many popular software products from leading software companies, e.g., PerfTrack [8] based on the Event Tracing for Windows (ETW) [3] platform from Microsoft. Based on such technologies, scenario-based tracing has been used as an automated and scalable solution for complex large-scale software products, e.g., Windows. Moreover, our approach can be fully automated with automatic GUI test scripts, reducing human efforts and improving scalability.

10. RELATED WORK

Model Inference using Multi-Profiles. Goldsmith et al. [19] propose an approach that fits performance measurements of clusters of basic blocks to workload sizes. Zaparanuks et al. [38] propose an approach that infers an empirical cost function of an application automatically, and Coppa et al. [16] propose an approach that measures the size of the input given to a generic code fragment. Unlike their model inference based on sorted or random inputs, our approach iteratively refines the inferred models based on the model accuracy from the previous iteration. Moreover, our approach uses context-sensitive analysis to address complex contexts in GUI applications. Westermann et al. [36] propose an approach to infer the prediction models between interdependent, performance-relevant configuration parameters and the performance metric of interest. All these approaches infer a single complexity model for a program, while our approach infers context-sensitive complexity models for locations inside a program, and identifies workload-dependent loops using complexity transitions.

Static Analysis. Chang et al. [14] propose an approach that combines taint analysis with control dependency analysis to detect high-complexity control structures, such as recursive calls and nested loops. Wang et al. [35] propose an approach that statically searches for the intersections of blocking and responsive invocations as the potential hang bugs based on patterns around method invocations. Approaches of purely static analysis face challenges in identifying implicit loops or resolving complex contexts in GUI applications, and in handling many runtime features, such as indirect method calls via function pointers.

Performance Analysis. Traditional performance analysis centers around analyzing the performance measurements obtained by profiling program executions, such as call-tree and call graph profiles [11]. There also exist approaches that assist searching [12] and summarizing [34] profiles to find performance problems. These approaches rely on manual efforts to explore traces or search to identify bottlenecks, while our approach infers WDPBs from multiple profiles.

Foo et al. [17] and Jiang et al. [27] propose approaches to learn signatures or baselines from previous runs, and then detect performance problems by comparing current runs against the derived performance signatures or baselines. Grechanik et al. [21] propose an approach that automatically clusters the input space into good and bad performance test cases, and drill down to the most significant methods to identify performance bottlenecks. Zhang et al. [39] propose a symbolic-execution-based approach to generate load tests for exposing performance bottlenecks. Our previous work [24] mines large-scale traces to identify performance bugs. All these approaches require WDPBs to surface on the analyzed executions, suffering from the insufficiency issue of identifying WDPBs. Nistor et al. [30] propose an approach to detect performance problems via identifying loops whose computation has similar memory-access patterns across loop iterations. Their approach relies on loop events for analysis, and thus cannot identify implicit loops without modelling UI libraries calls.

11. CONCLUSION

We have presented the ΔInfer approach that predicts WDPB loops under large workloads via inferring iteration counts of WDPB loops using complexity models for the workload size. ΔInfer incorporates the novel concept of *context-sensitive delta inference* that consists of two parts: *temporal inference* for inferring the complexity models of different program locations, and *spatial inference* for identifying WDPB loops as WDPB candidates. We conducted evaluations on ΔInfer with two popular open source GUI applications, 7-Zip and Notepad++. The results show that ΔInfer infers high-quality complexity models with iterative refinements, performs much better than the context-insensitive analysis, and effectively identifies highly impactful WDPBs that cause 10 performance bugs.

Acknowledgment

Tao Xie's work is supported in part by NSF grants CCF-0845272, CCF-0915400, CNS-0958235, CNS-1160603.

12. REFERENCES

[1] 7-Zip. http://www.7-zip.org/.

[2] ΔInfer. https://sites.google.com/site/asergrp/projects/deltainfer/.

[3] Event Tracing for Windows (ETW). http://msdn.microsoft.com/en-us/library/windows/desktop/bb968803(v=vs.85).aspx.

[4] https://sourceforge.net/projects/notepad-plus/forums/forum/331753/topic/5101818.

[5] https://sourceforge.net/projects/sevenzip/forums/forum/45797/topic/4941337.

[6] https://sourceforge.net/projects/sevenzip/forums/forum/45797/topic/4941343/.

[7] Notepad++. http://notepad-plus-plus.org/.

