LagAlyzer: A latency profile analysis and visualization tool

Andrea Adamoli
University of Lugano
Lugano, Switzerland
Email: andrea.adamoli@usi.ch

Milan Jovic
University of Lugano
Lugano, Switzerland
Email: milan.jovic@usi.ch

Matthias Hauswirth
University of Lugano
Lugano, Switzerland
Email: matthias.hauswirth@unisi.ch

***Abstract*—Many computer systems are interactive in some way, that means they are used by human users. A human user perceives the performance of a computer system primarily in terms of its response time. If a system does not respond to a user's input, such as a key press, a mouse motion, or a gesture on a touch screen, within roughly 100 ms, a user perceives the system as sluggish.**

Developers of interactive software and systems thus are interested in keeping response times below this perceptibility threshold. Existing tools allow the measurement of interactive response times, and the tracing of application behavior. In this paper we present a tool, LagAlyzer, which analyzes and visualizes the information gathered by such latency measurement tools. LagAlyzer enables the characterization of perceptible lag, for example by quantifying to what degree perceptible performance was caused by synchronization bottlenecks, by garbage collection, by the runtime libraries, or by the application. We use LagAlyzer to characterize the perceptible latency found in commonly used interactive Java applications. We believe that this is the first study giving insight into why interactive Java applications sometimes are perceived as sluggish.

I. Introduction

Interactive GUI applications are prevalent. Besides the traditional productivity applications running on personal computers, they span the entire spectrum from powerful special-purpose scientific visualization tools all the way to applications running on mobile devices. Given the software engineering benefits of managed languages, many of these applications are running on managed runtime environments such as Java.

Researchers in human-computer interaction have long shown that the aspect of performance that matters most for interactive applications is the lag perceptible by users. Shneiderman [11] showed that the lag in handling user events can induce frustration, annoyance, and anger, and that it significantly affects user productivity [10], and he finds that a lag beyond 100 ms is perceptible by users. MacKenzie and Ware [7] find that user performance in virtual reality environments significantly decreased as they increased lag up to 225 ms, and Dabrowski and Munson [1] find that users perceive lag longer than 150 ms for keyboard input and longer than 195 ms for mouse input.

In this paper we present LagAlyzer, a tool to analyze and visualize the causes of lag in interactive Java applications. LagAlyzer operates offline on traces gathered by existing latency profilers such as LiLa [6]. The analyses performed by LagAlyzer enable us to characterize the reasons for perceptible performance problems of Java applications. Our study involves 14 common open-source Java applications, ranging from small, focused programs such as a crossword puzzle editor, all the way to the most complex Java applications, such as the NetBeans IDE with its 45000 classes.

The remainder of this paper is structured as follows: Section II introduces LagAlyzer, our analysis and visualization tool. Section III presents the experimental methodology of our characterization study, and Section IV presents the results. Section V discusses threats and limitations of our approach, Section VI surveys related work, and Section VII concludes.

II. LagAlyzer

LagAlyzer is based on the concept of an *episode* [6], a time interval from the point a user request is dispatched until the point the request is completed. Episodes that take longer than a given threshold (100 ms in this paper) are considered *perceptible*, and thus have a negative impact on perceived performance.

A. Input to LagAlyzer

LagAlyzer is not a profiler. It neither instruments applications nor does it capture application behavior. LagAlyzer is a tool to visualize and analyze traces produced by other tools. LagAlyzer is an offline tool: it requires the complete traces to exist before it can start to analyze and visualize them. In this paper we use an extended version of LiLa [6], an existing listener latency profiler, to produce traces of user sessions with interactive applications. Besides information about the start and end of each episode, the traces LiLa provides to LagAlyzer also contain information about system behavior throughout the episode. In particular, the traces contain:

Listener notifications. Traces contain the call and return to each listener. Listener calls correspond to the handling of user input (e.g. mouse and keyboard activity).

Graphics rendering. Traces contain the call and return to the methods responsible for painting GUI components. These calls are responsible for output of the interactive application (to the screen).

Native calls. Traces contain all JNI calls and returns. This allows us to determine whether the lag was induced by Java code or by native libraries. All interactive Java applications use native libraries, because at a minimum they indirectly invoke native functions in the GUI toolkit.

Background-thread event dispatches. Traces contain the start and end of all dispatches of GUI events posted by background threads. Interactive applications use background threads for various purposes, for example to implement timers used to periodically recompute an animation, for reacting to information received over the network, or for performing long-running computations without blocking the user interface. Background threads update the user interface by posting an event to the event queue, and we track the processing of those events.

Garbage collection. Traces contain the start and end time of all garbage collections. This allows us to determine to what degree garbage collection, which in our experiments is performed by a stop-the-world collector, affects perceptible performance.

Call stack samples of all threads. Traces contain periodically captured call stacks of all threads. This enables us to estimate how much time in long latency episodes was spent in the application versus the runtime system. Moreover, it also provides insight into concurrency and synchronization issues without requiring heavy-weight instrumentation.

The above information can be grouped into two categories: intervals and events. Intervals correspond to activities that occurred over an extended period of time, and where the trace contains start and end time stamps. Events correspond to activities that essentially completed instantaneously. Except for the last item (call stack samples) in the above list, all information is modeled as intervals. **Table I** shows the types of intervals and the names we use to refer to them in this paper.

Name	Description
Dispatch	Start to end of a given episode
Listener	A listener notification call
Paint	A graphics rendering operation
Native	A JNI native call
Async	The handling of an event posted in a background thread
GC	A garbage collection

TABLE I
INTERVAL TYPES

LagAlyzer represents the activity of each thread as a tree of nested intervals. Given the type of intervals we collect, we can guarantee that the intervals of a given thread are properly nested (i.e. intervals either are nested or do not overlap at all). This is the case because all interval types except for garbage collection correspond to method calls and returns, and method calls and returns are properly nested for any given thread. Note that the instrumentation in the measurement tool we use maintains that property also in the presence of exceptions. Garbage collection intervals also are nested, because, in order for a garbage collection to start, all threads have to reach a safe point, and they are only released from that safe point after the garbage collection finished. Note that, because a GC stops all threads, for a given garbage collection we add a separate copy of the GC interval to the interval trees of each thread.

The core of LagAlyzer consists of an in-memory representation of the latency traces described above. This core provides the basis for the visualizations and analyses we present next. Developers who want to write their own analysis can implement it using the straightforward API provided by the core.

