Measuring the Performance of Interactive Applications with Listener Latency Profiling

Milan Jovic, Matthias Hauswirth
Faculty of Informatics, University of Lugano
Lugano, Switzerland
jovicm@lu.unisi.ch, Matthias.Hauswirth@unisi.ch

ABSTRACT

When Java developers need to improve the performance of their applications, they usually use one of the many existing profilers for Java. These profilers generally capture a profile that represents the execution time spent in each method. The developer can thus focus her optimization efforts on the methods that consume the most time. In this paper we argue that this type of profile is insufficient for tuning interactive applications. Interactive applications respond to user events, such as mouse clicks and key presses. The perceived performance of interactive applications is directly related to the response time of the program.

In this paper we present listener latency profiling, a profiling approach with two distinctive characteristics. First, we call it *latency* profiling because it helps developers to find long latency operations. Second, we call it *listener* profiling because it abstracts away from method-level profiles to compute the latency of the various listeners. This allows a developer to reason about performance with respect to listeners, also called observers, the high level abstraction at the core of any interactive Java application.

We present our listener latency profiling approach, describe LiLa, our implementation, validate it on a set of micro-benchmarks, and evaluate it on a complex real-world interactive application.

Categories and Subject Descriptors: D.2.8 [Software Engineering]: Metrics – *Performance measures*

General Terms: Measurement, Performance, Human Factors.

Keywords: GUI, Listeners, Latency, Profiling.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PPPJ 2008, September 9–11, 2008, Modena, Italy.
Copyright 2008 ACM 978-1-60558-223-8/08/0009 ...$5.00.

1. INTRODUCTION

In this paper we present a profiling approach for interactive applications that helps developers to quickly focus on the performance issues that actually are perceptible by users. We present our listener latency profiling approach and a concrete implementation called LiLa, and we evaluate LiLa on microbenchmarks and complex real-world applications.

A significant body of research in human-computer interaction has studied the connection between the human perception of system performance and the response latency of interactive systems. Shneiderman [14] finds that *lengthy* or *unexpected* response time can produce frustration, annoyance, and eventual anger. Moreover, he reports [13] that user productivity decreases as response time increases. Ceaparu et al. [1] write that users reported that they lost above 38% of their time working on computers due to frustration, and they identify time delays as a significat reason for such frustration. MacKenzie and Ware [8] study lag as a determinant of human performance in virtual reality environments. They tested lags up to 225 ms and found that user performance significantly decreased with increasing lag. Dabrowski and Munson [2] study the threshold of detection for responses of interactive applications, showing that for keyboard input, users notice response times greater than 150 ms, while for mouse input, the detection threshold is at 195 ms.

All this prior research shows that the key determinant of perceptible performance is the system's latency in handling user events. The goal of this paper is to introduce an approach to profile this latency for interactive Java applications. Most of today's interactive Java applications are written on top of one of two GUI toolkits: AWT/Swing[1] [15], which is part of the Java platform, or SWT [10], which is part of IBM's Eclipse rich-client platform [9]. Our LiLa profiler abstracts away from the

[1]Java's built-in GUI toolkit consists of two parts, the original AWT, which is still used as the foundation, and Swing, which reimplements the AWT widgets in Java and adds many more powerful widgets. In the remainder of this paper we are going to use the name "Swing" to refer to this combination of Swing running on top of AWT for brevity.

intricacies of these toolkits and can profile applications based on either of them.

The main contributions of this papers are:

- We present a measure of interactive application performance which we call "cumulative latency distribution".
- We provide a technique for instrumenting, tracing, analyzing, and visualizing the behavior of interactive applications in order to understand their performance.
- We describe LiLa, our implementation of that technique, that can profile applications written on either of the two dominating Java GUI toolkits.
- We develop a set of GUI microbenchmarks for both the Swing and SWT GUI toolkits.
- We evaluate our LiLa profiler on the microbenchmarks and on complex real-world applications (NetBeans and our LiLa profile viewer itself).

The remainder of this paper is structured as follows: Section 2 presents background on the architecture and the differences between the Swing and the SWT Java GUI toolkits. Section 3 presents our listener latency profiling approach and its implementation in form of the LiLa profiler. Section 4 evaluates our approach using microbenchmarks and real-world applications. Section 5 discusses related work, and Section 6 concludes.

2. SWING VS. SWT

Developers of interactive Java applications are faced with the choice between two powerful GUI toolkits. While Swing is part of the Java platform, and can thus be considered the default Java GUI toolkit, SWT is at the basis of the Eclipse rich client platform, probably the dominant rich client platform for Java, which forms the foundation of the well-known Eclipse IDE. In order to be useful for the majority of GUI applications, we aim at providing a latency profiling approach that works with both of those prevalent GUI toolkits.

In this section we identify the differences between these two toolkits that affect our profiling approach. But before highlighting the differences, we first point out an important commonality: Both toolkits allow only a single thread, the GUI thread, to call into their code[2]. Thus, all GUI activities are essentially single-threaded. The GUI thread runs in a loop (the "event loop"). In each loop iteration it dequeues an event from an event queue and dispatches that event to the toolkit or the application. As part of this dispatching process, listeners may be notified. These invocations of the listeners are the program points we instrument and track with our LiLa profiler.

Besides these commonalities, the Swing and SWT toolkits differ in important architectural aspects. They (1) use different approaches to event handling, they (2) use different approaches to modality[3], and they (3) delegate different kinds of responsibilities to native code. Because these differences affect our profiling approach, we briefly describe how Swing and SWT implement these aspects.

