# Catch Me If You Can: Performance Bug Detection in the Wild

Milan Jovic
University of Lugano
Milan.Jovic@usi.ch

Andrea Adamoli
University of Lugano
Andrea.Adamoli@usi.ch

Matthias Hauswirth
University of Lugano
Matthias.Hauswirth@usi.ch

## Abstract

Profilers help developers to find and fix performance problems. But do they find performance *bugs* – performance problems that real users actually notice?

In this paper we argue that – especially in the case of interactive applications – traditional profilers find irrelevant problems but fail to find relevant bugs.

We then introduce *lag hunting*, an approach that identifies perceptible performance bugs by monitoring the behavior of applications deployed in the wild. The approach transparently produces a list of performance issues, and for each issue provides the developer with information that helps in finding the cause of the problem.

We evaluate our approach with an experiment where we monitor an application used by 24 users for 1958 hours over the course of 3-months. We characterize the resulting 881 issues, and we find and fix the causes of a set of representative examples.

***Categories and Subject Descriptors*** [*Performance of Systems*]: Measurement techniques; D [*Software Engineering*]: Metrics—Performance measures

***General Terms*** Performance, Measurement, Human Factors

***Keywords*** Profiling, Latency bug, Perceptible performance

## 1. Introduction

Many if not most software applications interact with human users. The performance of such applications is defined by the users' perception. As research in human-computer interaction has shown, the key determinant of *perceptible* performance is the system's lag in handling user events [5, 6, 24, 25]. Thus, to understand the performance of such applications, developers have to focus on perceptible *lag*.



*OOPSLA'11,* October 22–27, 2011, Portland, Oregon, USA.



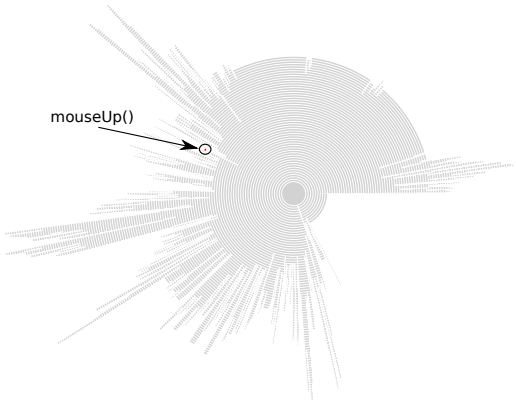


**Figure 1.** Perceptible Bug Hidden in Hotness Profile

However, when developers try to understand the performance of their applications, they use profilers to identify and optimize *hot* code. **Figure 1** visualizes the output of a traditional code hotness profiler in the form of a calling context tree rendered as a sunburst diagram[1]. The center of the diagram represents the root of the tree (the application's main method). The tree grows inside out. Each calling context is represented by a ring segment. The angle of a ring segment is proportional to the *hotness* of the corresponding calling context. The tree in Figure 1 consists of 224344 calling contexts and represents the production use of a real-world Java application (Eclipse).

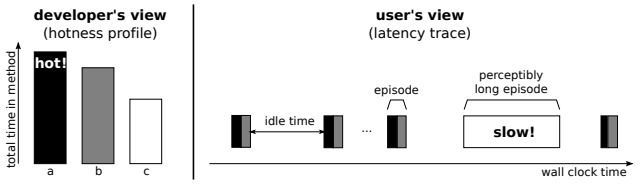


**Figure 2.** Developer vs. User-Centric View of Performance

Following a traditional performance tuning approach, a developer will focus on the hottest regions (the ring segments with the widest angles) of the calling context tree. In this paper we argue that this is the wrong approach for interactive applications. As **Figure 2** shows, code hotness profiles

[1] We use sunburst diagrams, also known as "calling context ring charts" [1], because they scale to the large calling context trees of real-world programs.

produced by traditional profilers do not properly reflect the perceptible performance of applications. The left part of the figure represents a traditional *hotness profile*, a developer-centric view of performance, showing the three hottest methods (methods that spent most CPU cycles). The right part represents a *trace* of the corresponding program execution. The trace shows a significant number of idle-time intervals (where the computer was waiting for user input). We use the term "episode" to denote the intervals of computational activity in-between idle intervals. More importantly, the trace shows that some (hot) methods were executed often but each invocation completed quickly, while other (cold) methods were executed rarely but an invocation might have taken a long time. This example highlights that perceptible performance problems are not necessarily caused by the hottest methods in a traditional profile.

The real-world hotness profile in Figure 1 confirms this point: the calling context tree contains a node corresponding to a method that repeatedly (30 times) exhibited an exceptionally long (at least 1.5 seconds) latency. However, that context (the EditEventManager.mouseUp method indicated with the arrow) is barely visible in the tree, because the invocations of this method make up less than 1% of the application's execution time. A developer looking at this profile would never have considered tuning that method to improve performance. Instead, he probably would have focused on hot methods that are frequently called but do not necessarily contribute to perceptible lag.

### 1.1 Measuring latency instead of hotness

In this paper we introduce an approach that automatically catches perceptible performance bugs such as the one hidden in Figure 1. We do this by focusing on the latency instead of the hotness of methods.

At first sight, measuring latency may seem simple: just capture time-stamps for method calls and returns. Unfortunately, for real-world applications, the instrumentation and data collection overhead for such an approach significantly slows down the application. This is the reason why commercial profilers use call-stack sampling instead of tracking method calls and returns. Moreover, the large overhead perturbs the time measurements and casts doubt on the validity of the measurement results.

A central goal of our approach is thus the ability to measure latency and to gather actionable information about the reason for long latency method calls, while keeping the overhead minimal.

### 1.2 Catching bugs in the wild

Performance problems are highly dependent on the context in which an application runs. Their manifestation depends on (1) the underlying platform, (2) the specific configuration of the application, and the (3) size and structure of the inputs the application processes. For example, applications may run on computers with different clock frequencies, numbers of cores, amounts of memory, and different amounts of concurrent activity. Users may configure applications by installing a variety of plug-ins, some of them might not even be known to the application developer (e.g. they may install a spell-checking plug-in into an editor, and that plug-in may perform unanticipated costly operations when getting notified about certain changes in the document). Or users may use an application to process inputs that are unexpectedly sized (e.g. they may browse a folder containing tens of thousands of files, they may use a web server to serve mostly multi-gigabyte files, or they may want to edit an MP3 file that contains 10 hours of audio). Without knowing the exact usage scenarios of widely deployed applications, developers are unable to conduct representative performance tests in the lab.

Thus, the second goal of our approach is the ability to directly detect performance problems in the deployed applications.

### 1.3 Contributions

In this paper we propose *Lag Hunting*. Our approach targets *widely-deployed* applications by gathering runtime information in the field. It targets *interactive* applications by focusing on perceptible lag. Without any user involvement, it produces a list of performance issues by combining data gathered from the different deployments. Moreover, it automatically produces an *actionable* [20] report for each issue that helps in identifying and fixing the issue's cause.

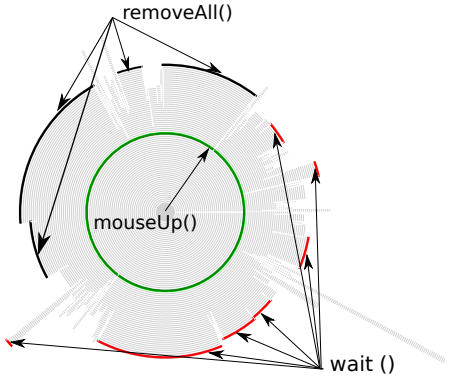


**Figure 3.** Perceptible Bug in Lag Hunting Profile

**Figure 3** shows the calling context tree that is part of the report our approach produces for the performance bug that is barely visible in Figure 1. Unlike the calling context tree derived from a traditional sampling profiler, our tree only represents the behavior that is related to the given bug, and it shows all the method calls that contributed to the long latency of the problematic mouseUp method.

