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

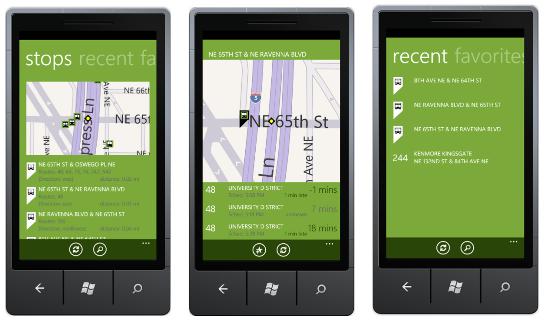
3-4 times for small inputs, while many other methods execute well over 4 times; however, for large inputs, the methods related to the problem can execute hundreds of times, creating performance problems. We propose to use only *string getters*—i.e., the methods that return values of string fields—both 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.



(a) `AllStops` (b) `OneStop` (c) `RecentViews`

Figure 1: Screenshots for `OneBusAway`

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 interactiveness of these navigations [19].

Suppose a developer wants to test the navigation between the `AllStops` and `OneStop` screens. The developer cap tap a bus stop in the list on the `AllStops` screen and, consequently, `OneBus-Away` displays the `OneStop` screen. The developer can now press

```
<TextBlock Grid.Row="1" x:Name="RouteInfo"
    Foreground="{StaticResource␣OBAForegroundBrush}"
    Text="{Binding␣Path=CurrentViewState.CurrentStop.name}"
    FontSize="{StaticResource␣PhoneFontSizeMedium}"/>
```

Figure 2: XAML declaration from `OneBusAway`

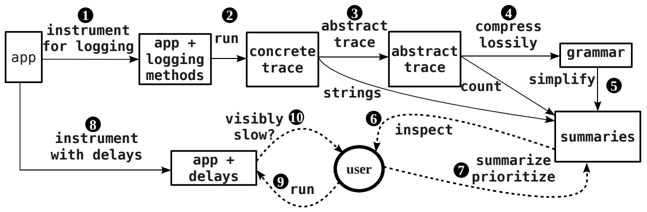


Figure 3: Process of using SunCat

the back button, and `OneBusAway` returns to the `AllStops` screen. Navigating back and forth among `AllStops` and `OneStop` screens, the developer can visit several `OneStop` screens. Note that the developer can repeat these GUI actions several times even if a part of the input is quite small (e.g., the list has only a few bus stops).

Using SunCat while navigating among `AllStops` and a *small number* of `OneStop` screens helped us to identify a performance problem that would affect the user for a *large number* of `OneStop` screens. Recall that SunCat logs string getter calls and computes a Performance Summary that includes repetition patterns of these calls, grouped by a grammar. Our analysis of the grammar and the rest of the summary for our run of `OneBusAway` showed that the number of the getter calls for bus stop name *unexpectedly* increases during the run (Section 4.3.1). To check if this increase would affect the user-perceived performance, we instructed SunCat to insert time delays for this getter. Navigating again among a *small number* of `OneStop` screens, we noticed a substantial slowdown.

Our further analysis showed that this slowdown is indeed due to a real performance problem that would be difficult to detect without SunCat for at least two reasons. First, the long delay is unlikely to manifest in regular testing. The developer (with or without using a profiler) cannot encounter the long delay unless she displays `OneStop` screen a very large number of times without closing the application. Displaying `OneStop` screen many times can easily happen for an end user but is less likely to happen during in-house testing, where individual tests tend to be short and stand alone.

Second, off-the-shelf static or dynamic tools that analyze code cannot be directly used because the root of the performance problem is not in the (compiled) C# code on which such tools typically operate. In addition to C#, `OneBusAway` uses the XAML declarative language [28]. Figure 2 shows a snippet of the XAML file that declares a certain field (`RouteInfo` of type `TextBlock`) and specifies that the Windows Phone runtime should update the field with the value of the `CurrentViewState.CurrentStop.name` field whenever the latter changes. Each time `OneBusAway` displays a `OneStop` screen, it creates a C# object (`DetailsPage`) for the current stop which will be automatically updated when the current stop changes. The runtime updates the `DetailsPage` object using reflection, and reflection is also difficult to analyze [9].

We found that `OneBusAway` has a leak where the number of `DetailsPage` objects grows over time, and they are never used by the application, leading to many unnecessary, slow updates. While the objects are small and do not take much memory, eventually their automatic updates can lead to a long delay in displaying `OneStop` screens. We reported this problem to the developers [34], who fixed it within hours. The information provided by SunCat helped us to provide a clear and convincing report describing *how the performance problem will affect the application for larger inputs*.

concrete event: a_1

```
  string value: NE 65TH ST & OSWEGO...
  stack:
    OneBusAway.ViewModel.BusServiceDataStructures.Stop.get_name()
    System.Reflection.RuntimeMethodInfo.InternalInvoke()
              [...more stack frames...]
```

concrete trace:

$a_1a_2a_3a_4b_1b_2c_1c_2d_1d_2e_1e_2e_3f_1f_2f_3g_1h_1g_2h_2g_3h_3g_4h_4i_1j_1i_2j_2i_3j_3i_4j_4i_5j_5i_6j_6$

abstract trace:

```
aaaabbccddeeefffghghghghijijijijijij
```

Figure 4: Sample concrete event and traces for `OneBusAway`

3. SunCat

SunCat works in several steps (Figure 3). ① SunCat instruments the application bytecode to log all string getters (Section 3.1). ② The developer then runs the modified binary in a test scenario, and the logging saves a trace of *concrete events* that include the call stack for getters and return values (e.g., in `OneBusAway`, concrete events include reads of *specific* bus station names). While developers typically test smartphone applications manually, SunCat could work equally well with automated testing. ③ SunCat creates a list of *abstract events* by ignoring the return values (e.g., in `OneBusAway`, abstract events include reads of *some* bus station name) and then ④,⑤ computes a *Performance Summary*, which summarizes repetitions of abstract events into *repetition patterns* and prioritizes these patterns (Section 3.2). ⑥,⑦ The SunCat user inspects the Prioritized Patterns and can further modify the summary in various ways to understand which patterns could create performance problems for runs with large inputs. ⑧ SunCat can add time delays in the application to allow the user to ⑨,⑩ check if the suspicious patterns can indeed affect the user-perceived performance (Section 3.3).

3.1 Logging

SunCat by default logs only calls to string getters, although one could also specify other events, e.g., network calls. Specifically, for each call, SunCat logs: (1) the call stack (as in calling context 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_1 for one call to `get_name` with the example string value ("`NE 65TH ST & OSWEGO...`").

SunCat uses off-the-shelf binary instrumentation of .NET to add logging methods to each string getter. Note that 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 reflection, based on the XAML file. For each update to the `DetailsPage` 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_i, 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 experience shows that it can be even more useful to compare how repetition 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, 6...) as we navigate to more `OneStop` screens. Creating small variations of GUI actions is natural for many smartphone applications, because they display data (e.g., bus stops, tweets, music songs), typically organized in groups of similar elements. Intuitively, actions for these elements should be similar and thus great candidates for *comparison*. For example, the `AllStops` screen in `OneBusAway` displays a list of bus stops, and navigating to their `OneStop` screens should be similar. To enable comparison across sub-traces, SunCat allows the user to specify them in the *Comparison Mode*, e.g., when testing the navigation to `OneStop` screens, the user can specify that navigating to and from each `OneStop` screen forms a start and stop point of a sub-trace.

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 & OSWEGO` in the concrete event $\mathtt{a}_1$ 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 '$\diamondsuit$' 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 strings in the concrete events (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 `h` 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.

```
<Grammar> ::= {<Rule> "\n"} <Rule>
<Rule> ::= <Nonterm> "->" {<Item>} <Item>
<Item> ::= <RepPattern> | <Ignore>
<Ignore> ::= "..."
<RepPattern> ::= (<Nonterm>|<Terminal>)["^"<RepCount>]
<RepCount> ::= <DetailedList> | <MaxVal>
<DetailedList> ::= {<num> "|"} <num>
<MaxVal> ::= "≤"<num>
<Terminal> ::= ? 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. $\mathtt{A}^4$ is shorthand for `AAAA`. $\mathtt{a}^{4|5|6|7}$ denotes that `a` can repeat 4, 5, 6, or 7 times. $\mathtt{C}^{\leq 6}$ 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 $\mathtt{S} \texttt{ -> } \mathtt{A}^4$ 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., `abbbabbbb` would be $\mathtt{A}^2$ with $\mathtt{A} \texttt{ -> } \mathtt{ab}^{3|4}$) 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 $\mathtt{a}^3\mathtt{b}^5\mathtt{a}^4\mathtt{b}^3$. The method $\underline{\mathtt{und}}$ 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., $\mathtt{a}^3\mathtt{b}^5\mathtt{a}^4\mathtt{b}^3$ is merged into $\mathtt{N}^2$ where $\mathtt{N} \texttt{ -> } \mathtt{a}^{3|4}\mathtt{b}^{5|3}$. This is lossy because the resulting expression encodes all (16) strings of form $\mathtt{a}^{\mathtt{n}_1}\mathtt{b}^{\mathtt{n}_2}\mathtt{a}^{\mathtt{n}_3}\mathtt{b}^{\mathtt{n}_4}$, where $\mathtt{n}_1, \mathtt{n}_3 \in \{3, 4\}$ and $\mathtt{n}_2, \mathtt{n}_4 \in \{5, 3\}$. The method `merge` preserves the order of repetition counts but removes the duplicates (so that the resulting lists are not too long).