Swing takes on the responsibility of performing the event loop, calling back to the application to handle each individual event. It automatically creates a GUI thread that runs the event loop and thus executes all GUI-related code, including the notifications of all listeners. Swing handles modality automatically, and does so in a "blocking" fashion: when the GUI thread makes a modal component visible (such as a modal dialog that prevents interaction with the underlying windows), that component's setVisible() method does not return until the modal component is made invisible. Finally, Swing reimplements all Swing components on top of "empty" native components, and thus even "built-in" components, such as message boxes or file dialogs are fully implemented in Java and thus instrumentable by our approach.

SWT leaves the implementation of the event loop to the application. A typical SWT event loop looks as follows:

```
while (!shell.isDisposed()) {
  if (!display.readAndDispatch()) {
    display.sleep();
  }
}
```

It loops until the top-level component (shell) gets disposed. In each iteration it reads and dispatches one event from the queue, or it sleeps if the queue is empty, to be woken up when a new event gets enqueued. The application developer decides which thread will be handling the GUI events: The first time a thread enters the GUI toolkit, it is turned into the GUI thread. SWT, like Swing, only supports a single GUI thread, no other thread is allowed to access the GUI. SWT applications may, and indeed often do, contain multiple event loops. Besides the main event loop which often serves the main window of an application, applications may mimic Swing's behavior of "blocking" modal components by placing an event loop right after the (non-blocking) open() method call that opens the modal component. The thread can then remain in that loop until the component is disposed. Finally, SWT uses the built-in components provided by the native platform. This means that components like message boxes or file dialogs do not use Java listeners in their implementation, and, in case these components are modal, implement their own event loop

[2]There are a few exceptions to this rule, most notably Display.asyncExec() in SWT and EventQueue.invokeLater() in Swing that allow other threads to enqueue a Runnable object for later execution in the event thread.

[3]A *modal* dialog is a window that disables user interaction with other windows while it is showing.

in native code. Thus, on the level of Java we cannot instrument their event loop or their listeners to include their behavior in the latency profile.

3. LISTENER LATENCY PROFILING

The goal of our listener latency profiling approach is the determination of (a) the overall characterization of the responsiveness of a given application in the form of a cumulative listener latency distribution, and (b) a listener latency profile that shows the long latency listeners of that application along with information about their context.

Our profiling approach consists of four steps. We (1) instrument the code, (2) run the instrumented application to generate a trace, (3) analyze the trace and build a profile, and (4) visualize the profile to identify perceptible performance bottlenecks. In the following subsections we describe these four steps along with our implementation in form of the LiLa profiling infrastructure.

3.1 Instrumentation

Our instrumentation consists of two parts: instrumenting listener calls and instrumenting the toolkit's event dispatching code. The main goal of our instrumentation is to capture all calls to and returns from listener methods. For this we automatically instrument all corresponding call sites in the application and runtime library classes. Additionally, we manually instrument the Swing and SWT event dispatching code in order to measure how much time was spent handling events *outside* Java listeners. This requires only minimal instrumentation in a single class of Swing resp. SWT.

The purpose of this manual instrumentation is to verify whether measuring the time spent in listeners gives us a representative picture of the overall time spent handling GUI events. Not all GUI events are necessarily handled by listeners, but all GUI events that reach Java are dispatched in a central location. Note that while instrumenting this central dispatch code gives us a more complete picture of the event handling latency, it does not provide us with much information about the code that is responsible for the long latency. Thus, also instrumenting listener calls is of critical importance.

Responsibility of instrumentation. The instrumentations we insert into the existing classes do not have to do much: their only purpose is to call a static method of a class that performs all the work. Below we show the outline of that Profiler class. The dispatchStart() and dispatchEnd() methods are invoked by our manual instrumentation of the event dispatch code. The listenerCall() and listenerReturn() methods are called by the instrumentation that surrounds each listener call. They have two arguments, the first receives the listener object that is the target of the call, and the second receives the name of the method invoked on that object. For example, the two arguments would allow us to determine that PrintAction.actionPerformed() was called, or that GraphCanvas.Highlighter.mouseMoved() returned.

```
class Profiler {
  static void dispatchStart() { ... }
  static void dispatchEnd() { ... }
  static void listenerCall(
    EventListener listener,
    String methodName) { ... }
  static void listenerReturn(
    EventListener listener,
    String methodName) { ... }
}
```

Manual instrumentation of event dispatch. To capture the start and end of event dispatching in Swing, we instrument the dispatchEvent() method of java.awt. EventQueue, surrounding it with calls to the Profiler.

```
protected void dispatchEvent(AWTEvent ev) {
  try {
    Profiler.dispatchStart();
    ... // original code here
  } finally {
    Profiler.dispatchEnd();
  }
}
```

In SWT, we capture event dispatching activity analogously by instrumenting the readAndDispatch() method of org.eclipse.swt.widgets.Display:

```
public boolean readAndDispatch () {
  try {
    Profiler.dispatchStart();
    ... // original code here
  } finally {
    Profiler.dispatchEnd();
  }
}
```

Automatic instrumentation of listener calls. To instrument all listener method invocations we perform an off-line instrumentation pass over all application and Java library classes. We implement our instrumentation using the ASM [11] library to binary rewrite the corresponding Java class files.

