SUNCAT: Helping Developers Understand and Predict Performance Problems in Smartphone Applications

Adrian Nistor  
Chapman University, USA  
anistor@chapman.edu

Lenin Ravindranath  
Massachusetts Institute of Technology, USA  
lenin@csail.mit.edu

ABSTRACT
The number of smartphones shipped in 2014 will be four times larger than the number of PCs. Compared to PCs, smartphones have limited computing resources, and smartphone applications are more prone to performance problems. Traditionally, developers use profilers to detect performance problems by running applications with relatively large inputs. Unfortunately, for smartphone applications, the developer cannot easily control the input, because smartphone applications interact heavily with the environment.

Given a run on a small input, how can a developer detect performance problems that would occur for a run with large input? We present SUNCAT, a novel technique that helps developers understand and predict performance problems in smartphone applications. The developer runs the application using a common input, typically small, and SUNCAT presents a prioritized list of repetition patterns that summarize the current run plus additional information to help the developer understand how these patterns may grow in the future runs with large inputs. We implemented SUNCAT for Windows Phone systems and used it to understand the performance characteristics of 29 usage scenarios in 5 popular applications. We found one performance problem that was confirmed and fixed, four problems that were confirmed, one confirmed problem that was a duplicate of an older report, and three more potential performance problems that developers agree may be improved.

Categories and Subject Descriptors
D.2.8 [Software Engineering]: Metrics—Performance measures;  
D.2.5 [Software Engineering]: Testing and Debugging

General Terms
Performance

Keywords
Predicting performance problems, smartphone applications

1. INTRODUCTION
Smartphones are extremely popular, and the number of smartphones shipped in 2014 is expected to be four times larger than the number of PCs [15]. Each of Apple’s and Google’s online App Stores has about 1.2 million applications [38], and each day 500 new applications are added to the Windows Phone Store [8]. Furthermore, smartphones are used primarily as devices for displaying data and not for making phone calls [11].

Compared to PCs, smartphones have very limited computing resources. Hence, even small code inefficiencies, that would go unnoticed when running on a typical PC, can create glitches and potentially long delays in a smartphone application. Most smartphone applications are intended to be highly interactive, and such glitches and delays create a negative user experience. For example, OneBusAway [3] is a popular smartphone application that displays bus traffic information for the Seattle area. The reviews of OneBusAway on Windows Phone Store are highly positive about its functionality, but one in every six reviews mentions performance, and 94% of the reviews mentioning performance are negative, i.e., only 6% of the reviews praise the current performance.

Traditionally, developers use profilers to detect performance problems [26, 40, 48]. The developer runs the code on a relatively large input and collects profile information, e.g., collects for each method the time taken for execution or the number of times the method was executed. The developer then focuses on optimizing the methods that were the most expensive. Unlike correctness bugs that can manifest for both small and large inputs, many performance problems only manifest for large inputs. Therefore, developers typically do not use small inputs for profiling.

A major challenge in testing many smartphone applications is that the developer cannot easily control the entire input because smartphone applications interact heavily with the environment. For example, OneBusAway communicates with a bus traffic server using a complex protocol, gets GPS data about the location, and uses current time. Modeling the entire environment—including protocol, GPS data, time—would be difficult. Therefore, system testing of smartphone applications is typically done with common, relatively small inputs, by manually performing GUI actions. For example, OneBusAway has automated tests only for unit testing; for system testing, developers can easily test OneBusAway with, say, 5 buses, but they cannot easily try it for a larger input with, say, 50 buses. However, such uncommon inputs can still happen in the actual use and can create performance problems.

What information can we extract from runs with small inputs to help developers understand how the application would behave for larger inputs? We develop SUNCAT, a specialized techniques that answers this question for smartphone applications, which are an increasingly important and popular domain.

The calling patterns of string getters help developers understand and predict performance in smartphone applications: Profiling all methods can give misleading results for small inputs. For example, for a previously unknown performance problem we found in OneBusAway, the methods related to the problem execute only
3-4 times for small inputs, while many other methods execute well over 4 times; however, for large inputs, the methods related to the problem can execute hundreds of times, creating performance problems. We propose to use only string getters—i.e., the methods that return values of string fields—both to enable effective profiling and to provide useful additional information.

The rationale for using string getters is twofold. First, the number of string getter calls is closely related with the computation being done and thus can help in correlating calls with performance. The reason is that many smartphone applications display and process text information (with video games being a notable exception). For example, a sequence of string getter calls for bus stop names reflects the computation that OneBusAway performs to save a bus route (which include bus stop names and a lot of other information); the more string getter calls there are, the more work OneBusAway needs to do. Even when displaying mainly non-text, smartphone applications process some text, e.g., to display the maps in figures 1(a) and 1(b), OneBusAway finds the bus stop names close to the current geographical location. Second, the return string values provide useful information for understanding the code. SUNCAT presents these strings to the developer, to help her relate the computation she is investigating with the information that she saw on the smartphone screen. For example, in OneBusAway, she can find that several calls were performed for the same bus stop that was viewed (rather than, say, for different nearby bus stops).

Prioritizing and understanding the repetition patterns observed in runs with small inputs: SUNCAT instruments an application to log events that consist of string getter calls and their return values. When the developer runs the application, SUNCAT records a trace. The main algorithmic contribution of SUNCAT is a novel algorithm for lossy grammar-based compression, which summarizes long and complex event traces in easy to understand execution overviews. SUNCAT presents to the user a Performance Summary that shows how groups of events repeat in the trace. Each repetition pattern shows repetition counts for events and how these counts vary in the trace. SUNCAT can prioritize the patterns using the maximum counts and/or count variations. Based on the expectations the developer has—e.g., that some computation should take constant time or have a small maximum count—the developer then chooses to investigate the most suspicious, unexpected patterns that are more likely to cause performance problems for large inputs.