We demonstrate the practicality of our approach by implementing it in *LagHunter*, our performance bug detector for Java GUI applications. In a three-month experiment we deployed LagHunter to find performance issues in one of the most complex interactive Java applications, the Eclipse

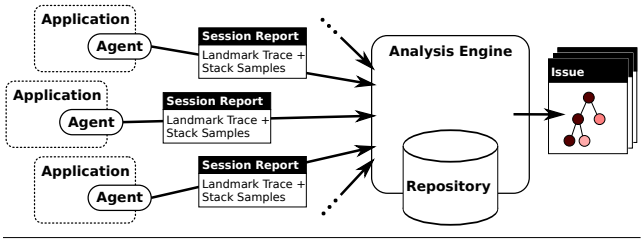


**Figure 4.** Lag Hunting

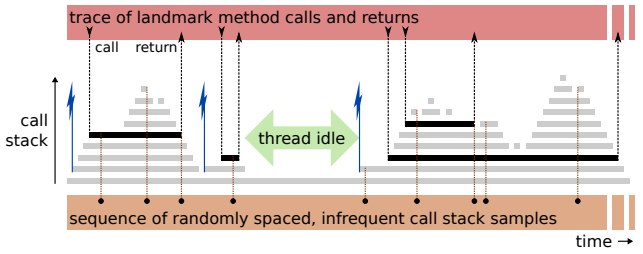


**Figure 5.** Session Report

IDE. In that experiment, LagHunter identified 881 performance issues. We characterized these issues and we performed three case studies to find and fix the performance bugs for representative issues.

The remainder of this paper is structured as follows: Section 2 introduces the Lag Hunting approach. Section 3 presents *LagHunter*, our tool for lag hunting in Java GUI applications. Section 4 characterizes the performance issues LagHunter identified in Eclipse. Section 5 evaluates our approach with three case studies. Section 6 discusses the limitations of our approach and our evaluation, Section 7 presents related work, and Section 8 concludes.

## 2. Approach

**Figure 4** provides an overview of our approach. Applications are deployed with an agent that collects performance information about each session. At the end of a session, the agent sends a session report to an analysis engine running on a central server. The server collects, combines, and analyzes the session reports to determine a list of performance issues. To the developers, the analysis engine appears like a traditional bug repository. The only difference is that the "bug reports" are generated automatically, without any initiative from the users, that all equivalent reports have been combined into a single issue, that related issues are automatically linked together, and that issues contain rich information that points towards the cause of the bug.

### 2.1 What data to collect?

Perceptible performance problems manifest themselves as long *latency* application responses. A complete trace of each call and return of all methods of an application would allow us to measure the latency of each individual method call, and to determine all the callees of each method and their contributions to the method's latency. However, such an approach would slow down the program by orders of magnitude and thus could not be used with deployed applications. The challenge of any practical latency bug detection approach is to reduce this overhead while still collecting the information necessary to find and fix bugs. Moreover, the information needs to be collected in a way to enable a tool to effectively aggregate a large number of session reports, each containing possibly many occurrences of long-latency behavior, into a short list of relevant performance issues.

**Figure 5** shows part of the execution of an interactive application. The x-axis represents time. The three flash symbols represent three incoming user requests (events) that need to be handled by the application. The call stack, shown in light-grey, grows from bottom to top. Each rectangle on the stack corresponds to a method activation. In our approach, the agent collects two kinds of information. First, it captures calls and returns from so-called *landmark* methods (black rectangles). This landmark trace contains the timing information necessary to measure lag. Second, it captures rare, randomly spaced call stack samples of the running threads. Each stack sample contains the complete calling context of the thread at the time the sample was taken. This sequence of stack samples provides information about what the thread was doing throughout a long-latency landmark method invocation.

Figure 5 shows information for only one thread. We do collect the same information for all threads. However, for many interactive applications, most threads are blocked most of the time. Moreover, for most current GUI toolkits, only one thread, the *event dispatch thread*, is allowed to interact with the user (process user input and update the GUI). For many applications, if the user is not interacting with the application, the GUI thread becomes idle, waiting for new user requests[2]. The figure shows such an idle interval in the center. While the thread is idle, no landmark method is on the stack, and we thus do not need any call stack samples[3].

### 2.2 Landmark methods represent performance issues

The most crucial parameter in our approach is the choice of landmark methods. Landmark methods serve a double purpose. First, their calls and returns represent the points where we measure latency. Second, each landmark method represents a *potential performance issue*. The lag hunting approach produces a list of potential performance issues that is identical to the list of landmark methods that incurred a

[2] For some applications, such as video players or games, the GUI thread receives periodic requests from a timer, which cause it to update the GUI from time to time.

[3] In a practical implementation of our approach, such as LagHunter, it can be beneficial to *always* sample the call stack and to drop the samples taken whenever the thread is idle, because the low sampling rates of our approach only cause negligible sampling overhead, but there is a cost of communicating the start and end of idle intervals to the call stack sampler.

non-negligible latency (e.g., at least one invocation of that method took more than 3 ms). Each method in that list is annotated with a rich set of information that includes its invocation latency distribution. A landmark method becomes a *real* performance issue, if a developer deems its latency distribution to be severe (e.g., over 100 ms on average).

A chosen set of landmarks should fulfill three properties: First, they need to *cover most of the execution* of the program. Given that we only measure the latency of landmarks, we will be unaware of any long activity happening *outside* a landmark. Second, they need to be *called during the handling of an individual user event*. We want to exploit the human perceptibility threshold (of roughly 100 ms) for determining whether a landmark method took too long, and thus such a method needs to correspond to a major activity that is part of handling a user event. A method executed in a background thread may take much longer than 100 ms, but it will not affect the responsiveness of the application, if the GUI thread continues to handle user events. Moreover, a top-level method of the GUI thread (e.g. the main method of the application represented by the bottom rectangle in Figure 5), may have a very long "latency", but it is not responsible for delaying the handling of individual user events. Third, landmarks should be *called infrequently*. Tracing landmarks that are called large numbers of times for each user event would significantly increase the overhead.

### 2.3 Landmark selection strategies

We now describe how to select good landmarks. While our approach focuses on interactive applications, just by changing the set of landmarks, many of these ideas would also apply to the analysis of transaction-oriented server applications.

**Event dispatch method.** One possible landmark method is the single GUI toolkit method that dispatches user events. This method will cover the entire event handling latency. However, when using this method as the *only* landmark, the analysis will result in a list with this method as a single issue. This means that many different causes of long latency will be combined together into a single report, which makes finding and fixing the causes more difficult. Even though the event dispatch method is not very useful when used in isolation, it is helpful in combination with other, more specific, landmarks. Any left-over issue not appearing in the more specific landmarks (methods transitively called by the dispatch method) will be attributed to the dispatch method.

**Event-type specific methods.** A more specific set of landmarks could be methods corresponding to the different kinds of low-level user actions (mouse move, mouse click, key press). However, these methods are still too general. A mouse click will be handled differently in many different situations. Having a single issue that combines information about mouse clicks in multiple contexts is often not specific enough.

**Commands.** Ideally, the landmarks correspond to the different commands a user can perform in an application. If the application follows the command design pattern [9] and follows a standard implementation idiom, it is possible to automatically identify all such landmarks.

**Observers.** Even an individual command may consist of a diverse set of separate activities. Commonly, a command changes the state of the application's model. In applications following the observer pattern [9], the model (synchronously) notifies any registered observers of its changes. Any of these observers may perform potentially expensive activities as a result. If the application follows a standard idiom for implementing the observer pattern, it is possible to automatically identify all observer notifications as landmarks.

**Component boundaries.** In framework-based applications, any call between different plug-ins could be treated as a landmark. This would allow the separation of issues between plug-ins. However, the publicly callable API of plug-ins in frameworks like Eclipse is so fine grained, that the overhead for this kind of landmarks might be too large.

**Application-specific landmarks.** If application developers suspect certain kinds of methods to have a long latency, they may explicitly denote them as landmarks to trigger the creation of specific issues.

### 2.4 Data collection

Given a specification of the set of landmark methods, the agent deployed with the application dynamically instruments the application to track all their invocations and returns. Moreover, the agent contains code that periodically samples the call stacks of all the threads of the application. It also collects information about the platform, the application version, and installed plug-ins. At the end of a session, the agent combines the information into a session report and uploads it to the analysis server.

### 2.5 Analysis

For each session report, the analysis engine extracts all landmark invocations from the landmark trace. For each landmark invocation, it computes the inclusive latency (landmark end time - landmark start time) and the exclusive latency (inclusive latency - time spent in nested landmarks), and it updates the statistics of the corresponding issue. The repository contains, for each issue, information about the distribution of its latencies, the number of occurrences, and the sessions in which it occurred. Moreover, to help in identifying the cause of the latency of a given landmark, the repository contains a calling context tree related to that landmark. This tree is built from the subset of stack samples that occurred during that landmark (excluding samples that occurred during nested landmarks). The tree is weighted, that is, each calling context is annotated with the number of samples in which it occurred.

