# LagHunter Tutorial

In this tutorial we present a new profiling technique which should help developers to understand and improve interactive application performance. 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.

Interactive applications respond to user events, such as mouse clicks and key presses. Perceptible performance problems manifest themselves as long latency application responses.

Having a complete trace of each method call and return of the entire application would allow us to measure the latency of each individual method call. However, such an approach would slow down the program by orders of magnitude and thus could not be used with deployed applications.

The challenge of any practical latency bug detection approach is to reduce this overhead while still collecting the information necessary to find and fix performance issues. We achieve this with selective instrumentation od methods, only those which are necessary to capture a complete interaction with a user during run-time.

## What do we collect?

To capture a complete picture of interaction with a user we instrument methods very selectively and we call them landmarks. Their choice is the most crucial parameter in our approach. Landmark methods serve a double purpose. First, their calls and returns represent the points where we measure latency. Second, each of them represents a potential performance problem in the application.

A chosen set of landmarks should fulfill a few properties: they need to *cover most of the execution* of the program. Given that we only measure the latency of landmarks, we will be unaware of any long activity happening *outside* a landmark.

Then, they need to be *called during the handling of an individual user event.* We want to exploit the human perceptibility threshold (of roughly 100 ms) for determining whether a landmark method took too long, and thus such a method needs to correspond to a major activity that is part of handling a user event. A method executed in a background thread may take much longer than 100 ms, but it will not affect the responsiveness of the application, given that the GUI thread will continue to handle user events. Moreover, a top-level method of the GUI thread (e.g. the main method of the application represented by the bottom rectangle in Figure 1), may have a very long "latency", but it is not responsible for delaying the handling of individual user events.

Finally, landmarks should be *called infrequently.* Tracing landmarks that are called large numbers of times for each user event would significantly increase the overhead.

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

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

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

**Commands.** Ideally, the landmarks correspond to the different commands a user can perform in an application. If the application follows the command design pattern [**?**] and follows a standard implementation

idiom, it is possible to automatically identify all such landmarks.

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

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

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

Moreover, we periodically samples the call stacks of all the threads of the application. It also collects information about the GC activity. At the end of runtime, we combine all the information into a report which are later on analyzed and visualized.

## How does it run?

To implement our approach on Java, we have developed a Java agent. It's task is to instrument the application classes as well as runtime library classes before they are loaded. Classes are modified on fly while being loaded. They are **not** stored back on the hard drive which makes the implementation very flexible and suiable for use.

Every class that Sun's VM attempts to load is given primarly to our profiling tool to transform the bytecode. The profiler returns instrumented bytecode array to the VM which then loads it. Thus, classes are not modified permanently but for the run-time. To run an application together with our profiler here are the extra command line agruments:

```
java -Xmx2048m -javaagent:LagHunter-4.jar=/useLiLaConfigurationFile -jar [your application jar]
or
java -Xmx2048m -javaagent:LagHunter-4.jar=/useLiLaConfigurationFile -cp [your application class]
```

It is not necessary to use such a big heap space as in the example. Such a huge space in memory is needed only if we run enormously complex system which has hundered thousands of classes and by itself is spatially demanding. Usually, 512 megabytes is enough to run both application and our profiler. Nevertheless, considering nowadays computer systems we allow ourselves to require 2 gigabytes as default.

Profiler agent is very easy to run. LiLaConfiguration.ini contains profiler settings. It should reside in the same folder as the profiler. Default one is always provided with the profiler, but still we may change it before each run. To understand the options which profiler offers, please type the following command line in the terminal:

```
java -javaagent:LagHunter-4.jar=/help
```

## How does the trace look like?

For each trace, the analysis engine extracts all landmark invocations from the landmark trace. For each landmark invocation, it computes the inclusive latency (landmark end time - landmark start time) and the exclusive latency (inclusive latency - time spent in nested landmarks), and it updates the statistics of the corresponding issue.