The goal of SUNCAT is to help the developer reason about the performance problems that can manifest in the future runs, even if these problems did not manifest in the current run. Research shows that providing only a prioritized list of suspicious patterns is not enough to explain even correctness bugs that did manifest—as Parnin and Orso phrase it: “Programmers want values, overviews, and explanations” [37]—and more information is needed to understand performance problems [6, 7, 17, 19, 41, 42]. Therefore, in addition to the Prioritized Patterns, the Performance Summary presents a grammar that hierarchically groups related events. The developer can choose the level of detail presented in the grammar (e.g., Full Grammar or Summarized Grammar) and can also see the concrete string values from the performed run.

While computing repetition patterns from an entire trace provides useful information for predicting performance problems, our experience shows that it can be even more useful to compare how repetition patterns vary across multiple sub-traces within a trace. For example, in OneBusAway, one could find that the number of certain getter calls increases across sub-traces. Such sub-traces can be naturally obtained in many smartphone applications by repeating GUI actions. For example, OneBusAway displays a list of bus stops, and the developer can navigate to several of them.

Assessing the impact on user-perceived performance: Developers of smartphone applications are aware of the limited computing resources of smartphones and try to enhance the user-perceived performance [14]. For example, to enhance the responsiveness of the threads producing data immediately shown to the user, code has other threads that prefetch some data or perform some of the expensive but non-critical work in the background. Hence, while some computation may be truly expensive, the user need not notice any performance problem. As a result, the important performance problems in smartphone applications are only those that do affect the user-perceived performance. We find that a simple solution helps to determine such computations: SUNCAT inserts time delays in some string getters selected by the developer, so when she reruns the application, she can check if it becomes slow.

Experience with previously unknown performance problems in real-world smartphone applications: We used SUNCAT to understand 29 usage scenarios in 5 real-world Windows Phone applications: Conference [25], Pex4Phone [27], OneBusAway [3], TouchDevelop [5], and Subsonic [4]. These applications can be downloaded from the Windows Phone Store and are quite popular, e.g., OneBusAway and TouchDevelop each have more than 270 reviews on Windows Phone Store. We were unfamiliar with these applications before starting the study.

SUNCAT helped us understand these applications, and we found nine performance problems: one problem we reported was already confirmed and fixed by developers [34], four problems were confirmed by developers [25], one problem we found was a duplicate of an older confirmed performance problem [35], and three more problems we found developers labeled as cases that could be improved but are not a high priority. In brief, while SUNCAT is a technique specialized for smartphone applications, it showed very good results for this increasingly important domain.

2. EXAMPLE

We describe in more detail our running example—a test scenario in OneBusAway [3]—and a performance problem we found in it using SUNCAT. OneBusAway displays bus information such as stops, routes, schedules, and arrivals in screens with various formats and levels of detail. Figure 1 shows three sample screenshots. The AllStops screen shows a map with the bus stops (top of screen) and a list of these stops (bottom of screen). The OneStop screen displays information about buses arriving at the selected bus stop. The RecentViews screen shows recently viewed bus stops and routes. The user can navigate among these screens in several ways, so it is natural to test the interactivity of these navigations [19].

Suppose a developer wants to test the navigation between the AllStops and OneStop screens. The developer can double-tap a bus stop in the list on the AllStops screen and, consequently, OneBusAway displays the OneStop screen. The developer can now press
the application. Displaying
SUN
a real performance problem that would be difficult to detect with-
OneBusAway
In addition to C#, OneBusAway
lem is not in the (compiled) C# code on which such tools typically
cannot be directly used because the root of the performance prob-
happen for an end user but is less likely to happen during in-house
OneStop
likely to manifest in regular testing. The developer (with or with-
of type
small
effect the user-perceived performance, we instructed S
during the run (Section 4.3.1). To check if this increase would af-
the developer can visit several
screens, it creates a C# object (
DetailsPage
of the input is quite small (e.g., the list has only a few bus stops).
We found that
OneBusAway
has a leak where the number of
get_name
other events, e.g., network calls. Specifi-
can add time delays in the application to allow the user to
SUNCAT can add time delays in the application to allow the user to
3.1 Logging
SUNCAT by default logs only calls to string getters, although one
could also specify other events, e.g., network calls. Specifi-
call log: (1) the call stack (as in calling
current con- text profiling [13]) and (2) the return string value. For example,
in OneBusAway, one of the getters is get_name for the field
CurrentViewState.CurrentStop.name. Figure 4 shows an
example concrete event a, for one call to get_name with the example string
("NE 65TH ST & OSWEGO...").
SUNCAT uses off-the-shelf binary instrumentation of .NET to
add logging methods to each string getter. Note that SUNCAT
SUNCAT instruments the method body (effectively the callee), which is in the
smartphone application binary itself, rather than the call site (in the
caller), which may be elsewhere. For example, for get_name, the
caller is in the Windows Phone runtime, making calls through re-
flection, based on the XAML file. For each update to the Details-
Page object, the runtime calls get_name to update the RouteInfo
field, and SUNCAT logs these calls to get_name as concrete events.
To illustrate an example trace, assume the developer navigates to
the OneStop screen when there are 4 DetailsPage objects in the
system; the runtime calls get_name 4 times to perform 4 updates to
the RouteInfo fields. Figure 4 shows the concrete event trace with
4 events a, corresponding to 4 calls to get_name, and several
other events, corresponding to the calls to other string getters.
3.1.1 Comparison Mode
While computing repetition counts from one trace can provide
useful information for predicting performance problems, our expe-
rience shows that it can be even more useful to compare how re-
petition counts vary across multiple sub-traces. For example, while it
helps to know there are 4 get_name calls for the current OneStop
screen, it helps even more to know that the number grows (4, 5,
3.2 Performance Summaries

SUNCAT abstracts each concrete event to an abstract event by ignoring the return string value and considering only the call stack. (In general, other abstractions could be used, e.g., taking only top $N$ entries from the stack [7].) However, SUNCAT still allows the user to inspect concrete string values because many of them help in understanding the application. For example, in OneBusAway, bus stop names (e.g., NE 65TH ST & OSPEGO in the concrete event $a_1$ in Figure 4) are easy to understand and relate with the input because they show on the phone screen. Figure 4 shows a simple abstract trace. Figure 5(a) shows an abstract trace for an example run of OneBusAway when navigating to four OneStop screens. The symbol ‘$\cdot$’ shows the points in the sub-trace that correspond to the four different screens in the Comparison Mode.