```
abstract trace:
aaaabbccddeeefffghghghghijijijijijij◊aaaaabbccddeeeeffffghghghghijijij◊aaaaaabbccddeeefffghghghghijijijijijij◊
aaaaaaabbccddeeeeffffghghghghijijijij
```

(a) Abstract trace

$a^4b^2c^2d^2e^3f^3$ghghghghijijijijijij ◊ $a^5b^2c^2d^2e^4f^4$ghghghghijijij ◊ $a^6b^2c^2d^2e^3f^3$ghghghghijijijijijij ◊ $a^7b^2c^2d^2e^4f^4$ghghghghijijijij

---

$a^4\ b^2\ c^2\ d^2\ e^3\ f^3\ B^4\ C^6$ ◊ $a^5\ b^2\ c^2\ d^2\ e^4\ f^4\ B^4\ C^3$ ◊ $a^6\ b^2\ c^2\ d^2\ e^3\ f^3\ B^4\ C^5$ ◊ $a^7\ b^2\ c^2\ d^2\ e^4\ f^4\ B^4\ C^4$

B -> g h

C -> i j (this grammar is also generated by the lossless algorithm)

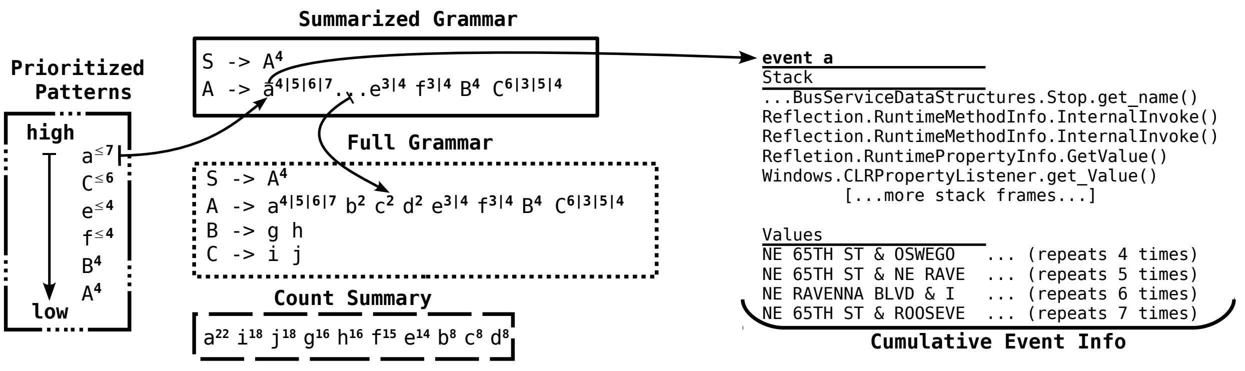
S -> A^4

A -> $a^{4|5|6|7}\ b^2\ c^2\ d^2\ e^{3|4}\ f^{3|4}\ B^4\ C^{6|3|5|4}$

B -> g h

C -> i j (this grammar is generated only by our lossy algorithm)

(b) Steps of the algorithm (some intermediate steps are not shown)



```
event a
Stack
...BusServiceDataStructures.Stop.get_name()
Reflection.RuntimeMethodInfo.InternalInvoke()
Reflection.RuntimeMethodInfo.InternalInvoke()
Refletion.RuntimePropertyInfo.GetValue()
Windows.CLRPropertyListener.get_Value()
        [...more stack frames...]

Values
NE 65TH ST & OSWEGO   ... (repeats 4 times)
NE 65TH ST & NE RAVE  ... (repeats 5 times)
NE RAVENNA BLVD & I   ... (repeats 6 times)
NE 65TH ST & ROOSEVE  ... (repeats 7 times)
```

(c) Performance summary with four levels of detail, linked to the concrete events with stacks and string values

Figure 5: Example input, steps, and output of our lossy grammar-based compression algorithm

Figure 5(b) shows several steps of the algorithm. Merging substrings of length 1 simply adds a repetition count. Merging longer substrings introduces non-terminals, e.g., `B -> g h` or `C -> i j`, which are reused when the same underlying sequence is encountered. For example, after `C` is introduced for C^6, the sequence `ijijij` is merged into C^3 (without introducing a new non-terminal).

Our lossy compression can produce substantially shorter grammars than the original lossless compression [46]. For example, the last step in Figure 5(b) is the output of our algorithm, and the second step would be the output of the lossless compression (with the first line starting with `S ->`). The lossless algorithm can only merge the exact repetitions (e.g., `aaa` into a^3 or $a^3b^5a^3b^5$ into M^2 where $M \rightarrow a^3b^5$), but it does not merge different repetitions such as $a^3b^5a^4b^3$. In contrast, our algorithm merges these *different repetitions with the same underlying sequence of symbols*.

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., $C^{6|3|5|4}$ with $C^{\leq 6}$. 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 $p^{4|10|8}$ and Q^{38}. 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 the identified methods. Effectively, SUNCAT inserts time delays 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