While traversing the entire code of a method, we look for bytecode instructions that represent method invocations. Calls to listeners usually correspond to INVOKEINTERFACE bytecode instructions[4]. However, we do not instrument all such method calls, as we are only interested in calls to (and returns from) *listener* methods. Collecting a full method call profile would be prohibitively expensive, and might significantly perturb the resulting measurements. We thus use a simple static analysis to determine whether the target of a call corresponds to a listener, and we only instrument the call if this is the case.

Our static analysis is based on the simple fact that listeners in Swing as well as SWT implement the java.util.

[4]In rare situations (if the static type of the callee is not an interface, but a subclass implementing such an interface) listener calls could also be implemented with INVOKEVIRTUAL. We also instrument such calls if they occur.

EventListener interface. We only instrument the call site if (1) the static type of the call target is a subtype of EventListener and (2) the target method is declared in an interface that extends EventListener. The second clause prevents the instrumentation of calls to non-listener methods of classes implementing EventListener.

We instrument each call site by surrounding it with extra instructions. These instructions prepare the necessary arguments (listener object and method name) and invoke (INVOKESTATIC) the Profiler class' listenerCall() resp. listenerReturn() method. Surrounding each call site with such instrumentation allows the Profiler to determine the start and end of a listener method execution.

3.2 Execution

Once we have instrumented the application and runtime library classes, we can run the application to collect a profile. In order to use the instrumented version of the Swing runtime library, we use the `-Xbootclasspath/p` command line argument to java to prepend the instrumented classes to the boot classpath. Since the SWT toolkit is not part of the Java library, we can just replace the normal SWT JAR file with the instrumented version (e.g. by using the `-classpath` argument to java). We include our Profiler class with the instrumented toolkit classes.

Once we run the application, each listener call and each event dispatch will trigger two notifications to the Profiler, one before and one after the call. Since Java supports multiple threads, and since listeners may be used even outside the single GUI thread, our profiler has to be thread-safe. We thus keep track of information between profile notifications in ThreadLocals.

The profiler is relatively straightforward. It basically writes a trace into a simple text file. Each listener call or return and each start and end of an event dispatch leads to one trace record that we represent as one line in the trace file. Each such line consists of several tab-delimited fields. For each of the four kinds of trace records, we include a field with the trace record type (e.g. "ListenerCall"), the thread that caused that record (using Thread.getId()), and a timestamp (using System.nanoTime()). ListenerCall and ListenerReturn trace records also include the listener's class name and the name of the called method.

Figure 1 shows ten trace records from a trace of our "MouseButton" microbenchmark (Section 4). Each record shows the record type, followed by the thread id and the timestamp in nanoseconds. The first two lines show that an event was dispatched by thread 14, and that handling that event did not involve any listeners (dispatchStart is immediately followed by dispatchEnd). The third line starts another dispatch, which ends with the last line. During the handling of that event, three listeners are notified. First we see a nesting of two listeners: while handling the eventDispatched listener notification, the SelectiveAWTEventListener notifies the LightweightDispatcher listener. Both of these classes are part of the Swing implementation. The third listener call, at the bottom, represents part of the application (as can easily be seen from the package name). It is implemented as an anonymous inner class ($1) in the MouseButtonLatencyCanvas, and the method invoked is mouseReleased. This listener has a particularly high latency of 903 milliseconds, as we can see from the timestamp field. It is these kinds of listener calls we intend to identify with our approach.

3.3 Trace Analysis

The trace analysis converts a trace into a cumulative latency distribution and a listener latency profile. The profile shows, for each listener method, the maximum, average, and minimum latency of all its invocations.

Analyzing the trace is relatively straightforward and is very similar to computing a traditional method profile from a trace of method calls and returns. We compute the latency of each call to a listener by subtracting the timestamp of the ListenerReturn record from the timestamp of the ListenerCall record, and we keep track of the number of calls and their latencies for each listener method.

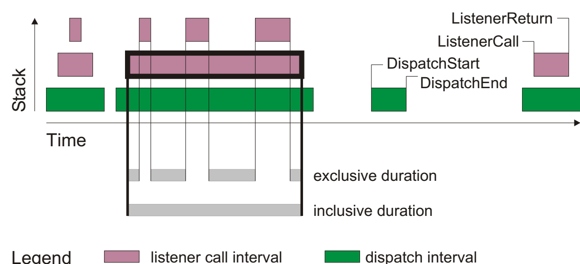

Figure 2: Trace Analysis

A slight complication arises from nested listener calls: the activity happening during a listener call may itself lead to a notification of other listeners. **Figure 2** gives a visual representation of a LiLa trace. Time goes from left to right. The y axis represents nesting (an abstraction of the call stack) of listener calls and/or event dispatches. The first dispatch (without annotations) includes a nested listener call, which itself includes another nested listener call. We will now focus on the listener call (bold outline) within the second dispatch. The annotations in the figure show how we compute its inclusive duration (end minus start timestamp) and its exclusive duration (inclusive duration minus inclusive durations of its children). The figure also shows that traces can contain dispatches that do not contain nested listener calls.