### 2.6 Connecting issues

Based on the information about individual issues, the analysis engine then establishes connections between related issues. It does that by computing the similarities between the issues' calling context trees. Two different issues (e.g. two different commands) may trigger the same cause of long latency. If that is the case, their calling context tree will be similar. If a developer investigates a performance issue, the connections to similar issues can provide two key benefits: it can simplify the search for the cause (by providing more information about the contexts in which it occurs), and it can help to prioritize the issue (given that a fix of the issue may also fix the related issues).

### 2.7 Call pruning

Given an issue, its calling context tree, latency distribution, and the list of related issues allow a developer to form hypotheses about the reason for the long latency. To test those hypotheses, developers would usually change the program and rerun it to confirm that the change indeed reduced the perceptible latency.

Our approach partially automates this step. We do this by re-executing the application while automatically omitting various parts of the expensive computation. In particular, we omit calls to *hot* methods in the calling context tree. Note that we do *not* omit the calls to the landmark method, but calls to hot methods that are (transitively) called by the landmark method. Note that pruning a method call does not usually correspond to a fix. It often leads to crashes or an incorrectly working application. However, it is effective in locating and confirming the cause of long latency behavior.

### 2.8 Discussion

The key idea behind our approach is to gather a large number of session reports and to convert them into a small set of issues familiar to the developer. In this way, the developer gains insight about the behavior of a landmark in many different execution contexts. Note that we do not require developers to explicitly specify landmark methods. All but the last class of landmarks we outline can be detected automatically. Moreover, commands and observers represent an infinite number of landmarks: the agent automatically recognizes any commands or observers as a landmark. The fact that we only trace a set of relatively infrequently executed landmarks keeps the performance impact of data collection low. Moreover, the fact that many users provide session reports enables a low call stack sampling rate with minimal overhead.

## 3. Implementation

LagHunter is a lag hunting tool targeting Java GUI applications. The LagHunter agent consists of a Java agent and a JVMTI agent that are shipped together with the application. The application does not have to be changed, the agents are activated using command line arguments to the Java virtual machine. The Java agent dynamically rewrites the loaded classes using ASM [22] to instrument all landmark method calls and returns. It is also responsible for removing method calls during call pruning. The JVMTI agent is written in native code and loaded into the Java virtual machine. It is responsible for call stack sampling. Moreover, it also traces the virtual machine's garbage collection activity. At the shutdown of the virtual machine, the JVMTI agent gathers the landmark trace, the stack sample sequence, and the garbage collection trace, compresses them, and ships them to the analysis server. If the upload fails, it keeps them on disk and tries to send them the next time the application shuts down.

### 3.1 Landmarks

LagHunter supports several types of landmarks: the event dispatch method, observers, paint methods, and native calls. The *event dispatch method* is easy to identify, it corresponds to a specific method in the SWT or Swing GUI toolkits (we support both). Using the event dispatch method as a landmark ensures that we capture *all* episodes incurring long latency. We instrument all invocations of *observer* notification methods that follow Java's EventListener idiom. This kind of landmark is easy to detect automatically and usually provides a meaningful level of abstraction. The *paint method* landmark correspond to any paint() method in a subclass of java.awt.Component. It tracks graphics rendering activity in Swing applications (SWT uses the observer pattern for painting). Especially in visually rich applications, graphics rendering can be expensive and paint method landmarks represent abstractions meaningful to a developer. Finally, we instrument all calls sites to *native* methods. They represent cross-language calls to native code via JNI, and thus may be responsible for long-latency input or output operations. While the kinds of landmarks supported by LagHunter are targeted specifically at interactive applications, LagHunter could be extended with further landmark types, e.g., to support event-based server applications.

### 3.2 Overhead and perturbation

Dynamic binary instrumentation takes time, and classes are loaded throughout the application run, so our Java agent's instrumentation activity might perturb its own timing measurements. To make our implementation practical, our agent uses a variety of optimizations, among them a persistent instrumentation cache that stores instrumented classes to avoid re-instrumentation in subsequent application runs.

Most method calls complete quickly. This is true even for landmark methods. If we built a complete landmark trace, with every call and return, the session reports would grow unreasonably large. Moreover, the constant tracing and I/O activity would also perturb our latency measurements. We thus filter out short landmarks online. Our filter maintains a shadow stack of active landmark methods. A call to a landmark method pushes a new element on the stack. A

return from a landmark method pops the top-most element. It drops it if the latency was below a desired threshold, and it writes it to the trace otherwise. Using this trace filtering approach, we can reduce the trace size by several orders of magnitude without losing relevant information.

**Evaluation.** To evaluate the overhead and perturbation caused by our agent we conducted controlled experiments on three different platforms. Three different users had to perform predefined tasks in two different large interactive applications, Eclipse and NetBeans, which we ran on top of our agent.

We first used the data collected by the JVMTI agent to verify that the Java agent did not cause excessive overhead. The stack samples collected in our initial experiment showed that we spent a significant amount of time in our trace filtering code, and the garbage collection trace showed an abnormally large number of garbage collections. We studied our trace filtering implementation and realized that we were triggering object allocations every time we processed an event. We eliminated these allocations and reran the experiments.

To evaluate the startup overhead due to our agent, we measured the startup time of the original applications (average across three platforms; Eclipse: 1.1 s, NetBeans: 2.85 s), the startup time of the application with our agent, but without the persistent instrumentation cache (Eclipse: 3.44 s, NetBeans: 7.28 s), and the startup time of the application with our agent, using the persistent instrumentation cache (Eclipse: 1.58 s, NetBeans: 3.2 s). All startup times represent the duration from launching the Java virtual machine until the dispatch of the first GUI event. The above results show that the persistent instrumentation cache is effective in reducing the startup time of the application *with* the LagHunter agent to values close to those of the original application.

We also measured the cost of stack sampling. The median time to take a stack sample varies between 0.5 ms and 2.23 ms. This is a small fraction of the perceptible latency of 100 ms, and thus is negligible with a low-enough sampling rate. Finally, we measured the effect of the filter threshold on trace size: a threshold of 3 ms reduces trace size by a factor of over 100. This shows that trace filtering is essential in reducing the size of the session reports shipped from the users to the central repository.

We have deployed our agent with three different applications (BlueJ[4], Informa[5], and Eclipse) which the students in our undergraduate programming class have been using extensively during their normal course work, and the only feedback we received was that the applications did not shut down immediately. This is to be expected, because we process and upload the session report when the application quits.

[4] `http://bluej.org`

[5] `http://informaclicker.org`

### 3.3 Analysis

Our repository collects all session reports as well as the generated issue information. We automatically run the analysis every night to update the issues with newly received information. We have developed a small web interface to access our issue repository. This allows us to rank issues according to various criteria, and to look at the reports for each specific issue. Those reports contain visualizations of latency distributions and interactive visualizations of the calling context tree. We render the calling context trees, which can contain hundreds or thousands of nodes, with the calling context ring chart visualization of the Trevis [1] interactive tree visualization and analysis framework. When establishing connections between related issues, we use Trevis' weighted multiset node distance metric to compute the similarity between calling context trees.

## 4. Issue Characterization

In this section we characterize the repository of the 1108 session reports we gathered during a three-month experiment. Our experiment involved 24 users working for a total of 1958 hours with Eclipse, one of the largest interactive Java applications. We configured LagHunter to take a call stack sample every 500 ms on average, and to drop all samples taken during idle time intervals.