```
Map⟨List⟨Symbol⟩, Pair⟨List⟨RC⟩, Nonterminal⟩⟩ rules
method Grammar main(String 𝒮)
  List⟨RPList⟩ l = [] // empty list
  foreach (String s : 𝒮.split('◇')) // substrings for sub-traces
    l = l @ computeRPList(s) // append to the list
  endfor
  Grammar g = new Grammar("S", "−>", l) // starting rule
  foreach(⟨sl, ⟨rc, N⟩⟩ : rules) do g.add(N, "−>", new RPList(sl, rc)) done
  return g
method RPList computeRPList(String 𝒮)
  RPList X = new RPList(𝒮, 1) // repetition count 1 for each terminal
  int n = 1 // length of substring
  while (n < X.len / 2)
    // find all occurences of all substrings of length n
    Set⟨Set⟨int⟩⟩ r={{i}∪{j∈{i+1..X.len-n|X.und(i,n).equals(X.und(j,n))}|
                      i∈{1..X.len-n}}}
    // find repeated adjacent substrings of length n
    Set⟨Set⟨int⟩⟩ a = {s ⊆ s' ∈ r | ∀ i ∈ s. ∃ j ∈ s. i=j+n∨ i=j-n}
    if (a is empty)
      n++
    else
      Set⟨int⟩ Q = from a, pick set with the largest number of elements
      if (n == 1)
        // invariant: repetitionsCount is 1 for all elements in Q
        Symbol N = X.get(min(Q)).symbol
      else
        List⟨List⟨RC⟩⟩ e = X.getRepetitionsCounts(Q, n)
        Symbol N = getNonterminal(X.und(min(Q),n),e)
      endif
      X = X.prefix(min(Q)) @ new RP(N, [|Q|]) @ X.suffix(max(Q)+n)
      n = 1
    endif
  endwhile
  return X
method Nonterminal getNonterminal(List⟨Symbol⟩ s, List⟨List⟨RC⟩⟩ e)
  if (rules.containsKey(s))
    Pair⟨List⟨RC⟩ c, Nonterminal N⟩ = rules.get(s)
    e = c @ e
  else
    Nonterminal N = createNewNonterminal()
  endif
  rules.put(s, new Pair(merge(e), N))
  return N
method List⟨RC⟩ merge(List⟨List⟨RC⟩⟩ e)
  List⟨RC⟩ r = []
  for (int i = 1..e.getFirst().size())
    RC rc = new RC()
    foreach (List⟨RC⟩ l : e)
      foreach (int count : l.get(i))
        if (! rc.contains(count)) rc.add(count)
      endforeach
    endforeach
    r = r @ rc
  endfor
  return r
struct RC { List⟨int⟩ counts; } // repetitions count
struct RP {Symbol symbol; RC repetitionCount; } // repetition pattern
class RPList
  method RP get(int i) {...} // return RP at the offset i
  method RPList sub(int start, int len) {...} // sublist [start, start+len)
  // return underlying sequence of symbols without repetition counts
  // − lossy compression ignores repetition counts
  // − lossless compression would preserve counts ("und" same as "sub")
  method List⟨Symbol⟩ und(int start, int len)
    List⟨Symbol⟩ l = []
    foreach (int i = start..start+length-1) do l = l @ get(i).symbol done
    return l
  method List⟨List⟨RC⟩⟩ getRepetitionCounts(Set⟨int⟩ Q, int n)
    List⟨List⟨RC⟩⟩ r = []
    foreach (int i : sorted(Q))
      List⟨RC⟩ l = []
      foreach (int j = i..i+n-1) do l = l @ X.get(j).repetitionCount done
      r = r @ l
    endforeach
    return r
```

Figure 7: *Lossy* grammar-based compression

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.

Application	Source	#Rev	#Class	LOC
Conference	Microsoft	0	61	3,271
Pex4Phone	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. 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` displays the program for research conferences and was used at ICSE, FSE, ISSTA, and MSR. It displays information such as the authors that published at the conference, papers presented in each session, details about each paper, etc. The user can navigate between various screens, e.g., from a screen displaying details about one paper to the screen displaying details for an author of the paper. `OneBusAway` is our running example that displays public transportation information such as bus stops, routes, schedules, and arrivals. `TouchDevelop` lets the user build scripts for smartphones in a user friendly GUI, i.e., the user can visually build scripts. `Pex4Phone` lets the user interact with the Pex4Fun [27] web site to solve "code puzzles", i.e., write C# code that matches a secret implementation. `Subsonic` enables the user to interact with a server where the user stored favorite songs: the user can play songs from the server, see the most recently played songs and albums, get info about the albums, songs, and artists, etc.

4.2 Scenarios

Figure 9 (first three columns) lists the test scenarios that we analyzed using SunCat. The name for each scenario has the application name and an ID. For each scenario, we first navigate to some application screen (shown in a `typewriter` font), then start SunCat logging, perform a user action (shown in a **bold** font), potentially press back button and repeat the action multiple times in the Comparison Mode (column *'R?'*), and stop SunCat logging. For example, for `OneBusAway` #4, we first go to the `AllStops` screen and then perform the action "go to `OneStop`" multiple times.