Given an abstract trace, SUNCAT computes a Performance Summary that summarizes the repetition patterns in the trace. The goal is to help developers understand how the execution cost may evolve for larger inputs. To achieve this, SUNCAT can count the number of event occurrences, even non-consecutive ones, or can provide additional information by hierarchically grouping related events and counting consecutive occurrences. The SUNCAT user can summarize and prioritize these patterns in various ways to determine which repetition patterns are likely to create performance problems.

Figure 5(c) shows several kinds of summaries for the example abstract trace from Figure 5(a). The simplest summary is Count Summary, which just counts the number of events in the entire trace. The core of the advanced summary is the Full Grammar, which is a context-free grammar (Section 3.2.1) obtained by a novel lossy grammar-based compression of the trace (Section 3.2.2). The previous grammar-based compression algorithms [20,22,23,31,46] were lossless: their goal was to compress a string into a grammar that can generate only one string, but that results in large and hard to read grammars. In contrast, our goal is a short and intuitive summary of execution, so we allow grammars that can generate many strings. The user can further omit some details from Full Grammar to obtain a smaller Summarized Grammar and can prioritize the terms from the grammar to obtain the list of Prioritized Patterns.

These forms show different levels of detail that allow the developer to “zoom in/out” while inspecting the patterns. For all levels, the developer can follow the abstract events to the call stack and concrete events in the concrete traces (the right-most part of Figure 5(c)).

Both Count Summary and grammars are much more succinct and easier to inspect than full traces. While grammars are longer than Count Summary, they offer several additional pieces of information. First, they group symbols together, effectively providing a context for understanding correlated events. Second, they preserve ordering among events. Third, they provide a hierarchical organization of events, e.g., in Figure 5(c), $g$ and $b$ are together in $B$ which is nested in $A$ which is repeated in $S$. Fourth, they provide a trend that shows how repetition counts vary during execution.

$$\text{<Grammar> ::= \{\text{<Rule> \"n\"} \} \text{<Rule>}
        \text{<Rule> ::= \{\text{<Nonterm> \"\rightarrow\"} \text{<Items} \} \text{<Items>}
        \text{<Items> ::= \{\text{<RepPattern} \| \text{<Ignore>\}} \text{<Ignore>}
        \text{<RepPattern> ::= \{\text{<Nonterm>\{\text{<Terminal>\[\"\rightarrow\"}\text{<RepCount}\} \text{<RepCount>}
        \text{<RepCount> ::= \{\text{<DetailedList> \| \text{<MaxVal>\}} \text{<MaxVal>}
        \text{<DetailedList> ::= \{\text{<nume> \[\"\rightarrow\"}\text{\nume}\} \text{\nume>}
        \text{<MaxVal> ::= \{\text{\nume}\[\"\rightarrow\"}\text{\nume}\}
        \text{\Terminal> ::= ? \text{abstract event ?}}$$

Figure 6: EBNF for grammars produced by SUNCAT

3.2.1 Grammars

Figure 6 shows the full meta-grammar for the grammars that SUNCAT computes. It is best explained on an example, so consider the Summarized Grammar from Figure 5. The uppercase letters are non-terminals, and lowercase letters are terminals that correspond to abstract events. $A^4$ is shorthand for $AAAA$. $a^{151617}$ denotes that $a$ can repeat 4, 5, 6, or 7 times. $c^{\infty}$ denotes that $c$ can repeat up to 6 times. ‘...’ denotes ignored unimportant details, e.g., the user can decide to ignore repetition counts smaller than 3, and the rules for $B$ and $C$ are omitted because their right-hand side is only ‘...’.

3.2.2 Computing the Performance Summary

SUNCAT performs lossy grammar-based compression on the abstract event trace. We modify a known lossless algorithm [46] with the goal to produce more compact grammars. Figure 7 shows the pseudo-code of our algorithm. The input to main is a sequence of terminals, and the output is a context-free grammar that can generate this input string and many other strings (for our lossy compression) or only the input string (for the lossless compression). The algorithm maintains a map rules that is used to create the rules for the output grammar. To create the starting rule for the output grammar, main computes for each sub-trace a sequence of repetition patterns—where each repetition pattern RP is a symbol (terminal or non-terminal) with its list of repetition counts—and appends these repetition patterns. For example, the rule $S \rightarrow A^4$ in Figure 5 comes from merging four sequences, each being $A$, of four sub-traces. Note that the non-terminal symbols from rules are reused across sub-traces, so in this example all repetition counts come from merging across sub-traces. However, in general, repetition counts can come from within one sub-trace (e.g., $abbaabba$ would be $A^2$ with $A \rightarrow ab^3$) due to our algorithm being lossy.

The computeRPList method takes the string for each sub-trace and proceeds as follows. For increasing substring length (lines 13–20), it finds repeated substrings that are adjacent (lines 15–18), merges them to represent repetition (lines 22–30) and then starts again from the beginning (line 31). The process repeats until there are no more repeated adjacent substrings (line 13). Along the way, our algorithm attaches to each symbol (terminal or non-terminal) a list of repetition counts.