### 4.1 Parameters

LagHunter's analysis engine detected 881 issues in the given reports. **Table 1** characterizes five key parameters of this set of issues. For each parameter it shows the mean, first quartile, median, second quartile, 90-th percentile, and maximum value. The first parameter, the average exclusive latency, represents the severity of a given issue: the longer its latency, the more aggravating the issue. The next three parameters describe how prevalent a given issue is. From the perspective of a developer, the need to fix an issue increases when it is encountered by many users, occurs in many sessions, and occurs many times. The last parameter, the number of collected call stacks, is indicative of the chance of fixing the issue. The more call stack samples available about a given issue, the more information a developer has about the potential cause of the long latency.

| | Avg | Q1 | Med | Q3 | p90 | Max |
|---|---|---|---|---|---|---|
| Avg.Excl.[ms] | 320 | 4 | 11 | 31 | 119 | 38251 |
| Users | 5 | 1 | 2 | 8 | 19 | 24 |
| Sessions | 49 | 1 | 3 | 14 | 115 | 1069 |
| Occurrence | 896 | 2 | 6 | 48 | 449 | 338622 |
| Stacks | 65 | 0 | 0 | 3 | 42 | 28613 |

**Table 1.** Univariate characterization

The average exclusive time of an issue varies between less than a millisecond and over 38 seconds. However, 90% of these issues take less than 119 ms. This shows that most

"issues" are benign. They can easily be filtered out in the web-based front-end to our repository. Note that we capture information about benign issues on purpose[6]. This allows us to see whether a given landmark *always* had a long latency, or whether its long latency invocations were exceptional. The number of *users* who encounter an issue, the number of *sessions* in which an issue appears, and the number of overall *occurrences* of an issue follow equally skewed distributions. Only a small minority of issues are prevalent, which means that developer effort can be focused on fixing a small number of high-impact performance bugs. The table also shows that the *stack* sample distribution is equally slanted: half of the issues receive no stack samples, and 10% of the issues receive 42 or more stack samples.

### 4.2 Landmark location

Most of the reported landmarks correspond to listener notifications. In which parts of Eclipse are those listeners located? First, we noticed that a significant fraction of them (275 of 881) are located *outside* the standard Eclipse classes. Many of those are inside optional or 3rd party plug-ins, but some are also in the class libraries, and a few are in dynamically generated proxy classes. The majority of the 606 landmarks in the standard Eclipse distribution are located in a few dominating plug-ins. **Figure 6** shows the top-15 of these plug-ins. The dominant plug-in (with 111 landmarks) implements the Eclipse workbench user-interface. Moreover, given that the users in our experiment were developing software in Java, it is not surprising to notice many landmarks in the Java Development Tools (JDT) user interface (84) and the JFace text editor (76) plug-ins.

Notice that the above distribution does not quantify the *latency* encountered at these landmarks. It just summarizes the landmarks that LagHunter instrumented, which were executed, and where the latency was at least 3 ms, so they ended up in at least one session report. Moreover, the distribution does not specify where the *causes* of long latency were located, but it focuses on the locations where long latency can be *measured*.

### 4.3 How prevalent are long-latency issues?

**Figure 7** shows a bubble plot of issue occurrence vs. average exclusive latency. Both scales are logarithmic. Each bubble corresponds to an issue. The size of the bubble is proportional to number of users who encountered the issue.

Most issues are clustered below 100 ms. Eclipse is a mature product that has been extensively tuned for performance, and the Eclipse team, explicitly or implicitly, must have cared about the 100 ms perceptibility threshold. Notice, however, that the y-axis corresponds to the *exclusive* latency of an issue, which means that the application's response time may have been significantly larger due to nested landmarks. Moreover, the axis represents the issue's *average* latency, and thus some individual occurrences of an issue might have taken significantly longer.

[6] To limit trace size and overhead, LagHunter's online filtering approach does eliminate "very short" landmark method invocations (in the configuration used for this experiment, invocations with an *inclusive* time of less than 3 ms).

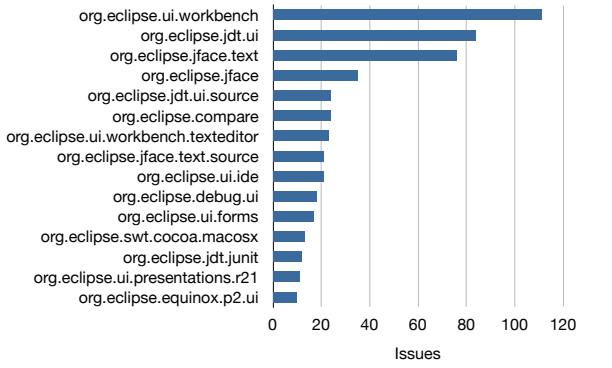


**Figure 6.** Distribution of landmarks to plug-ins

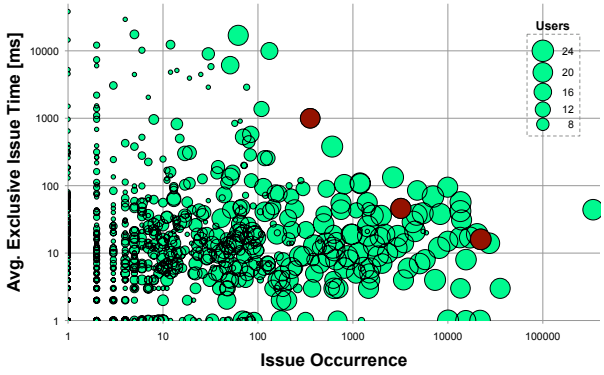


**Figure 7.** Number of users encountering an issue

Issues with few occurrences are encountered by a small number of users. The same is the case for some of the long latency issues. We found that some of these issues correspond to rare user operations (e.g. project creation), while others correspond to activity in non-standard plug-ins (e.g. Android) installed only by a small set of users.

### 4.4 Availability of stack samples

**Figure 8** has the same axes as Figure 7, but the bubble size is proportional to number of collected stack samples. This figure confirms that issues that occur often or exhibit a long latency have a higher chance of being encountered by the JVMTI stack sampler. These are precisely the issues developers want to fix, and thus our approach automatically produces the information where it is needed most, and omits it where it is irrelevant.

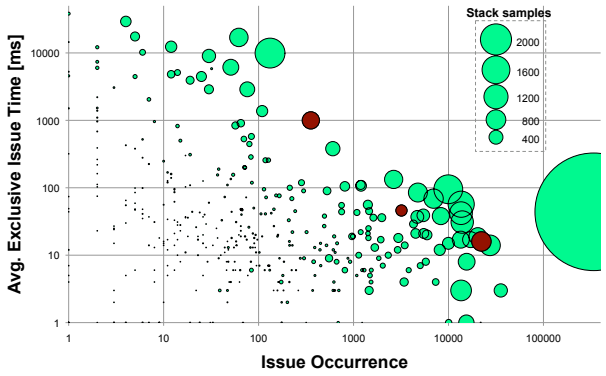


**Figure 8.** Number of stack samples per issue

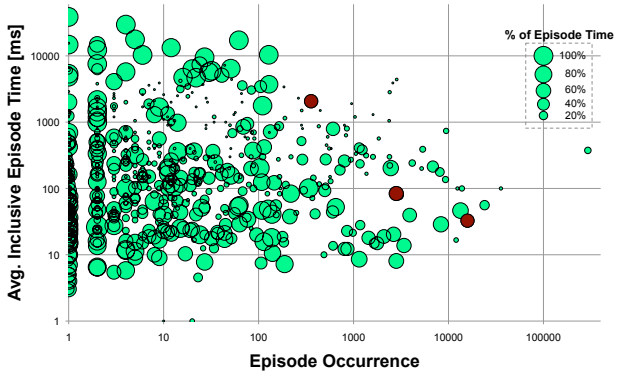


**Figure 9.** Severity of issues (x/y), and benefit of fix (size)

## 5. Case Studies

In this section we describe case studies resolving three of the 881 reported issues. We picked three interesting issues that occurred frequently and had a non-negligible latency.

**Figure 9** shows all issues in the repository and highlights the three case study issues. In this figure, the x-axis represents the total number of episode occurrences containing the issue. The y axis shows the average inclusive time of episodes containing the issue. The circle size represents the fraction of inclusive episode time due to the issue.

Like in Figure 7 and 8, each circle represents an issue. However, unlike in those prior figures, the x and y axes both represent the *episodes*. This corresponds to a user's view of performance: users are not interested in the exclusive time of issues, but they primarily perceive the inclusive time of complete episodes. The x/y space of Figure 9 thus represents the user's perspective (the severity as perceived by the user). The x-axis represents *how often* a user is annoyed, and the y-axis represents *how much* a user is annoyed each time. The third dimension, the size, represents the developer's perspective: a developer could say "I know this circle represents a frequent (x) and big (y) annoyance for the user, but how much can I reduce this annoyance if I fix this issue?" The size of the circle represents the answer to that question. A circle of 50% size (like the three circles representing our three case studies) means that by fixing that issue, the latency of the episodes that contain that issue can be reduced by 50% on average.