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

[9] Preventing Hangs in Windows Applications. http://msdn.microsoft.com/en-us/library/dd744765.

[10] SourceForge. http://sourceforge.net/.

[11] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proc. PLDI*, pages 85–96, 1997.

[12] G. Ammons, J. deok Choi, M. Gupta, and N. Swamy. Finding and removing performance bottlenecks in large systems. In *Proc. ECOOP*, pages 170–194, 2004.

[13] T. Ball, O. Kupferman, and M. Sagiv. Leaping loops in the presence of abstraction. In *Proc. CAV*, pages 491–503, 2007.

[14] R. Chang, G. Jiang, F. Ivancic, S. Sankaranarayanan, and V. Shmatikov. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *Proc. CSF*, pages 186–199, 2009.

[15] S. Chatterjee and A. Hadi. *Regression Analysis by Example*. Wiley series in probability and mathematical statistics. Wiley-Interscience, 2006.

[16] E. Coppa, C. Demetrescu, and I. Finocchi. Input-sensitive profiling. In *Proc. PLDI*, pages 89–98, 2012.

[17] K. C. Foo, Z. M. Jiang, B. Adams, A. E. Hassan, Y. Zou, and P. Flora. Mining performance regression testing repositories for automated performance analysis. In *Proc. QSIC*, pages 32–41, 2010.

[18] Q. Fu, J.-G. Lou, Q.-W. Lin, R. Ding, D. Zhang, Z. Ye, and T. Xie. Performance issue diagnosis for online service systems. In *Proc. SRDS*, pages 273–278, 2012.

[19] S. F. Goldsmith, A. S. Aiken, and D. S. Wilkerson. Measuring empirical computational complexity. In *Proc. ESEC/FSE*, pages 395–404, 2007.

[20] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *Proc. PLDI*, pages 120–126, 1982.

[21] M. Grechanik, C. Fu, and Q. Xie. Automatically finding performance problems with feedback-directed learning software testing. In *Proc. ICSE*, pages 156–166, 2012.

[22] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In *Proc. POPL*, pages 127–139, 2009.

[23] R. J. Hall. CPPROFJ: Aspect-capable call path profiling of multi-threaded Java applications. In *Proc. ASE*, pages 107–116, 2002.

[24] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie. Performance debugging in the large via mining millions of stack traces. In *Proc. ICSE*, pages 145–155, 2012.

[25] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic knobs for responsive power-aware computing. In *Proc. ASPLOS*, pages 199–212, 2011.

[26] R. Jain. The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling. Wiley professional computing, 1991.

[27] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automated performance analysis of load tests. In *Proc. ICSM*, pages 125–134, 2009.

[28] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *Proc. PLDI*, pages 77–88, 2012.

[29] I. Molyneaux. *The Art of Application Performance Testing - Help for Programmers and Quality Assurance*. O'Reilly Media, 2009.

[30] A. Nistor, L. Song, D. Marinov, and S. Lu. Toddler: detecting performance problems via similar memory-access patterns. In *Proc. ICSE*, pages 562–571, 2013.

[31] A. Pathak, Y. C. Hu, and M. Zhang. Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices. In *Proc. HotNets*, pages 5:1–5:6, 2011.

[32] H. Song, Y. Qi, X. Tian, and D. Xu. Navigating and visualizing long lists with fisheye view and graphical representation. In *Proc. DMAMH*, pages 123–128, 2007.

[33] X. Song, H. Chen, and B. Zang. Why software hangs and what can be done with it. In *Proc. DSN*, pages 311–316, 2010.

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

[35] X. Wang, Z. Guo, X. Liu, Z. Xu, H. Lin, X. Wang, and Z. Zhang. Hang analysis: fighting responsiveness bugs. In *Proc. EuroSys*, pages 177–190, 2008.

[36] D. Westermann, J. Happe, R. Krebs, and R. Farahbod. Automated inference of goal-oriented performance prediction functions. In *Proc. ASE*, pages 190–199, 2012.

[37] G. H. Xu, D. Yan, and A. Rountev. Static detection of loop-invariant data structures. In *Proc. ECOOP*, pages 738–763, 2012.

[38] D. Zaparanuks and M. Hauswirth. Algorithmic profiling. In *Proc. PLDI*, pages 67–76, 2012.

[39] P. Zhang, S. Elbaum, and M. B. Dwyer. Automatic generation of load tests. In *Proc. ASE*, pages 43–52, 2011.