Many if not most computer systems are used by human users. The performance of such interactive systems ultimately affects those users. Thus, when measuring, understanding, and improving system performance, it makes sense to consider the human user's perspective.

Essentially, the performance of interactive applications is determined by the perceptible lag in handling user requests. So, when characterizing the runtime of an interactive application we need a new approach that focuses on the perceptible lags rather than on overall and general performance characteristics. Such a new characterization approach should enable a new way to profile and improve the performance of interactive applications.

Imagine a way that would seek out these perceptible lags and then investigate the causes of these lags. Performance analysts could simply optimize responsible parts of the software, thus eliminating perceptible lag for interactive applications. Unfortunately, existing profiling approaches either incur significant overhead that makes them impractical for an interactive scenario, or they lack the ability to provide insight into the causes of long latencies.

An effective approach for interactive applications has to fulfill several requirements such as an accurate view of the causes of performance problems and insignificant perturbation of the interactive application.

We propose a new profiling approach that helps developers to understand and improve the perceptible performance of interactive applications and satisfies the above needs.

B

Understanding the Performance of Interactive Applications

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Milan Jovic

under the supervision of
Matthias Hauswirth

September 2011

Contents

Part I

Prologue

This part contains two chapters. Chapter 1 introduces the problem and formulates the thesis statement. Chapter 2 discusses work related to our research.

Chapter 1

Introduction

Should we take software performance for granted? As Millsap wrote in his article "Thinking clearly about performance" [86], "we should treat it as a software application feature, which like any other has to be designed and built".

Interactive applications are prevalent on computer systems, from servers to desktops all the way down to mobile devices. Developers of interactive applications and systems want to focus their performance efforts on aspects that are actually perceptible by users. A body of research in human-computer interaction has studied the connection between the human perception of system performance and the response latency of interactive applications [103; 41]. They found that human performance decreases with increase of a system response time [82], and that lengthy or unexpected response time produces frustration, annoyance, and eventual anger of the users [103].

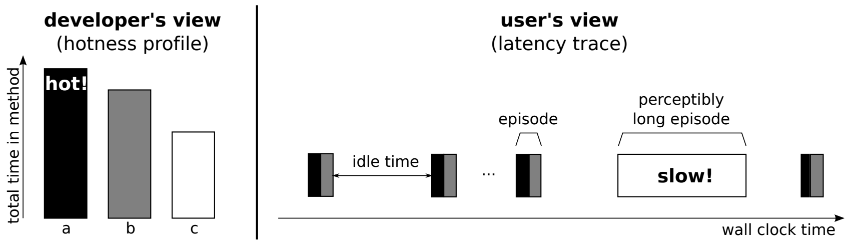


Figure 1.1. Method Hotness Profiles are Abstractions

Thus, given that human-computer interaction research has shown that the key determinant of perceptible performance is the system's lag in handling user events, the "code hotness" profiles produced by traditional profiling tools [57; 108; 14; 7] do not necessarily reflect perceptible performance for interactive applications. **Figure 1.1** shows an illustration. The left part represents a traditional hotness profile showing the three hottest methods (methods that spent most CPU cycles). The right part represents a trace of the corresponding program execution. The trace shows a significant number of idle-time intervals (where the computer was waiting for user input). We use the term "episode" to denote the intervals of computational activity in-between idle intervals. More importantly, the trace shows that some (hot) methods were executed often but each invocation completed quickly, while other (cold) methods were executed rarely but an invocation might have taken a long time. This example highlights that perceptible performance is not necessarily caused by the hottest methods in a traditional profile.

The real-world hotness profile in Figure 1.2 confirms this point: The center of the diagram represents the root of the tree (the application's main method). The tree grows inside out, and each stack frame is represented by a ring segment. The angle of a ring segment is proportional to the hotness of the corresponding calling context. The tree in Figure 1.2 represents the production use of a real-world Java application (Eclipse). We use the non-traditional circular tree visualization because it scales to large calling context trees. It has 224344 ring segments (calling contexts)[88] in this example. The calling context tree contains a node corresponding to a method that repeatedly (30 times) exhibited an exceptionally long (at least 1.5 seconds) latency. However, that context (the *EditEventManager.mouseUp()* method indicated with the arrow) is barely visible in the tree, because the invocations of this method make up less than 1% of the application's execution time. A developer looking at this profile would never have considered tuning that method to improve performance. Instead, he probably would have focused on hot methods that are frequently called but do not necessarily contribute to perceptible lag.

We want to solve problems like the one above. We show a new profiling approach, landmark latency profiling, that helps developers understand and improve the perceptible performance of interactive applications.

A landmark latency profile consists of latency information for a small set of "landmark" methods. The set of landmark methods has to be small enough to keep profiling overhead low, but large enough to provide meaningful information to the developer.

The remainder of the dissertation is structured as follows: The work related to the dissertation is discussed in Chapter 2. Together with Chapter 1 they form Part I of the dissertation. Part II consists of Chapter 3, Chapter 4, and Chapter 5. They discuss the approach presented in the thesis. Further, Chapter 6, Chapter 7, and Chapter 8 discuss the evaluations of the approach done in the lab and with real developers out of the lab, and form Part III. Chapter 9 discusses additional

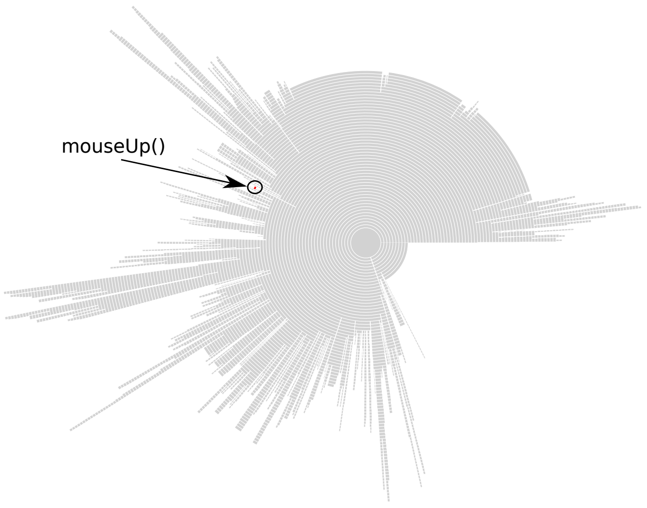


Figure 1.2. Perceptible Bug Hidden in Hotness Profile

applications of our approach while Chapter 10 discusses the limitations of the approach and the implementation. Finally, we conclude our thesis in Chapter 11. These chapters form the last Part IV of the dissertation.

1.1 Thesis Statement

We define our thesis as follows:

Landmark latency profiling helps developers to understand and to improve the perceptible performance of their interactive applications.

To demonstrate the validity of the thesis statement, we will adress the following questions as well:

- *What are relevant landmarks for interactive application performance?* Responding to a user event, an interactive application updates its state, changing the data model and/or updating the output on screen. Input events triggered by a user are usually just the initial ones followed by many internal events, until the application possibly updates the output and becomes again idle, ready to handle a new event. Questions that need to be answered are: Which methods have to be monitored and marked as landmarks? What information should be in a profile? Which metrics best represent interactive application performance?

- *How can the profiling approach be used in production?* Having the implementation of the approach in production use requires its easy deployment, seamless installation, and transparent profiles collection from the users. Questions to answer on this level are: What is the acceptable slowdown due to profiling? What information to collect, and how to represent that information, to allow acceptable trace size? What to collect from users?

- *How to validate the approach?* Validation of the approach would require both validation of collected data correctness and validation that the approach helps performance problem localization and resolution. Developing a set of micro-benchmarks to produce each event individually and validating their presence in a profile would help answering the first part of the question. Questions that need to be answered are: Which events have to be validated? How much time does a developer need to locate the problem, with and without the profiling approach? How much time does a developer need to resolve the problem, with and without the profiling approach?

Chapter 2

Related Work

Work related to perceptible performance measurement of interactive applications can be divided into four parts.

First we discuss the work which defines perceptible performance and important metrics for our measurements. When time needed to handle user requests is imperceptible to the user, performance can be considered good enough. Problems with the performance of interactive application appear with perceptibility. Human-computer interaction researchers addressed many of these problems. Results of their research, such as the establishment of a perceptibility threshold, were instrumental for our approach. We discuss the relevant aspects of HCI research in Section 2.1.

Second, important work comes from the existing approaches that deal with software performance. Existing profiling approaches inspired us to come up with our low overhead profiling approach for interactive applications. We discuss them in Section 2.2.

An outcome of any profiling technique is either a trace or a profile. In both cases their content is some profiled application runtime data. The third part is the work related to behavioral data collection and their analysis. We discuss this work in Section 2.3.

Finally, to understand the performance of interactive applications we should understand their behavior and their internal characteristics. Section 2.4 describes the work related to the characterization of interactive applications.

2.1 Human-Computer Interaction

HCI research impacts our work from three perspectives. Relevant questions which they try to answer are: How does a user perceive the change of system response? Second, how does anxiety of a user change due to system response time? Third, how does performance of a user change with system response?

2.1.1 Human Perception

HCI researchers have studied the impact of system response time on the human perception of system performance. Shneiderman [104] defined the response time as the time it takes from the moment a user initiates an activity until the computer presents results on the display. This is exactly what we think is important to a user. Time the user thinks before entering the next action he defined as user think time. Indeed, during this period of time a user can't perceive an application as sluggish because her doesn't know if it hangs or it is ready to react on action. Shneiderman further finds that unexpected long response time can produce frustration, annoyance, and eventual anger. Moreover, he reports that more experienced users expect shorter response times [103], and that while users adapt their working style to the system's response time, user productivity decreases as response time increases. In a study of end-user frustration Ceaparu et al. [37] found that users reported that they lost above third of their time working on computers due to frustration. They identify time delays as a significant reason for such frustration. These results motivated our work to focus on the performance in terms of response time, too. Moreover, just striving for better performance is not often a goal as there we could always ask for more. In terms of interactive applications, imperceptibility to a user is the goal. It means that response time below the human perceptiblity threshold is the optimal one we want to achieve.

HCI researchers have also studied the conrete thresholds of humans for various application responses. Dabrowski and Munson [41] studied the human detection thresholds for interactive applications responses of keyboard and mouse inputs. They showed that users notice response times for both greater than 150 ms, while for mouse input, the detection threshold is at 195 ms. On the other hand, if the system response time is below any defined perceptiveness, a user won't be able to notice it. For example, the movie standard was set to 24 frames per second [10] as experimental research has shown that displaying 24 images per second a human eye perceives each as instantaneous. Hypothetically, if system response time, by Shneiderman [104], is always within the 24th part of second then an application would imperceptibly execute from the user perspective. Thus, these results of the researchers in the field helped us defining the perceptibility threshold.

2.1.2 Human Anxiety

Guynes [58] studied the relationship between system response time and human anxiety. He classified subjects into two groups by patterns. The first pattern was primarily of competitiveness, excessive drive, and an enhanced sense of time urgency. The second pattern was defined as the relative absence of the first pattern characteristics. He experienced that personality of the first group would exhibit a greater change in anxiety than the second when system response time was prolonged. McAdoo [83] found that physiological changes due to excitement of the autonomic nervous system, include increased heart rate, systolic blood pressure, respiration, and muscle tension. Some experienced a muscle tension increase, while other a marked increase in heart rate and systolic blood pressure. If users are exposed to poor or variable system response times for longer period, they may even change the expectations. Turner [112] found an increase in mental strain in workers who performed a task on an interactive system which provided almost immediate response, versus workers who performed the same task on a system with delayed response of a few minutes. He shows that system response time is one of the most important factors that impact humans' anxiety and thus their response. Referring to this work we find perceptible performance as even more important than it looks like. Many people employed in the industry perform their daily job on a computer in an office using an interactive applications dedicated for the specific tasks. These studies showed that perceptibly long response time may impact negatively their productivity. How human's performance changes in the sluggish environments we discuss in Section 2.1.3.

2.1.3 Human Performance

As humans are affected by system response time, their performance may change as well while performing the tasks. However, empirical measurements of effect of lag on human performance were rare. MacKenzie and Ware [82] studied lag as a determinant of human performance in virtual reality environments. They found that user performance significantly decreased with increasing lag. Gibbs [55] measured an effect of lag relating to the design of controllers for remote manipulations systems. However, tested lags were extremely long (up to several seconds) and were programmed as an exponential time constant preventing the cursor from fully reaching a target. Sheridan et al. [102] conducted the same type of experiment having a mechanical apparatus. In both cases movement time increased due to lag.

This work as well as the work related to human anxiety just confirmed that response time is crucial metric for performance of interactive applications.

2.1.4 Challenges

Even though significant work has been done in the field, there are still many uncertainties. As participants are necessary in this kind of research the experiment outcome widely depends on them. The usual key issues are examinee structure, their habits and expectations, their current condition, their previous experience, but also validation of their answers. This information has to be precise and accurate to enable statistically significant claims afterward as for the case of keyboard and mouse latency perceptibility thresholds. The described work has variable numbers of participants in the experiments, starting from 4 [103] up to 86 [58]. Thus, overall results and conclusions may highly depend on this factor, too.