Figure 9 shows that there is a relatively small number of issues a developer would want to explore. The most promising issues are those that occur frequently (to the right), lead to long latency episodes (to the top), and can significantly reduce the containing episodes' latency when fixed (large circle). The three issues we picked lie more or less on the frontier of Figure 9, which means that they incurred significant latency or a significant number of occurrences.

### 5.1 Methodology

In each case study we followed the same methodology:

**Identify.** The web front-end to our LagHunter repository shows a table of all known issues. We pick an issue and study the information provided in its report, which includes the landmark method name, the latency distribution, the number of sessions, users, and occurrences, and the number of call stack samples. Moreover, each issue with at least one call stack sample contains a calling context tree.

**Reproduce.** Given the information in the issue report, we try to reproduce the problem on our own computer. The landmark name and the calling context tree help greatly in this step. We reproduce each issue to confirm its presence and to understand how to trigger it. During reproduction, we run the application together with the LagHunter agent, collecting a session report for analysis. Analyzing such an individual report helps us to understand an issue in isolation.

**Prune.** Before fixing the bug, we try to speed up the application by *pruning* the issue's calling context tree. We rerun the application, and we instruct the LagHunter agent to prune calls for that issue. This way, we do not have to edit the source code and rebuild the application, but we can just rerun it with an extra option to let the agent do the work.

**Resolve.** To fix the bug, we change the source code manually to decrease latency. This is the most challenging step. We need to keep the expected functionality while reducing execution time. This may require the use of caching of prior computations, or the strengthening of conditions to eliminate computation that is not absolutely essential. It may also require turning costly computations or synchronous input/output operations into asynchronous background activities that do not block the user interface thread. Due to the complexity of some of these fixes and our limited experience with the Eclipse code base, in the following case studies we sometimes eliminate or sim-

| Landmark | Avg. Excl. Latency [ms] | | |
|---|---|---|---|
| | Identified | Reproduced | Fixed |
| mouseDoubleClick | 999 | 149 | 4 |
| verifyText | 16 | 458 | 13 |
| keyPressed | 46 | 30 | <3 |

**Table 2.** Resolved issues

plify a particularly costly application feature instead of providing an optimized alternative implementation.

**Verify.** To verify the fix, we again rerun the application on top of the agent. We repeat the same interaction we performed during reproduction. We should already notice, even before looking at the collected session report, the absence of any perceptible latency, and the analysis of the session report should confirm that the latency fell below the perceptibility threshold.

### 5.2 Issues under study

**Table 2** lists the three issues we fix in the following case studies. The first column shows the name of the landmark method (we omit the class name for brevity). The second column shows the average exclusive latency as identified by LagHunter. The third column shows the same metric, but obtained during the reproduction step. The last column shows the latency after applying our fix. The small exclusive latency of only 16 ms for the second issue may be striking. However, this value is an average, and a significant number of this issue's occurrences were perceptibly long. More generally, the table shows a significant difference between the latencies identified in the field and the latencies observed during the reproduction in the lab. The goal of a reproduction in the lab is to confirm the presence of a problem, and to gather initial hypotheses about the causes. A reproduction in the lab thus is not necessarily a faithful reproduction of *all* of a user's activity. Moreover, it takes place in a different environment, with a potentially different application configuration, and thus necessarily leads to measurement results that can differ significantly from the average observed in the field. The important aspect of Table 2 is the drastic reduction in latency between the reproduced and the fixed measurements. Moreover, in the real-world use of our approach, the ultimate measure of success will be the reduction of the latencies observed *in the field*, after the deployment of the fixed version of the application.

The following three subsections describe each case study in detail.

### 5.3 Maximize and Restore

Sorting issues by their average time, one of the top ranked issues was the `mouseDoubleClick` method in the AbstractTabFolder class. It occurred 355 times and took 999 ms on average. Its latency distribution shows a wide range of latencies starting from 46 ms up to more than 10 seconds.

**Reproduciton.** Eclipse implements a multi-page text editor. Maximizing or restoring the active page tab is followed by an animation. This action is reported as perceptible. To reproduce, we open a document from a workspace and perform the maximize and restore actions with a double-click on the document tab.

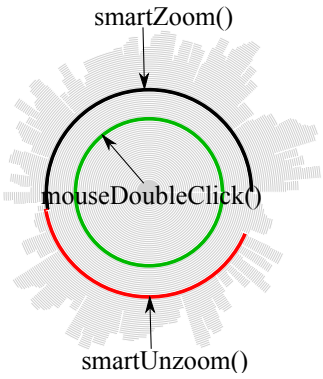


**Figure 10.** Maximize and restore

**Pruning and Resolution.** The calling context tree of this issue is shown in the ring chart [1] of **Figure 10**. The root of the tree (the main method) is represented by the center, and the branches grow towards the outside. Each calling context is represented by a ring segment. A segment's angle corresponds to the number of times the given calling context was sampled. The chart points out two equally "heavy" branches: `smartZoom` and `smartUnzoom`. It shows that the double-click executed both of these methods. One solution is to remove animations to simplify the zoom and unzoom actions. We eliminate lag by cutting the animation from the source code. A redesign of the solution for the animation to make it faster would require deeper changes.

**Validation.** We reran Eclipse on our agent and repeated the experiment performing double-clicks on the page tabs. The latency decreased to only 4 ms, as shown in Table 2.

### 5.4 Rename Refactoring

Sorting issues by their total exclusive latency, one of the top ranked issues is the `verifyText` method in the TextViewer class. It occurred 22335 times and appeared in 446 sessions.

**Reproduction.** A user of Eclipse can perform an in-place rename refactoring. The name is directly editable in the text editor. Every key typed during such a refactoring is immediately applied to all occurrences of that identifier in the open text editor. Each key pressed causes a percetible lag. This behavior was easily reproducible on our computer.

**Pruning. Figure 11** shows the calling context tree of this issue. The black ring segments towards the leaves of the tree represent the root cause of the problem. They correspond to calls to the `packPopup` method of the RenameInformationPopup class. As the most sampled method in this issue, it consumed most of the time. Its class implements a tooltip-style popup that appears below the identifier while in rename

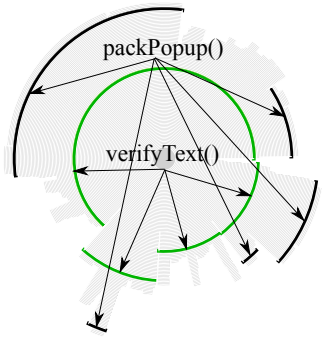


**Figure 11.** Rename refactoring

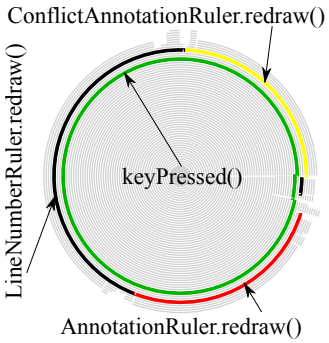


**Figure 12.** Scrolling

rafactoring mode. The popup has a special shape of a call-out. The packPopup method recomputes the shape of the non-rectangular popup after each key press. We used call pruning to eliminate the expensive method calls. This significantly reduced the latency, however, the popup window did not appear anymore.

**Resolution.** To fix the bug, we cached the results after the first packPopup invocation, which avoided the unnecessary computations.

**Validation.** After fixing this bug, in-place refactoring became much faster. The perceptible lag disappeared, and the measured average latency was reduced to 13 ms.

### 5.5 Scrolling

The 100 ms perceptibility threshold applies to individual user requests. However, movies need to be rendered at a rate of roughly 25 frames per second to appear smoothly. The same applies to animations, such as the scrolling of a document. Thus, in these instances, the latency should be below 40 ms to be imperceptible.

When sorting the issues by occurrences, the `keyPressed` method in the ViewportGuard class appears near the top of the list, with more than 343000 occurrences. It is called every time a key is pressed in the text editor. It took 46 ms on average.

**Reproduction.** The calling context tree in **Figure 12** shows that most of the time in this issue is spent redrawing three components: LineNumberRuler, AnnotationRuler, and ConflictAnnotationRuler. Knowing about the landmark method and these three classes points us towards the action the user must have performed: scrolling the viewport of the source code editor. To reproduce this issue we open a long Java source file that can't fit on the screen. We hold down the arrow key to scroll. We notice that the scrolling is not as smooth as it could be. The bigger the opened project and the more views we open, the bigger the latency becomes.