We came up with these scenarios by ourselves, while using the applications in an intuitive manner. We explored each application a bit to see what screens it has; if a screen showed a list, we thought it natural to test it in the Comparison Mode, e.g., displaying the details of one bus stop should be comparable to displaying the details of another bus stop. The application developers themselves could prepare a longer list of scenarios, covering all screens.

Test Scenario	Description: `Screen`. **Action**	R?	PP?	Ev.	#CS	Number of patterns and top 3 patterns				Str	T(s)
`Conference` #1	`Authors`. **Go to** `OneAuthor`	Y	√	1489	6	2	$B^{\leq 203}$	$C^{\leq 5}$		2	1.1
`Conference` #2	`Session`. **Back to** `Main`	Y	√	8912	10	4	$B^{\leq 718}$	$C^{\leq 316}$	A^{21}	4	7.1
`Conference` #3	`Authors`. **Tap letter & pop-up**	Y	√	15580	8	3	$C^{\leq 20}$	A^{1279}	B^{1279}	3	14.0
`Conference` #4	`Session`. **Go to** `OnePaper`	Y	√	1344	4	2	$a^{\leq 444}$	$A^{\leq 5}$		1	2.1
`Pex4Phone` #1	`Main`. **Go to** `Leaderboards`	N		303	3	1	$A^{\leq 13}$			0	0.5
`Pex4Phone` #2	`Main`. **Go to** `Play`	N		1287	12	2	C^{21}	B^{3}		2	2.7
`Pex4Phone` #3	`Play, navigate away`. **Back to** `Play`	N		395	8	5	$C^{\leq 3}$	B^{21}	B^{11}	5	0.5
`Pex4Phone` #4	`Main`. **Go to** `Learn`	N		175	10	2	A^{17}	B^{15}		2	0.2
`Pex4Phone` #5	`Learn, navigate away`. **Back to** `Learn`	N		280	10	3	A^{17}	B^{15}	B^{15}	3	0.4
`Pex4Phone` #6	`Learn`. **Go to** `OneCourse`	Y	√	320	14	4	$C^{\leq 7}$	D^{4}	A^{4}	4	0.5
`Pex4Phone` #7	`Play`. **Go to** `Training`	N		156	8	2	$B^{\leq 15}$	A^{4}		2	0.2
`OneBusAway` #1	`All Routes`. **Go to** `OneRoute`	Y	√	2481	24	8	$C^{\leq 82}$	$E^{\leq 81}$	$D^{\leq 6}$	7	118.5
`OneBusAway` #2	`One Route`. **No action**	N		1302	6	3	$a^{\leq 12}$	B^{11}	c^{6}	2	13.0
`OneBusAway` #3	`All Stops`. **Go to the map**	N		179	5	2	e^{62}	d^{31}		0	0.3
`OneBusAway` #4	`All Stops`. **Go to** `OneStop`	Y	√	312	13	5	$a^{\leq 16}$	$C^{\leq 11}$	$e^{\leq 10}$	5	0.8
`OneBusAway` #5	`Recent Views`. **Go to** `OneStop`	Y	√	362	9	5	$a^{\leq 28}$	$b^{\leq 16}$	$c^{\leq 16}$	5	0.9
`OneBusAway` #6	`Main`. **Go to** `AllStops`	N		88	3	5	$a^{\leq 5}$	$B^{\leq 3}$	a^{6}	5	0.1
`TouchDevelop` #1	`Welcome`. **Go to** `Main`	N		339	18	8	f^{46}	e^{24}	d^{23}	6	1.6
`TouchDevelop` #2	`Tile`. **Go to** `Script`	N		161	7	1	A^{23}			1	0.2
`TouchDevelop` #3	`Script`. **Go to** `Tile` **and return**	Y		645	11	5	e^{46}	d^{23}	a^{7}	4	1.0
`TouchDevelop` #4	`Scripts`. **Create new script**	Y	√	5497	10	12	$b^{\leq 352}$	a^{38}	a^{38}	6	30.9
`TouchDevelop` #5	`New Script`. **Go to** `MainScript`	N		38	3	1	a^{3}			1	0.1
`TouchDevelop` #6	`Main Script`. **Create new IF stmt.**	N		21	3	2	a^{7}	A^{7}		2	0.1
`TouchDevelop` #7	`IF stmt. wizard`. **Create IF stmt.**	N	√	478	25	10	e^{48}	B^{24}	d^{24}	9	6.2
`Subsonic` #1	`Music Library`. **Tap group letter**	N		18	1	2	a^{10}	a^{8}		2	0.1
`Subsonic` #2	`Music Library`. **Go to** `OneArtist`	N		149	19	2	C^{14}	D^{4}		2	0.5
`Subsonic` #3	`Artist`. **Go to** `OneAlbum`	N		280	30	3	D^{11}	c^{4}	a^{3}	2	1.0
`Subsonic` #4	`Music Library`. **Go to** `Newest`	N		183	17	2	A^{10}	B^{6}		2	0.6
`Subsonic` #5	`Newest`. **Go to** `OneAlbum`	N		77	35	5	B^{10}	b^{10}	C^{5}	3	0.7
OVERALL			**10**		**43X**	**121X**				**84%**	**7.1**

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 useful. *"T(s)"* = time to process the trace (in seconds).

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, 7, 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 generates for this trace. From Prioritized Patterns and Summarized Grammar, several patterns stand out: $\mathtt{a}^{4|5|6|7}$, $\mathtt{e}^{3|4}$, and $\mathtt{f}^{3|4}$ 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); $\mathtt{C}^{6|3|5|4}$ 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 OneBusAway.WP7.ViewModel.BusServiceDataStructures.Stop.get_name(), i.e., a is a call to the string getter for a bus stop name. Hence, $\mathtt{a}^n$ 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 $\mathtt{a}^n$ 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 & I) 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.

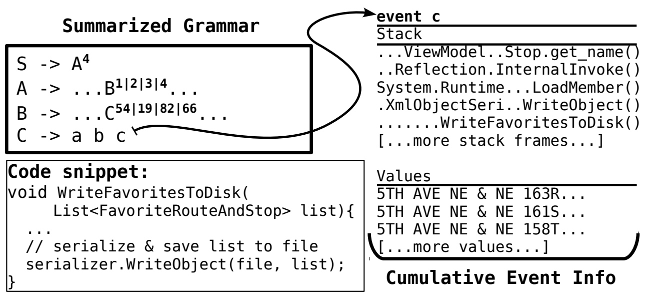
Problem Description: Putting all this together, we concluded that navigating to OneStop screens triggers more and more data-binding updates for the current page to which the user navigates. Since the number of these updates seems to grow without limit, after enough time, there will be a very large number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

Inspection Step (Run SunCat *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 SunCat 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 SunCat, 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. $\mathtt{C}^{54|19|82|66}$ 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.