A crucial part of our algorithm is in lines 15 and 28: when deciding whether to merge two adjacent substrings, the algorithm ignores the repetition counts, e.g., it allows merging $ab^3b^5a^3b^3$. The method until returns the underlying sequence of symbols, effectively $abab$ in this example (and even the lossless algorithm would merge $abab$). To merge the repeated adjacent substrings of length greater than 1 (lines 27, 28), a non-terminal is used, and the repetition counts are merged from the corresponding lists for each symbol, e.g., $a^3b^5a^4b^3$ is merged into $A^7$ where $A \rightarrow a^{348}b^{313}$. This is lossy because the resulting expression encodes all (16) strings of form $a^{n_1}b^{n_2}a^{n_3}b^{n_4}$, where $n_1, n_2 \in \{3, 4\}$ and $n_3, n_4 \in \{5, 3\}$. The method merge preserves the order of repetition counts but removes the duplicates (so that the resulting lists are not too long).
3.2.3 Summarizing and Prioritizing Patterns

After SUNCAT computes the entire grammar, the user can modify it in various ways (e.g., replace some sequences with "..." or replace repetition counts with the maximum values) and can build different prioritization lists for inspection. By default, SUNCAT replaces all (non-constant) repetition counts with the maximum values, e.g., \(c^{6131014}\) with \(c^{56}\). The user can sort the patterns based on whether/how they vary (constant, increasing, varying) and what their maximum values are. SUNCAT does not automatically predict which patterns are more likely to lead to performance problems in the future runs. Indeed, many patterns naturally grow as a program input gets larger, e.g., we expect more computation to process 500 bus stops than 50 bus stops than 5 bus stops. Not every large/growing/varying pattern indicates a performance problem.

However, the user often has some expectation for how the patterns should vary and can look for the most suspicious patterns that violate this expectation. Consider, for example, two hypothetical patterns \(p^{32}3^8\) and \(Q^{28}\). If the user expects computation to take constant time, then \(p\) is more suspicious than \(Q\); \(p\) varies, so it may repeat many more times, while \(Q\) seems to always repeat a constant number of times. In contrast, if the user expects the maximum value to be small (e.g., the phone screen showed a small number of elements), then \(Q\) is more suspicious than \(p\). Likewise, patterns with a monotonically increasing vs. varying number of repetitions may be more or less suspicious, based on the expectation.

3.3 User-Perceived Performance

An important problem in evaluating suspicious patterns and methods is to establish whether they affect user-perceived performance. Recall that smartphone applications hide the latency of expensive computation, e.g., in OneBusAway some threads prefetch data from the server and store it in local memory. Statically determining whether a method is on a critical path for GUI is extremely hard because smartphone applications use many asynchronous event handlers, making it hard even to statically build a call graph. Using large inputs to dynamically evaluate problems is not an option.

SUNCAT helps the developer to use common, small inputs to decide which expensive patterns could impact the user-perceived performance for larger inputs. After the developer identifies a set of suspicious methods (e.g., \(get\_name\) in our example), SUNCAT can instrument the application binary to insert time delays only in some locations where it previously inserted logging methods for the events. The developer then runs the modified application binary for a similar input as the original run (e.g., navigating among
AllStops screen and a small number of OneStop screens as described in Section 2). If the delays are noticeable, the developer decides that the selected methods can indeed create problems.

Figure 7: Lossy grammar-based compression

4. EVALUATION

We implemented SUNCAT following the description from Section 3. For logging and inserting delays, we used CCI [2], an off-the-shelf static binary rewriter for .NET bytecode. SUNCAT can work both on the Windows Phone simulator and on real phones.

4.1 Applications

We used SUNCAT to understand the performance characteristics of 29 usage scenarios in five real-world applications (Figure 8). We selected these applications from the applications developed at Microsoft Research and from popular applications hosted on Microsoft’s CodePlex web site [1]. We needed to have the source code available; while SUNCAT instruments bytecode, to decide if the problem detected is real, we needed to double check with the source code. We did not know in advance if these applications have performance problems or not. Also, we were not familiar with any of these applications before this study and did not contact the developers until after we found the performance problems.

Conference
Microsoft
0
61
3,271
Pez4Phone
Microsoft
80
40
3,355
OneBusAway
Open Src
272
152
4,107
TouchDevelop
Microsoft
332
1,005
>14,753
Subsonic
Open Src
n/a
252
6,105

Figure 8: Programs used in evaluation. #Rev is the number of reviews on Windows Phone Store; #Class and LOC are the numbers of classes and lines of code.
4.3 Performance Problems Found

For ten usage scenarios from Figure 9 (column ‘PP?’) we found nine unique performance problems (OneBusAway #4 and OneBusAway #5 exposed the same problem), eight of which were previously unknown. One problem we reported was already confirmed and fixed (OneBusAway #4/OneBusAway #5) [34], four problems were confirmed (Conference #1, Conference #2, Conference #3, Conference #4), one problem we found was a duplicate of an older confirmed performance problem (OneBusAway #1) [35], and three more problems we found developers labeled as cases where performance could be improved but is not a high priority (Pex4Phone #6, TouchDevelop #4, TouchDevelop #7).

We describe our experience in using SUNCAT to identify these performance problems. Since Performance Summaries generated by SUNCAT are a new type of information for understanding performance, our presentation follows the style of papers that present new information for profiling [6, 17, 19, 41, 42, 47]. Namely, we present key steps in the process of analyzing the summaries, i.e., navigating among the grammar, terminals, non-terminals, call stacks, strings, and code to understand the potential performance problems. To make our presentation specific, we present for several test scenarios (1) a concrete run, (2) sample key steps in the process of using SUNCAT, and (3) the problem description.