**Pruning.** We used call pruning to eliminate the notifications of the various rulers, which decreased the latency to 3 ms.

**Resolution.** As a resolution, we decided to prevent the redraw of the rulers only when the up or down arrow keys are pressed (but to allow repaints for all other keys). This avoids redraws while scrolling, but it permits redraws in all other situations.

**Validation.** We reran the fixed Eclipse and validated the reduced latency when scrolling.

**Discussion.** The calling context tree in Figure 12 shows three rulers that react to key presses. However, the calling context tree of the reproduced issue contained only `redraw` invocations for two of those ruler classes. The reason for this is that the ConflictAnnotationRuler class is not part of any standard Eclipse plugin. Sometimes, such additional plug-ins installed by users may significantly reduce the responsiveness of an application. Without the help of LagHunter, developers would not be able to discover such issues.

### 5.6 Linking issues by similarity

When fixing the issues in the above case studies, we realized that by fixing one of the issues, some other issues disappeared, too. For example, by solving the `keyPressed` (scrolling) issue, we also solved one of the most dominant issues: the largest bubble (with most call stack samples) in Figure 8. This realization lead to the idea of linking issues by the similarity of their calling context trees.

Now that we have implemented issue-linking, LagHunter reports, for each issue, a small set of the most similar issues. This idea is similar to a recommender system: "if you are interested in issue X, you may also want to have a look at issue Y". To compare two issues, we measure the weighted multiset node distance [1] between their calling context trees. Moreover, instead of presenting the related issues in a flat list ordered by similarity, we visualize them in the form of a dendrogram, using hierarchical agglomerative clustering to cluster issues by similarity. This approach compares each issue to each other issue and forms a hierarchy of clusters. The dendrogram represents this hierarchy of clusters as a tree. The leafs of the tree correspond to issues, and the internal nodes of the tree correspond to clusters. The root node of the tree corresponds to the single cluster containing all issues. **Figure 13** shows the clusters of related issues LagHunter produced for the above three case studies (top to bottom). The x-axis on the dendrograms shows the dissimilarity: 0

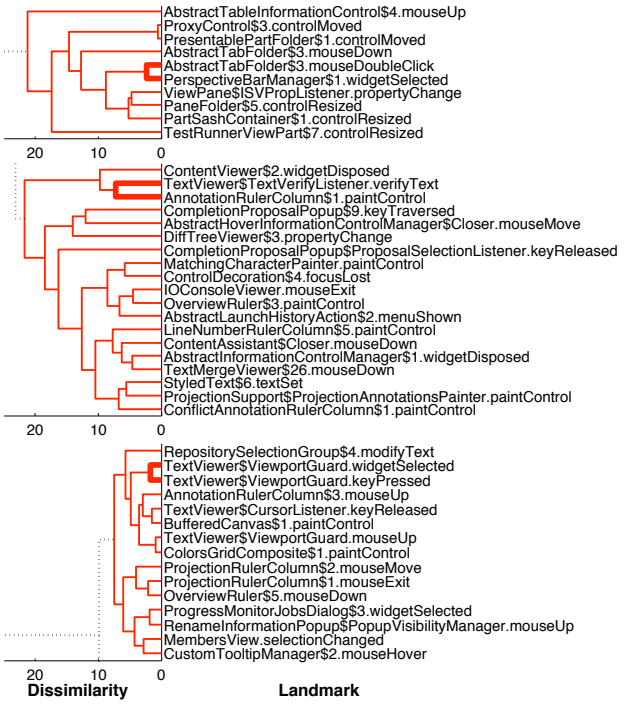


**Figure 13.** Issues clustered by dissimilarity

means that the calling context trees of two issues are identical, and 100 means that the two calling context trees are entirely different. Note that the axis does not go to 100, because we only show the subset of the dendrogram with the most similar issues. The bold lines in the dendrograms show the connections between the issue we solved and its most similar peer.

The top dendrogram shows that AbstractTabFolder.mouseDoubleClick is similar to other issues related to tabs and resizing. Knowing this ahead of time would have helped us in reproducing and fixing the bug. The middle dendrogram shows that TextViewer.verifyText is similar to several issues related to painting controls, including some related to various rulers. Finally, the bottom dendrogram shows that ViewportGuard.keyPressed is most similar to ViewportGuard.widgetSelected, the dominant issue in Figure 8. This information would have helped us to predict that fixing the verifyText issue would have the added benefit of fixing the (even more relevant) keyPressed bug.

## 6. Limitations

This section discusses (1) the limitations of the lag hunting approach and our implementation, and (2) the limitations of our evaluation studies.

### 6.1 Limitations of Approach

**User-relevant issues.** Do the issues reported by our approach correspond to those performance problems users really care about? Our approach exploits the fact that there is a known threshold at which users start to perceive latency. This is a unique situation: we have a performance metric (event handling latency), *and* we have an absolute value for that metric (around 100 ms for discrete events), established by prior research [5, 6, 24, 25], at which performance is sufficient. However, it would be interesting to study, in the context of general interactive applications, whether users indeed would report perceptible latencies above the threshold as performance issues. One could do that by combining automatic lag hunting with the manual fly-by profiling approach [1], where users can click a button to immediately report a performance problem when it occurs.

**False positives.** Does LagHunter inadvertently catch butterflies instead of just bugs? LagHunter's list of issues includes *all* traced landmark method calls. Instead of taking a binary decision and reporting only the "bugs", LagHunter allows the developer to rank the issues according to criteria such as the landmark's inclusive time, the number of its occurrences, or the number of sessions in which it occurred. It is up to the developer to select the ranking approach, and to decide how far down the list he will go in his performance tuning effort. These decisions are not easy, and they are less a matter of correctness and more a matter of developer priorities and resources.

**False negatives.** LagHunter does not necessarily catch all perceptible performance bugs. While it does highlight all individual landmark method calls that exhibit a long latency, it does not currently point out contiguous bursts of short landmark method calls. Such bursts can occur in applications that use animation (such as games or video players). If subsequent calls in such bursts are not separated by any idle time interval, this is indicative of an animation where rendering an individual frame takes too long. It could be useful to also automatically detect and report such bursts.

LagHunter detects long latency only in landmark methods. If a method *outside* a landmark incurs long latency, LagHunter will not detect it. To circumvent this problem, we include the event dispatch method as a landmark, and thus any latency in the GUI thread will be captured (all methods during episodes are transitively called by the event dispatch method). However, if *only* the dispatch method was defined as a landmark, all such long-latency episodes would appear as a single issue (the event dispatch method). Such a catch-all issue is difficult to fix. A developer would have to look at hot subtrees of the dispatch method's CCT, introduce new landmark definitions accordingly, and then wait for new issues to appear.

**Concurrency.** LagHunter targets interactive applications. Prior work has shown that such applications rarely exhibit significant concurrent behavior [2]. However, our profiler does work with concurrent applications, and it tracks the behavior of all threads, so it is possible to see what other threads were doing while the user interface thread exhibited lag. Moreover, our lag hunting approach is not limited to in-

teractive applications. For example, it could also be used to track lag in transaction-oriented server applications, where concurrency is much more prevalent. In that situation, it might be beneficial to extend LagHunter to track dependencies between threads. That way it could also highlight the reasons for why a thread blocked or waited.

**No automatic optimization.** Our approach helps developers to catch relevant latency bugs, however it does not automatically optimize the application to eliminate those bugs. While feedback-directed performance optimizations usually target a very specific class of performance issue (e.g., to improve memory locality by object co-location), the goal of our approach is to catch all perceptible performance bugs, independent of their underlying cause. We thus follow an ends-based approach ("what needs to be fixed?") instead of a means-based approach ("what can we fix?"). This entails, however, that we do require the involvement of a human developer.

**Difficulty of issue reproduction.** How difficult is it for developers to reproduce an issue given just a lag hunter report? While the information in the report, especially the name of the landmark method and the calling context tree, help in understanding the activity the user must have performed in the wild, additional information (such as screenshots of the user's GUI at the time the problem occurred) would certainly be helpful. Our agent could capture screenshots when it detects long latency behavior and ship those screenshots to the analysis engine as part of the session report. Moreover, it could use a GUI record and replay tool to capture user interactions, to partially automate the reproduction of latency bugs. For privacy reasons, we did not capture screens or user interactions in our initial implementation of the lag hunting approach.