B. Episode Sketch

LagAlyzer visualizes individual episodes using an *episode sketch*. Episode sketches are an extension of the *trace timeline* visualizations implemented in LiLa Viewer [6]. **Figure 1** shows an example episode sketch. Like a trace timeline, an episode sketch represents all the information available about a given episode along a time axis. The sketch has three parts. (1) The *time axis* at the bottom of the figure shows the time in the execution of the application when this episode occurred. (2) Above the time axis, LagAlyzer shows the *tree* structure of the different intervals. LagAlyzer uses the different interval types shown in Table I and renders each interval type in a different color. (3) Unlike LiLa Viewer's trace timelines, LagAlyzer's episode sketches also integrate a visualization of *call stack samples* that were taken of the GUI thread during the episode. Each sample is represented by a point colored according to the thread state. Those points are shown along the top edge of the figure. A user can hover over any point to see the complete call stack trace and thread state information of that sample.

An episode sketch's combination of stack samples with the trace timeline provides previously missing information to the developer. It provides answers to questions like whether, during perceptibly slow episodes, the GUI thread was mostly runnable, blocked, waiting, or sleeping, what fraction of time the thread spent in native or in Java code, and what fraction of time it was executing in the runtime library or in the application. It is exactly these questions that we want to answer in the characterization study in Section IV of this paper.

The episode shown in Figure 1, represented by the "dispatch" interval at the bottom of the interval tree, took 1705 ms and is thus clearly perceptible by the user. The visualization shows that the entire duration of that episode is attributable to a call to the JFrame.paint method, which called JRootPane.paint. This call then lead to JLayeredPane.paint (which took 1533 ms), all the way to JToolBar.paint (1347 ms). During that rendering of the toolbar, roughly in the center of the episode, is a 843 ms long interval representing a native call to the DrawLine function of the sun.java2d.loops.DrawLine class. While this long call indicates a potential inefficiency in the native code for drawing lines, the interval tree shows a further interval nested inside this native call. That 466 ms interval represents a garbage collection.

The episode sketch shows that during the garbage collection no call stack samples were taken (there are no squares at the top of the visualization during the garbage collection interval). This is because during GC the JVMTI-based call stack sampling, like any other mutator activity, is stopped. A closer look at the figure shows that sampling activity actually is stopped for almost the entire duration of the native call, not just during the garbage collection interval. We were initially surprised about this behavior. However, the wording in the JVMTI specification [8] of the "Garbage Collection Start" and "Garbage Collection End" events indicates that the notifications of the begin and end of garbage collection (which our tracing tool uses) bracket only the phase where all threads are stopped and collection takes place. Thus the GC interval does not cover

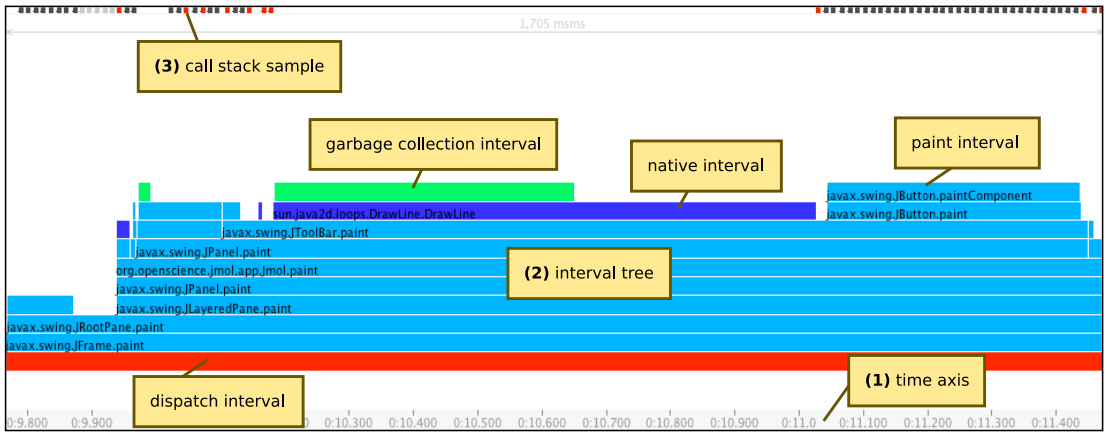

Fig. 1. Episode sketch: temporal visualization of activity during an episode

the time where *some* threads are stopped while others still are running towards (or already are running away from) their safe point before (or after) the actual collection. As a result, the behavior we observe in the figure most probably shows that the GUI thread still was at the safe point, waiting to get its first time slice. That also would imply that the native method is not to blame for the lag, because it only got to run shortly before it completed.

C. Episode Classification

Note that the fact that in the above example the GC is triggered while the thread is inside the native method does not mean that the native method is responsible for that GC. It could be that prior code executed in this thread, or code executed in different threads, was responsible for most of the memory allocations that ultimately triggered the collection. Moreover, it could also be that background threads monopolized the CPU during the given episode, such that the GUI thread, while runnable, was not actually executing for a while. Finally, it could be that threads in other processes, outside the monitored Java virtual machine, where using the CPU.

For all these reasons, looking at an individual episode is not usually enough to determine the cause of long latency. LagAlyzer thus groups episodes into equivalence classes according to the structure of their interval trees. If an equivalence class contains many episodes, and if all episodes in that equivalence class exhibit significant lag, then this equivalence class represents a common *pattern* of bad performance. It is these patterns that developers should target when optimizing the performance of their interactive applications. Moreover, by grouping equivalent episodes into patterns, LagAlyzer can present a developer with a small number of patterns, and the developer does not need to look at the complete set of episodes.

D. Episode Patterns

We define patterns (equivalence classes on episodes) based on the structure of the episode's interval tree. By structure we mean the type of interval and its symbolic information (e.g. the class and method names of a listener interval).

In comparing two trees, we exclude the GC nodes. This is because, as stated above, garbage collection may or may not be blamed to the interval surrounding the GC interval. Thus, by ignoring the GC intervals, a developer can determine whether a given equivalence class always or rarely contains GC intervals. If it always contains GC intervals, then the developer may want to investigate the cause of the GC, and reduce the amount of allocation to reduce the number of GC intervals.

Moreover, we also exclude timing information in the tree comparisons. That is, two structurally equivalent episodes are in the same equivalence class even if their durations differ greatly. This allows us to find equivalence classes where all episodes are perceptible, or classes where only one or a few episodes are perceptible. Equivalence classes where all episodes are perceptible are interesting for developers. Equivalence classes with only few perceptible episodes are less interesting, because the few perceptible episodes might be slow due to unrelated background activity. Equivalence classes with only one perceptible episode, especially if that episode is the first episode of that class, can indicate that there was some initialization activity (e.g. class loading) that slowed down the first episode.