```
void WriteFavoritesToDisk(
    List<FavoriteRouteAndStop> list){
  ...
  // serialize & save list to file
  serializer.WriteObject(file, list);
}
```

```
...ViewModel..Stop.get_name()
..Reflection.InternalInvoke()
System.Runtime...LoadMember()
.XmlObjectSeri..WriteObject()
.......WriteFavoritesToDisk()
[...more stack frames...]
```

```
5TH AVE NE & NE 163R...
5TH AVE NE & NE 161S...
5TH AVE NE & NE 158T...
[...more values...]
```

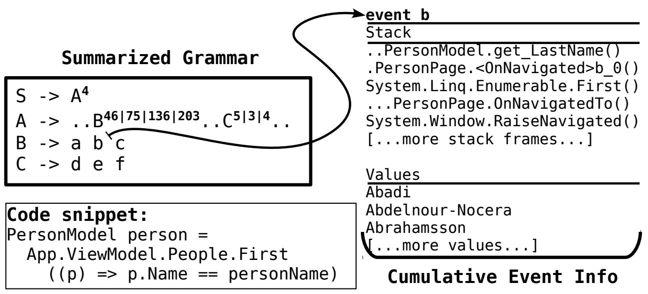
Figure 10: Performance Summary for OneBusAway #1

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 S 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 $\mathtt{C}^n$ 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. We infer that $\mathtt{C}^n$ represents a bus route.

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. From the code of that method (Figure 10), we see it saves the favorite routes (and stops) to disk, and in the class for routes we indeed find that fields have annotations for serialization.

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 $\mathtt{C}^{54|19|82|66}$, 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 $\mathtt{B}^{1|2|3|4}$ 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 $\mathtt{B}^{1|2|3|4}$ 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 $\mathtt{a}^{503}$ (or at best $\mathtt{a}^{54|73|155|221}$ 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.



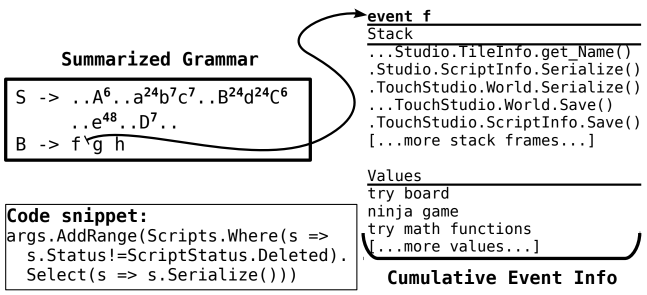
$S \to A^{4}$
$A \to ..B^{46|75|136|203}..C^{5|3|4}..$
$B \to a\ b\ c$
$C \to d\ e\ f$

```
event b
Stack
..PersonModel.get_LastName()
.PersonPage.<OnNavigated>b_0()
System.Linq.Enumerable.First()
...PersonPage.OnNavigatedTo()
System.Window.RaiseNavigated()
[...more stack frames...]

Values
Abadi
Abdelnour-Nocera
Abrahamsson
[...more values...]
```

```
Code snippet:
PersonModel person =
  App.ViewModel.People.First
    ((p) => p.Name == personName)
```

Figure 11: Performance Summary for `Conference` #1



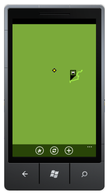
$S \to ..A^{6}..a^{24}b^{7}c^{7}..B^{24}d^{24}C^{6}$
$\quad ..e^{48}..D^{7}..$
$B \to f\ g\ h$

```
event f
Stack
...Studio.TileInfo.get_Name()
.Studio.ScriptInfo.Serialize()
.TouchStudio.World.Serialize()
...TouchStudio.World.Save()
.TouchStudio.ScriptInfo.Save()
[...more stack frames...]