However, to alleviate the above privacy concerns, we have developed a dialog that can automatically be shown to the user after a latency bug occurred. That dialog presents a screenshot and allows the user to black out arbitrary regions of the screen (e.g. to remove passwords or other private information that is visible). Moreover, we overlay the history of the mouse coordinates over that screenshot, and allow the user to navigate back in time in terms of mouse position, and thus to indicate when the bug occurred. Finally, we ask the user to point where (position/GUI component in the screenshot) the bug occurred. We did not use these features in our study, because we had not yet fully implemented them at that time. However, our hypothesis[7] is that they might help to gather useful information aiding developers in bug reproduction, while preserving user privacy.

[7] Unknown to us, Google independently developed a similar privacy-preserving screenshot feature, which they added to their new Google+ web application. They show a "Send feedback" button at the bottom right of the web page, and when a user clicks the button, they allow them to black out screen regions and to highlight problem areas. Google's introduction of this approach confirms our hypothesis that our idea can be beneficial.

**Cost of stack sampling.** Each time our stack sampling agent takes a sample it slightly increases the latency of the program. In Section 3.2 we have shown that the median cost of taking a sample is between 0.5 ms and 2.23 ms. Using a sampling rate of 1000 Hz would thus add between 50 ms and 223 ms to a 100 ms episode. This would significantly perturb the measurement results and perceptibly slow down the application. Moreover, the large number of samples would lead to larger traces. However, because we perform lag hunting in the wild, we can reduce the sampling rate proportionally to the number of users running the application, while still receiving the same number of samples for each landmark. Moreover, when a developer reproduces a reported issue in the lab, we found that a sampling rate of 10 Hz is still enough to gather enough stack samples to confirm the issue by repeating the perceptibly slow activity a few times.

**Inaccuracy of stack sampling.** Mytkowicz et al. [20] have shown that current Java stack sampling profilers are inaccurate due to the placement of safe-points by the virtual machine's JIT compilers. Our approach depends on the same infrastructure underlying the profilers studied in that paper, and we thus are prone to the same inaccuracies in the calling context profiles collected for each issue. The fact that we collect samples in many different environments (different users, hardware, operating systems, virtual machines), over many different program runs, might mitigate part of the problem. Moreover, our landmark tracing approach is based on bytecode instrumentation, and thus the latency we report is not affected by that problem. Finally, once virtual machine implementers correct the problems pointed out by Mytkowicz et al., our sampled calling context profiles will automatically become more accurate.

**Limitations of call pruning.** Call pruning can lead to crashes, usually after the omitted method call. Pruning is not calling-context sensitive. Assume we prune a call site o.x() in method m() that would call a method x(). If method m() is called from a context outside the given landmark, this pruning could lead to crashes even outside this landmark. We have not observed this in practice, but we could easily construct a case where this would occur. Note that the goal of pruning is to provide the developer with an idea of the consequences of *not* executing some call (some call on the path to a hot CCT subtree). Pruning the call is not a fix, it normally leads to missing functionality or even to a crash. Thus, the developer will still have to replace the omitted code with a more efficient implementation.

### 6.2 Limitations of Study

**Application selection.** We evaluate LagHunter by catching latency bugs in only one single application (Eclipse). This carries the risk that our approach might not be effective in finding bugs in other programs. However, Eclipse is an order of magnitude larger and richer than most other interactive Java programs. Moreover, it consists of a large number of independent plug-ins, and some of those plug-ins are more

complex than many normal stand-alone applications. Eclipse also is the only Java application that uses two separate Java GUI toolkits: its default toolkit is SWT, but some Eclipse plugins (including the plugin responsible for the bug in Figure 3) use AWT/Swing. Besides the three case studies presented in Section 5, we have been using LagHunter to catch performance bugs in a range of other applications, including Informa [13]. Most recently LagHunter has been adopted by the authors of CodeBubbles [3] and is now included in the current CodeBubbles pre-release.

**Issue selection.** We selected three issues for our case studies. Our selection was not a blind, random process, and it was not based on strict, pre-determined criteria. While we took care in selecting severe and interesting issues, we may have been biased towards issues that are easier to understand or fix. We believe that Figure 9 provides a good starting-point for methodologically selecting issues to analyze in future empirical studies. Ideally, such studies would take a random sample of issues with a pre-determined minimum number of episode occurrences, average inclusive episode time, and percentage of inclusive episode time due to the issue.

**Self evaluation.** We evaluated the benefits of our tool by using it ourselves. Thus, our case studies may be biased by our desire to see our approach perform well. Moreover, we were not familiar with the Eclipse code base, and thus our approach to fixing a bug is not necessarily representative of the approach of an experienced Eclipse developer. We would love to better quantify how using LagHunter improves developer productivity over not using LagHunter. However, we believe that a worthy answer to this question will require studies with real developers. To get quantitative measures of productivity increase with reasonable statistical significance, we would need many developers willing to invest a significant amount of time.

## 7. Related Work

We discuss four areas of related work: post-deployment problem detection, call graph profiling, generic dynamic profiling and tracing, and latency profiling.

**Post-deployment error detection.** Liblit et al. [17] present an approach for finding correctness bugs by correlating the value of predicates inserted at specific program points with the occurrences of failures. They reduce the cost of the potentially expensive instrumentation by employing a code-duplication-based sampling approach, and they improve coverage by collecting the information from deployed applications. Hauswirth and Chilimbi [14] developed an approach for memory leak detection in the field based on low-overhead dynamic instrumentation. Nagpurkar et al. [21] focus on detecting performance problems in the field. However, they focus on embedded applications, rely on hardware profiling support, and identify hot code instead of long latencies. Adamoli and Hauswirth [1] present a fly-by profiling approach which maintains a ring buffer of periodically taken call stack samples and sends them to the developers when a user wants to complain about performance. Like Nagpurkar et al., they cannot identify long latencies. Glerum et al. [11] describe Windows Error Reporting, an infrastructure that gathers information on the client computer at the time of a failure, and sends it to a central server. They categorize failure reports by classifying them into "buckets", based e.g., on the application, module, and program counter value at the time of the failure. Unlike our approach, their work does not include data about what happened before the crash (only information available at the time of the crash). Ganapathi and Patterson [10] found that 44% of the automated Windows Error Reports in their university network represented hang bugs. However, their definition of hang bug only includes bugs where a user explicitly killed an application. It thus excludes all hang bugs with less extreme latencies.

**Call graph profiling.** Call graph profiling has been used for a long time to understand application performance. In 1982 Graham et al. first described gprof [12], the profiler that is now part of the prevalent GNU tool chain. While gprof only maintains one level of context (one edge in the call graph), more recent profilers can track more of the calling context, up to the entire calling context tree. Later work, such as Spivey's [26], addresses the high cost associated with gathering accurate context-sensitive profiles.

Our approach relates to call profiling in two ways. First, we use a sampling based call profiler for root cause analysis within long-latency behavior. Using a sampling-based approach is essential to reduce perturbation. Second, we use an instrumentation based approach to find long-latency behavior, however, we only instrument a subset of important landmark method calls. In general, LagHunter differs from classical call profilers in that it collects traces instead of profiles. Our online trace filtering approach enables the effective reduction of trace size while keeping the latency information necessary for lag hunting.

To provide information about the root cause of a performance bug, our approach reruns the program while pruning the calling context tree. Our idea for call pruning is based on Misailovic et al.'s work on loop perforation [19]. While they skip specific loop iterations, we skip specific method calls. However, unlike loop perforation, call pruning is a focused approach: we explicitly prune methods that are known to contribute significantly to perceptible latency.

**Generic dynamic profiling and tracing.** Existing dynamic profiling and tracing techniques such as DTrace [4] or BTrace [18] provide means to easily instrument running applications. Similarly, aspect-oriented programming implementations such as AspectJ [16] provide dynamic weaving approaches. All these generic approaches provide means to specify *what* to instrument and what to *do* in the instrumentation.

However, neither approach is sufficient for our needs. First, the approaches do not support the identification of all the landmark methods we need to trace. While they can instrument method calls matching specific patterns, or even all methods overriding or implementing specific supertype methods, they are unable to identify all listener method invocations, because it is not sufficient to know the name, class, or supertype of a call target to determine whether it is a listener method. While one could instrument *all* call sites, or all call sites that invoke a method in some subtype of EventListener (something that is possible with the above techniques), and then *decide at runtime* in the trace action whether the specific call indeed is a listener notification, such an approach would lead to significant overhead. Moreover, neither of the two low-overhead approaches (DTrace and BTrace) supports arbitrary computations in its trace actions (e.g., they disallow loops and recursion), and thus they would be unable to do the online trace filtering necessary for a lightweight approach.