2.2 Software Performance

To deal with the performance of interactive applications we need tools that measure relevant metrics of their execution. Researchers tried to measure performance on both hardware and software levels.

On the hardware level, modern processors usually have a few programmable built-in counters allowing user to count instructions, branch miss-predictions, cache misses, etc. Several library packages provide access to them such as the HPM toolkit [99],PerfCtr [95], Perfmon [96], and PAPI [32]. These libraries provide interfaces to allow program instrumentation, to record hardware counter data, and to analyze the results. The important aspect of using these libraries is that we leave the application code intact. Thus, all the measures we take correspond fully to the application runtime and not to our impact. Using these libraries to count available events during runtime will be surrogate measures that we don't need to help developers finding the cause of bad performance. Knowing that there is increased number of cache misses or any other overall measure will mostly not be able to point developer to the right place in the code. The application may have many cache misses and still suffer from complex drawing or synchronization problems. Thus, giving the overall performance picture of runtime can't effectively and directly help as we need concrete information of each perceptible action and in the first place their latency as crucial measure.

Zaparanuks et al. [123] evaluated counters and found significant inaccuracies when using them to measure instructions and cycles especially when measuring the behavior of short segments of code, hundreds or thousands of instructions. Even though there can be inaccuracy while measuring total number of executed instructions, they are hard to use to understand the performance of interactive applications where response time is the metric that matters.

On the software level, most of the approaches require interventions either in the execution binary code or in the bytecode. There is a number of binary instrumentation tools developed for different platforms such as VTune [116] for Inter processors, PIXIE [73] for MIPS architecture performing a basic block counting, PIN [38] for Intel processors and ATOM [109] as a binary rewriting toolkit for Alpha. Other tools work on the Java bytecode level such as BCEL [42]-Bytecode Engineering Library, BIT [78]-Bytecode Instrumenting Tool, JikesBT [64]-Jikes Bytecode Toolkit, and Soot [113] as Java compiler and optimization framework. Moret et al. [87] presented polymorphic bytecode instrumentation which enables complete bytecode coverage, that is, any method with a bytecode representation can be instrumented.

Our approach requires lightweight instrumentation to modify only methods marked as landmarks. Directly, these libraries and tools do not provide data we would need to correctly measure interactive application performance. Indirectly, we could use libraries to instrument important methods in the code. In another words, we could write specific profiler for the domain of interactive application.

Generally, to deal with software performance developers use profilers or build a specific one that fit their needs. Work in this area inspired us to come up with our approach. Most of the work in this field we could split into a work related to standard (non-distributed) and distributed applications on one side, and a work about instrumentation techniques and sampling techniques on the other side. How their work fits our needs and how we differ from them we explain in the following subsections.

2.2.1 Standard Applications

Both industry and the academic literature have broadly described different method profiling tools dedicated to this type of applications. On the level of Java virtual machine, a number of profiling tools performs collection of data from application run-time, such as CPU time or heap memory usage, trying to achieve more accurate profiles with less overhead. We tried to find the closest approach which could help developers to find causes of perceptible performance.

Gprof [57] hooks into the compilation and adds profiling code to an application. It builds caller/callee function pairs which allows a dynamic call graph generation. This work was later extended to more precisely handle programs with mutual recursion and dynamic method binding by Spivey [108]. However, collecting of such metrics from the application runtime is not useful to us for two reasons. First, overall CPU time distribution will not show to a developer slow response time to a user requests. Second, these tools significantly perturb the execution. The task we have requires the lightweight behaviour

DJProf [2] uses AspectJ to investigate heap and object lifetime, and time-spent in methods. It is an interesting implementation but doesn't fit our approach. HProf [13] works as dynamically-linked native library and uses JVM TI [12]. It generates information through calls to JVM TI, through event callbacks from JVM TI, and through byte code insertion (BCI) on all class file images loaded into the VM. At the first place, this instrumentation approach doesn't allow us to measure time spent in each call which we claim is necessary metric to have. Java Memory Profiler [98], JMemProf [4], and Visual VM [11] trace heap objects usage and visualize it during application execution and thus can not directly help locating responsible code. Nevertheless, such approaches may help revealing why already identified method cause a slow down. JRat [6] uses AspectJ and weaves instrumentation instructions into an application. Using the aspects may help instrumenting specific methods (landmarks). However, even though it is possible to define landmark type with aspect, we couldn't use these approach for two main reasons. Current implementations do not allow us to express all of the landmark types. Also, their generic approach reflects with significant perturbation.

Some profiling tools fully instrument code capturing the time spent in each method by wrapping every call. Both the NetBeans Profiler [7] and Eclipse TPTP [14] profiler work in this way with thousand times application slowdown when every method in application is instrumented. For this reason, these profilers may only instrument the places explicitly specified by a user. Significant slowdown they may cause during runtime disabled us to use them.

Still, there are many other commercial and non commercial profiling tools such as JProfiler [5], Profiler4J [8], and Extensible Java Profiler [3]. Usually they use sampling together with application code instrumentation to collect data from the application run-time. Again, they mostly capture time spent in methods, create call graphs, and trace memory consumption, thus none of them we could use in our approach.

2.2.2 Distributed Systems

Work related to performance in the area of distributed systems has latency as important metric to gather in profiles. Thus, it represents one of the crucial research body that we studied to design our approach. Here, we present an overview of the most important work that impact our approach which is based on latency.

Miller et al. presented Paradyn [85] as a performance measurement tool for parallel and distributed programs. It is based on a dynamic performance instrumentation, thus the code is injected during execution. It counts events and measures time spent in fragments of code. The idea they use is to add or remove instrumentations on request to reduce the profiling overhead. Metrics in Paradyn are everything countable or measurable in terms of time. Extensions of this work have been done by Xu et al. [120] and Newhall et al. [92] to add support for the multi-threaded applications and for Java, respectively. Kind of measures this approach has is exactly what we need to collected and afterward analyze to quickly locate slow parts.

Bakic et al. [21] presented BRISK, a distributed instrumentation system. It has the built-in clock synchronization to align events and dynamic online sorting algorithms to allow lower overhead. We find this work interesting as it shows a way to align event order with extremely low overhead which is necessary in our approach. Snelick's presented S-Check [106], an automated program sensitivity analysis tool for parallel and distributed programs. It predicts how changes in parts of a program are going to affect performance. Spezialetti [107] presented technique that exploit program and event semantics to reduce the overhead of monitoring a distributed computation for ordered and concurrent events.

SCALEA [111] as a part of ASKALON [47], is a performance instrumentation, measurement, and analysis tool. It supports the automatic analysis of a series of experiments, not just single runs. Kappa-PI [46] is a knowledge-based performance analyzer for parallel MPI (Message Passing Interface) and PVM (Parallel Virtual Machine) programs while a performance analysis environment Peridot [54] is designed to work teraflop computers. The idea presented in these approaches to have an automatic analysis of multiple runs beside instrumentation, motivates our work to have our profiler distributed in the field. Relevant performance problems may occur there but not in the lab. Also, a problem may manifest only on specific platform and context which we ca not create artificially in the lab.

Focusing on latency, ETE [62] identifies key events that occur in transaction execution. Events are routed to a separate component called the transaction generator, that uses external definitions of transactions to construct transactions records from event streams. Nevertheless, one would need to identify events of interest.

Barham et al. [23] developed Magpie, an approach to characterize the workload of a distributed system. Beside latency they observe concurrency and communication focusing on responsiveness rather than throughput. Their goal was to understand relationship between throughput and response time. Fonesca et al. [50] developed an integrated tracing framework called X-Trace. It is a cross-layer and cross-application tracing framework designed to reconstruct the user's task tree. A user invokes X-Trace when with a web request, by inserting X-Trace metadata with in the request. This metadata is then propagated down to lower layers through protocol interfaces and also along all recursive requests that result from the original task. The trace data generated by X-Trace is published to a reporting infrastructure. Dapper [105], Google's production distributed systems tracing platform, conceptually shares similarities particularly with Magpie and X-Trace.

In general, these approaches identify important events for the performance of the distributed systems, but also try to measure their latencies. It motivated us to identify important aspects for the performance of interactive applications and to come up with the corresponding measurement approach.

2.2.3 Instrumentation

Researchers came up with the various ways to instrument the application code. Instrumentation could be inserted manually, by assistance of a compiler, by binary translation using a tool to instrument compiled binary, by runtime instrumentation where code is instrumented before execution, or by runtime injection where code is modified at runtime to jump at our functions. Depending on application type, created profiles may contain different kinds of data about the behavior of the application run-time. As discussed in Section 2.2, distributed application profiles report latency beside other information while standard application (non-distributed) profiles mostly report the hottest methods during run-time.

Previous work on instrumentation showed that a significant body of research tried to focus on the important places of the control flow graphs. Ball and Larus presented the algorithm [22] that picks places in methods using the spanning tree to determine a minimal, low-cost set of edges to instrument in the control flow graphs. Still, their idea tends to cover a complete code with lower overhead no matter what it is. We claim that such an approach will not give benefits for our domain. Later on Roy et al.[100] presented generalization covering loops in the paths. Using this work, Vaswani et al. [114] presented a preferential path profiling to reduce the overhead of path profiling by observing only the most consumed, interesting paths. Similarly, selective path profiling [19] is an approach to profile only subsets of desired entities but also the execution of other entities which they interact with. The idea of instrumenting only a desired subset of entities fits into our approach but they do not provide a way to define landmarks as we need. Bond et al. [29] presented a path profiling approach that uses a bit-tracing where a 1-bit value is associated with the outcome of each two-way branch. When branch executes, instrumentation code appends a bit to a trace buffer that records branch outcomes. This idea brings up benefits in terms of overhead but it is for general use and the outcome of the profile does not provide with data we could use to recognize perceptible method execution. Duesterwald et al. [44] showed that hot paths could be successfully predicted. The goal is to predict what will be one of the most frequently executing paths based on a limited amount of execution history. They had the hit rate average is 97% when only 10% of execution was profiled but had a lot of false positives, too. To reduce run-time overhead Sun's JFluid [43] exploits dynamic bytecode instrumentation and code hot-swapping to turn profiling on and off dynamically, for the whole application or just a subset of it. All of these approaches showed a significant decrease of the instrumentation cost when only specific parts of the application code are instrumented.

Even though none of the approaches above target latency, the idea to instrument only part of the application code shows importance to instrument only relevant part of the code. Less instrumentation leads to less overhead and may bring more benefits.

Some of the existing dynamic profiling and tracing tools such as DTrace [36] or BTrace [1] provide facilities to easily instrument running applications. But, neither of the two low-overhead approaches supports arbitrary computations in its trace actions (e.g. they disallow loops and recursion), and thus they would be unable to do the online trace filtering necessary for a lightweight approach.

Pearce et al. [94] investigated whether aspect-oriented programming with AspectJ can be used for efficient profiling of Java programs. He showed AspectJ's limitations as inability of handling load-time weaving standard libraries, state association, synchronization, or array allocation join point. Nevertheless, Vilazon et al. [115] extended AspectJ to enable weaving of standard java class libraries. As discussed in Section 2.2.1 we couldn't use aspects due to the unacceptable overhead with such a generic approach and limitation to identify landmarks correctly.

Further, researchers developed the instrumentation frameworks such as FERRARI [28], a framework for Java that allows performance analysts to write instrumentation code on top of it. They tried to ease and simplify instrumentation having underneath framework to insert the instrumentation code correctly. Instrumentation code snippets one can write in AspectJ [72], in BCEL [42], ASM [33], or any other bytecode instrumentation library. Drawback of the approach is potential unnecessary overhead. Due to this reason we couldn't use it in our approach. Kerninst [110] as a framework for dynamic code injection into a running kernel, allows programmers to work on OS level and construct their own C++ tools for dynamic kernel instrumentation.

In summary, most of the approaches tried to reduce the overhead and still to provide accurately hot methods information in a profile. Their focus on capturing a profile that represents the execution time spent in each method is irrelevant for interactive applications. We argue that this data is insufficient for performance tuning. 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 (latency) of the program and not to the measured CPU time or hot calling context tree paths.

2.2.4 Sampling

A common technique used to achieve lower overhead in comparison to instrumentation techniques is sampling. Unfortunately, sampling may decrease accuracy. Usually, sampling is used to build a calling context and call path profiles. Walking the run-time call stack is one way to capture calling context. The Shark [9] performance tool provides sampled stack walking to identify hot call paths. Zhuang et al. [124] introduced an adaptive approach to sample only hot call paths. They walk the runtime call stack and stop when they hit a part of the calling context tree that they already have sampled. Mytkowicz et al. [90] brought an inferred call path profiling to collect minimal information from the running program using sampling and to infer the full call paths afterward offline. Bond et al. [30] implemented probabilistic calling context that continuously maintains a probabilistic unique value representing the current calling context with low overhead. He reported that for millions of unique contexts, a 32-bit PCC value has few conflicts. These approaches shares the idea of being standalone approach to detect slow part in the code. We think that such approach can't help any better than full instrumentation. It only saves unnecessary overhead while providing overall runtime picture. Sampling should happen only while interesting parts of the code relevant to slow response execute.