Values
try board
ninja game
try math functions
[...more values...]
```

```
Code snippet:
args.AddRange(Scripts.Where(s =>
  s.Status!=ScriptStatus.Deleted).
  Select(s => s.Serialize()))
```

Figure 12: `TouchDevelop` #7 Performance Summary



Inspection Step (Run SUNCAT *with Delays):* We finally determined if this writing to disk affects the user-perceived performance. We instructed SUNCAT to insert delays in the `get_direction`, `get_id`, and `get_name` getters, and then reran the same test scenario, navigating from the `AllRoutes` screen to an `OneRoute` screen. We got the `OneRoute` screen shown to the right of the text, which persists for some time before the route info is fully displayed. Unlike in the previous scenario where the application blocks on the current screen, in this scenario the application proceeds to the next screen but shows *some* partial information (route line, the location) that is inadequate for the user.

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 $\mathtt{B}^{46|75|136|203}$ and $\mathtt{C}^{5|3|4}$ 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 $\mathtt{B}^{n}$ 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.

Inspection Step (Identify the Pattern to Explore): Figure 12 shows the simplified grammar for the above run: $\mathtt{e}^{48}$ stands out because its repetition count is large. From the Cumulative Event Info, we find that `e` corresponds to `get_BaseScriptIdFileName` in the class `ScriptInfo`. From the method name, we see that the repetition is over script files, although it is unclear what `BaseScriptId` stands for. Unfortunately, the string values for `e` do not help as they are cryptic, e.g., `46e7d1ac-5b54-4c7b-9`. However, the call stack for `e` contains the method `System.Diagnostics.Contracts.Contract.ForAll()`, which is used for .NET assertions (such as post-conditions and object invariants) and was enabled in our debug run. We stopped the investigation of this pattern, because such assertions would be presumably disabled in production runs.

Inspection Step (Identify the Pattern to Explore): The patterns that stand out next are $\mathtt{a}^{24}$, $\mathtt{B}^{24}$, and $\mathtt{d}^{24}$. We see from their contexts—$\mathtt{B}^{24}\mathtt{d}^{24}\mathtt{C}^{6}$ and $\mathtt{a}^{24}\mathtt{b}^{7}\mathtt{c}^{7}$—that the former appears more interesting because `B` and `d` seem correlated, as they are adjacent and have the same repetition number.

Inspection Step (Understand What a Non-Terminal Represents): The rule for `B` has three terminals, corresponding to `get_Name`, `get_LinkName`, and `get_Counter` from the class `Microsoft.TouchStudio.TileInfo`. We focused on `get_Name` because we were not sure what `get_LinkName` and `get_Counter` return. (It is interesting that `d`, as `g`, corresponds to the `get_LinkName` from the `TileInfo` class, but `d` and `g` differ because their call stacks differ.) `TouchDevelop` has many buttons shaped like tiles, but it was not clear to which tiles `B` refers. From the strings for `f` we immediately recognized the names of the 24 scripts that come preloaded with `TouchDevelop`, because we saw these names while we used `TouchDevelop`. We concluded that `B` corresponds to scripts.

Problem Description: Intuitively, displaying an `if/then/else` statement should not iterate over all scripts in the system. Moreover, when there will be many scripts in the system, this action can become slow. Looking at the full stack traces in the terminals of `B` (Figure 12), it appeared from the method names that `TouchDevelop` serializes (and presumably saves to disk) all the scripts in the system. Indeed, Figure 12 shows a code snippet with a LINQ expression that serializes each script that is not deleted. While this code is easy to understand *after* we explained the problem, it is not obvious to detect *before* we understood the problem [37].

4.4 Other Scenarios and False Alarms

For the other 19 scenarios from Figure 9, we did not find any performance problem. SUNCAT still reported some repetition patterns, the same way that a traditional profiler reports a profile for any program run, whether the run's input is small or large, representative or not. Developers do not consider that profilers give "false alarms" [30]. Likewise, SUNCAT *helped us to realize there are no real performance problems* in those scenarios. In particular, the string values and call stacks helped to understand 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 $\mathtt{B}^{23}$, $\mathtt{e}^{24}$, and $\mathtt{C}^{5}$, 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 $\mathtt{f}^{46}$ 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 SunCat 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 SunCat 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 SunCat 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 `OneBusAway`, but we realized later that strings starting with `1_` represent bus stop IDs. Similarly, strings such as `2ca1-1659-7132-4297-b89d-da624ab72db2.ts-base` in `TouchDevelop` would probably be recognized by developers as names for scripts stored on disk. `Subsonic` 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 `Subsonic` #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 easily 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, SunCat 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, SunCat analyzes a single input, which makes using SunCat very easy. Furthermore, SunCat 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 SunCat complements these techniques. SunCat 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 SunCat, these techniques do not require the performance problem to slow down the observed run. Unlike SunCat, 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 SunCat, 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 Shan 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