E. Pattern Browser

LagAlyzer presents the user with a table of patterns. For each pattern, it shows the number of episodes and the minimum, average, maximum, and total lag encountered over all the pattern's episodes. The developer can filter the pattern table by eliding any patterns that do not have any perceptible episodes. By selecting a pattern in the table, the developer can reveal a list of all the episodes in that pattern as well as an episode sketch of the first episode of that list. The developer can then browse through the sketches of all episodes in the pattern to get a quick grasp of the timing variations between episodes.

III. Methodology

To characterize the perceptible performance of interactive applications, we chose a suite of real-world Java applications. We ran these applications under an extended version of LiLa to gather latency profiles. We then used LagAlyzer to perform the offline analyses needed to characterize the applications based on the recorded latency profiles.

A. Platform

We ran all our experiments on MacBook Pro computers with Core 2 Duo processors with 4 MB of cache, 2 GB of RAM, and an NVidia 128-bit built-in graphics card with 128 MB of RAM. We ran the benchmarks on Mac OS X 10.5 Leopard, using the 64-bit version of Java 1.6.0_15 in server mode[1].

B. Applications

We selected the 14 Swing applications shown in **Table II**. They represent a diverse selection of popular open-source Java GUI programs, spanning the range from small and focused tools (like CrosswordSage) to large framework-based applications (like NetBeans).

Application	Version	Classes	Description
Arabeske	2.0.1	222	Arabeske texture editor
ArgoUML	0.28	5349	UML CASE tool
CrosswordSage	0.3.5	34	Crossword puzzle editor
Euclide	0.5.2	398	Geometry construction kit
FindBugs [4]	1.3.8	3698	Bug browser
FreeMind	0.8.1	1909	Mind mapping editor
GanttProject	2.0.9	5288	Gantt chart editor
JEdit	4.3pre16	1150	Programmer's text editor
JFreeChart (Time)	1.0.13	1667	Chart library (time data)
JHotDraw (Draw)	7.1	1146	Vector graphics editor
Jmol	11.6.21	1422	Chemical structure viewer
Laoe	0.6.03	688	Audio sample editor
NetBeans (Java SE)	6.7	45367	Development environment
SwingSet	2	131	Swing component demo

TABLE II
APPLICATIONS

C. Interactive Sessions

We manually performed four similar sessions with each interactive application. We planned the sessions to cover a reasonable and realistic usage scenario for each application. On average each session took roughly 8 minutes. For some applications (e.g. JFreeChart) our sessions lasted less than 8 minutes because the limited functionality of the specific applications did not allow for longer sessions without excessively repeating the same activity.

IV. Lag Characterization

In this section we introduce our approach to characterizing the perceptible lag in interactive applications and we use our approach to characterize a set of real-world Java applications. The fully automated analysis of about 7.5 hours of interactive sessions (roughly 250'000 episodes) took 15 minutes (including the generation of MATLAB graphs).

A. Overview

Table III provides an overview of our 14 applications: it contains one row per benchmark application and ends with a row with the mean across all applications. Each row represents the average over the four interactive sessions we performed with each application. The table consists of three blocks: the name of the benchmark, a quantification of session duration ("Exp. time"), a characterization of episode duration ("Episodes"), and a characterization of patterns mined from the episodes ("Patterns").

Session duration. The "E2E" column shows the end-to-end execution time, this is the time from the start to the end of a session. We see that the average session durations range between 250 seconds and 630 seconds. These raw end-to-end times, however, do not quantify to what degree we exercised the application: they include all the user think time (potentially we could have started the application, walked away from the computer, and come back after 8 minutes). Column "In-Eps" thus shows the percentage of time spent in episodes (the total time the system spent handling user requests). This value ranges from 8% to 47%. In most interactive applications, a user is unable to keep the system busy at all times. For example, typing at a rate of 100 characters per second would leave the system with 10 ms to process each entered character – if processing a character only takes 1 ms, then the system is idle 90% of the time. We would thus observe an in-episode time of 10% even if the user was continuously typing.

Episodes. To reduce measurement overhead and perturbation, our tracing tool automatically filters out episodes that are shorter than 3 ms. LagAlyzer thus never gets to see such episodes, it only is able to see how many such short episodes occurred. We show this number in column "$<$ 3ms". The large counts in this column, up to over 1.2 million for Laoe, show that most episodes are short (and thus uninteresting for understanding perceptible lag). Column "$\geq$ 3ms" shows the number of episodes, between 1173 and 9676, that were explicitly represented in our session traces. Most of these traced episodes are still not perceptible. Column "$\geq$ 100ms" thus shows how many episodes per session were actually perceptible by the user. These are the episodes we need to characterize, because it is these episodes that one has to eliminate or shorten to improve the perceptible performance of these applications. We observed between 24 and 706 perceptibly long episodes in each interactive session, which highlights a considerable potential for performance optimizations. Column "Long/min" shows the fraction of perceptibly long ($\geq$ 100ms) episodes per minute of in-episode time[2]. This number, between 18 to 180 perceptible episodes per minute, provides a notion of how frequently a user becomes aware of the system's lag, and it allows a comparison of perceptible performance across applications and sessions of different durations. While 180 perceptible episodes per minute (3 perceptible episodes per second) seems high, a program rendering a complex animation can easily reach this number, given that it would aim at repainting the screen roughly 24 times per second. According to this measure of perceptible episodes per minute, our sessions with JMol had the worst perceptible

[1]We used the server version of the virtual machine (instead of the client version, which would be optimized for interactive applications), because some applications depend on Java 1.6, and Apple only provides a server version of Java 1.6 (and only in 64-bit).

[2]We divide by in-episode time, and not by end-to-end time, because in-episode time is a more stable denominator. If a user was to think for a long time during an interactive session, the end-to-end time would drastically increase, but the in-episode time would stay stable, while the number of (short or long) episodes would also stay stable.