Moseley et al. [89] approximate the frequency of call paths, sampling long traces of instrumented code in parallel with normal execution, taking advantage of the trend of increasing number of cores. All the ideas above share one common thought that is to take samples only at the specific moments and not for the whole runtime. This way overhead is drastically reduced. Driven by this thought, we designed sampling profiler to take samples only if the response to the user request is slow. Unnecessary overhead is avoided as samples are not taken if a request handling is imperceptible.

Nevertheless, virtual machine takes samples in the safe points of execution and a consequence is lower accuracy. Thus, a profile picture might be completely wrong [91] if profiler accuracy is low. Imagine a profiler that produce a profile always showing method m() consumes 100% of run-time, even though m() is not invoked. One could classify this profiler as precise, but not as accurate. Binder tried to develop high accuracy sampling profiler [27]. He did a periodic sampling based on bytecode counting. The periodic invocation of the profiling agent depends on the number of executed bytecodes and not on an external timer. It profiles long running programs on application servers, where exact profiling would not be feasible due to the extreme overhead. This technique could be taken as trade-off between high accuracy of profiles and overhead.

Arnold and Grove worked on high accuracy call graph profiles [20] using event based sampling (method counter) but had high overhead. Later, they presented the work of combined timer and event based sampling where the overhead was below 1% but accuracy was about 60%.

In addition to instrumentation of landmark methods a sampling based call profiling approach would help us to complete the picture of taken path when long latency episode occurs. In general, it is essential to reduce perturbation. Experiences and findings in this area helps us with a design of our sampling mechanism.

2.3 Collecting Behavioral Data in the Field

Capturing and analyzing profiling data from multiple users is highly beneficial for discovering bugs and performance problems in the application run-time. Liblit et al. [79] showed how to gather random samples with very low overhead for users in distributed program sampling, and how to make use of the gathered information. They proposed an infrastructure where information of each user execution is transmitted to the central database. Collected data from these executions is analyzed to extract information that may help engineers to find and fix problems quicker. Moreover, some problems appear only in the specific context which is impossible to predict and reproduce in the lab.

Gathering information from multiple user provides a more accurate picture of how the software is actually used. It also allows better decisions about how to spend scarce resources on modifications. In particular, bugs that affect a large number of users are of higher priority than rare bugs. This kind of information is almost impossible to obtain from anywhere else than from the actual user executions. Automatically gathering data from the user executions allows automated analysis and ease problem identification. An added benefit of this approach might be that developers could easily see which application features are most commonly used.

Hangal implemented DIDUCE [59] (Dynamic Invariants Detection with Checking Engine) a system that extracts invariants dynamically from Java program executions. It is used to identify the root causes of programming errors. In particular, DIDUCE isolates timing dependent bugs. Google's Dapper [105] is a distributed systems tracing platform that collects and analyses data on their server. Their developers find the platform very useful helping them to identify and resolve problems they couldn't do before.

2.3.1 Bug Repositories

To our knowledge, there are a few studies done on the quality of bug reports. Bettenburg et al. [25] presented preliminary results for the Eclipse project using their manual bug quality prediction model. Later on they trained prediction models and did a survey with Apache and Mozilla as two additional projects.

Several studies used bug reports to automatically assign developers to bug reports [18] [35], to assign locations to bug reports [34], to track features over time [48], to recognize bug duplicates [101], or to predict effort for bug reports [117]. Ko et al. [77] conducted a linguistic analysis of the bug report titles. His results suggest that future bug tracking systems should collect data in a more structured way. Antoniol et al. [52] stressed that version archives and bug databases are detached. An integration would allow queries to locate the most faulty places easier and Kersten presented Mylyn [71], the tool that attaches a task context to bug reports. Bettenburg et al. [26] showed in his survey that the inability to reproduce errors is one of the biggest problems for developers. Implicitly, there is a need for tools that can capture the execution of a program on the side of a user and replay the execution on the side of a developer [65; 93; 119]. Moreover, users could help developers to fix bugs without creating bug reports and wasting time.

Glerum et al. [56] described Windows Error Reporting (WER), a system for collecting and classifying crash reports. It has been in operation for over ten years and is running on over one billion client computers. Two main benefits of post-deployment problem detection, which they demonstrate, are gathering information about problems in the context in which they occur, and combining the collective information from a large user base. Also, having information from multiple runtimes allows random sampling techniques with extremely low frequency and still high coverage of the runtime. We supported our approach with exactly this sampling technique when it is deployed with an application.

Liblit et al. [81] presented statistical debugging, where they came up with the idea to distribute their software slightly changed to report back about runtime. Their approach works for crashes, and we could imagine a button that sends the state of a slow program or even sends automatically generated reports with recognized slow events using our approach. Still, it is unclear how to extract sufficient information for rarely occurring bugs.

2.4 Interactive Application Characterization

To understand performance of interactive applications, a necessary component is the understanding of their behavior and internal characteristics. Zaparanuks et al. [121] studied whether real-world interactive applications have different characteristics from existing benchmarks. They used dynamic and static platform-independent metrics in addition to the micro-architectural metrics provided by hardware performance counters. The platform independent approach by Dufour et al. [45] proposes dynamic metrics for characterizing the behavior of Java applications but doesn't work for interactive applications. In their case metrics collection slows down program execution by several orders of magnitude, thus any reasonable interaction between user and application is impossible.

Flautner et al. [49] studied thread-level parallelism in interactive applications. They inspected whether desktop users should use multiprocessor machines for everyday tasks. They reported that existing Linux workloads exhibit thread-level parallelism, but also that more than 2 processors will not likely improve performance of the interactive application. Further, they found that shorter interactive episodes tend to have higher thread-level parallelism while thread-level parallelism is higher in the interactive episodes than the average for the entire run.

We did initial studies to characterize interactive Java applications [16] showing where application spent time, what is the average concurrency, how events distribution looks like and of what type they are.

To obtain statistically correct results the experiments have to be repeated. As it is impossible to manually repeat exact sequence it is hard to provide significant claims. We believe that a capturing tool able to correctly replay recorded user sessions [66] with interactive applications could be used to automate interactive application performance testing using our approach.

Part II

Approach

This part contains three chapters. Chapter 3 discusses the inception of the idea presented in this dissertation. Chapter 4 discusses the characterization of interactive applications. Finally, from the drawbacks learned in Chapter 3 and from the performance insights of the interactive applications discussed in Chapter 4, we discuss the final profiling approach in Chapter 5.

Chapter 3

Concept

To investigate idea of landmark latency profiling in-depth we implemented important aspects of the approach within a profiling tool developed in Java. As one of the most popular managed languages, Java runs on virtual machines that introduce many layers of abstraction between the application and the underlying hardware. Java supports agents which can transform the application code during run-time, which further allows dynamic instrumentation and easy deployment. However, a full-fledged implementation would require significant optimizations to make the profiling tool useful with acceptably low perturbation. That is why we have first implemented a static profiling tool to verify our statements and avoid unnecessary suspects in the results caused by perturbation.

3.1 Java GUI Toolkits

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.

Hereby, 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[1]. Thus, all GUI activities are essentially single-threaded.

[1] There are a few exceptions to this rule, most notably Display.asyncExec() in SWT and Event-

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, observers may be notified. These invocations of the observers are the program points we instrument and track with our 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[2], 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 observers. 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

Queue.invokeLater() in Swing that allow other threads to enqueue a Runnable object for later execution in the event thread.

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

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 observers in their implementation, and, in case these components are modal, implement their own event loop in native code. Thus, on the level of Java we cannot instrument their event loop or their observers to include their behavior in the latency profile.

3.2 Landmarks

As discussed in Section 1, perceived performance of interactive applications is directly related to the response time of the program in terms of latency. Our approach, in general, is based on the concept of a latency profile. A latency profile represents performance information from a single run of an interactive application. The goal is to determine and to characterize responsiveness of the application in the form of a cumulative latency distribution [67], and a latency profile that shows long latency events of the application along with information about their context.

Choice of methods is important in this approach. Their calls and returns thus represent the points where we should measure latency because each of them might represent a potential performance issue. As discussed in Section 1, we call them *landmarks*. A chosen set of landmarks should fulfill several properties.

First, they need to cover most of the execution of the program. Given that we only measure the latency of landmarks, we will be unaware of any long activity happening outside a landmark.

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

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

Landmarks in our approach are similar to pointcuts in aspect oriented programming. Pointcuts allow a programmer to specify join points which are defined places in a program execution such as method calls. Further, advices allow a programmer to specify code to run at a join point matched by a pointcut. The actions can be executed before, after, or around the specified join point. Thus, an advice corresponds to a code of our profiler which executes before each call and after each return of a landmark, and a pointcut corresponds to a landmark. However, we couldn't use aspects for two reasons: First, general purpose aspect weavers include overhead that may significantly perturb our results. Second, in some cases aspects can't precisely express a definition of a landmark. We discuss this issue in Chapter 5.

To make a choice of landmarks we looked at the design of the interactive applications. Modern interactive applications are based on component frameworks. Components in such applications observe each other following the "observer" design pattern. As a result, the sequence of operations happening during a single episode usually looks as follows: A user event (e.g. pressing a keyboard key) is dispatched to the focused GUI component. (e.g. a text field). When this component receives the event, it updates its internal state, and it notifies any registered observers of that state change. The observers then react to that notification by updating their own state, and by notifying their own observers. Thus, a single user event may percolate throughout the entire application, sometimes triggering a snowslip effect of observer notifications.

Moreover, the observers can register and unregister their interests dynamically. For example, as long as the "Outline View" in Eclipse is visible, it will observe the "Java Editor". As soon as the user disables the "Outline View", it will unregister and will not receive further notifications. The operations (and method calls) that occur as a result of a user event thus depend on the user's current arrangement of windows and views. As a result, the same latency bug (say, a slow "Java Editor" component) can lead to significantly different calling context trees, depending on the set of additional components (e.g. the "Outline View") that were reacting to the event that triggered the bug. Thus, we classified observers as important and good candidates for landmarks.

Nevertheless, observers are not the only promising candidates. Most interactive applications update their interfaces whenever something is changed. For example, Java's AWT/Swing toolkit GUI refresh is handled by a few methods of the Component and JComponent classes. Swing always calls at least one of them whenever a component needs to be painted. If rendering of such a component is complex, it may take a perceptibly long time to execute. Further more, all managed languages often execute a part of the application code that reside in the native libraries. Many of the standard library classes depend on JNI to provide functionality to the developer, e.g. file I/O and sound capabilities. Execution of these calls may take long and be the actual cause of perceptible performance. Both GUI refresh and native methods are another good candidates to be classified as landmarks. Additionally, garbage collectors can cause significant pause times. Tracing of garbage collections may denote which method execution time was not due to the implementation but the stop-the-world activity.

As we discussed eariler in Section 3.1, developers of interactive Java applications are faced with the choice between two powerful GUI toolkits, Swing and SWT, and we aimed at providing a latency profiling approach that works with both.

The instrumentation consists of two parts instrumenting particular methods which are marked as landmarks and instrumenting the toolkit's event dispatching code. The main goal of our instrumentation is to capture all calls to and returns from chosen set of landmarks. For this we automatically instrument all corresponding call sites in the application and runtime library classes.

Further, not all GUI events are necessarily handled by some of the mentioned landmarks, but all GUI events that reach Java are dispatched in a central location. Thus, the event dispatch location with both toolkits is a crucial place to mark as landmark. While instrumenting, this central dispatch code gives us a more complete picture of the event handling latency but it does not provide us with much information about the code that is responsible for the long latency. That is why instrumenting other landmark calls is also of critical importance.

3.3 Stack Sampling

While discussed landmarks provide great insight into which components are responsible for the perceptible latency episodes, the relatively coarse granularity of their profiles can make the identification of the precise cause of long latency difficult. Using a finer granularity by instrumenting *all* method calls and returns is too costly and can cause significant perturbation. Instrumenting only the methods that are *relevant* for detecting latency bugs is hard to impossible. Our solution is to augment the latency profile with a profile generated by periodically sampling the call stack.

Stack sampling of all the threads during run-time would allow calling context tree construction. The offline analysis will then find all call stack samples that took place during a long-latency episode, and merge them into a sampled calling context tree. Like a traditional calling context tree, this tree would represent the hotness of methods, showing only the hot methods that contributed to the given long latency episode. Such a tree thus will combine the user-centric view of episode latency with the developer-centric view of code hotness, and allows a developer to find the methods responsible for perceptible latency.

Finally, because a given landmark calling context tree is built from the stack samples taken during all of its notifications, this will allow us to reduce the sampling rate, and thus the overhead and perturbation due to call stack sampling, while still getting a realistic approximation of the complete calling context tree.