[1] CodePlex. http://www.codeplex.com/.

[2] Common Compiler Infrastructure. http://research.microsoft.com/en-us/projects/cci/.

[3] One Bus Away. http://onebusawaywp7.codeplex.com/.

[4] Subsonic. http://subsonic.codeplex.com/.

[5] TouchDevelop. http://www.touchdevelop.com/.

[6] E. A. Altman, M. Arnold, S. Fink, and N. Mitchell. Performance analysis of idle programs. In *OOPSLA*, 2010.

[7] G. Ammons, J.-D. Choi, M. Gupta, and N. Swamy. Finding and removing performance bottlenecks in large systems. In *ECOOP*, 2004.

[8] J. Belfiore. Scaling Windows Phone, evolving Windows 8. http://blogs.windows.com/windows_phone/b/windows-phone/archive/2014/02/23/scaling-windows-phone-evolving-windows-8.aspx.

[9] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE*, 2011.

[10] M. D. Bond and K. S. McKinley. Continuous path and edge profiling. In *MICRO*, 2005.

[11] E. Chan. Smartphones for data, not for calling. http://www.businessweek.com/technology/content/mar-2011/tc20110316_121017.htm/.

[12] E. Coppa, C. Demetrescu, and I. Finocchi. Input-sensitive profiling. In *PLDI*, 2012.

[13] D. C. D'Elia, C. Demetrescu, and I. Finocchi. Mining hot calling contexts in small space. In *PLDI*, 2011.

[14] D. Diaz and M. Zilberman. Build data-driven apps with Windows Azure and Windows Phone 7. http://msdn.microsoft.com/en-us/magazine/gg490344.aspx.

[15] D. Graziano. Smartphone, tablet and PC shipments to surpass 1.7 billion in 2014. http://bgr.com/2013/06/11/smartphone-tablet-pc-shipments/.

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

[17] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind. Vertical profiling: Understanding the behavior of object-oriented applications. In *OOPSLA*, 2004.

[18] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *PLDI*, 2012.

[19] M. Jovic, A. Adamoli, and M. Hauswirth. Catch me if you can: Performance bug detection in the wild. In *OOPSLA*, 2011.

[20] J. C. Kieffer and E.-H. Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46(3):737–754, 2000.

[21] Y. Kwon, S. Lee, H. Yi, D. Kwon, S. Yang, B.-G. Chun, L. Huang, P. Maniatis, M. Naik, and Y. Paek. Mantis: Automatic performance prediction for smartphone applications. In *ATC*, 2013.

[22] J. R. Larus. Whole program paths. In *PLDI*, 1999.

[23] E. Lehman and A. Shelat. Approximation algorithms for grammar-based compression. In *SODA*, 2002.

[24] H. Malik, H. Hemmati, and A. E. Hassan. Automatic detection of performance deviations in the load testing of large scale systems. In *ICSE*, 2013.

[25] Microsoft Corp. Conference App. http://research.microsoft.com/en-us/projects/confapp/.

[26] Microsoft Corp. How do I use the profiler in Windows Phone Mango? http://msdn.microsoft.com/en-us/windowsmobile/-Video/hh335849.

[27] Microsoft Corp. Pex4phone. http://www.pexforfun.com/.

[28] Microsoft Corp. XAML Overview (WPF). http://msdn.microsoft.com/en-us/library/ms752059.aspx.

[29] T. Mytkowicz, D. Coughlin, and A. Diwan. Inferred call path profiling. In *OOPSLA*, 2009.

[30] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Evaluating the accuracy of Java profilers. In *PLDI*, 2010.

[31] C. G. Nevill-Manning and I. H. Witten. Linear-time, incremental hierarchy inference for compression. In *DCC*, 1997.

[32] K. Nguyen and G. H. Xu. Cachetor: Detecting cacheable data to remove bloat. In *FSE*, 2013.

[33] T. H. D. Nguyen, M. Nagappan, A. E. Hassan, M. N. Nasser, and P. Flora. An industrial case study of automatically identifying performance regression-causes. In *MSR*, 2014.

[34] A. Nistor. Performance. Data binding updates to DetailsPage. http://onebusawaywp7.codeplex.com/workitem/16297.

[35] A. Nistor. Performance. Disk writes when navigating to the details page. http://onebusawaywp7.codeplex.com/workitem/16299/.

[36] A. Nistor, L. Song, D. Marinov, and S. Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *ICSE*, 2013.

[37] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *ISSTA*, 2011.

[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/.

[39] K. Srinivas and H. Srinivasan. Summarizing application performance from a components perspective. In *FSE*, 2005.

[40] Sun Microsystems. HPROF JVM profiler. http://java.sun.-com/developer/technicalArticles/Programming/HPROF.html.

[41] N. R. Tallent and J. M. Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In *PPOPP*, 2009.

[42] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield. Analyzing lock contention in multithreaded applications. In *PPOPP*, 2010.

[43] K. Vaswani, A. V. Nori, and T. M. Chilimbi. Preferential path profiling: Compactly numbering interesting paths. In *POPL*, 2007.

[44] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *OSDI*, 2004.

[45] Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, C. Yuan, and Z. Zhang. STRIDER: A black-box, state-based approach to change and configuration management and support. In *LISA*, 2003.

[46] Q. Xu and J. Subhlok. Efficient discovery of loop nests in communication traces of parallel programs. Technical report, University of Houston, UH-CS-08-08, 2008.

[47] D. Yan, G. Xu, and A. Rountev. Uncovering performance problems in Java applications with reference propagation profiling. In *ICSE*, 2012.

[48] Yourkit LLC. Yourkit profiler. http://www.yourkit.com.

[49] D. Zaparanuks and M. Hauswirth. Algorithmic profiling. In *PLDI*, 2012.

[50] X. Zhang and R. Gupta. Whole execution traces. In *MICRO*, 2004.