Note that both inclusive and exclusive durations represent important information. The inclusive latency shows how long that listener (and all its nested listeners) took. It is the time noticed by the user. The exclusive

```
...
dispatchStart    14 296380441352951
dispatchEnd      14 296380443289230
dispatchStart    14 296380665901043
 listenerCall    14 296380666001335 java.awt.Toolkit$SelectiveAWTEventListener eventDispatched
  listenerCall   14 296380666030878 java.awt.LightweightDispatcher eventDispatched
  listenerReturn 14 296380666055671 java.awt.LightweightDispatcher eventDispatched
 listenerReturn  14 296380666080605 java.awt.Toolkit$SelectiveAWTEventListener eventDispatched
 listenerCall    14 296380666120135 ch.unisi.inf.performance.test.awt.MouseButtonLatencyCanvas$1 mouseReleased
 listenerReturn  14 296381569377812 ch.unisi.inf.performance.test.awt.MouseButtonLatencyCanvas$1 mouseReleased
dispatchEnd      14 296381569494865
...
```

Figure 1: Segment of a LiLa Trace

time helps to find the listener responsible for the long latency. Usually listeners are losely coupled (their entire purpose is to decouple observers from observables), that is, a listener (e.g. ActionListener.actionPerformed()) that triggers the notification of another listener (e.g. DocumentListener.documentSaved()) usually is not aware of who its nested listener is, and what it does. Blaming a listener for the latency caused by nested listeners may thus be wrong.

Modal phase injection. Unfortunately, the above analysis is incomplete, as it does not address the problem of modality. If a listener opens a modal dialog, the above approach may attribute the entire period where the modal dialog was open (and the user interacted with it) to the latency of that listener. This clearly is wrong.

Figure 3 shows, on the left side, an example of such a problem trace. The annotations show when the user opened a modal dialog, when the dialog actually appeared on the screen, and for how long the dialog stayed open. The trace shows that the listener that was responsible for opening that dialog does not return until after the dialog is closed. This is because modal dialogs contain their own event loop, and the thread opening the dialog enters that nested event loop when the dialog is shown. It only returns after the dialog disappears.

This is a problem for our accounting, as we will charge that listener (resp. its inclusive and exclusive time) for all the time the dialog was showing. That includes all the user's thinking time in that interval (highlighted in Figure 3 with surrounding ovals). The listener should only be accountable for the time it took to open up (and maybe close) the dialog, but not for the time during which the user interacted with it. We want to exclude such periods of modal activity. However, detecting modal activity with instrumentation (e.g. to capture when a modal dialog appears and disappears) is complicated, and in some cases is impossible on the Java level. We thus use a simple heuristic to detect such nested modal periods. This heuristic works with just the information available in our traces. It makes use of the fact that during modal dialogs we encounter dispatch intervals nested within the listener interval that opened the dialog.

Figure 3 shows how we use our heuristic to transform the original trace into a trace including a new type of intervals: a modal phase. We call this transformation "modal phase injection". The left side of the figure shows the trace before phase injection. The right side shows the result of phase injection, consisting of the same trace with one additional interval injected between the listener call interval and its nested dispatch intervals. This modal phase interval represents the interval during which the modal dialog blocked our listener, starting with the start of the first nested dispatch interval and ending with the last nested dispatch interval.

After modal phase injection we can compute the listener call duration in a new way: we compute *end-to-end duration* as the end minus start timestamp, *inclusive duration* as end-to-end duration minus the duration of children that are modal phases, and *exclusive duration* as end-to-end duration minus the duration of all children.

Cumulative latency distribution. Given the transformed trace we compute the cumulative latency distribution. This distribution represents how long the user had to wait for a response to a user action. We compute it by measuring the latencies of all episodes. An episode corresponds to an event dispatch interval or to a listener call interval. It either is at the very bottom of the stack (in the trace), thus representing the handling of an event in the main window, or it is a child of a modal phase, representing the handling of an event in a modal dialog. The latency distribution (a histogram) simply consists of the episode counts binned by inclusive latency (e.g. 30 episodes with an inclusive duration of 100ms, 40 episodes with 110ms, ...). We get the *cumulative* distribution from the basic distribution by adding all counts of episodes with a latency as long or longer than the given bin.

The cumulative latency distribution allows us to quickly see how responsive an application is: We want to have few episodes with a latency greater than 100 milliseconds, and even fewer episodes longer than 1 second.

Listener latency profile. The listener latency profile allows a developer to find the causes of long-latency episodes. It groups all listener call intervals by their respective listener class and method name. Thus, all calls to a given listener method (e.g. to ReSortAction.actionPerfomed()) are grouped together. For each such group we determine the maximum, average, and