Benchmarks	Exp. time		Episodes				Patterns				
	E2E [s]	In-Eps [%]	< 3ms	≥ 3ms	≥ 100ms	Long/min	Dist	#Eps	One-Ep [%]	Descs	Depth
Arabeske	461	25	323605	6278	177	95	427	5456	62	7	5
ArgoUML	630	35	196247	9066	265	75	1292	8011	66	10	5
CrosswordSage	367	8	109547	1173	36	80	119	1068	46	5	4
Euclide	614	35	109572	9676	96	26	202	9053	35	5	4
FindBugs	599	21	39254	6336	120	56	245	6128	44	6	4
FreeMind	524	11	325135	3462	26	30	246	3326	55	7	5
GanntProject	523	47	126940	2564	706	168	803	2373	70	18	12
JEdit	502	9	117615	2271	24	33	150	1610	50	5	4
JFreeChart	250	26	77720	1658	175	164	114	1581	44	6	5
JHotDraw	421	41	246836	5980	338	114	454	5675	70	8	5
JMol	449	46	110929	3197	604	180	187	3062	52	7	5
Laoe	460	47	1241198	3174	61	18	226	3007	58	8	5
NetBeans	398	27	305177	3120	149	82	642	2911	66	10	5
SwingSet	384	20	219569	4310	70	57	444	4152	59	9	6
Mean	470	28	253525	4447	203	84	396	4101	56	8	5

TABLE III
OVERALL STATISTICS

performance. This conforms to our subjective experience: when we used JMol to perform three dimensional animations of molecules with significant complexity and difficult to compute surfaces, the animations exhibited a perceptible lag and their frame rate dropped significantly.

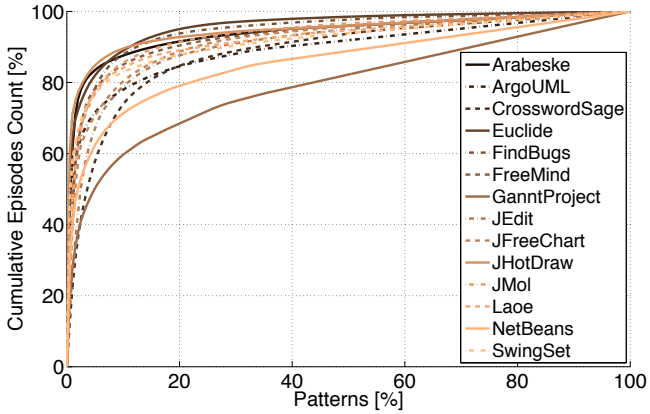

Fig. 3. Cumulative distribution of episodes into patterns

Patterns. Column "Dist" shows the number of distinct patterns (according to the classification in Section II-C) we observed in each application. This number is an order of magnitude lower than the number of traced episodes, thus an average pattern summarizes roughly 10 episodes. Column "#Eps" shows the number of episodes actually covered in patterns. This number is smaller than the number of traced episodes, because we exclude episodes that have no internal structure (where the trace contains a dispatch interval without any children). The column shows that most episodes are covered by a pattern. Column "One-Ep" shows the percentage of singleton patterns: patterns for which we found only a single episode. This number is quite high, 56% over all benchmarks, but these singleton patterns only account for 10% of episodes. The last two columns in the table further characterize the information available for each pattern. They show the size ("Descs" – number of descendants of the dispatch interval), and the average depth ("Depth") of the interval tree, averaged over all patterns, again excluding episodes without any children. We can see that some benchmarks provide much richer patterns. GanttProject shows an average of 18 intervals in an episode's interval tree. The episode sketch in **Figure 2** shows the reason for this rich tree structure: GanttProject has a complex, deeply nested structure of GUI components, and a paint request to its main window triggers recursive paint requests throughout its component tree, leading to a deep nesting of paint intervals in the episode sketch.

Figure 3 presents the cumulative distribution of episodes into patterns. The x-axis represents the percentage of patterns that cover a certain percentage of episodes (y-axis). Each line represents a different benchmark (see legend). The figure shows that the patterns follow the Pareto rule: roughly 80% of episodes are covered by only 20% of the patterns. Thus, some patterns occur very frequently and our classification approach is able to condense a large fraction of episodes into a small summary consisting of a few prevalent patterns.

B. Occurrence of Lag: Always, Sometimes, Once, or Never

LagAlyzer's Pattern Browser presents a developer with a list of episode patterns that summarize the events that occurred during the application's execution. This browser helps the developer to quickly focus on the few problematic patterns. In this section we characterize *how* problematic the identified patterns are in terms of how many *perceptible* episodes each pattern represents. **Figure 4** shows the results of this characterization.

In the worst case, all episodes of a given pattern are perceptible ("always" in the figure). Such a pattern represents a deterministic problem which probably can be understood quickly. Often, only some of a pattern's episodes are perceptible ("sometimes"). Such patterns may represent a nondeterministic phenomenon and may be difficult to understand. It can happen that only one of the episodes of a pattern is perceptible ("once"), for example because this was the pattern's first episode which triggered expensive initialization activities. Ideally, none of a pattern's episodes are perceptible ("never"). We classify singleton patterns (patterns with only one episode) as "always" if their only episode was perceptible.

The two extreme cases we see in this figure are GanttProject and FreeMind. In *GanttProject*, 57% of patterns are always