3.4 What data to collect?

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

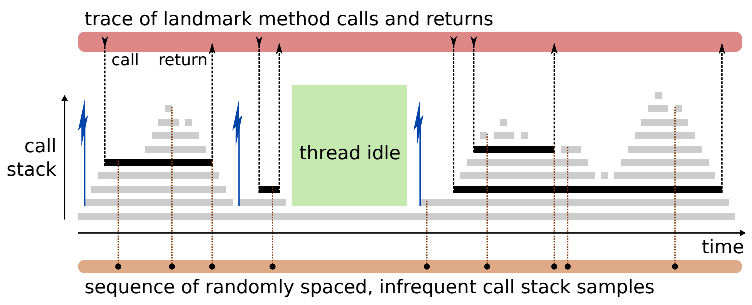

trace of landmark method calls and returns
call
return
call stack
thread idle
time
sequence of randomly spaced, infrequent call stack samples

Figure 3.1. Session Report

Figure 3.1 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 a method activation. In our approach, the agent collects two kinds of information. First, it captures calls and returns from so-called *landmark* methods (black rectangles). This landmark trace contains the timing information necessary to measure lag. Second, it captures rare, randomly spaced call stack samples of the running threads. Each stack sample contains the complete calling context of the thread at the time the sample was taken. This sequence of stack samples provides information about what the thread was doing throughout a long-latency landmark method invocation.

Figure 3.1 shows information for only one thread. We do collect the same information for all threads. However, for many interactive applications, most threads are blocked most of the time. Moreover, for most current GUI toolkits, only one thread, the *event dispatch thread*, is allowed to interact with the user (process user input and update the GUI). For many applications, if the user is not interacting with the application, the GUI thread becomes idle, waiting for new user requests[3]. The figure shows such an idle interval in the center. While a thread is idle, the agent does not need to sample the thread's stack. 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 observer call, paint call, native 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 observers and native methods 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 observer, paint, or native call and 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

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

of the eight kinds of trace records, we include a field with the trace record type (e.g. “ObserverCall”), the thread that caused that record (using Thread.getId()), and a timestamp (using System.nanoTime()). ObserverCall, ObserverReturn, PaintCall, PaintReturn, NativeCall, and NativeReturn trace records also include the class and method name of their call targets.

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

Figure 3.2. Segment of a trace

Figure 3.2 shows ten trace records from a trace of our “MouseButton” microbenchmark (Section 3.7). 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 observers (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 observers are notified. First we see a nesting of two listeneners: while handling the eventDispatched observer notification, the SelectiveAWTEventListener notifies the LightweightDispatcher observer. Both of these classes are part of the Swing implementation. The third observer 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 observer has a particularly high latency of 903 milliseconds, as we can see from the timestamp field. It is these kinds of calls we intend to identify with our approach.

3.5 Trace Analysis

The trace analysis converts a trace into a cumulative latency distribution and a latency profile. The profile shows, for each landmark, 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 landmark and we keep track of the number of calls and their latencies for each landmark.

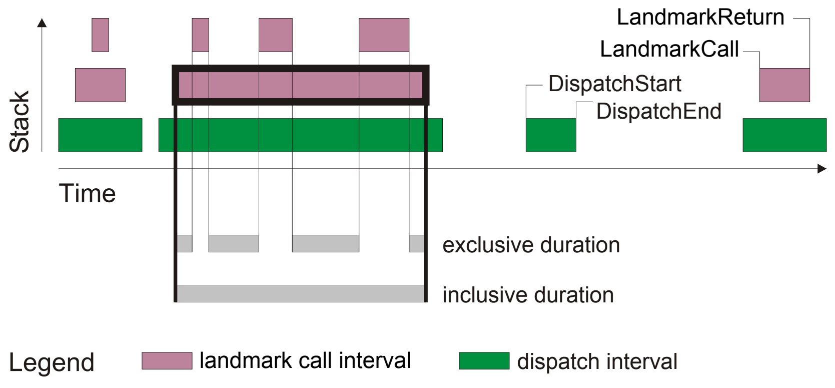


Figure 3.3. Trace Analysis

A slight complication arises from nested observer calls: the activity happening during a landmark call may itself lead to a calls of other landmarks. **Figure 3.3** gives a visual representation of a trace. Time goes from left to right. The y axis represents nesting (an abstraction of the call stack) of landmark calls and/or event dispatches. The first dispatch (without annotations) includes a nested landmark call, which itself includes another nested landmark call. We will now focus on the landmark call (bold outline) within the second dispatch. The annotations in the figure show how we compute its inclusive duration (end minus start time stamp) 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 landmark calls.

Both inclusive and exclusive durations represent important information. The inclusive latency shows how long that landmark (and all its nested landmarks) took. It is the time noticed by the user. On another side, the exclusive time helps to find the landmark responsible for the long latency. As observers are usually loosely coupled, it means that an observer that triggers the notification of another observer usually is not aware of who its nested observer is, and what it does. Depending on the situation, blaming an observer for the latency caused by nested observer might thus be wrong. Thus, exclusive duration might be more important to a developer to understand better where the time is consumed.

As same observers may be invoked from different contexts executing through the completely different paths it could end up with completely different behavior. Thus, knowing information about observer's ancestors can be crucial for overall analysis. One way is to instrument each method call in the application code. Nevertheless, it would enormously perturb the execution and thus impact all the latencies significantly. A promising compromise would be to use a sampling technique to obtain calling context trees of landmarks.

3.5.1 Modal phase injection

Traditional call profilers provide information about inclusive and exclusive time spent in each method. Unfortunately, in the context of our latency profiling of interactive applications, the analysis is incomplete, because it does not address the problem of modality. If an observer 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 observer. This clearly is wrong.

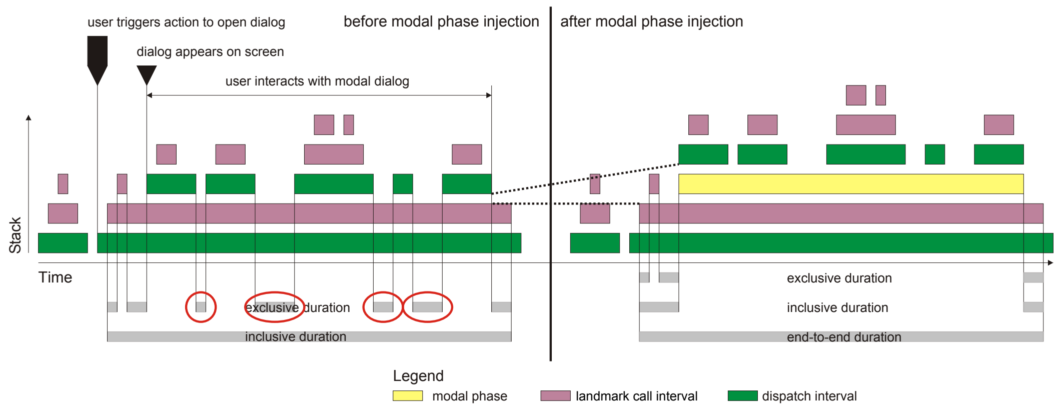


Figure 3.4. Trace Analysis and Injection of Modal Phases

Figure 3.4 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 observer landmark 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 observer (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 3.4 with surrounding ovals). The observer 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 observer interval that opened the dialog.

Figure 3.4 shows how we use our heuristic to transform the original trace into a trace including a new type of interval: 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 observer call interval and its nested dispatch intervals. This modal phase interval represents the interval during which the modal dialog blocked our observer landmark, 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 observer landmark call duration in a new way: we compute *end-to-end duration* as the end minus start time stamp, *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.

3.5.2 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 a landmark 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 is a histogram that simply consists of the episode counts binned by inclusive latency. Example distribution is shown on Figure 3.5. In this example 10 episodes are with an inclusive duration of 100 ms, approximately 5 episodes with an inclusive duration of 300 ms, etc. 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.