### 4.3.1 OneBusAway #4

#### Concrete Run: We describe in more detail our experience with the running example (Section 2), the OneBusAway #4 test scenario. The number of events and the repetition counts in Figure 9 are larger than in Figure 5 because the abstract trace in Figure 5 is shortened for clarity. We first used SUNCAT to instrument the OneBusAway application for logging and opened the instrumented OneBusAway. We ran several other scenarios before navigating to the AllStops screen (Figure 1(a)). In this run, the AllStops screen showed bus stops NE 65TH ST & OSWEGO, NE 65TH ST & NE RAVE, NE RAVENNA BLVD & I, NE 65TH ST & ROOSEVE, and several more. For each bus stop, one can see detailed information by tapping the respective stop and getting to the OneStop screen (Figure 1(b)). We started the SUNCAT logging and tapped the first bus stop. While OneBusAway is navigating to the OneStop screen, SUNCAT logs concrete events. After the screen was displayed, we stopped the SUNCAT logging and pushed the back button, which brought us back to the AllStops screen. We next visited the other three stops, repeating the same process: start the instrumentation, tap the stop, wait for the OneStop screen to be displayed, stop the instrumentation, and press the back button. After navigating to these four OneStop screens, we obtained an event trace.

#### Inspection Step (Identify the Pattern to Explore): Figure 5 (Section 3) shows the various pieces of information that SUNCAT gen-

---

**Figure 9:** Performance characteristics. “R?” = is the action in Scenario Description repeated? “PP?” = was a performance problem found? “Ev.” = number of concrete events. “#CS” = number of terminals in Count Summary. “Str” = number of patterns in the Summarized Grammar for which strings were used. “T(s)” = time to process the trace (in seconds).

<table>
<thead>
<tr>
<th>Test Scenario</th>
<th>Description: Screen. Action</th>
<th>R?</th>
<th>PP?</th>
<th>Ev.</th>
<th>#CS</th>
<th>Number of patterns and top 3 patterns</th>
<th>Str</th>
<th>T(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conference #1</td>
<td>Authors. Go to OneAuthor</td>
<td></td>
<td>✓</td>
<td>1489</td>
<td>6</td>
<td>2 b^203 c^5 A^21</td>
<td>2.1</td>
<td>1.1</td>
</tr>
<tr>
<td>Conference #2</td>
<td>Session. Back to Main</td>
<td></td>
<td>✓</td>
<td>8912</td>
<td>10</td>
<td>4 h^718 c^316 A^21</td>
<td>4.7</td>
<td>7.1</td>
</tr>
<tr>
<td>Conference #3</td>
<td>Authors. Tap letter &amp; pop-up</td>
<td></td>
<td>✓</td>
<td>15580</td>
<td>8</td>
<td>3 c^20 A^279 B^1279</td>
<td>3.5</td>
<td>14.0</td>
</tr>
<tr>
<td>Conference #4</td>
<td>Session. Go to OnePaper</td>
<td></td>
<td>✓</td>
<td>1344</td>
<td>4</td>
<td>2 c^444 A^21</td>
<td>1.2</td>
<td>2.1</td>
</tr>
<tr>
<td>Pex4Phone #1</td>
<td>Main. Go to Leaderboards</td>
<td>N</td>
<td></td>
<td>303</td>
<td>3</td>
<td>1 A^13</td>
<td>0</td>
<td>0.5</td>
</tr>
<tr>
<td>Pex4Phone #2</td>
<td>Main. Go to Play</td>
<td>N</td>
<td></td>
<td>1287</td>
<td>12</td>
<td>2 C^21</td>
<td>2</td>
<td>2.7</td>
</tr>
<tr>
<td>Pex4Phone #3</td>
<td>Play, navigate Away. Back to Play</td>
<td>N</td>
<td></td>
<td>395</td>
<td>8</td>
<td>5 C^21</td>
<td>5.5</td>
<td>0.5</td>
</tr>
<tr>
<td>Pex4Phone #4</td>
<td>Main. Go to Learn</td>
<td>N</td>
<td></td>
<td>175</td>
<td>10</td>
<td>2 A^17</td>
<td>2</td>
<td>0.2</td>
</tr>
<tr>
<td>Pex4Phone #5</td>
<td>Learn, navigate Away. Back to Learn</td>
<td>N</td>
<td></td>
<td>280</td>
<td>10</td>
<td>3 A^17</td>
<td>3</td>
<td>0.4</td>
</tr>
<tr>
<td>Pex4Phone #6</td>
<td>Learn. Go to OneCourse</td>
<td>Y</td>
<td>✓</td>
<td>320</td>
<td>14</td>
<td>4 C^4</td>
<td>4.5</td>
<td>0.5</td>
</tr>
<tr>
<td>Pex4Phone #7</td>
<td>Play. Go to Training</td>
<td>N</td>
<td></td>
<td>156</td>
<td>8</td>
<td>2 B^15</td>
<td>2</td>
<td>0.2</td>
</tr>
<tr>
<td>OneBusAway #1</td>
<td>All Routes. Go to OneRoute</td>
<td>Y</td>
<td>✓</td>
<td>2481</td>
<td>24</td>
<td>8 C^82</td>
<td>7</td>
<td>118.5</td>
</tr>
<tr>
<td>OneBusAway #2</td>
<td>One Route. No action</td>
<td>N</td>
<td></td>
<td>1302</td>
<td>6</td>
<td>3 a^12</td>
<td>13</td>
<td>1.0</td>
</tr>
<tr>
<td>OneBusAway #3</td>
<td>All Stops. Go to the map</td>
<td>N</td>
<td></td>
<td>179</td>
<td>5</td>
<td>2 e^62</td>
<td>6</td>
<td>0.3</td>
</tr>
<tr>
<td>OneBusAway #4</td>
<td>All Stops. Go to OneStop</td>
<td>Y</td>
<td>✓</td>
<td>312</td>
<td>13</td>
<td>5 e^8</td>
<td>0.5</td>
<td>0.8</td>
</tr>
<tr>
<td>OneBusAway #5</td>
<td>Recent Views. Go to OneStop</td>
<td>Y</td>
<td>✓</td>
<td>362</td>
<td>9</td>
<td>5 a^28</td>
<td>5.9</td>
<td>0.9</td>
</tr>
<tr>
<td>OneBusAway #6</td>
<td>Main. Go to AllStops</td>
<td>N</td>
<td></td>
<td>88</td>
<td>3</td>
<td>5 a^5</td>
<td>5</td>
<td>0.1</td>
</tr>
<tr>
<td>TouchDevelop #1</td>
<td>Welcome. Go to Main</td>
<td>N</td>
<td></td>
<td>339</td>
<td>18</td>
<td>8 r^66</td>
<td>6</td>
<td>1.6</td>
</tr>
<tr>
<td>TouchDevelop #2</td>
<td>Tile. Go to Script</td>
<td>N</td>
<td></td>
<td>161</td>
<td>7</td>
<td>1 A^73</td>
<td>0.2</td>
<td>1.0</td>
</tr>
<tr>
<td>TouchDevelop #3</td>
<td>Script. Go to Tile and return</td>
<td>Y</td>
<td></td>
<td>645</td>
<td>15</td>
<td>5 e^9</td>
<td>1</td>
<td>0.3</td>
</tr>
<tr>
<td>TouchDevelop #4</td>
<td>Scripts. Create new script</td>
<td>Y</td>
<td>✓</td>
<td>5497</td>
<td>10</td>
<td>12 h^352</td>
<td>3</td>
<td>6.9</td>
</tr>
<tr>
<td>TouchDevelop #5</td>
<td>New Script. Go to MainScript</td>
<td>N</td>
<td></td>
<td>38</td>
<td>3</td>
<td>1 a^6</td>
<td>1.0</td>
<td>0.1</td>
</tr>
<tr>
<td>TouchDevelop #6</td>
<td>Main Script. Create new IF stmt.</td>
<td>Y</td>
<td></td>
<td>21</td>
<td>3</td>
<td>2 a^7</td>
<td>0.5</td>
<td>0.1</td>
</tr>
<tr>
<td>TouchDevelop #7</td>
<td>IF stmt. wizard. Create IF stmt.</td>
<td>Y</td>
<td>✓</td>
<td>478</td>
<td>25</td>
<td>10 c^66</td>
<td>9</td>
<td>6.2</td>
</tr>
<tr>
<td>Subsonic #1</td>
<td>Music Library. Tap group letter</td>
<td>N</td>
<td></td>
<td>18</td>
<td>1</td>
<td>2 a^10</td>
<td>2</td>
<td>0.1</td>
</tr>
<tr>
<td>Subsonic #2</td>
<td>Music Library. Go to OneArtist</td>
<td>N</td>
<td></td>
<td>149</td>
<td>19</td>
<td>2 C^14</td>
<td>1.0</td>
<td>0.5</td>
</tr>
<tr>
<td>Subsonic #3</td>
<td>Artist. Go to OneAlbum</td>
<td>N</td>
<td></td>
<td>280</td>
<td>30</td>
<td>3 D^11</td>
<td>1</td>
<td>0.3</td>
</tr>
<tr>
<td>Subsonic #4</td>
<td>Music Library. Go to Nearest</td>
<td>N</td>
<td></td>
<td>183</td>
<td>17</td>
<td>2 A^10</td>
<td>2</td>
<td>0.6</td>
</tr>
<tr>
<td>Subsonic #5</td>
<td>Nearest. Go to OneStop</td>
<td>N</td>
<td></td>
<td>77</td>
<td>35</td>
<td>5 B^10</td>
<td>3</td>
<td>0.7</td>
</tr>
<tr>
<td><strong>OVERALL</strong></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td>43X</td>
<td>121X</td>
<td>84%</td>
<td>7.1</td>
</tr>
</tbody>
</table>
erates for this trace. From Prioritized Patterns and Summarized Grammar, several patterns stand out: a4151617, e314, and a314 are especially interesting, because their repetition counts increase, suggesting that they may increase even further (e.g., 8, 9, 10... for a, or 5, 6, 7... for e); c6131614 is interesting because its repetition count varies (increases and decreases), suggesting that other values may be possible (say, 20).