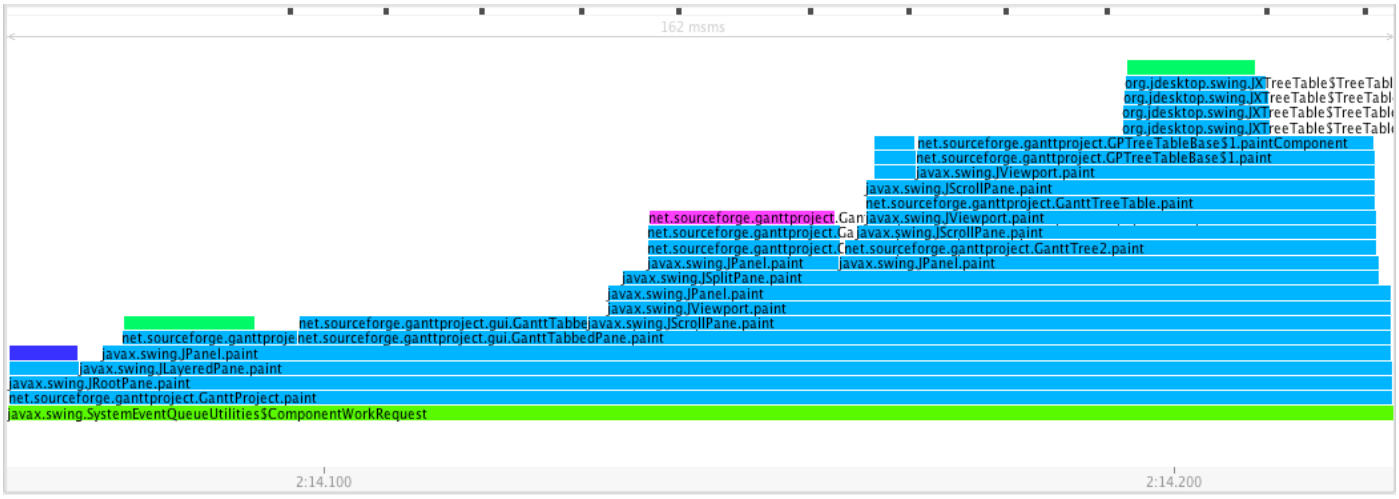

Fig. 2. Episode sketch of GanttProject, showing deep nesting of paint intervals

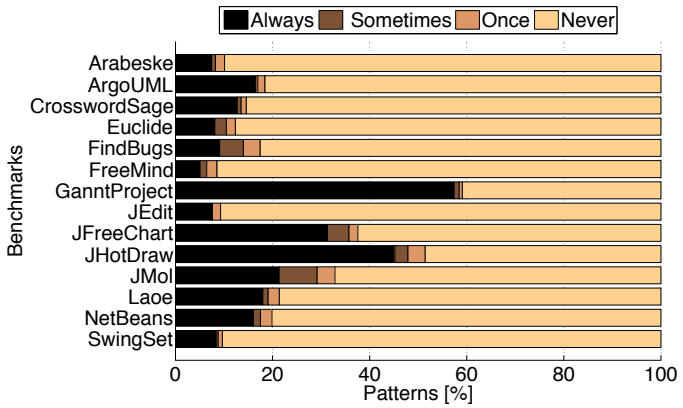

Fig. 4. Long-latency episodes in patterns

perceptibly slow. This high percentage is a consequence of our classification of singletons: GanttProject has many singleton patterns. This is related to the complexity of the interval tree structure in GanttProject's episodes, which causes a much larger set of distinct patterns. As we have discussed in Section IV-A, GanttProject has many deeply nested paint episodes, and many of them are perceptibly long. In *FreeMind*, 92% of patterns are never slow. This is due to FreeMind having only a small number (26 out of 3462) of perceptible episodes.

Overall, Figure 4 shows that for the 14 applications we studied, most patterns (96% on average) are either consistently slow or consistently fast, and a relatively small fraction of patterns (22%) are (once, sometimes, or always) perceptible.

C. Trigger: Input, Output, or Asynchronous Events

Lag can occur in handling input, e.g. by processing mouse and keyboad events. It can occur in producing output, e.g. in rendering data to the screen. Finally, it can occur in events triggered by asynchronous activity, such as when a background thread notifies the GUI thread of a state change in the application's model. In this section we characterize episodes according to these different triggers.

To determine the trigger of an episode we perform a pre-order traversal of its interval tree. The type of the first "listener", "paint", or "async" interval determines the trigger. A "listener" interval implies the episode was triggered by *input* (listeners usually are responding to user input events such as mouse clicks). A "paint" interval implies the episode was triggered by the system requesting an update of the program's *output* (paints represent activity that renders to the screen). An "async" interval implies that an *asynchronous* event was triggering the episode[3]. Finally, some episodes contain no children at all, or at least no "listener", "paint", or "aysync" child that was long enough to pass the tracing infrastructure's filter ($>$ 3 ms). We classify those episodes as *unspecified* in this analysis.

Figure 5 categorizes all episodes according to their trigger. The figure contains two graphs: the upper graph shows the data for all traced episodes and the lower graph shows the data for only the perceptible episodes. A performance analyst is mostly interested in the lower graph, because she needs to optimize the episodes with perceptible latency and does not care about the large number of imperceptible episodes. Each graph shows one stack of bars for each application. The x-axes show the percentage of episodes (resp. of perceptible episodes) grouped by their triggers.

On average across all benchmarks, 40% of the perceptible lag is due to input processing, 47% due to output, and 7% due to the handling of asynchronous notifications.

We now discuss the applications that exhibit particularly striking characteristics. In Arabeske, 57% of the perceptible episodes have no specific trigger. An investigation with LagAlyzer shows that many perceptible episodes are empty (and

[3]We found that the Swing GUI toolkit's repaint manager sometimes causes "asynchronous" intervals, even though it does not run in a different thread, due to the way it enqueues requests for repainting a component. We can identify these special episodes by their tree structure: they contain an "async" interval containing a "paint" interval. We remove those episodes from the group of asynchronous episodes and reclassify them as output episodes.

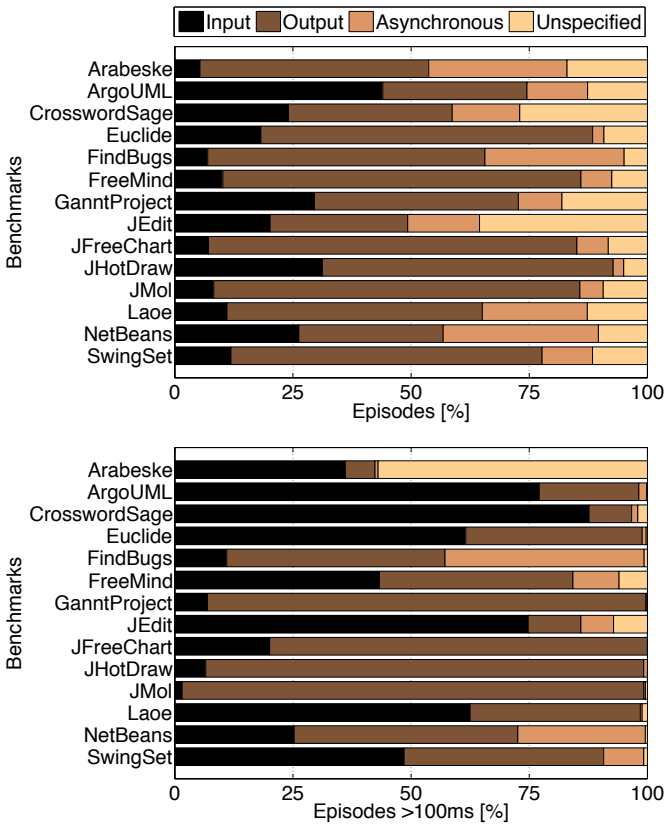

Fig. 5. Triggers of (perceptible) episodes

therefore appear as unspecified). Most of these episodes consist of a long garbage collection interval. A look at the call stack samples during these episodes shows that the program calls System.gc() and thereby explicitly triggers a major garbage collection. Explicitly requesting a garbage collection during an interactive episode could be considered a performance bug.