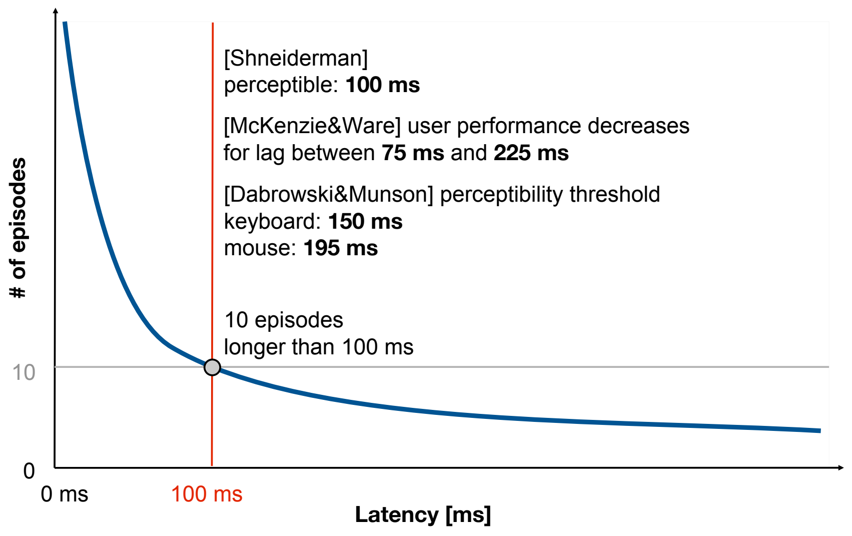


Figure 3.5. Cumulative Distribution

3.6 Trace Viewer

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
```

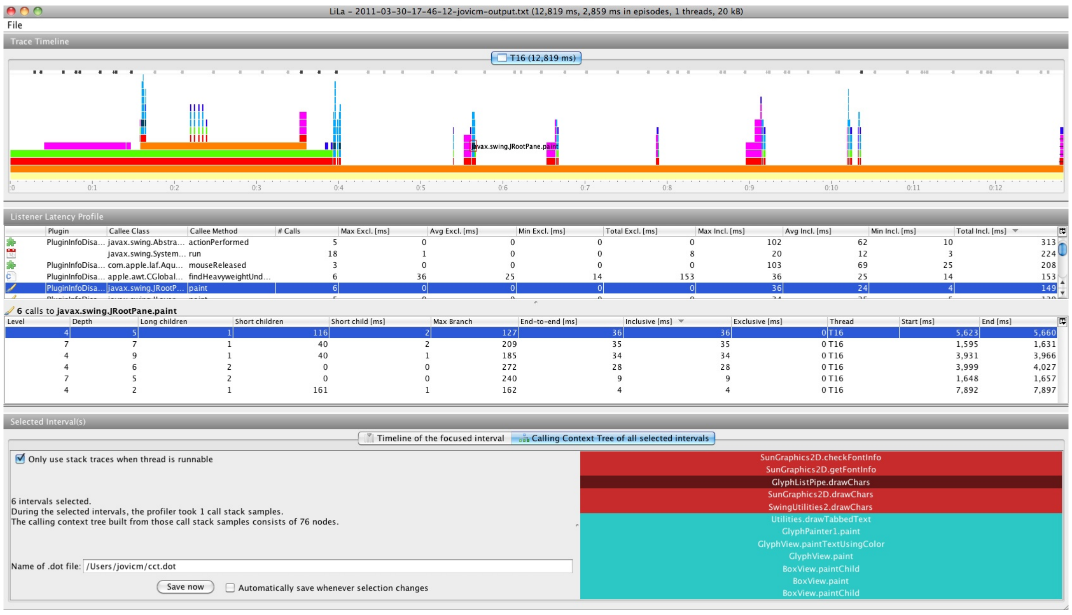
Figure 3.6 shows the interface of LagAlyzer. It is simple and straight forward for use. It has a single **File** menu with a single **open** choice. It opens standard file open dialog where we may navigate to a desired trace file.

LagAlyzer interface has 4 distinctive areas split horizontally. The first one from the top represents full runtime of GUI thread for the chosen trace file. It is shown separately in **Figure 3.7**.

The X axis represents the execution time. For example, **Figure 3.7** shows a very short run of an application. By looking at the x-axis value we may say that the run length is roughly 13 seconds.

Not only one landmark can execute during some period of time. When an observer method occurs which handles mouse clicks, it may trigger some other observer notifications, like a 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.


LiLa - 2011-03-30-17-46-12-jovicm-output.txt (12,819 ms, 2,859 ms in episodes, 1 threads, 20 kB)

	Plugin	Callee Class	Callee Method	# Calls	Max Excl. [ms]	Avg Excl. [ms]	Min Excl. [ms]	Total Excl. [ms]	Max Incl. [ms]	Avg Incl. [ms]	Min Incl. [ms]	Total Incl. [ms]
	PluginInfoDisa…	javax.swing.Abstra…	actionPerformed	5	0	0	0	0	102	62	10	313
		javax.swing.System…	run	18	1	0	0	8	20	12	3	224
	PluginInfoDisa…	com.apple.laf.Aqu…	mouseReleased	3	0	0	0	0	103	69	25	208
	PluginInfoDisa…	apple.awt.CGlobal…	findHeavyweightUnd…	6	36	25	14	153	36	25	14	153
	PluginInfoDisa…	javax.swing.JRootP…	paint	6	0	0	0	0	36	24	4	149

6 calls to **javax.swing.JRootPane.paint**

Level	Depth	Long children	Short children	Short child [ms]	Max Branch	End-to-end [ms]	Inclusive [ms]	Exclusive [ms]	Thread	Start [ms]	End [ms]
4	5	1	116	2	127	36	36	0	T16	5,623	5,660
7	7	1	40	2	209	35	35	0	T16	1,595	1,631
4	9	1	40	1	185	34	34	0	T16	3,931	3,966
4	6	2	0	0	272	28	28	0	T16	3,999	4,027
7	5	2	0	0	240	9	9	0	T16	1,648	1,657
4	2	1	161	1	162	4	4	0	T16	7,892	7,897

Selected Interval(s)
Timeline of the focused interval
Calling Context Tree of all selected intervals

☑ Only use stack traces when thread is runnable

6 intervals selected.
During the selected intervals, the profiler took 1 call stack samples.
The calling context tree built from those call stack samples consists of 76 nodes.

Name of .dot file: /Users/jovicm/cct.dot

☐ Automatically save whenever selection changes

SunGraphics2D.checkFontInfo
SunGraphics2D.getFontInfo
GlyphListPipe.drawChars
SunGraphics2D.drawChars
SwingUtilities2.drawChars
Utilities.drawTabbedText
GlyphPainter1.paint
GlyphView.paintTextUsingColor
GlyphView.paint
BoxView.paintChild
BoxView.paint
BoxView.paintChild

Figure 3.6. LagAlyzer interface

Each landmark type is visualized with a different color. For example, event dispatch intervals 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 can zoom into the specific region of the runtime. When the desired time range is selected, we release the mouse button and it will zoom into the selected region. To zoom out, we use the context menu which appears on the right mouse click on the time line.

Note that the visualization 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 execution. That is why periodically taken samples during runtime are used. On the top of **Figure 3.7** 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 start and end times of landmark methods and stack dumps with information about all other methods. For example, let our GUI thread start from the main method and call some observer notification method which further calls other method. If the stack is dumped when our application was in that other method then the stack will contain all three methods, while the runtime line will have only one rectangle shown for the only landmark method which is in this case an observer.

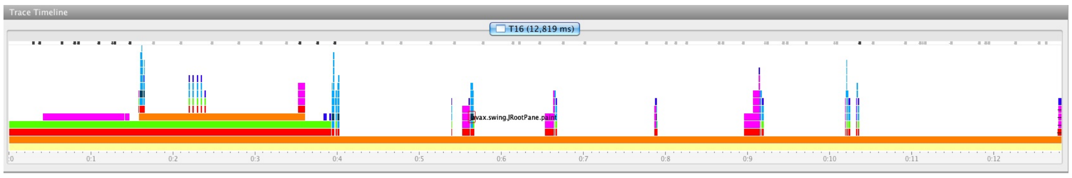
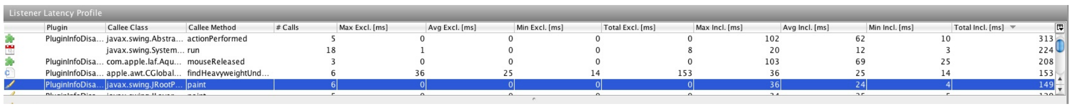


Figure 3.7. LagAlyzer run-time time line

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



Listener Latency Profile

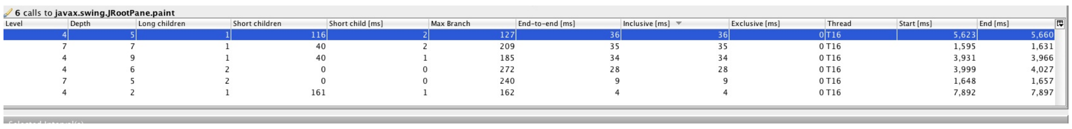
	Plugin	Callee Class	Callee Method	# Calls	Max Excl. [ms]	Avg Excl. [ms]	Min Excl. [ms]	Total Excl. [ms]	Max Incl. [ms]	Avg Incl. [ms]	Min Incl. [ms]	Total Incl. [ms]
	PluginInfoDisa...	javax.swing.Abstra...	actionPerformed	5	0	0	0	0	102	62	10	313
		javax.swing.System...	run	18	1	0	0	8	20	12	3	224
	PluginInfoDisa...	com.apple.laf.Aqu...	mouseReleased	3	0	0	0	0	103	69	25	208
	PluginInfoDisa...	apple.awt.CGlobal...	findHeavyweightUnd...	6	36	25	14	153	36	25	14	153
	PluginInfoDisa...	javax.swing.JRootP...	paint	6	0	0	0	0	36	24	4	149

Figure 3.8. LagAlyzer Landmarks

Figure 3.8 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 have a brush icon. The second cell shows the name of JAR file where the class of the corresponding landmark resides. The third and the fourth cells represent the landmark class and method name. The next cell shows landmark occurrence count. This cell is followed by eight cells in two groups of four. The first group shows information about landmark exclusive execution time. The second group 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 the total time consumed while handling all the invocations during runtime for a specific landmark.

Figure 3.9 shows detailed information for occurred landmark calls during runtime. Every table row stands for one call of the corresponding landmark. This table is tightly connected to the previous table shown in **Figure 3.8**. Information in this table always relates to a landmark selected in the table shown in

Figure 3.8. It breaks down collected landmark information in one row per each occurrance.



6 calls to **javax.swing.JRootPane.paint**

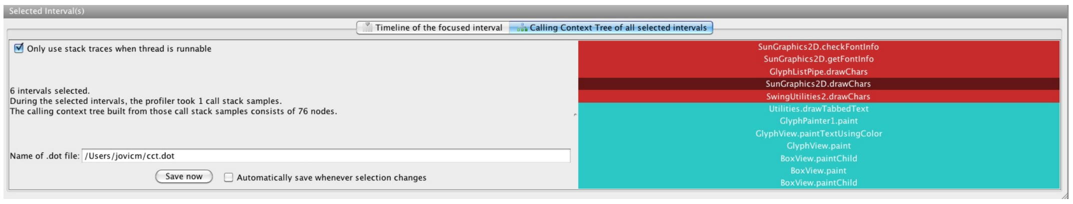
Level	Depth	Long children	Short children	Short child [ms]	Max Branch	End-to-end [ms]	Inclusive [ms]	Exclusive [ms]	Thread	Start [ms]	End [ms]
4	5	1	116	2	127	36	36	0	T16	5,623	5,660
7	7	1	40	2	209	35	35	0	T16	1,595	1,631
4	9	1	40	1	185	34	34	0	T16	3,931	3,966
4	6	2	0	0	272	28	28	0	T16	3,999	4,027
7	5	2	0	0	240	9	9	0	T16	1,648	1,657
4	2	1	161	1	162	4	4	0	T16	7,892	7,897

Figure 3.9. LagAlyzer Individual Landmark Execution

"Level" column tells how deep some landmark invocation was on the stack, while "Depth" column tells how many more were nested. They are followed by three columns related to the landmark's children (other landmarks). The column "Long children" shows how many were long, while "Short children" and "Short child" show how many were short and their execution time. The terms long and short come from the thresholds of 3 ms and 100 ms. If it ran longer than 100 ms then it is treated as long. This threshold has been shown to be the border of human perceptiblity. On the other hand, if it ran in less than 3 ms, then that call is omitted from the trace (avoid space problems) but basic execution information is saved to be shown in the table. "Max Branch" column shows the total count of the landmark's nested children. "End-to-End" column represents time needed to handle the landmark. This time value represents the time needed to execute the landmark method including its dispatch interval. The following three columns are "Exclusive time", "Inclusive time", and "Thread" columns that show exclusive execution time, inclusive execution time, and thread ID of the thread that executed the landmark method. The last two columns are "Start" and "End". They show the exact time (in milliseconds) when a landmark method execution started and ended.

In general, this table may help developers in a few aspects. For example, let's assume that there were 10 occurrences 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. This is a sign of an initialization problem. For these reasons it is good practice to first look at this table and individual call information, and to confirm the suspects.

The bottom region of the interface is shown in the **Figure 3.10**. The left tab shows a landmark call stretched to the whole screen which is selected either in the table or on the runtime shown in the **Figure 3.7**. This way we can quickly see concrete names of classes and methods at readable size. The right tab shows


Selected Interval(s)
Timeline of the focused interval
Calling Context Tree of all selected intervals
Save now

☑ Only use stack traces when thread is runnable

6 intervals selected.
During the selected intervals, the profiler took 1 call stack samples.
The calling context tree built from those call stack samples consists of 76 nodes.

Name of .dot file: /Users/jovicm/cct.dot

☐ Automatically save whenever selection changes

```
SunGraphics2D.checkFontInfo
SunGraphics2D.getFontInfo
GlyphListPipe.drawChars
SunGraphics2D.drawChars
SwingUtilities2.drawChars
Utilities.drawTabbedText
GlyphPainter1.paint
GlyphView.paintTextUsingColor
GlyphView.paint
BoxView.paintChild
BoxView.paint
BoxView.paintChild
```

Figure 3.10. LagAlyzer Calling Context Tree

the calling context tree which is constructed from the stack dumps taken for the selected landmark. The calling context tree is not constructed only from the stack dumps of a landmark's 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. Our representation of the tree is given in **Figure 3.10**. The width of a calling context's rectangle corresponds to percetage of stack dumps taken while being in that context as a fraction of the total number of stack dumps taken in all the contexts. Thus, it is very important not mix this representation with the one shown in **Figure 3.7**. The x-axes of the previous figures represent time, while this x-axis represents hotness.

3.7 Validation

In this section we validate the approach on a set of micro-benchmarks. We used our visualization tool, discussed in Section 3.6 to verify that content of the trace files is as expected. More, we evaluate our approach for profiling real-world applications.

3.7.1 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) the profiler mistakenly reports a short activity as a long latency observer call, or where it is unable to capture and report long-latency activities.

Benchmark	Swing Feature	SWT Feature
Keyboard input	KeyListener	KeyListener
Mouse motion	MouseMotionListener	MouseMoveListener
Mouse button	MouseListener	MouseListener
Painting	paint call	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 3.1. Validation Micro-Benchmarks for Swing and SWT

Table 3.1 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 observers, and the observers simulate expensive computations by waiting for a specifiable delay (using a call to Thread.sleep()). We expect any delays occuring in these observers to directly appear in the profile.

3.7.1.1 Handling keyboard and mouse events

The first three benchmarks test observers that handle mouse or keyboard input. Such observers 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 observers registered for the respective mouse or keyboard events.

As expected, studying the traces from these benchmarks with the profiler we find that the profiler points out the long-latency observers at the very top of its profile.

3.7.1.2 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 Viewer's Trace Timeline).

This benchmark demonstrates that, unlike in SWT, painting a custom component in Swing is not done by registering a observer, but by overriding a paint method in the component's class. Correspondingly, the profiler does not report any long-latency *observer* calls in its profile. However, thanks to our paint call instrumentations the profile correctly reports long-latency paint calls. For the SWT version of this benchmark, the profiler shows all the PaintListener.paintControl() notifications as expected.

3.7.1.3 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 profile for SWT does not show any long latency observer invocations, but only shows long-latency nondescript event dispatches. This is because SWT does not use observers for handling timer firings, but uses a Runnable instead[4]. As expected, the Swing version of this benchmark shows all the actionPerformed() notifications corresponding to the timer firings.

3.7.1.4 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 observer, they erroneously inflate the lateny of that observer, a problem that our modal phase injection might fix.

The 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 observer, and so do not introduce an error into the observer 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 3.5, 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).

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

3.7.1.5 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 observer 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 an observer (as is normal), it blocks the thread and does not cause nested event dispatches visible within Java. This affects the observer latency profile (our observer is charged for the entire duration the user looked at the message box), as well as the latency distribution (adding one overly long episode).

3.7.1.6 Accuracy

In all of our micro-benchmarks we deliberately introduced latencies between 10 ms and 1 second by calling Thread.sleep(). We thus approximately[5] know the actual latency. We have found that the latency reported by the profiler 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.

3.7.1.7 Summary

In conclusion, the observer latency profile computed by the profiler provides detailed information about most of the important interactive events. It lacks detailed information about SWT timers, but still captures the latency of such events because it reports all event dispatches, no matter whether they include observer 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 observer 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

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

message box, such long latency observer calls are easy to spot and skip by the performance analyst. Moreover, SWT dialogs implemented in Java are properly handled by our heuristic.

3.7.2 Real-World Applications

In this subsection we evaluate the applicability of our profiler to real-world applications. We picked our own trace viewer and the NetBeans IDE, which, with its over 45000 classes is one of the most complex open-source interactive Java applications.

3.7.2.1 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 work 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 observer latency distribution of the instrumented and the uninstrumented application. Unfortunately, we cannot measure the observer 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 micro-benchmarks 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 trace 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 the profiler with a study of real users, and to use online analysis approaches to further reduce the overhead where necessary.

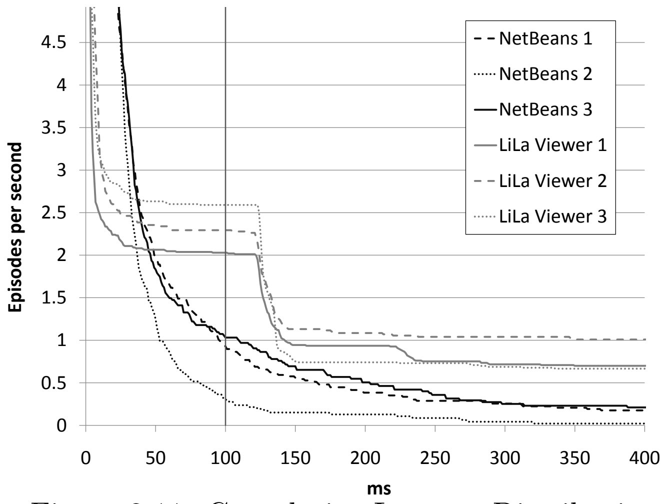


Figure 3.11. Cumulative Latency Distribution

3.7.2.2 Results

Figure 3.11 shows the perceptible performance of NetBeans and our trace viewer. It includes three curves for each benchmark[6]. 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 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 the profiler: Every second the profiler 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 the viewer. The profiler pointed out several causes for the inferior performance of the trace viewer; the most significant was a observer reacting to changes in a table model. The observer 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 trace viewer.

[6] For clarity we only include three of the nine runs for each benchmark in Figure 3.11. The other runs have similar curves.

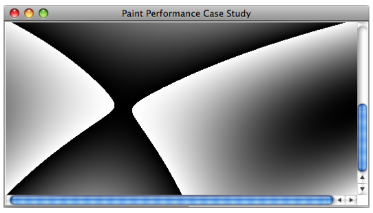
3.8 Case Studies

The goal of our profiling approach is to support developers or performance engineers in quantifying, understanding, and improving the perceptible performance of their applications. In this section we present two concrete case studies highlighting the use of our approach. The first study is concerned with developing a responsive visualization of a large 2-dimensional heat map. The second study shows the impact of different approaches to retrieving results produced by querying a remote server for display in a large table.

3.8.1 Rendering a Heat Map

We want to visualize the data stored in the following straightforward data model. The data model represents a rectangular matrix, and each cell represents a heat value as an int.