Inspection Step (Understand a Terminal Symbol from String Values): Since the pattern for a has both increasing and the largest values, we wanted to understand what it represents and if growing its repetition counts can lead to a performance problem. Looking at the top of the call stack for a, we find on eBusAway.WP7.ViewModel.BusServiceDataStructures.Stop.get_name().i.e., a is a call to the string getter for a bus stop name. Hence, a10 represents an iteration over n bus stops. To understand what this iteration represents, we inspect the strings corresponding to a (Figure 5). One would expect these strings for bus stop names to be related in some way, e.g., be part of the same bus route or be close to our current location. However, we were surprised to see one a10 pattern iterate n times over the same bus stop, i.e., one string (NE 65TH ST & OSWEGO) repeated 4 times, another string (NE 65TH ST & NE RAVE) repeated 5 times, the third string (NE RAVENNA BLVD & 1) repeated 6 times, and the fourth string (NE 65TH ST & ROOSEVE) repeated 7 times. We immediately noticed these are the bus stops for which we had opened the OneStop screens.

Inspection Step (Understand a Terminal Symbol from Call Stacks and Code): Having inspected the string values for a, we look at its full call stack. All the stack frames, except the get_name itself, are from the Windows Phone runtime, some from the reflection classes. We deduced that the runtime invoked get_name due to some data-binding update (discussed in Section 2). The method names on the call stack do not show the reason for the data-binding update: where it is declared or what action triggered it. We used an automatic search of all OneBusAway project files for references to the OneBusAway.WP7.ViewModel.BusServiceDataStructures.Stop class and the name field, and we found the XAML declaration shown in Figure 2.

Problem Description: Putting all this together, we concluded that navigating to OneStop screens triggers more and more data-binding updates for the current page to which the user navigates. Since the number of these updates seems to grow without limit, after enough time, there will be a very large number of updates.

Inspection Step (Run SUCAT with Delays): It is not obvious from the application source if these updates are on the critical path for the user (or performed in some background thread). To determine how repeated updates affect the user-perceived performance, we instructed SUCAT to insert delays in get_name. We then reran the same test scenario, navigating from the AllStops screen to OneStop screen. We saw that AllStops indeed persists for some time before OneStop is displayed.