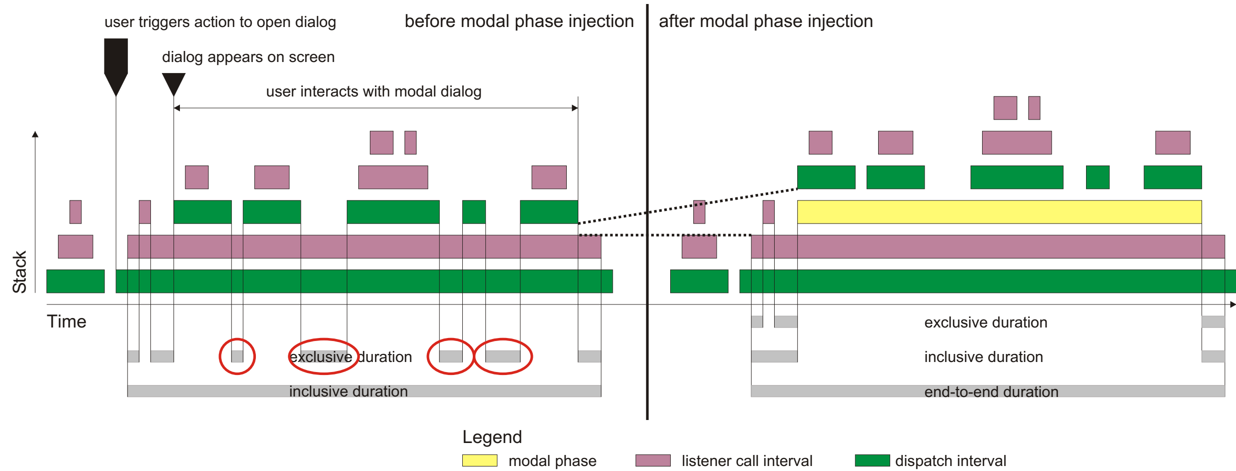

Figure 3: Trace Analysis and Injection of Modal Phases

minimum inclusive and exclusive latency, as well as the number of listener calls of that kind.

3.4 Visualization

In our implementation, LiLa, analysis and visualization happen in the same program, the LiLa Viewer. The LiLa Viewer analyzes the trace when the user opens the trace file. It then provides a multitude of interacting visualizations of the trace, the latency distribution, and the listener latency profile, in graphical, tabular, and tree form. **Figure 4** shows the LiLa Viewer visualizing a listener latency profile of a 207-second run of a JHotDraw application. The viewer consists of four panels. For easier reference we have numbered these panels in the figure from top (1) to bottom (4) on their right side.

The "Latency Distribution" (1) panel presents the cumulative latency distribution graph. The x-axis shows the latency, from 0 to 10 seconds. The y-axis represents the number of episodes. A point (x, y) on the curve shows that y episodes took x seconds or longer. This graph gives an overview of the perceptible performance of the program run. It includes dark vertical lines at 100 ms and 1 second, indicating Shneiderman's latency limits. This graph gives a quick overview of the interactive performance of the application. A responsive application would lead to an L-shaped curve, where all episodes are below 100 ms. If episodes take longer than 100 ms, or even longer than 1 second, this is a reason for performance tuning. The developer can use the remaining panels of the LiLa Viewer to identify such long-latency episodes. As a first step, the developer can use the cumulative latency distribution graph as a filter, by clicking on it to set a latency threshold. This will focus the remaining panels on the episodes that are longer than the given threshold.

The "Trace Timeline" (2) panel gives an overview of the entire trace. The x-axis corresponds to time, from the start to the end of the program run. The small strip at the top of the panel shows all episodes, and episodes longer than the chosen latency threshold are highlighted. This allows a developer to quickly determine when in the program execution the long-latency episodes took place. Below that strip is a collection of tabs (three tabs in our case), one tab for each Java thread that executed any listener code. Usually the GUI thread is responsible for most calls to listeners, but some applications also call listeners from within other threads. Each thread tab shows the timeline of that thread. As in the strip above, the x-axis corresponds to the entire execution time. The y-axis corresponds to the call stack of the thread. Listener calls can be nested, and this nesting is visually indicated by stacking these calls. The outermost call appears at the bottom, while nested calls are stacked on top.

The "Intervals" (3) panel provides three tabs with different tabular presentations of the individual intervals in the trace. The most important tab, "Listener Profile", presents the actual profile: The top table contains one row per listener method. Each row shows the name of the listener (e.g. the highlighted row shows org.jhotdraw.util.CommandMenu) and its method (e.g. actionPerformed), how many times the method got called (32), as well as the maximum (388 ms), average (29 ms), and minimum (0 ms) latency of these calls. The latency is shown as inclusive (including the latency of nested listener calls) or exclusive (excluding the time spent in nested listener calls). All tables in the LiLa Viewer are sortable by any of the columns. The table shown is sorted by the maximum exclusive lateny. This leads to the most costly listeners appearing at the top, allowing the developer to quickly focus on the problem cases. The bottom table shows one row per invocation of the listener chosen in the top table (in the example, it contains 32 rows). This allows the investigation of how the listener's behavior varies across different invocations.