While the above techniques do not support lag hunting, it would be interesting to extend them to support our approach.

**Latency profiling.** Existing methods to measure response time can be grouped into two categories: *intrusive* instrumentation-based approaches, and *non-intrusive* indirect approaches. While non-intrusive approaches do not require the instrumentation of applications or GUI frameworks and are more portable, they do not provide the level of detail available using instrumentation-based approaches.

Endo et al. [7] measure the response time of applications running on Windows with a non-intrusive approach. They detect idle time intervals by running a low-priority probe thread that will only get scheduled when the interactive application is idle. Their measurements of short interactions with Microsoft Office applications show that Word, which they assume uses asynchronous threads to hide latency, handles about 92% of requests in less than 100 ms, while Powerpoint performs significantly worse.

Zeldovich et al. [27] devised a different non-intrusive approach to measure response times of interactive applications: they instrument VNC [23], an infrastructure that allows the remote interaction with applications, to track the communication between the application and the display server. They then correlate user input and screen updates to determine response times. Using VNC they can replay tracked sessions under different system configurations and determine whether changes in configuration impact response times. Since VNC is available on various window systems, their approach allows the comparison of response times on different platforms. The application and system with the best response time they report on handles only 45% of requests in less than 100 ms.

The non-intrusive mechanism to measure response time in the work by Fláutner et al. [8] keeps track of all processes that communicate as a result of a user event in the X server and measures the time it takes for all these processes to become idle.

In our own prior work [15] we introduced the idea of measuring the latency of observers and we presented a profiler for use by developers in the lab. Its static-instrumentation approach and the size of its traces made it impractical for use in the wild. Moreover, it did not provide any information about the causes of the long latency of listeners. In subsequent work we presented a tool that visualizes latency profiles [2] in a visualization similar to the sketch in Figure 5, and we studied the profiles of 14 open-source Java GUI applications. One of the key findings in that prior work was the low amount of concurrency in those applications: only 1.2 threads are runnable on average during interactive episodes, and for perceptibly long episodes that number is even lower. The lag hunting approach and the LagHunter tool presented in this paper were significantly influenced by our prior work. However, the core idea of lag hunting is different: instead of creating a single heavy-weight profile in the lab, we need to collect a large number of light-weight profiles in the field. This new approach enabled us, for the first time, to study the perceptible performance of real-world applications in production environments, and to automatically produce performance bug reports that are helpful in locating and fixing real performance bugs.

## 8. Conclusions

Wouldn't it be great if developers had actionable information about the performance, as perceived by their users, of their deployed applications?

In this paper we argue that performance bugs are bugs, too. Performance bugs are particularly sensitive to context, which means that they may manifest themselves in the wild but may escape detection in a testing lab. Thus, to catch performance bugs, post-deployment detection approaches are essential.

We present such an approach and a tool, LagHunter, which focuses on latency bug detection in interactive applications. LagHunter combines a low-overhead approach to latency profiling with call stack sampling, automatically computes information about similar latency bugs, and it performs semi-automated call pruning, to efficiently help developers find the causes of long latency.

LagHunter is lightweight enough for deployment in production settings. We deploy it in such a setting to find performance bugs of the Eclipse IDE. By monitoring 24 student developers during a 3-month project, LagHunter gathered 1108 Eclipse sessions representing 1958 hours of usage. Based on that data, our automatic analysis reported 881 issues. In this paper we characterize these issues based on their severity and the information content of their reports. In a case study we pick three representative issues, and we use the generated reports to find and eliminate the corresponding performance bugs.

Our characterization shows that even production-quality software like Eclipse still contains considerable performance bugs, and our case study shows how LagHunter helps developers to fix these bugs. We have already used LagHunter to successfully find and fix performance bugs in other applications, among them a large educational software tool we wrote ourself. Moreover, LagHunter has been adopted by the developers of CodeBubbles for catching performance bugs in a pre-release of their software. We have made LagHunter available for download[8], and we hope that developers of interactive Java applications will start to use it to catch their own perceptible performance bugs in the wild.

## Acknowledgements

We would like to thank the anonymous reviewers for their valuable feedback. This work has been conducted in the context of the Binary Translation and Virtualization cluster of the EU HiPEAC Network of Excellence. Milan Jovic has been funded by the Swiss National Science Foundation under grant number 200020_125259.

## References

[1] Andrea Adamoli and Matthias Hauswirth. Trevis: A context tree visualization & analysis framework and its use for classifying performance failure reports. In *SoftVis '10: Proceedings of the ACM Symposium on Software Visualization*, 2010.

[2] Andrea Adamoli, Milan Jovic, and Matthias Hauswirth. Lagalyzer: A latency profile analysis and visualization tool. In *ISPASS '10: Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 2010.

[3] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. Code bubbles: rethinking the user interface paradigm of integrated development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 455–464, New York, NY, USA, 2010. ACM.

[4] Bryan M. Cantrill, Michael W. Shapiro, Adam H. Leventhal, and Sun Microsystems. Dynamic instrumentation of production systems. In *USENIX 2004 Annual Technical Conference (USENIX'04)*, pages 15–28, June 2008.

[5] I. Ceaparu, J. Lazar, K. Bessiere, J. Robinson, and B. Shneiderman. Determining causes and severity of end-user frustration. *International Journal of Human-Computer Interaction*, 17(3), 2004.

[6] J. R. Dabrowski and E. V. Munson. Is 100 milliseconds too fast? In *CHI '01 extended abstracts*, 2001.

[7] Y. Endo, Z. Wang, J. B. Chen, and M. Seltzer. Using latency to evaluate interactive system performance. In *OSDI '96*, 1996.

[8] K. Flautner, R. Uhlig, S. Reinhardt, and T. Mudge. Thread-level parallelism and interactive performance of desktop applications. In *ASPLOS-IX*, 2000.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[10] Archana Ganapathi and David Patterson. Crash data collection: A windows case study. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 280–285, Washington, DC, USA, 2005. IEEE Computer Society.

[11] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. Debugging in the (very) large: ten years of implementation and experience. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 103–116, New York, NY, USA, 2009. ACM.

[12] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. *SIGPLAN Not.*, 39(4):49–57, 2004.

[13] Matthias Hauswirth and Andrea Adamoli. Solve & evaluate with informa: a java-based classroom response system for teaching java. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 1–10, New York, NY, USA, 2009. ACM.

[14] Matthias Hauswirth and Trishul M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. *SIGPLAN Not.*, 39(11):156–164, 2004.

[15] Milan Jovic and Matthias Hauswirth. Measuring the performance of interactive applications with listener latency profiling. In *PPPJ '08*, pages 137–146. ACM, 2008.

[16] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP'01*, 2001.

[17] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. *SIGPLAN Not.*, 38(5):141–154, 2003.

[18] Sun Microsystems. BTrace. https://btrace.dev.java.net/, 2009.

[19] Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin Rinard. Quality of service profiling. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 25–34, New York, NY, USA, 2010. ACM.

[20] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter Sweeney. Evaluating the accuracy of java profilers. In *PLDI '10: Proceedings of the ACM SIGPLAN 2010 conference on Programming language design and implementation*, New York, NY, USA, 2010. ACM.

[21] Priya Nagpurkar, Hussam Mousa, Chandra Krintz, and Timothy Sherwood. Efficient remote profiling for resource-constrained devices. *ACM Trans. Archit. Code Optim.*, 3(1):35–66, 2006.

[22] ObjectWeb. ASM. Web pages at `http://asm.objectweb.org/`.

[8] `http://sape.inf.usi.ch/laghunter`

[23] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper. Virtual network computing. *IEEE Internet Computing*, 02(1), 1998.

[24] B. Shneiderman. Response time and display rate in human performance with computers. *ACM Comput. Surv.*, 16(3), 1984.

[25] B. Shneiderman. *Designing the user interface: strategies for effective human-computer interaction*. Addison-Wesley, 1986.

[26] J. M. Spivey. Fast, accurate call graph profiling. *Softw. Pract. Exper.*, 34(3):249–264, 2004.

[27] N. Zeldovich and R. Chandra. Interactive performance measurement with VNCplay. In *FREENIX Track: USENIX Annual Technical Conference*, April 2005.