4.3.2 OneBusAway #1

Concrete Run: The previous scenario navigates among screens for bus stops, and this scenario navigates among screens for bus routes. In our example run with SUCAT, the AllRoutes screen showed several routes, and we navigated to four OneRoute screens.

Inspection Step (Identify the Pattern to Explore): Figure 10 shows the simplified Summarized Grammar for this run. c54119182166 stands out with the largest maximum values (even larger than the parts ignored in ...). For each non-terminal, such as C, we can choose to explore the context in which it appears in the grammar (on the right-hand side of another rule) or the sequence that it represents (what its right-hand side is). Indeed, as discussed in Section 3.2, a key benefit of grammars is that they provide context and hierarchical organization of symbols.

Inspection Step (Understand Where a Non-Terminal Appears): We first looked in what context C appears and find it in the rule for B. We then find B in the rule for A, with an increasing repetition count. A itself repeats four times in the rule for B due to the Comparison Mode, corresponding to our navigation to four bus routes. It appears that, inside A, B could continue increasing from 4 to 5, 6, 7, etc., and inside B, C varied seemingly randomly.

Inspection Step (Understand What a Non-Terminal Represents): We next wanted to understand what C represents, and if it can grow to large numbers. Since C maps to three terminals, we look at their call stacks in the Cumulative Event Info. We find on top the getters get_direction, get_id, and get_name from the structure oneBusAway.WP7.ViewModel.BusServiceDataStructures.Stop, i.e., these string getters correspond to a bus stop direction, ID, and name. Since these getters are adjacent in C, we know that they are always executed together, so we deduce that C11 represents an iteration over n bus stops. To understand what this iteration represents, we inspected the string values corresponding to the terminal c. We chose to look at c rather than a or b because get_name promises to give more information than get_direction or get_id. From the strings for the bus stop names, it stands out they start with 5TH AVE NE & NE followed by some different numbers, which suggests that these bus stops are consecutive, like a bus route along the 5th Ave.

Inspection Step (Understand a Terminal Symbol from Call Stacks and Code): Having inspected the string values for c, we looked at its full call stack. It has calls of methods from a .NET serialization class and then a call of the method WriteFavoritesToDisk in OneBusAway.WP7.ViewModel.BusServiceDataStructures.Stop, i.e., these string getters correspond to a bus stop direction, ID, and name. Since these getters are adjacent in C, we know that they are always executed together, so we deduce that C11 represents an iteration over n bus stops. To understand what this iteration represents, we inspected the string values corresponding to the terminal c. We chose to look at c rather than a or b because get_name promises to give more information than get_direction or get_id. From the strings for the bus stop names, it stands out they start with 5TH AVE NE & NE followed by some different numbers, which suggests that these bus stops are consecutive, like a bus route along the 5th Ave.

Problem Description: Putting all this together, we concluded that OneBusAway saves to disk entire routes with all their bus stops. By itself, this serialization can become slow if a route has many stops; from c54119182166, we see that some routes have as few as 19 stops while others have as many as 82. Moreover, the number of serializations grows over time, as shown by the b121314 pattern, so even if serializing any one given route is not slow, there is an increasing number of routes to serialize. Looking at the code, we confirm that OneBusAway keeps a list of the most recently viewed bus routes; the b121314 pattern comes from the fact that the code saves the entire list for each navigation, and the list can grow with each new navigation. Without a grammar, it would be much harder to know there is a nesting of repetitions for B and C; the Count Summary would only show a503 (or at best a54119182166 in the Comparison Mode), and likewise for b and c, so it would not be clear there is a growing list that can create a performance problem.
4.3.3 Conference #1

Concrete Run: We used the ICSE 2011 data to test Conference. In the Conference #1 scenario, we first navigate to the Authors screen that lists all paper authors. Tapping an author shows an AuthorInfo screen with the details such as institution and the list of papers in ICSE 2011. We used SUNCAT in the Comparison Mode and visited four authors, randomly scrolling down the list (ending up with Bae, Baysal, Brand, and Claessen).

Inspection: Figure 11 shows the simplified grammar for this run; the patterns \( B^{-4}\) and \( C^{-3}\) stand out as their repetition counts vary, but B has larger variations. We inspected B by looking at the call stacks and string values for the terminals a, b, and c, similarly as described for OneBusAway #1. We found that B represents an author, and the strings involved in the \( B^{-2}\) repetition showed it iterated over the author names, stopping at the author that we navigated to. (While we randomly scrolled down the list, we would find the same even if we scrolled up the list.)

Problem Description: We suspected that the model data structures did a linear search for the author. Indeed, looking at the code, we found that this repetition is in a LINQ query\(^1\) that searched for the name of the author in a list of objects (People). This repetition grows as the index of author grows. The solution to this problem is to use a dictionary instead of a list.

4.3.4 TouchDevelop #7

Concrete Run: We ran TouchDevelop instrumented for SUNCAT logging and first navigated to the Scripts list. We pushed the NewScript button and from a list of statement types selected if/then/else. At this point, a wizard appears that displays all the available options for constructing an if/then/else statement. We simply stopped SUNCAT logging without selecting any information for then and else branches.

\(^1\)LINQ is a declarative SQL-like language integrated into C#. Figure 11 shows an example query that finds the first Person in the list with the matching name.
individual method calls, grammars helped to understand grouping of method calls, and time delays helped to understand the effect on user-perceived performance.

For example, for TouchDevelop #1 and the repetition patterns g^{23}, e^{24}, and c^{5}, we find that even with the delays the application runs normally, without any noticeable slowdown. This means that when these repetitions increase, the user will not necessarily observe any performance problems. The reason is that the computation is performed in a background thread, and the main thread does not wait for the computation to finish (because the computation saves some state, which is not critical for the regular user actions). In contrast, delays in \( t^{46} \) show that this pattern may impact user-perceived performance.