For JMol, 98% of the perceptible episodes are output episodes. Most of these episodes conform to a single pattern, which represents the rendering of the complex three dimensional molecule visualization. Given that JMol shows a timer-based animation, it triggers a repaint roughly every 40 ms, leading to a large number of output episodes even without user input activity.

In ArgoUML, 78% of perceptible episodes are input episodes. These episodes belong to many different patterns, representing the complexity of the application. Most of these patterns are related to input that needs to update the UML model maintained in the application, and some of these updates trigger expensive computations and checks.

FindBugs shows the largest fraction of asynchronously triggered episodes (42%). These episodes are mostly due to a background thread that periodically triggers an update of the animated progress bar. One of the patterns of this kind represents episodes that spend a significant amount of time in the toolkit's progress bar animation code, and where, in each such episode, a garbage collection is triggered at that time. Given that the garbage collections in this pattern often take several hundred milliseconds, it would be worthwhile for a performance analyst to investigate the allocation behavior in the progress bar animation code.

D. Location: Application, Library, Garbage Collector, or Native Code

Lag may be induced by the application, the runtime libraries, the garbage collector, or by native code. We first distinguish between the percentage of time spent in *application* versus *library* code. We analyze the call stack samples taken in Java code during episodes, and we partition them into application and runtime library samples. We distinguish between application and library samples based on the fully qualified class name of the method that was executing when the sample was taken. We use a different approach to distinguish between *garbage collection*, *native* code, and Java code time: Garbage collections and native calls are explicitly represented in traces as intervals, and thus we can directly compute the fraction of episode time spent in these two areas.

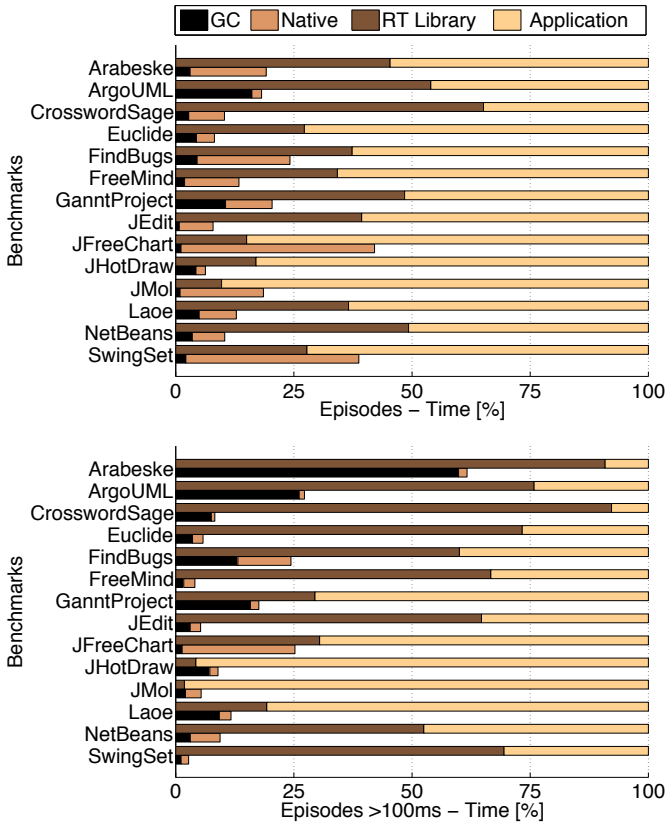

Fig. 6. Location where time was spent during (perceptible) episodes

Figure 6 shows the results of this analysis. The upper graph shows the data for all episodes and the lower graph focuses on the perceptible episodes. For each benchmark a graph shows two stacks of bars. The first stack shows the fraction of episode time spent in application versus runtime library code. The second stack shows the fraction of episode time spent in GC and native code.

On average over all applications, 52% of perceptible lag occurs in the runtime libraries and 48% in the application. The fact that more than half of the lag is due to time spent in the runtime libraries shows the importance of an efficient class library implementation for perceptible performance. 11% of perceptible lag is due to garbage collections, and 5% due to native calls. The contribution of native calls is thus relatively minor. The lag due to garbage collection is significant, however the average is greatly affected by two outliers.

We now discuss the applications that stand out in this characterization. As we have already discussed in Section IV-C, Arabeske explicitly triggers major garbage collections. As we can see in Figure 6, these garbage collections are responsible for about 60% of the perceptible lag.

Roughly 26% of the perceptible lag in ArgoUML is due to garbage collection. A look at LagAlyzer shows that minor garbage collections are spread throughout many of the episodes in many of the patterns. The upper graph of Figure 6 shows that over *all* episodes, ArgoUML spends 16% of time in GC: thus, GC is prevalent throughout program execution and is not uniquely concentrated in long episodes. These findings indicate that ArgoUML has a generally high allocation rate that necessarily leads to relatively frequent garbage collections.

24% of the perceptible lag in JFreeChart is due to native code. As we have seen in Figure 5, a large fraction of JFreeChart's lag is due to output. Using LagAlyzer, we can see that many episodes contain calls to native rendering methods. While individual rendering calls complete quickly, the many calls add up to the 24% contribution we see.

In Euclide, roughly 73% of perceptible lag is due to time spent in the runtime library. A look at the call stack samples during these episodes shows that Euclide was particularly slow in reacting to events in combo box controls. At the end of Section IV-E we will provide a more detailed explanation of the cause of this long lag for Euclide.

96% of the perceptible lag in JHotDraw is due to time spent in application code. LagAlyzer shows that a large fraction of the call stack samples were taken in code related to drawing handles and outlines of bezier curves. In our JHotDraw session we drew bezier curves of considerable complexity, and it looks like JHotDraw does not easily scale in that respect.

E. Cause: Concurrent Activity, Synchronization, Sleep, and Work

In this section we characterize the causes of perceptible latency. We group these causes into four categories: (1) concurrent activity in a background thread might slow down the GUI thread, (2) the GUI thread might be *blocked* entering a monitor, (3) the GUI thread might be *waiting* in Object.wait() or LockSupport.park(), (3) the GUI thread might voluntarily *sleep*, or (4) the GUI thread might have a lot of *work* to do.

Concurrent Activity. Figure 7 shows, for each benchmark, the average number of runnable (not necessarily running) threads during episodes. We compute this measure of concurrency by analyzing each call stack sample taken during episodes. For each sample we count the number of runnable threads.