```
public interface HeatMap {
  public int getWidth();
  public int getHeight();
  public int getValue(int x, int y);
}
```



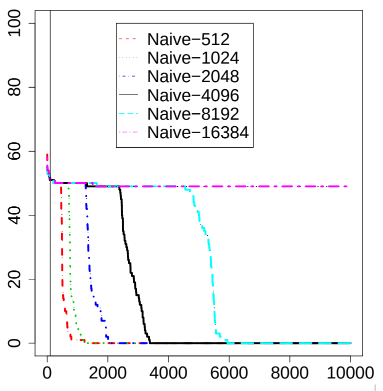
The heat map visualization shall show each cell in the heat map using a color representing its value. Dark colors represent small values, while bright colors represent large values. Given that heat maps can be large (hundreds to thousands of values along both axes), we want to place the visualization inside a scroll pane, so the user can scroll to different regions of interest. We do not want to provide support for zooming, and we map each value of the matrix to one pixel on the screen. This simple problem has a surprisingly rich set of possible solutions, and each solution has its specific impact on perceptible performance.

We create a GUI canvas (a subclass of Swing's JComponent class) to visualize the heat map, and we place that canvas inside a Swing JScrollPane. The canvas does not need to know about scrolling, it just has to know the width and height (in pixels) it needs in order to draw the entire heat map, and it has to be able to paint its contents. The scroll pane handles the scrolling and clipping of the visible part of the heat map canvas.

3.8.1.1 Naive implementation

We start with a naive implementation of the canvas, which, whenever it is asked to paint itself, will iterate through the entire heat map and render each value

```
public class NaiveMatrixCanvas extends JComponent {
 private HeatMap map;
 //...
 protected void paintComponent(Graphics g) {
  for (int x=0; x<map.getWidth(); x++) {
   for (int y=0; y<map.getHeight(); y++) {
    int value = map.getValue(x, y);
    Color c = new Color(value, value, value);
    g.setColor(c);
    g.drawLine(x, y, x, y);
   }
  }
 }
}
```



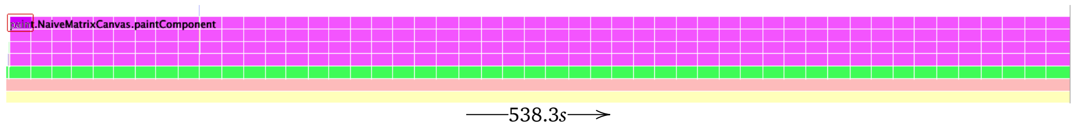


Figure 3.12. Naive Canvas and its Perceptible Performance

as a pixel. **Figure 3.12** shows an excerpt of the source code and the resulting perceptible performance for different map sizes. The perceptible performance, is visualized in two ways, (1) in the form of a cumulative latency distribution on the right, and (2) in the form of a trace at the bottom.

The x-axis of the cumulative latency distribution plot represents latency in milliseconds. In this figure it extends to 10 seconds. The y-axis shows the absolute number of episodes that took longer than x. We wrote a small program that embeds the canvas inside a JScrollPane that has a visible area of 256 by 256 pixels, and adds that scroll pane into a JFrame. The program then programmatically scrolls left and right by 256 pixels, thus triggering a total of 50 paint requests. Given that, a program run will contain at least 50 episodes, and the curves in the cumulative latency distribution extend to above 50 at $x = 0$. The cumulative latency distribution plot shows 6 curves representing six program runs with different heat map sizes. We fixed the height of the heat map to 512, and we show one curve for each of the six widths we studied.

The trace at the bottom visualizes the run with the biggest heat map of 16384 by 512 pixels. The x-axis of the trace represents wall-clock time. The y-axis represents an abstraction of the call stack. The listener, paint, and native calls are represented as rectangles stacked up along the y-axis. The event dispatches also show up as rectangles, usually towards the bottom of the stack. The bottom of the stack consists of one wide rectangle representing the entire lifetime of the GUI thread. On top of this is a rectangle representing the modal phase that corresponds to the lifetime of the main window. At the bottom of the trace we indicate the end-to-end execution time with an arrow.

For this naive canvas, our program ran for 538.3 seconds for the biggest heat map. The cumulative latency distribution in Figure 3.12 shows that the responsiveness of the program clearly depends on the size of the heat map: The curve for the 16384 pixel wide heat map extends past 10 seconds, meaning that each of the 50 repaint requests took longer than 10 seconds. This is over 100 times above the 100 ms perceptibility threshold. Smaller maps lead to more responsive behavior, but even the map that is 512 pixels wide takes a clearly perceptible 514 ms for each paint request.

3.8.1.2 Using a pool of color objects

A look at the NaiveMatrixCanvas class shows that this naive widget creates a new Color object for each pixel it paints. Given the large number of pixels (our biggest heat map has eight megapixels), these object allocations may impact perceptible performance. As a straightforward optimization, we thus introduce a pool of 256 pre-allocated colors, one of each possible heat value.

Figure 3.13 shows this optimized variant of the canvas and the resulting perceptible performance. Unfortunately this optimization has little effect on the latency distribution. The six curves still extend to almost the same latency. The average paint latency for the largest heat map only was reduced to 10382 ms (from the 10933 ms for the naive version).

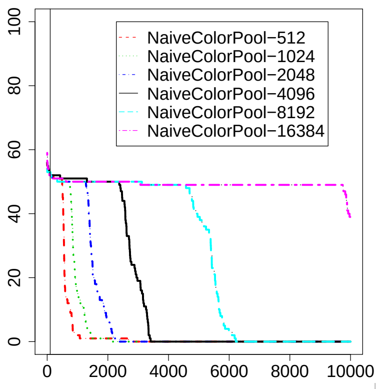
3.8.1.3 Clipping

The next optimization is based on the finding that we always only see a small piece (the size of the scroll pane's viewport, which we fixed to 256 by 256 pixels in our experiments) of the large heat map. To render the entire map at every paint request thus constitutes a large waste of time. Class ClippedColorPoolMatrixCanvas in **Figure 3.14** thus uses the clipping information provided by the scroll pane to focus the painting to only the pixels that can actually be seen. Thus, in our usage scenario, instead of rendering 8 megapixels we render only 64 kilopixels per paint request.

The cumulative latency distribution in Figure 3.14 shows the big gain in perceptible performance due to this simple optimization. It also shows, as expected, that the latency does not depend on the size of the heat map anymore. The aver-

```
public class NaiveColorPoolMatrixCanvas extends JComponent {
 private HeatMap map;
 private Color[] colorPool;

 //...
 protected void paintComponent(Graphics g) {
  for (int x=0; x<map.getWidth(); x++) {
   for (int y=0; y<map.getHeight(); y++) {
    int value = map.getValue(x, y);
    Color c = colorPool[value];
    g.setColor(c);
    g.drawLine(x, y, x, y);
   }
  }
 }
}
```



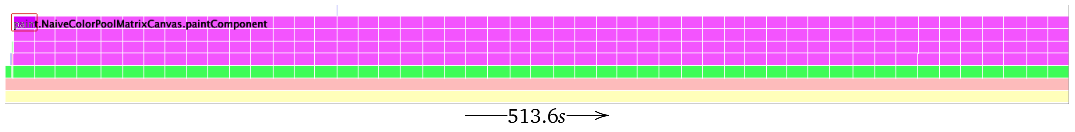


Figure 3.13. Performance with Color Pool

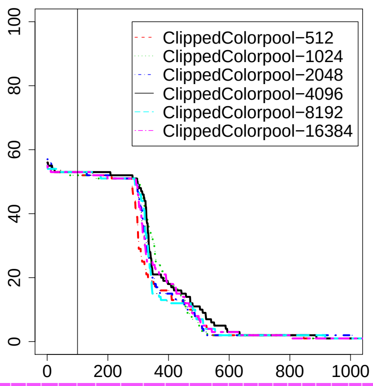
age paint request latency for the largest heat map size now dropped to 367 ms. While this is far better than the 10382 ms of the prior approach, it still is above the 100 ms perceptibility threshold.

3.8.1.4 Image buffer

Each time we paint in ClippedColorPoolMatrixCanvas, we iterate through all visible pixels and invoke Graphics.drawLine() to render each pixel. The Graphics class does not provide any methods to set a pixel, thus we need to render some shape to affect a pixel. We render a line from point (x,y) to point (x,y), causing just one pixel to be affected. This approach seems potentially slow. We thus use the following optimization: we render the entire map into an image buffer (a color bitmap) in the beginning, and at each paint request we just copy the necessary part of that bitmap to the screen. Given that copying pieces of bitmaps is fast, this should drastically reduce latency.

Figure 3.15 shows this optimized widget and the resulting perceptible performance. The latency curves now drop down to almost 0 already below the 100 ms perceptibility threshold. This means that the painting now essentially became imperceptible. Indeed, our own interactions with this version of the

```
public class ClippedColorPoolMatrixCanvas extends JComponent {
 private HeatMap map;
 private Color[] colorPool;
 //...
 protected void paintComponent(Graphics g) {
  Rectangle b = g.getClipBounds();
  for (int b.x=0; x<b.x+b.width; x++) {
   for (int b.y=0; y<b.y+b.height; y++) {
    int value = map.getValue(x, y);
    Color c = colorPool[value];
    g.setColor(c);
    g.drawLine(x, y, x, y);
   }
  }
 }
}
```



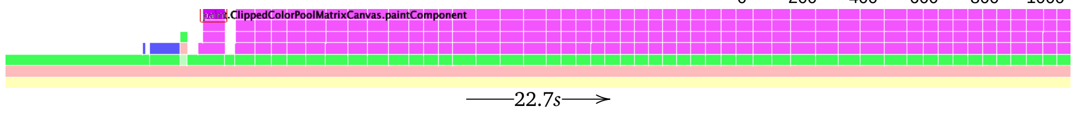


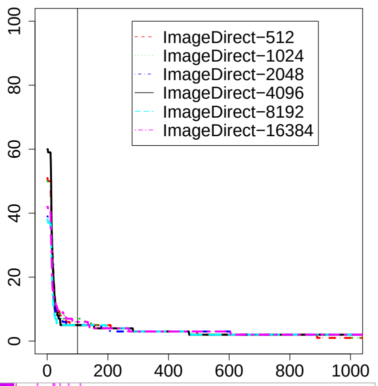
Figure 3.14. Performance with Clipping

program, unlike with any of the prior versions, feel smooth and responsive. However, as the trace at the bottom of Figure 3.15 shows, this optimization comes at a cost: the first time we want to paint the canvas, we first have to render the heat map into the background image buffer. This leads to the long latency episode highlighted in the trace: while all subsequent paint calls take less than 43 ms (many of them even take less than 10 ms and are thus omitted from the trace visualization), this initial paint call takes 5479 ms for the largest heat map.

3.8.1.5 Discussion

It was *not* the goal of this case study to produce a heat map visualization with good perceptible performance. The goal was to demonstrate the use of our profiling approach, and to relate the output of our latency profiler to the causes of bad performance in the source code. One could obviously conduct the same case study by manually instrumenting strategic points in the program (e.g. measuring the time at the entry and exit of the paintComponent methods in the given classes). However, our approach gathers and visualizes all that information automatically. It instruments *all* strategic points (observer notifications, paints,