The bottom panel, "Selected Interval" (4), shows de-

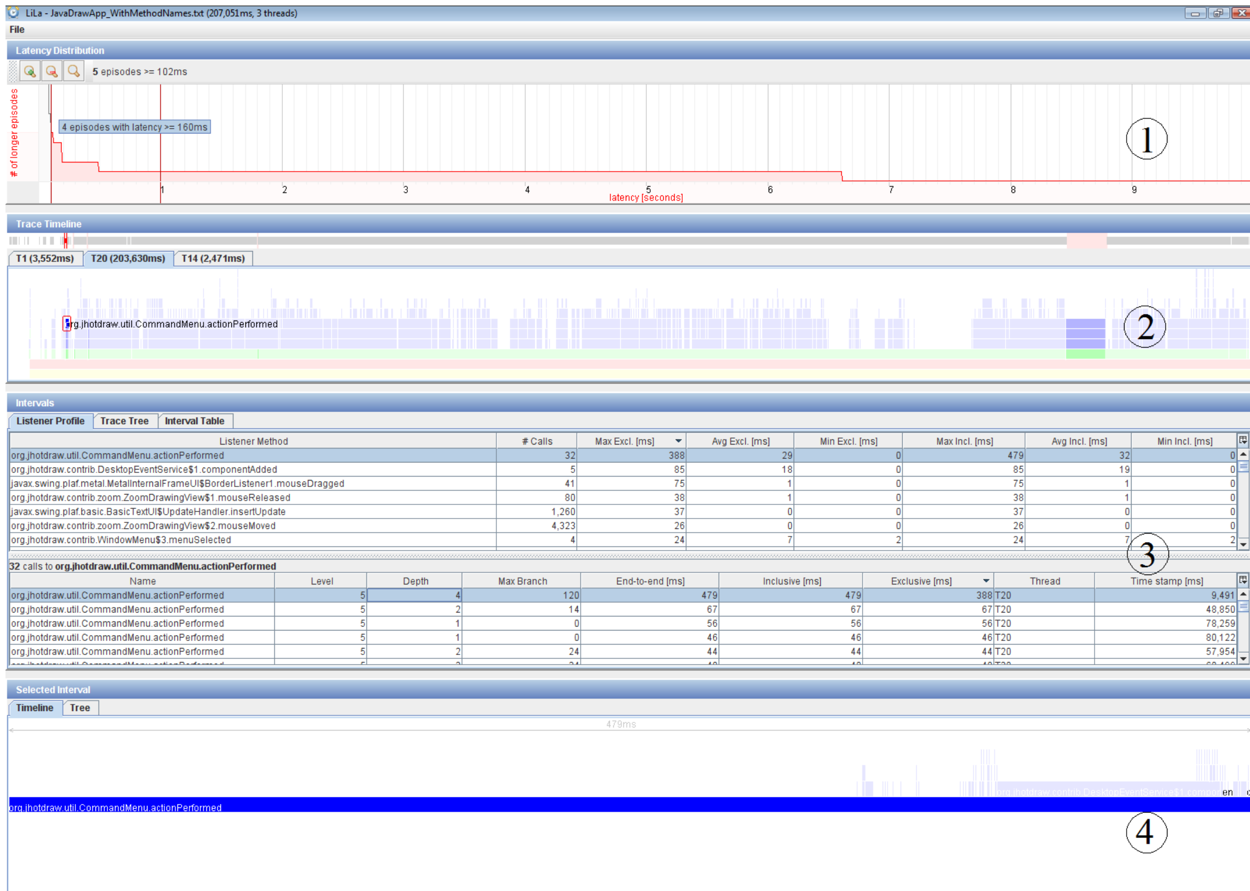

Figure 4: LiLa Viewer

tails about the interval that the user selected in the "Trace Timeline" or "Intervals" panel. These details can be shown as a timeline zoomed in to just that interval (as seen in the figure), or as a tree showing the nested listener calls with all their information. This detailed information is important for finding the cause of the long latency.

4. VALIDATION

In this section we validate LiLa on a set of microbenchmarks and then evalaute LiLa for profiling real-world applications.

4.1 Micro-Benchmarks

To determine the generality of our profiling approach, we implemented a set of validation benchmarks on top of both Swing and SWT. Note that it is not the goal of this evaluation to compare the performance of the Swing or SWT toolkits, nor of their respective widgets. The goal of this validation is to determine the limitations of our profiling approach in measuring the responsiveness of applications in all situations on both frameworks. In particular, we are interested to determine whether (and where) LiLa mistakenly reports a short activity as a long latency listener call, or where it is unable to capture and report long-latency activities.

Table 1 shows the different microbenchmarks. Each row shows one benchmark. The first column lists the benchmark name, while the second and third columns present the features covered by that benchmark in Swing resp. SWT terminology. The different microbenchmarks register different kinds of listeners, and the listeners simulate expensive computations by waiting for a specifiable delay (using a call to Thread.sleep()). We expect any delays occuring in these listeners to directly appear in the LiLa listener latency profile.

Handling keyboard and mouse events. The first three benchmarks test listeners that handle mouse or keyboard input. Such listeners are common in custom components, and they may involve significant computations (e.g. to find an item in a complex data model that corresponds to the coordinates at which the mouse was clicked) causing a perceptible delay. Our benchmarks involve a custom component with listeners registered for the respective mouse or keyboard events.

As expected, studying the traces from these bench-

Benchmark	Swing Feature	SWT Feature
Keyboard input	KeyListener	KeyListener
Mouse motion	MouseMotionListener	MouseMoveListener
Mouse button	MouseListener	MouseListener
Painting	JComponent.paintComponent()	PaintListener
Timer	Timer/ActionListener	Display.timerExec()/Runnable
Menu item	JMenuItem/ActionListener	MenuItem/SelectionListener
Combo box	JComboBox/ItemListener	Combo/SelectionListener
Two frames	JFrame	Shell
Non-modal dialog	JDialog/ModalityType.MODELESS	Shell/SWT.MODELESS
Modal dialog	JDialog/ModalityType.APPLICATION_MODAL	Shell/SWT.APPLICATION_MODAL
Built-in dialog	JOptionPane.showMessageDialog()	MessageBox/SWT.APPLICATION_MODAL

Table 1: Validation Micro-Benchmarks for Swing and SWT

marks with LiLa we find that LiLa points out the long-latency listeners at the very top of its profile.

Painting. The fourth benchmark covers painting. Painting a custom component can be expensive, especially if the component renders a complex visualization based on a large data model (such as the LiLa Viewer's Trace Timeline).