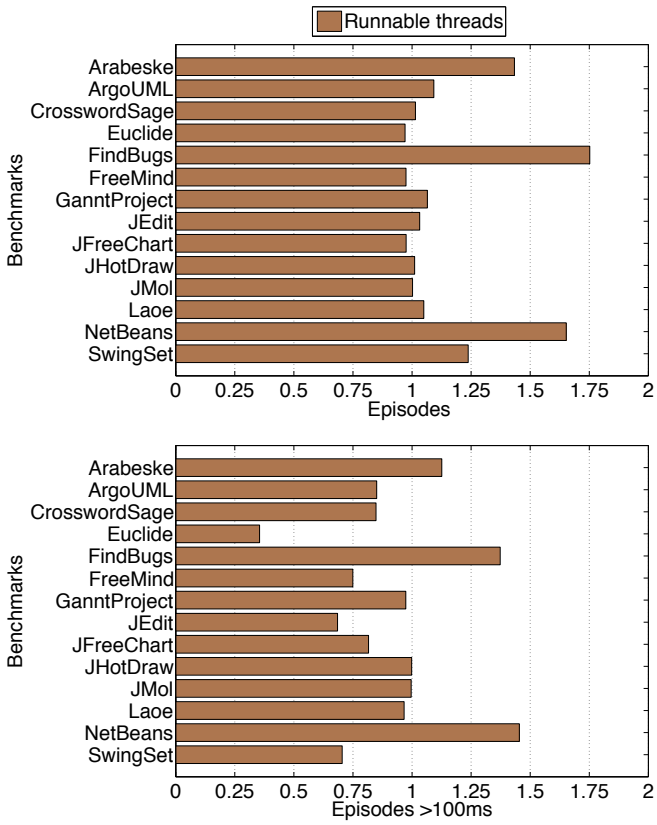

Fig. 7. Concurrency in episodes (average # of runnable threads)

A value of *one* means that exactly one thread was runnable, and given that episodes execute in the GUI thread, this has to be the GUI thread. A value *below one* means that the GUI thread sometimes was blocked, which might have caused the observed lag. A value *above one* means that other Java threads were competing with the GUI thread for the CPU core(s), and thus might have caused the lag by limiting the progress of the GUI thread.

The small amount of concurrency in these applications is surprising: only 1.2 threads are runnable on average over all episodes (upper chart). We believe this is due to the single-threaded design of modern GUI toolkits: all interactions with the toolkit have to happen in the designated GUI thread. As a result, developers prefer to also perform all computation in the GUI thread, and will only extract computation to background threads if absolutely necessary (if computation causes a perceptible lag). LagAlyzer helps to find exactly these cases.

For perceptible episodes, the average number of runnable threads is even lower than 1. The only applications with a concurrency greater than one during long-latency episodes were Arabeske, FindBugs, and NetBeans. These applications seem to make use of background threads. For example, we use FindBugs to open a project containing more than 1600 classes. Loading that project takes roughly 3 minutes, and given that

loading happens in a background thread, throughout the loading activity that thread competes with the GUI thread for the CPU.

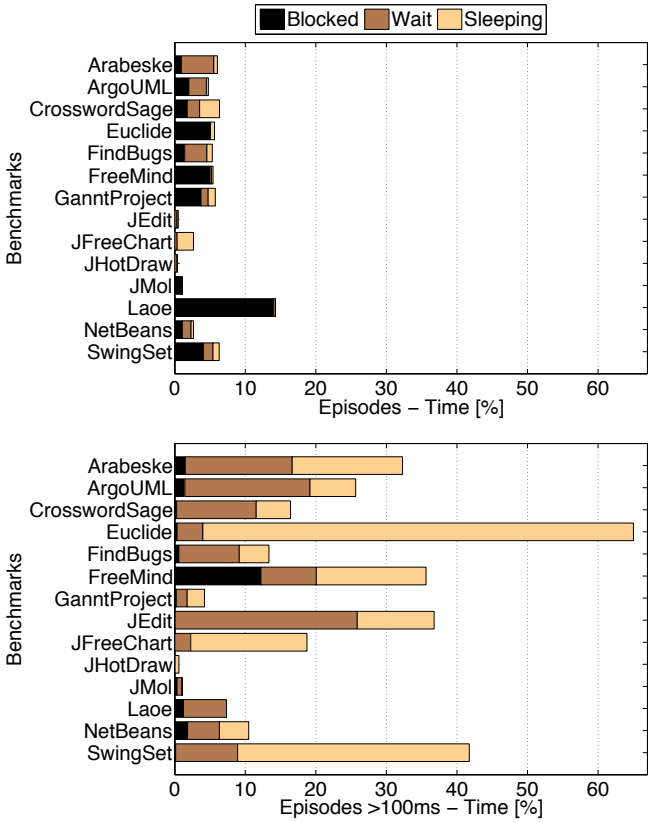

Fig. 8. Synchronization and sleep during episodes

Synchronization, Sleep, and Work. To determine to what degree event handling latency is caused by synchronization, voluntary sleep, or productive work, we count the fraction of call stack samples taken while the GUI thread was in each of these states. **Figure 8** partitions the time the GUI thread spent in episodes into four components represented in a stacked bar: time blocked while trying to enter contended monitors (first part of bar), time waiting in Object.wait() and LockSupport.park() (second part of bar), time sleeping (last part of bar), and time the thread was runnable (the remainder). Note that we zoomed the x-axis of the figure to 60% to better show the different parts of the bars: even though the bars seem big, the GUI thread is runnable most of the time. Also note that the upper graph in the figure, which shows the same statistics for *all* episodes (not just the perceptibly slow ones), shows almost no time spent in the blocked, wait, or sleep states. This difference demonstrates that aggregate information across the entire program execution is not necessarily helpful in pinpointing the causes of perceptible lag in interactive applications.

Synchronization does not seem to significantly affect perceptible performance: in most benchmarks the GUI thread spends less than 10% of the time of perceptible episodes blocked or waiting. The most notable exception is jEdit, where over 25% of perceptible lag is due to waits. Looking at the corresponding stack traces in LagAlyzer, we can see that the synchronization is related to event processing inside jEdit's modal dialogs. In FreeMind, 12% of the perceptible latency is due to monitor contention, and the stack traces indicate that the cause lies in the display configuration code of the runtime library.

We were surprised that voluntary sleep was responsible for a significant fraction of the perceptible lag. For Euclide, over 60% of perceptible lag was due to the GUI thread's sleep. LagAlyzer pointed us to the cause of this call to Thread.sleep: All stack traces where the GUI thread was sleeping were inside a method in Apple's GUI toolkit. That method caused a blinking effect in Apple's combo box implementation. The same issue appeared across all benchmarks: all time spent in Thread.sleep was due to Apple's implementation of this blinking animation.