**Figure 1** shows part of the execution of an interactive application. The x-axis represents time. The three flash symbols represent three incoming user requests (events) that need to be handled by the application. The call stack, shown in light-grey, grows from bottom to top. Each rectangle on the stack corresponds to

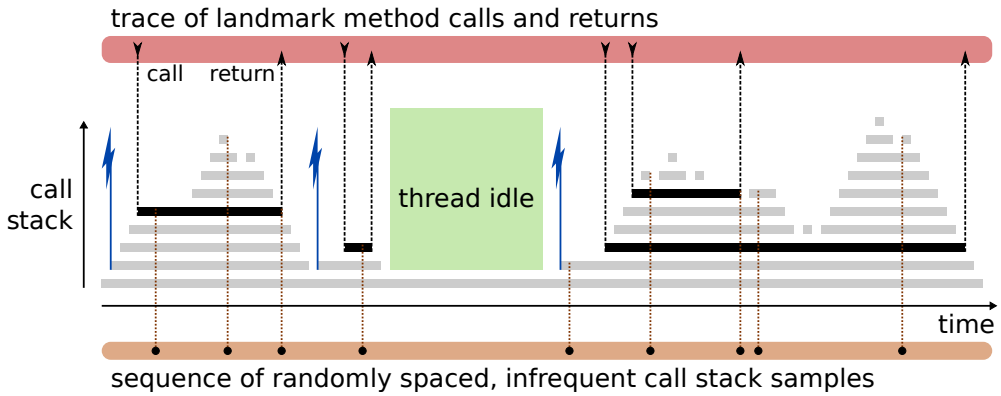


Figure 1: Session Report

a method activation. In our approach, the agent collects two kinds of information. First, it captures calls and returns from so-called *landmark* methods (black rectangles). This landmark trace contains the timing information necessary to measure lag. Second, it captures rare, randomly spaced call stack samples of the running threads. Each stack sample contains the complete calling context of the thread at the time the sample was taken. This sequence of stack samples provides information about what the thread was doing throughout a long-latency landmark method invocation.

## LagAlyzer

To visualize trace files we developed **LagAlyzer**. It is a standard Java application JAR-ed in one file. To run the tool from terminal it is enough to type the following command line:

```
java -Xmx2048m -jar LagAlyzer-4.jar
```

**Figure 2** shows the interface of LagAlyzer. It is simple and pretty straight forward for use. It has a single **File** menu with a single **open** choice. It opens standard file open dialog where we may navigate to a desired trace file.

LagAlyzer interface has 4 distinctive areas splitted horizontally. The first one from the top represents full runtime of GUI thread for the chosen trace file. It is shown separately on **Figure 3**.

X axis represents the execution time. For example, **Figure 3** shows very short run of an application. By looking at the x-axis value we may say that the run lengthis roughly 13 seconds.

Not only one landmark can execute during some period of time. When an observer method occurs which handle mouse click action, it may trigger some other observer notifications, like dialog open action, which will cause a component redraw handled by yet another landmark.

Y-axis shows stacked landmark methods in the order they were executing. Each landmark type is visualized with different color. For example, event dispatch intervarls are colored red, observers are painted pink, asynchronous calls are green, while methods that redraw AWT+Swing/SWT components are shown in blue color.

A rectangle's x-axis left corner corresponds to the x-axis time value when it started with execution. Respectively, the rectangle's x-axis right corner corresponds to the x-axis time value when it finished with execution. We could zoom into the specific region of the runtime. To achieve it, we should press the left mouse button when the cursor is at a desired place on the time line and to hold it while moving mouse cursor along the time to the right. When the desired time range is selected we should release the mouse button and it will zoom into the selected region. To zoom out, we should use the context menu which appears on the right mouse click on the time line.


Figure 2: LagAlyzer interface

Note that it does **not** show all the methods on the thread stack but only landmark methods. Indeed to identify the root cause of some landmark's long latency, we may need the rest of the picture from the exection. That is why periodically taken samples during runtime are used. On the top of the **Figure 3** there is a horizontal strip of dark dots where each one corresponds to a single stack dump. Their position along the x-axis shows when the stack dump is taken. Thus, a stack trace contains both landmarks and others. For example, let our GUI thread starts from the main method and it calls some observer notification method which further calls method foo (). If the stack is dumpped when our application was in the foo method then the stack will contain all three methods, while runtime line will have only one rectangle shown for the only landmark method which is in this case an observer.

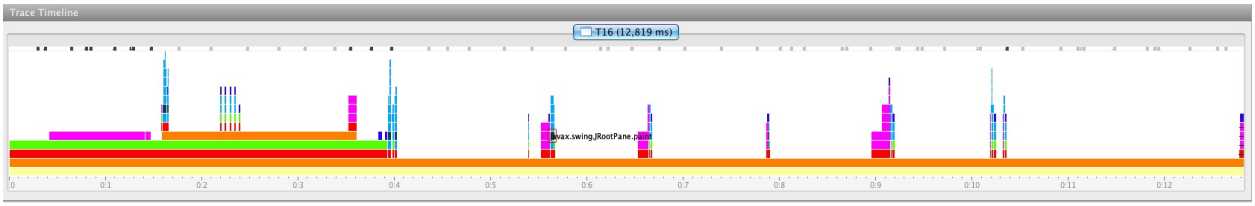


Figure 3: LagAlyzer run-time time line

This information is very valuable when some landmark takes long to execute but is actually not the "guilty one". Not rare, landmarks are just an initial methods which handle some user actions and the root cause is actually some other far nested method. Thus, combining the information for a slow landmark execution together with the stack dumps taken while it has running may reveal the real problem much easier.