This benchmark demonstrates that, unlike in SWT, painting a custom component in Swing is not done by registering a listener, but by overriding the paintComponent() method. Correspondingly, LiLa does not report any long-latency listeners in its profile. It does, however, show the long-latency event dispatches with the expected latency in its "Intervals" table[5]. For the SWT version of this benchmark, LiLa shows all the paintControl() notifications as expected.

Handling timers. The fifth benchmark is concerned with reacting to timer events. Timers are often used for animations, where a component is periodically repainted. Both frameworks provide support for setting up timers that periodically invoke a method in the event handling thread.

The LiLa profile for SWT does not show any long latency listener invocations, but only shows long-latency nondescript event dispatches. This is because SWT does not use listeners for handling timer firings, but uses a Runnable instead[6]. As expected, the Swing version of this benchmark shows all the actionPerformed() notifications corresponding to the timer firings.

Popups. The sixth and seventh benchmarks are concerned with widgets that possibly pop up a modal window. A combo box can activate a drop-down list, and a menu item can activate a pull-down menu. If these popups are modal and block the event handling thread while that thread is inside a listener, they erroneously inflate the lateny of that listener, a problem that our modal phase injection might fix.

The LiLa traces for Swing's JComboBox and JMenu show that they are not modal and do not block the event handling thread, thus their use does not introduce any error into our profiles. SWT's Menu and Combo are modal, but they are not opened from within a Java listener, and so do not introduce an error into the listener profile either. However, the Menu blocks the event handling thread, causing one single long dispatch interval (for the duration the menu is open). As we have described in Section 3.3, the episode used for the latency distribution include dispatch intervals. Thus, opening Menus in SWT negatively affects the cumulative latency distribution (leading to a erroneously high count of long episodes).

Modality. The last four benchmarks constitute tests for examining different modalities of top-level components (Swing JFrame and JDialog vs. SWT Shell).

Using two main windows (JFrame or non-modal Shell) or opening a non-modal dialog, does not in any way affect us, as there is no modal activity at all. While opening a Swing JOptionPane, modal JDialog, or a modal SWT Shell with its own separate event loop, blocks the thread, our modal phase injection corrects for this, and keeps the listener latency profile and the latency distribution intact. The remaining benchmark, the SWT MessageBox, however, cannot be detected using our heuristic. When we open it from within a listener (as is normal), it blocks the thread and does not cause nested event dispatches visible within Java. This affects the listener latency profile (our listener is charged for the entire duration the user looked at the message box), as well as the latency distribution (adding one overly long episode).

Accuracy. In all of our micro-benchmarks we deliberately introduced latencies between 10 ms and 1 second by calling Thread.sleep(). We thus approximately [7] know the actual latency. We have found that the latency reported by LiLa and the requested sleeping time usually differ by less than 3 ms. This accuracy is sufficient for our purposes, as we are interested in finding episodes with latencies significantly longer than 100 ms.

Summary. In conclusion, the listener latency profile computed by LiLa provides detailed information about most of the important interactive events. It lacks de-

[5]We could instrument calls to paintComponent() to also enable the profiling of painting code in Swing.

[6]It would be possible to instrument calls by a timer to a Runnable to measure that time.

[7]Thread.sleep() is not completely accurate, and listeners also execute a minimal amount of code, such as the call to Thread.sleep(), besides sleeping.

tailed information about Swing painting and SWT timers, but still captures the latency of such events because it reports all event dispatches, no matter whether they include listener notifications or not. Our heuristic to infer intervals of modal activity is successful, but it is not able to determine such periods when they happen entirely in native code (such as when showing an SWT MessageBox). We thus can mistakenly attribute the latency of showing a message box to the listener that opened up that dialog. Since showing a message box is a relatively rare event, and since it usually takes a user several seconds before she dismisses a message box, such long latency listener calls are easy to spot and skip by the performance analyst. Moreover, SWT dialogs implemented in Java are properly handled by our heuristic.

4.2 Real-World Applications

In this subsection we evaluate the applicability of our profiler to real-world applications. We picked our own LiLa Viewer and the NetBeans IDE, which is one of the most complex open-source interactive Java applications.

Overhead. The slowdown due to our instrumentation is hard to quantify, because we would need to measure the performance of the uninstrumented application. While we could measure the end-to-end execution time of the uninstrumented application, this would be meaningless (we argue in this paper that the end-to-end execution time of an interactive application is not a useful measure of its performance). What we really would want to compare is the listener latency distribution of the instrumented and the uninstrumented application. Unfortunately we cannot measure the listener latency distribution *without* our instrumentation, and thus we cannot precisely evaluate the impact of our instrumentation on perceptible performance.

However, the evaluation of the accuracy with our microbenchmarks suggests that for these programs the instrumentation does not significantly affect performance. Moreover, we measured the rate at which the instrumented real-world applications produced traces to quantify one relevant aspect of the overhead. For the 18 runs we captured (9 of NetBeans, 9 of the LiLa Viewer), we observed data rates from 100 kB/s to 1.4 MB/s. Note that we could reduce these rates, according to our experiments by about a factor of 15, by compressing the traces before writing them to disk. In future work we plan to evaluate the overhead of LiLa with a study of real users, and to use online analysis approaches to further reduce the overhead where necessaray.