V. Limitations and Threats to Validity

The previous section demonstrated the usefulness of LagAlyzer by characterizing the perceptible performance of realistic interactive Java applications. We now discuss the threats to the validity of our results.

The interactive sessions we performed may not be representative of the sessions real users would perform in production use. We were constrained in our ability to produce traces by the limitations of the LiLa profiler. LiLa requires significant effort in instrumenting the application and produces relatively large traces for real-world sessions. This is because LiLa's main purpose is to help a developer to find specific performance problems in a laboratory scenario, not to trace long-running sessions of users in the field. We addressed this issue by conducting interactive sessions, that, while being relatively short, performed realistic operations on realistic inputs (e.g. we analyzed a 1500 class project in FindBugs, or we edited a complete song in form of an MP3 file for LaOE).

LagAlyzer is an offline tool that needs to load the complete session trace into memory for analysis and visualization. To allow LagAlyzer to handle our sessions of around 8 minutes, given the considerable size of LiLa traces, we first filtered out all episodes shorter than 3 ms. Note that to analyze multiple sessions, for example for the analysis that produced the charts of Section IV, LagAlyzer can analyze one session at a time.

LagAlyzer itself does not perturb the measurements. The LiLa profiler, however, could potentially exhibit measurement perturbation. For example, it could slow down the application due to its instrumentation, or increase the frequency of garbage collections by allocating a significant amount of temporary data. We plan to study the perturbation of LiLa in future work.

Our study focuses on a single thread, the GUI thread. However, LagAlyzer already supports traces based on multiple concurrent event dispatch threads. It defines the notion of an episode as the time interval from the point where a given thread starts handling a GUI event until that thread finishes handling that event. The fact that we only have one GUI thread is a design property of the GUI toolkit and not a limitation of our approach.

VI. Related Work

To measure perceptible performance of modern interactive applications, we previously introduced the idea of listener latency profiling [6], an approach that attributes lag to source code artifacts called listeners, and we built a profiler, LiLa, to measure the lag of user requests of interactive Java applications. In this paper, we use an extended version of LiLa for gathering the traces which we analyze with LagAlyzer.

In the same prior paper, we also presented LiLa Viewer, a visualization tool that draws trace timelines showing the start and end of each interactive request and the tree of listener calls and returns that handled each of the requests. The goal of our prior work was to help developers to find the listeners responsible for long latency. LagAlyzer extends the timeline-based visualization of episodes by correlating intervals with call stack samples to give a more complete picture of system behavior during long latency episodes. The combined analysis of listener latency traces with call stack samples allows LagAlyzer to determine whether, during perceptibly long episodes, the GUI thread was mostly runnable, blocked, waiting, or sleeping, what fraction of time the thread spent in native or in Java code, and what fraction of time it was executing in the runtime library or in the application. Moreover, LagAlyzer groups episodes into equivalence classes, and it integrates multiple traces in its analysis, and thus helps to uncover repeating patterns of bad performance. This use of patterns also reduces the developer's effort when studying a performance problem.

Endo et al. [2] present a non-intrusive approach for measuring the response time of interactive applications running on Microsoft Windows. Their approach is based on a probe thread that runs with a low priority in the background. Given that the operating system only schedules the probe thread when the (higher priority) interactive application is idle, this approach allows them to indirectly measure idle time intervals. They find that Microsoft Word handles roughly 92% of user requests in less than 100 ms, and that PowerPoint performs significantly worse. However, they do not investigate the reasons of high latency user requests.

Fláutner et al. [3] take a more intricate approach to measuring interactive response times. They keep track of all the communication between processes as a result of a user event in the X window server (the user-space process in UNIX systems responsible for all GUI input and output).

Zeldovich et al. [12] instrument VNC [9], a software and protocol for interacting with a GUI application over the network. By monitoring the communication between the VNC client and server, they can deduce the response time between user inputs and screen updates. The best case scenario in their study shows that only 45% of requests are handled in less than 100 ms.

Howell et al. [5] compare the latency of application startup and four specific GUI events (that display the next screen in a database frontend) on two different Java GUI toolkits (the standard Swing and the open-source Thinlet). They focus on comparing the latency differences between different GUI toolkits, and they do so in a tightly controlled experiment with very limited interactions, while we characterize the contributions to perceptible latency observed in realistic interactions with real-world applications.

VII. Conclusions

In this paper we present the first study that gives insight into *why* interactive Java applications are sometimes perceived as slow. We present a tool, LagAlyzer, that analyzes offline traces gathered with a latency profiler and mines them for repeated patterns. It allows developers to browse through those patterns and to visualize perceptible episodes in the form of "episode sketches" that combine the necessary information to help solve performance problems. We use LagAlyzer to characterize the performance of 14 interactive applications ranging from a small crossword puzzle editor all the way to NetBeans, a professional IDE consisting of over 45000 classes. Our results show that the reasons for bad perceptible performance in these real-world applications are diverse and vary between applications. They include significant time spent in the Java runtime libraries, excessive time spent processing input or producing output, time spent in garbage collection, in native code, and even time spent voluntarily sleeping. We find surprisingly little concurrent activity in these applications and as a result also only a minor impact of synchronization on perceptible performance.

References

[1] 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*, pages 317–318, New York, NY, USA, 2001. ACM.

[2] 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*, pages 185–199, New York, NY, USA, 1996. ACM.

[3] 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*, pages 129–138, New York, NY, USA, 2000. ACM.

[4] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[5] C. J. Howell, G. M. Kapfhammer, and R. S. Roos. An examination of the run-time performance of gui creation frameworks. In *PPPJ '03: Proceedings of the 2nd international conference on Principles and practice of programming in Java*, pages 171–176, New York, NY, USA, 2003. Computer Science Press, Inc.

[6] M. Jovic and M. Hauswirth. Measuring the performance of interactive applications with listener latency profiling. In *PPPJ '08: Proceedings of the 6th international symposium on Principles and practice of programming in Java*, pages 137–146, New York, NY, USA, 2008. ACM.

[7] I. S. MacKenzie and C. Ware. Lag as a determinant of human performance in interactive systems. In *CHI '93: Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems*, pages 488–493, New York, NY, USA, 1993. ACM.

[8] S. Microsystems. Java Virtual Machine Tool Interface (JVMTI) 1.1.109. http://java.sun.com/javase/6/docs/platform/jvmti/jvmti.html, 2007.

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

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

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

[12] N. Zeldovich and R. Chandra. Interactive performance measurement with vncplay. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 54–54, Berkeley, CA, USA, 2005. USENIX Association.