**Figure 4** shows a table of all the landmarks executed at least once during runtime. Each landmark row contains information related to its execution. The first cell shows the icon which identifies landmark type. For example all GUI repaint landmarks has a brush icon. The second cell is dedicated to a specific tasks

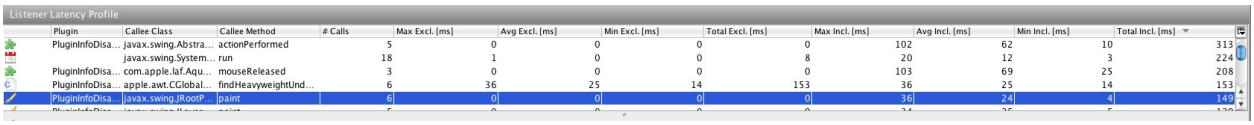


Figure 4: LagAlyzer Landmarks

and please omitted it. The third and the fourth cells represent the landmark class and method name. Next cell shows landmark occurance count. This cell is followed by eight cells in two groups of four. The first group shows information about landmark exclusive execution time. Respectively, the second groups shows information about inclusive execution time. Each group contains maximum, average, minimum and total time respectively for landmark exclusive and inclusive time. In both cases the last cell shows time consumed while handling oll the invocations during runtime in sum for a specific landmark.

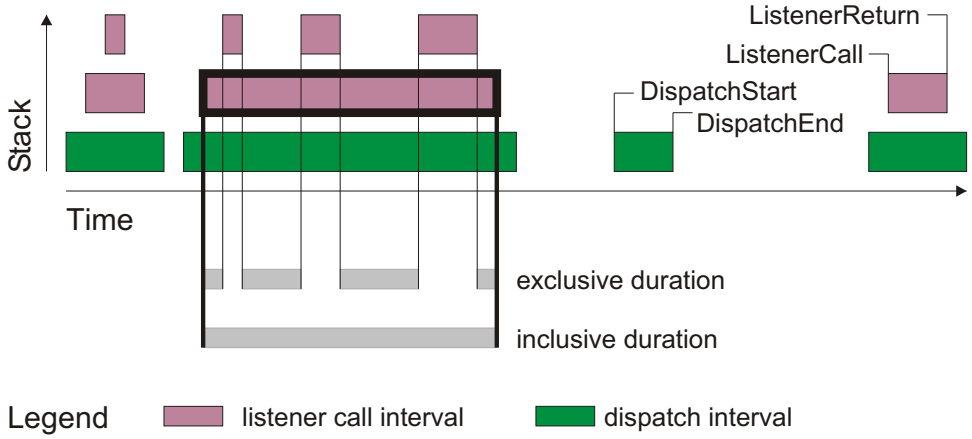


Figure 5: LagAlyzer interval analysis

**Figure 5** shows difference between exclusive and inclusive time on the example of one listener invocation. The diagram Y-axis shows the stack only for the landmark methods. X-axis represents time. Green bars represent event dispatch intervals, while pink bars represent listener calls. As it is shown in the figure, any landmark may call another landmarks. Consequently, consumed time (latency) is not spent only in the initial landmark. It could fairly spent time in some other landmarks.

Thus, exclusive time is consisted of execution time slices spent in the actual listener, while inclusive time takes complete time range from the start until the end of its exection. Both information are equally important to analyze results. Inclusive time shows what is potentially slow to a user as it represnts the continuous time spent handling some action. To the user, it is not important how time is consumed but only if it is perceptible or not. On another hand, developers may benefit from the exclusive time information. It shows where and how much time our application spent while handling the action which may point her closer to the actual problem. It is generarly good to first look at the inclusive time to understand what is perceptible to a user and then to break down the problem through with exclusive time information.

**Figure 6** shows individual information for occurred landmark calls during runtime. Every table row stands for one call of selected landmark. This table is tightly connected to the previous table showed in **Figure 4**. Information in this table always relates to a landmark selected in the table showed in **Figure 4**. It breaks down collected landmark information in one row per each occurrance.