```
public class ImageDirectMatrixCanvas extends JComponent {
 private HeatMap map;
 private BufferedImage image;
 //...
 private void paintToImage() {
  image = (BufferedImage)createImage(
            map.getWidth(), map.getHeight());
  //... paint entire map into image
 }
 protected void paintComponent(Graphics g) {
  if (image==null) {
   paintToImage();
  }
  Graphics2D g2 = (Graphics2D)g;
  g2.drawImage(image, null, 0, 0);
 }
}
```



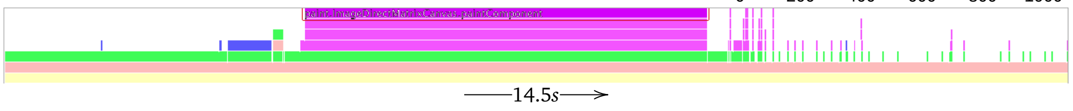


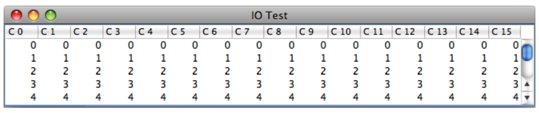
Figure 3.15. Performance with Image Buffer

and native calls), it measures the times, builds traces, latency distributions, and profiles, without signifcantly perturbing the measurements. While for the purpose of simplicity in this case study we discussed the performance problem (the canvas widget) in isolation, our profiling approach would find the problem even if this widget was embedded in a complex real-world application with hundreds or thousands of other components.

3.8.2 Retrieving Query Results

The previous case study investigated the performance of a visualization application. In this case study we investigate a different and possibly more common type of interactive Java application, a client application that queries a database and presents the results in a table. The database could run on a database server accessed through JDBC, but for the purposes of this case study we implemented a simple server as a Java RMI application. To keep the example simple, the server does not actually provide a way to specify the query to be executed. It only allows us to retrieve the results of a predefined hard-coded query.

```
public interface TableModel {
  //...
  Object getValueAt
      (int row, int col);
}
```



IO Test

C 0	C 1	C 2	C 3	C 4	C 5	C 6	C 7	C 8	C 9	C 10	C 11	C 12	C 13	C 14	C 15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4

Figure 3.16. Swing's TableModel Interface and GUI for Browsing Query Results in Tabular Form

Figure 3.16 shows the GUI of such an interactive application. It essentially consists of a JTable placed inside a JScrollPane. The table has a given number of columns. The number of rows corresponds to the number of records in the query's result. In Swing, a JTable determines its contents by accessing a TableModel. The TableModel provides methods to determine the number of colums, the title of each column, the data type of each column, the number of rows, and the value of each cell. In this case study we provide multiple implementations of TableModel and use our listener latency profiler to compare the perceptible performance of different approaches for retrieving remote query results. The only difference between the approaches is in the implementation of the TableModel interface.

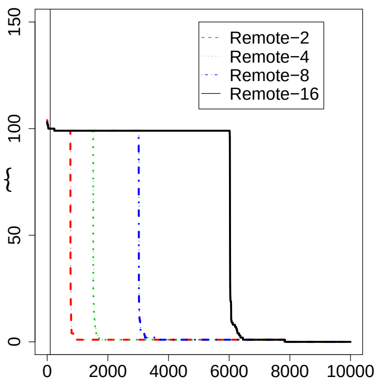
The usage scenario of our application consists of scrolling through the table by paging down 100 rows at a time. We navigate a result set of 10000 rows, and thus end up with 100 requests to scroll. This causes a sequence of 100 paint requests to the JTable, and on each request the table will query the model for the data of each newly visible cell. We perform different runs, using 2, 4, 8, and 16 column-tables. Our scroll pane allows the table to show 35 rows and all the 16 columns. With the 16-column table, one repaint thus needs to render 16*35 cells, which means it invokes TableModel.getValueAt() 560 times per paint request.

3.8.2.1 Naive approach

A common approach in software development is to keep the code as simple and clean as possible, and to only introduce more complexity if there is a demonstrated need. Following this philosophy, we first write the most straightforward implementation of TableModel we can think of. This naive implementation, shown in **Figure 3.17**, immediately forwards all getValueAt() calls to the remote server. That is, for each table cell value a JTable requests from the TableModel, there will be a separate remote request, which will gather just the value of that cell. Before spending time (and possibly introducing errors) with an optimized

```
interface QueryResultServer {
 public int getCell(int row, int col);
}

class RemoteTableModel implements TableModel {
 private QueryResultServer server;
 //...
 public Object getValueAt(int row, int col) {
   return server.getCell(row, col);
}
```



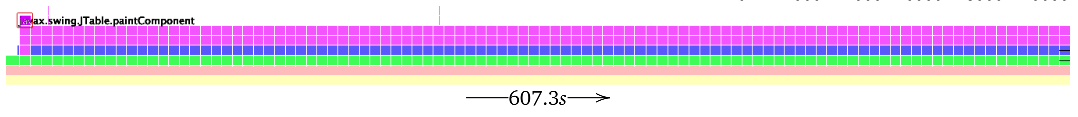


Figure 3.17. Naive TableModel

TableModel, we measure the performance of the naive implementation to see whether there actually is a need for performance optimization.

The cumulative latency distribution on the right of Figure 3.17 shows the perceptible performance of this approach. It contains one curve for each table width (2, 4, 8, and 16 columns). The average latency for painting the 16-column table is 6034 ms, and even the narrow 2-column table still requires 855 ms per repaint on average. The fact that the 2-column table is significantly more responsive points towards a possible optimization.

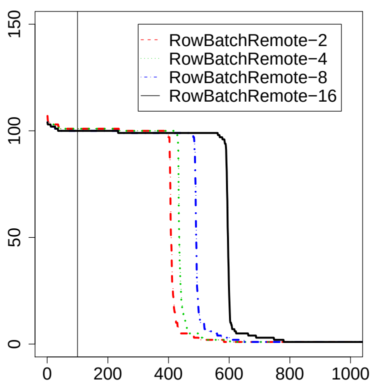
3.8.2.2 Row caching

Given that JTables render tables row-by-row, each row the table needs to paint will cause a consecutive sequence of invocations to getValueAt() with the same row number (one invocation for each cell in the row). We can improve the performance, in particular of tables with many columns, by always fetching a row from the server, and by caching the last accessed row. **Figure 3.18** outlines this implementation.

The cumulative latency distribution in Figure 3.18 shows a roughly 10-fold improvement in perceptible performance for the 16-column table. The optimization reduced the average latency for painting from 6034 ms to 593 ms. The performance gain for the 2-column table, a reduction from 855 ms to 406 ms, is less pronounced, as expected, because we only observe one hit for each cached row we fetch.

```
interface QueryResultServer {
 public int[] getRow(int row);
}

class RowBatchRemoteTableModel implements TableModel {
 private QueryResultServer server;
 private int[] cachedRowFields;
 private int cachedRow;
 //...
 public Object getValueAt(int row, int col) {
  if (row!=cachedRow) {
   cachedRowFields = server.getRow(row);
   cachedRow = row;
  }
  return cachedRowFields[col];
 }
}
```



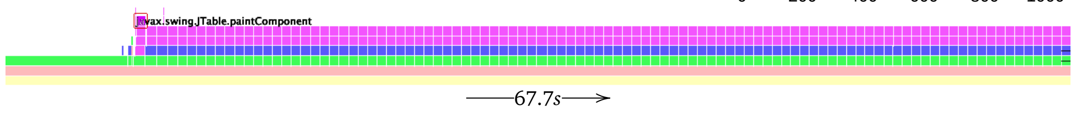


Figure 3.18. TableModel with Row Caching

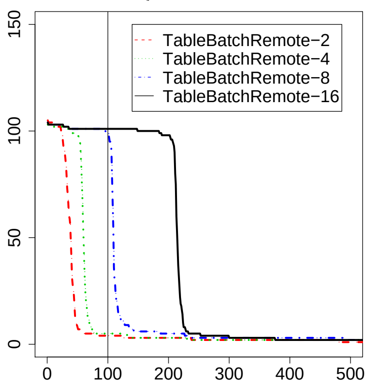
3.8.2.3 Table caching

To further reduce paint latency, we can drive the above idea to the extreme, and retrieve the entire table instead of a single row from the server. This means that the first getValueAt() call will be slow, requesting a potentially large amount of data from the server, and subsequent getValueAt() calls will be fast, directly returning the value from the locally cached table. **Figure 3.19** shows this approach.

The cumulative latency distribution in Figure 3.19 shows a significant improvement in perceptible performance. The average latency for painting the 16-column table is 227 ms. However, the first time the table is painted, its entire contents has to be fetched from the server, which causes a latency of 1872 ms for that first paint. Subsequent paints take 210 ms on average. While the first paint request does show up in the cumulative latency distribution, its visual impact is relatively minor, as it only shifts the curve up by on unit (1 episode) along the y-axis. However, the latency trace at the bottom of Figure 3.19 clearly shows the problematic paint call (highlighted). This first paint operation is definitely perceptible and significantly affects the user.

```
interface QueryResultServer {
 public int[][] getTable();
}

class TableBatchRemoteTableModel implements TableModel{
 private QueryResultServer server;
 private int[][] cachedTable;
 //...
 public Object getValueAt(int row, int col) {
  if (cachedTable==null) {
   cachedTable = server.getTable();
  }
  return cachedTable[row][col];
 }
}
```



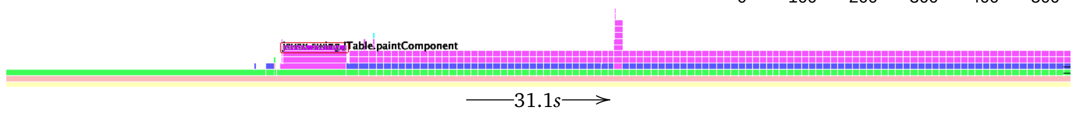


Figure 3.19. TableModel with Table Caching

3.8.2.4 Row set caching

The previous optimization lead to a single very long latency episode. If, instead of caching the entire table, we cached only the set of rows that are currently visible, we could improve rendering speed without inducing such a long wait time. **Figure 3.20** shows the corresponding implementation.

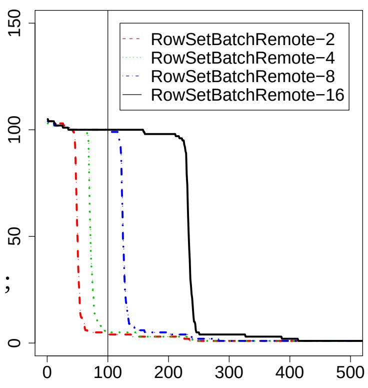
The average latency for painting the 16-column table is 229 ms. While this is higher than the 210 ms for the second and subsequent paints we observerd for the prior approach, we now have eliminated the initial very long lantency paint that copied the entire table, as we can see in the latency trace at the bottom of Figure 3.20.

3.8.2.5 Discussion

It is interesting to note that the end-to-end exeuction time (shown at the bottom of the latency traces), and even more importantly, the total time spent in episodes (the sum of all episode latencies) did not significantly change between the last two approaches. The table caching approach spent 31147 ms in episodes, while the row set caching approach spent 31181 ms. Nevertheless, our profiler clearly shows a difference in perceptible performance between the