Results. **Figure 5** shows the perceptible performance of NetBeans and our LiLa Viewer. It includes three curves for each benchmark[8]. The x-axis corresponds to the latency, and the y-axis corresponds to the number of episodes $\geq$ the given latency, *per second of working time.* Working time is the time spent within

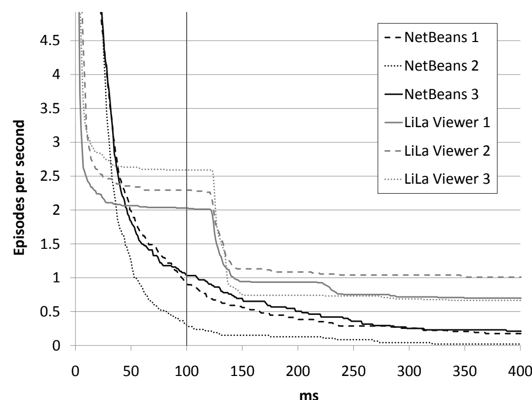

Figure 5: Cumulative Latency Distribution

episodes. We normalize by working time to be able to compare different distributions. We chose working time (instead e.g. end-to-end trace duration) for fairness: it would be unfair to divide by a long end-to-end time, if most of that time the system was idle, waiting for a user request. We notice that NetBeans has a better performance than LiLa: Every second LiLa causes between 2 and 2.5 episodes that are longer than the 100 ms perceptibility threshold. NetBeans, on the other hand, only causes between 0.3 and 1 such episodes. These results confirm our expectations: The professional-grade NetBeans application, which had been especially optimized for interactive performance, outperforms our prototype-quality LiLa Viewer. LiLa pointed out several causes for the inferior performance of the LiLa Viewer; the most significant was a listener reacting to changes in a table model. The listener was sorting the rows whenever the underlying model changed, which was expensive with large models. Given this information, we can now optimize this task to improve the perceptible performance of the LiLa Viewer.

5. RELATED WORK

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. [4] 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.

[8]For clarity we only include three of the nine runs for each benchmark in Figure 5. The other runs have similar curves.

Zeldovich et al. [16] devised a different non-intrusive approach to measure response times of interactive applications: they instrument VNC [12], 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. [5] 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.

Instrumentation-based approaches to measure latency are generally used ad-hoc, by developers of specific applications. For example, the Eclipse platform [7] provides a minimal infrastructure for measuring the time spent in certain regions of code [6]. Developers of applications for that platform can instrument their event-handling code with calls to that infrastructure to collect the latency due to their event handlers.

TIPME [3] is a system for finding the causes of long latency that lie in the operating system. TIPME runs on BSD Unix using an X server. It traces X events and scheduling and I/O activity in the operating system. Users can press a hot key to indicate irritatingly high latency, causing TIPME to dump the trace buffers for later manual causal analysis.

6. CONCLUSIONS

In this paper we introduced listener latency profiling, an approach for profiling the perceptible performance of interactive Java applications. Our approach is applicable for both Swing and SWT applications. We have presented the cumulative latency distribution as a means to characterize performance, and the listener latency profile as a means to highlight the components responsible for sluggish application behavior. We have implemented our idea for Swing and SWT platforms and validated it on a new suite of microbenchmarks. Finally, we have shown that our approach works with all of the complexities of real-world Java GUI applications by using it to profile our own LiLa profile viewer as well as the professional-grade NetBeans IDE.

We intend to improve the usability of LiLa and to release[9] it open source, so Java developers can start to use it for understanding the perceptible performance of their applications.

[9] `http://www.inf.unisi.ch/phd/jovic/MilanJovic/Lila/Lila.html`

7. ACKNOWLEDGEMENTS

This work has been conducted in the context of the Binary Translation and Virtualization cluster of the EU HiPEAC Network of Excellence. It has been funded by the Swiss National Science Foundation under grant number 200021-116333/1.

8. REFERENCES

[1] 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.

[2] J. R. Dabrowski and E. V. Munson. Is 100 milliseconds too fast? In *CHI '01: CHI '01 extended abstracts on Human factors in computing systems*, 2001.

[3] Y. Endo and M. Seltzer. Improving interactive performance using TIPME. In *Proceedings of the SIGMETRICS international conference on Measurement and modeling of computer systems*, 2000.

[4] Y. Endo, Z. Wang, J. B. Chen, and M. Seltzer. Using latency to evaluate interactive system performance. In *OSDI '96: Proceedings of the second USENIX symposium on Operating systems design and implementation*, 1996.

[5] K. Flautner, R. Uhlig, S. Reinhardt, and T. Mudge. Thread-level parallelism and interactive performance of desktop applications. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, 2000.

[6] Eclipse Foundation. The PerformanceStats API: Gathering performance statistics in Eclipse. http://www.eclipse.org/eclipse/platform-core/documents/3.1/perf_stats.html, 2005.

[7] Eclipse Foundation. Eclipse Project. http://www.eclipse.org/eclipse/, 2006.

[8] I. S. MacKenzie and C. Ware. Lag as a determinant of human performance in interactive systems. In *CHI '93: Proceedings of the SIGCHI conference on Human factors in computing systems*, 1993.

[9] J. McAffer and J.-M. Lemieux. *Eclipse Rich Client Platform : Designing, Coding, and Packaging Java Applications*. Addison-Wesley, 2005.

[10] S. Northover and M. Wilson. *SWT: The Standard Widget Toolkit, Volume 1*. Addison-Wesley, 2004.

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

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

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

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

[15] K. Walrath, M. Campione, A. Huml, and S. Zakhour. *The JFC Swing Tutorial: A Guide to Constructing GUIs, 2nd Edition*. Addison-Wesley, 2004.

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