The first cell tells us how deep some landmark invocation was on the stack while the second cell tells how many more were nested. These cells are followed by three cells related to its children (other landmarks). The first of those three cells shows how many were long while the second and the third show how many

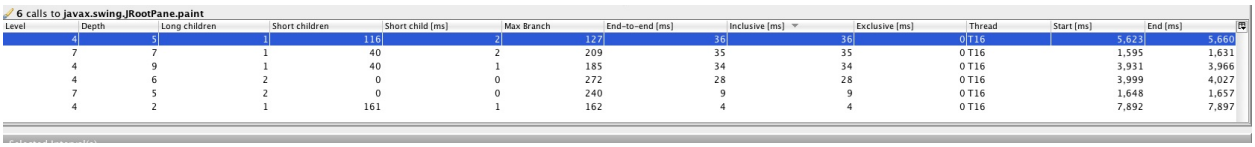


Figure 6: LagAlyzer Individual Landmark Execution

were short and their executioon time. Terms long and short comes from the thresholds of 3 ms and 100 ms. If it ran longer than 100 ms then it is threated as long. This threashold is scientifically shown to be the border of human perceptiblity. On another hand, if it ran in less than 3 ms than such call is ommitted from the trace (avoid spatial problems) but basic execution information are saved and here shown in the table. "Max Branch" cell is currently not used. End-to-end time cell represents time needed to handle specific landmark including complete dispatch interval. In most of cases this value is same as inclusive time that is shown in the successive cell. Still a difference may happen and it could help understanding the problem. After exclusive time cell and the cell that shows thread ID within the landmark call was running, there are two cells more: landmark start time and end time. They show the exact time (in milliseconds) when a call started and ended.

In general, this table may help us reveal a few more things. For example, let's assume that there were 10 occurrances of some landmark with average execution time of 1 s. It may easily happen that 9 of them took less than a 100 ms while only one took 10 s due to some external activity in the operating system or due to garbage collection. The average will be high which is not exactly true. It may happen that the first execution is slow, while others are fast. It is a sign of initialization problem. For these reasons it is good practice to first look at this table and individual call information, and to confirm the suspects.

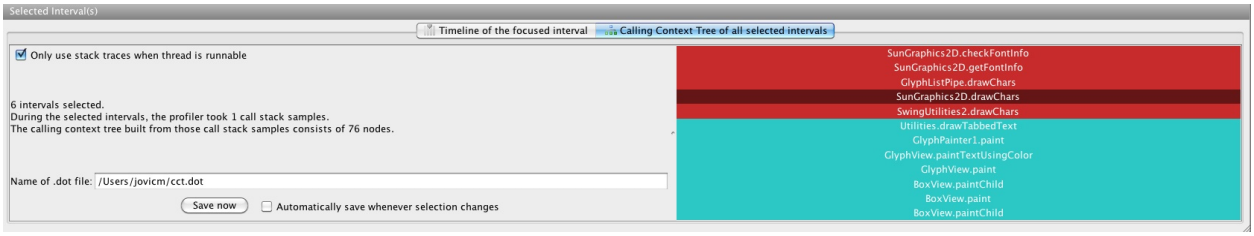


Figure 7: LagAlyzer Calling Context Tree

The bottom region of the interface is shown in the **Figure 7**. Left tab shows a landmark call stretched to the whole screen which is selected either in the table or on the runtime showed in the **Figure 3**. This way we can quicly see concrete names of classes and methods at readable size. The right tab shows the calling context tree which is constructed from the stack dumps taken for the selected landmark on either way. Calling context tree is not constructed only from the stack dumps of a landmark individual execution but from all executions. It is a tree where every path from the root to a leaf represents one context. Every node represents one method and it contains aggregated information of its calls. This tree is weighted tree. Our representation of the tree is given in **Figure 7**. A context rectangle width corresponds to percetange of stack dumps taken while being in that context out of total number of stack dumps taken in all the contexts. Thus, it is very important not mix this representation with the one shown in the figure **Figure 3**. All previous figures x-axis represents time, and this figure x-axis represents percentage.

## Conlusion

Use of LagHunter approach with LagAlyzer should help a developer to quickly nevigate to important performance problems and to responsible code parts. It can be also used to confirm eventual problem fix. To do so, a developer could use following methodology:

- make a change in the code which should fix the problem
- run the application again with our profiler
- trigger the same action which caused the performance issue
- close the application
- run LagAlyzer
- open the trace
- verify that time has dropped for the corresponding event handler