Note that some of these 19 scenarios could also create performance problems, but they would affect the user-perceived performance only in unusual or unrealistic cases with very large inputs. For example, while a conference can have 1279 authors, it would be highly unusual for a paper to have 1279 authors. Similarly, the number of programming constructs in a scripting language such as used in TouchDevelop cannot become very large. Hence, such performance problems are unlikely to be fixed because the complexity involved in modifying the code may not be warranted by how frequently the end users experience performance problems.

4.5 Computing the Performance Summary

Figure 9 shows specific quantitative results for one sample trace per each test scenario. We tabulate the number of events in the trace, the number of terminals (with repetition counts larger than two) in Count Summary, the number of repetition patterns in Summarized Grammar computed using our lossy compression algorithm, and the top three patterns in the Prioritized Patterns. On average (geometric mean), a Count Summary has 43X fewer elements and a Summarized Grammar has 121X fewer patterns than a trace has events, which illustrates the compression achieved by encoding traces into patterns. For two scenarios (OneBusAway #1 and #5), our current S\textsc{unCat} implementation does not automatically infer some patterns due to noise in the trace, but the patterns are easily seen in the Full Grammar and Summarized Grammar.

The column \( T(s) \) shows the time that S\textsc{unCat} took to process the traces. We ran the experiments on an Intel Xeon 2.8 GHz desktop with 6 GB of memory running Windows 7. Most experiments finish in under 30 seconds, which means developers can easily run S\textsc{unCat} interactively, during the development and testing process.

4.6 Using String Values

Figure 9 also shows for how many repetition patterns we found the string values returned by getters easy to understand. Some strings (e.g., bus stop names or author names) were quite clear to us even though we did not develop the application and only used it, but some other strings were fairly cryptic to us, although the application developer would probably understand them much easier. For example, it was initially not obvious to us what the string \( 1\_23580 \) represents in OneBus\textsc{away}, but we realized later that strings starting with \( 1\_ \) represent bus stop IDs. Similarly, strings such as 2cal-1659-7132-4297-b89d-da624ab72db2 were easily recognized by developers as names for scripts stored on disk. Sub\textsc{sonic} has some URL strings, and when we tried them in a browser, we got pictures of music album covers, which were easy to correlate with Sub\textsc{sonic} #2. Some strings would be difficult to understand even for the original developer, e.g., just small numerals or empty strings. Overall, we found strings easy to understand in 84% of the patterns, confirming our intuition that logging string values helps in understanding applications.

5. RELATED WORK

There is a rich body of research on performance profiling. Much of this work [6, 10, 13, 16, 17, 22, 29, 30, 43, 50] focuses on identifying execution subpaths that take a long time to execute during an observed run. Several techniques [7, 39] focus on how to manipulate and present these subpaths to the user. Other techniques detect performance problems that manifest as anomalous behavior [44, 45], deviations in load tests [24], or performance regressions [33]. For all these techniques, the performance problems need to manifest in the observed runs. In contrast, S\textsc{unCat} provides information to the developer to help reason about performance problems that would occur in unobserved runs.

Mantis [21] is a very recent technique that predicts performance in smartphone applications. The predictions made by Mantis have high accuracy. However, Mantis requires developers to provide many training inputs for its machine learning algorithm. Furthermore, Mantis was evaluated on CPU-intensive applications that have little to no user interaction. Unlike Mantis, S\textsc{unCat} analyzes a single input, which makes using S\textsc{unCat} very easy. Furthermore, S\textsc{unCat} was evaluated on highly interactive applications.

Two projects [12, 49] also propose specialized techniques for performance problems that may occur in unobserved runs. These techniques plot method execution cost by input size. These techniques cannot find performance problems in code that cannot be instrumented, such as the example in Section 2. Smartphone applications make heavy use of the runtime system and asynchronous events, so S\textsc{unCat} complements these techniques. S\textsc{unCat} also helps developers assess the impact on user-perceived performance.

Several techniques [18, 32, 36, 47] detect various code and execution patterns that may be indicative of performance problems. Like S\textsc{unCat}, these techniques do not require the performance problem to slow down the observed run. Unlike S\textsc{unCat}, these techniques do not give information about how the execution cost may evolve for larger inputs.

6. CONCLUSIONS

The use of smartphone applications is increasing, and the user experience they create is determined by performance as much as by functionality. Unfortunately, testing performance for smartphone applications is difficult because it is hard to control the inputs as code extensively interacts with the environment. We have presented S\textsc{unCat}, a novel technique that helps developers use common, small inputs to understand potential performance problems that smartphone applications could have for larger inputs. The key novelties include identifying string getters as important methods to count, using lossy grammar-based compression to obtain succinct repetition patterns to inspect, and providing a delay-based mechanism to check the effect on user-perceived performance. Our analysis of 29 test scenarios in 5 Windows Phone applications showed highly promising results as we found nine performance problems.

7. ACKNOWLEDGMENTS

We thank Madanlal Musuvathi and Thomas Ball for their invaluable help and support throughout this project. We thank Jonathan de Halleux, Nikolai Tillmann, and Christian Bird for useful discussions and help with the Windows Phone applications. We thank Herman Venter and Mike Barnett for helping with the CCI infrastructure. We thank Peter Jipsen, Atanas Radenski, and Peiyi Zhao for their feedback on this paper. We thank Guoqing (Harry) Xu, Atanas (Nasko) Rountev, and Shun Lu for early feedback on this paper. We thank Darko Marinov for feedback and useful discussions. This work was partially done while the authors were in internship at Microsoft Research.
8. REFERENCES

[38] S. Perez. iTunes app store now has 1.2 million apps, has seen 75 billion downloads to date. http://techcrunch.com/2014/06/02/itunes-app-store-now-has-1-2-million-apps-has-seen-75-billion-downloads-to-date/.