```
interface QueryResultServer {
 public int[][] getRows(int startRow, int endRow);
}

class RowSetBatchRemoteTableModel implements TableModel{
 private QueryResultServer server;
 private int rowSetSize;
 private int[][] cachedRowFields;
 private int cachedSRow; // startRow
 private int cachedERow; // endRow
 //...
 public Object getValueAt(int row, int col) {
  if (row<cachedSRow || row>=cachedERow) {
   cachedSRow = row;
   cachedERow = row+rowSetSize;
   cachedRowFields =
     server.getRows(cachedSRow, cachedRow);
  }
  return cachedRowFields[row−cachedSRow][col];
 }
}
```



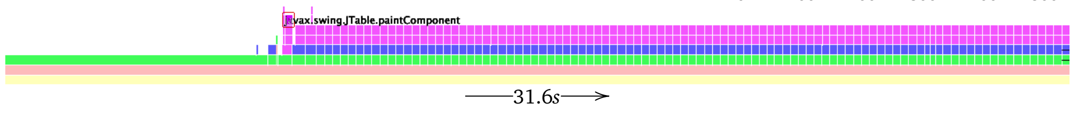
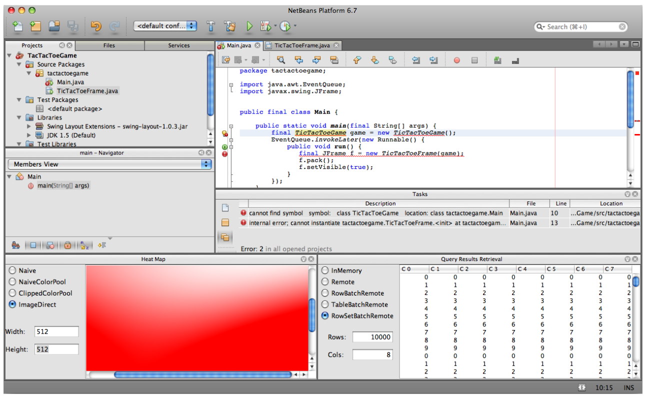


Figure 3.20. TableModel with Row Set Caching

two approaches. This difference shows up in the tabular output of the profiler, which we did not show in this paper, and it stands out in the latency traces we showed at the bottom of the figures. Looking at average times spent in methods would not reveal this difference in perceptible performance, and thus we need a profiling approach such as ours to identify and eliminate such performance issues.

3.8.3 Finding Perceptible Performance Problems in Large Applications

To see how our profiling approach can identify perceptible latency in the context of a real-world application, we have picked NetBeans, with over 45000 classes the biggest Java application we are aware of, and ported our two prior case studies into a NetBeans module. The right side of **Figure 3.21** shows that our module adds two new views at the bottom of the NetBeans window, containing



```
package tactactoegame;

import java.awt.EventQueue;
import javax.swing.JFrame;


public final class Main {

    public static void main(final String[] args) {
        final TicTacToeGame game = new TicTacToeGame();
        EventQueue.invokeLater(new Runnable() {
            public void run() {
                final JFrame f = new TicTacToeFrame(game);
                f.pack();
                f.setVisible(true);
            }
        });
```

Description	File	Line	Location
cannot find symbol symbol: class TicTacToeGame location: class tactactoegame.Main	Main.java	10	...Game/src/tactactoega
internal error; cannot instantiate tactactoegame.TicTacToeFrame.<init> at tactactoegam...	Main.java	13	...Game/src/tactactoega

Error: 2 in all opened projects

Heat Map: Naive [ ] NaiveColorPool [ ] ClippedColorPool [ ] [✓ ImageDirect]; Width: 512; Height: 512

Query Results Retrieval: InMemory [ ] Remote [ ] RowBatchRemote [ ] TableBatchRemote [ ] [✓ RowSetBatchRemote]; Rows: 10000; Cols: 8

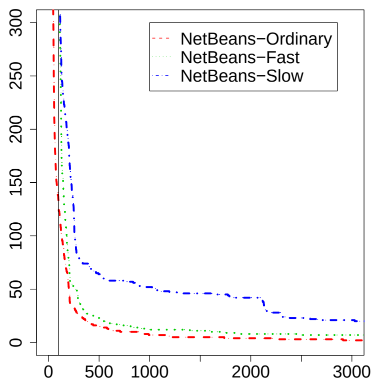


Figure 3.21. Performance of NetBeans Including Heat Map and Query Results

the two GUIs of our prior case studies. We then ran the complete NetBeans Java SE development environment including our module under our LiLa profiler.

We performed three separate runs of NetBeans. In each run we performed the same 17 minute long sequence of activities: We created a new project, opened the project's Main class, and programmed some array and list sorting and printing routines. During this, we ran the program we developed, configured the preferences, refactored class and method names, and searched for code.

The first run ("Ordinary") consist of exactly this normal interaction with NetBeans and does not use our module at all. The second run ("Fast") consists of the same interaction, interleaved with activities in the heat map and query results views. In that run we only used the fastest implementations of our GUI ("ImageDirect" for the heat map, and "RowSetBatchRemote" for the query results). The third run ("Slow") consists of the same interaction with the NetBeans proper, and with interactions with our module using the various slow implementations.

The resulting cumulative latency distributions are shown on the right of Figure 3.21. They clearly show that the ordinary NetBeans session has the best perceptible performance. The use of the fast version of our module slightly reduces the performance (dotted curve), while the slow version drastically impacts perceptible performance (dash-dot curve), showing 21 episodes longer than 3 seconds.

3.9 Summary

In this chapter we introduced the latency profiling approach as a way of dealing with the perceptible performance of interactive applications. Also, we introduced a way to characterize performance from the collected runtime data with the cumulative latency distribution. The latency profile with carefully selected landmarks highlights the code responsible for sluggish application behavior. We discuss the applicability on both Swing and SWT platforms and validate it on a new suite of microbenchmarks. 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 profile viewer as well as the professional-grade NetBeans IDE. Finally, we demonstrate our new profiling approach in two performance tuning case studies, and we show that we can identify performance problems even if they lie deep inside a complex real-world application.

Chapter 4

GUI Application Characterization

Aspects such as garbage collection activity, concurrency, or thread contention might influence the performance of interactive applications and cause perceptible lag during run-time. In this chapter we study the prevalence of these phenomena in interactive applications to ensure how crucial are the factors above. We chose a suite of real-world Java applications and ran them under an extended version of our profiler to gather latency profiles. We then perform the offline analyses needed to characterize the applications based on the recorded latency profiles.

4.1 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].

4.2 Applications

We selected the 14 Swing applications shown in **Table 4.1**. 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).

[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).

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	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 4.1. Applications

4.3 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 1/2 hours of interactive sessions (roughly 250'000 episodes) took 15 min, inclusive the generation of MATLAB graphs.

4.3.1 Overview

Table 4.2 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 two blocks: the name of the benchmark, a quantification of session duration ("Exp. time") and a characterization of episode duration ("Episodes").

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 ap-

	Exp. time		Episodes			
Benchmarks	E2E [s]	In-Eps [%]	< 3ms	≥ 3ms	≥ 100ms	Long/min
Arabeske	461	25	323605	6278	177	95
ArgoUML	630	35	196247	9066	265	75
CrosswordSage	367	8	109547	1173	36	80
Euclide	614	35	109572	9676	96	26
FindBugs	599	21	39254	6336	120	56
FreeMind	524	11	325135	3462	26	30
GanntProject	523	47	126940	2564	706	168
JEdit	502	9	117615	2271	24	33
JFreeChart	250	26	77720	1658	175	164
JHotDraw	421	41	246836	5980	338	114
JMol	449	46	110929	3197	604	180
Laoe	460	47	1241198	3174	61	18
NetBeans	398	27	305177	3120	149	82
SwingSet	384	20	219569	4310	70	57
Mean	470	28	253525	4447	203	84

Table 4.2. Overall statistics

plication, 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 “≥ 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 “≥ 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 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.

4.3.2 Trigger: Input, Output, or Asynchronous Events

Lag can occur in handling input, e.g. by processing mouse and keyboard 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

[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.

[3] We found that the Swing GUI toolkit’s repaint manager sometimes causes “asynchronous”

"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 4.1 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 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

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.

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.

4.3.3 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.

Figure 4.2 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 4.3.2, Arabeske explicitly triggers major garbage collections. As we can see in Figure 4.2, 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 4.2 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 4.1, 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 4.3.4 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.

4.3.4 Cause: Concurrent Activity, Synchronization, Sleep, 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, (3) the GUI thread might waiting in synchronization, (3) the GUI thread might voluntarily sleep, or (4) the GUI thread might have a lot of work to do.

Concurrent Activity. Figure 4.3 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.

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 (left 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.

Synchronization, Sleep, and Work. Figure 4.4 shows synchronization and sleeping time during episodes. The figure partitions the time the GUI thread spent in episodes into four components represented in a stacked bar: time blocked in monitor contention (first part of bar), time waiting in synchronization (second part of bar), time sleeping (last part of bar), and time working (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 spends the majority of its time working. 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 blocked, wait or sleep state. This demonstrates that aggregate information across the entire program execution is not necessarily helpful in pinpointing the causes of perceptible lag in interactive applications.

We measure the percentage of time in blocked, resp., waiting, resp., sleeping state by counting the fraction of call stack samples where the GUI thread was blocked in monitor contention, resp., waiting in Object.wait or LockSupport.park, resp., sleeping.

Synchronization cost does not seem to significantly affect perceptible performance (usually less than 10% of the time of perceptible episodes is wasted in synchronization). The most notable exception is jEdit, where over 25% of perceptible lag is due to synchronization. Looking at the corresponding stack traces in LagAlyzer, we can see that the synchronization is related to event processing inside jEdit's modal dialogs.

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.

4.4 Summary

In this chapter we characterized a suite of real-world interactive applications, monitoring their garbage collection activity, concurrency, or thread contention impact. We find that 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.

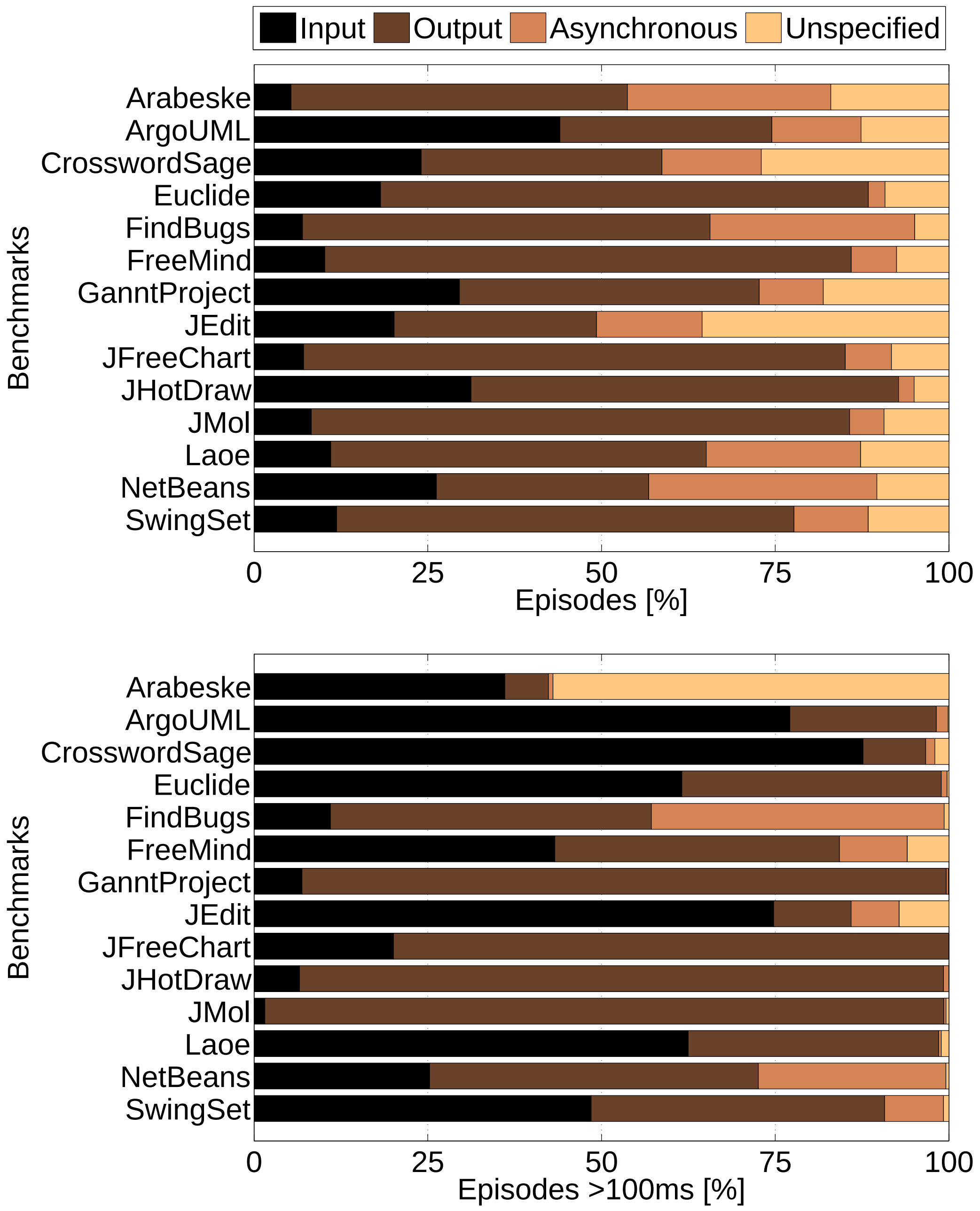


Figure 4.1. Triggers of (perceptible) episodes

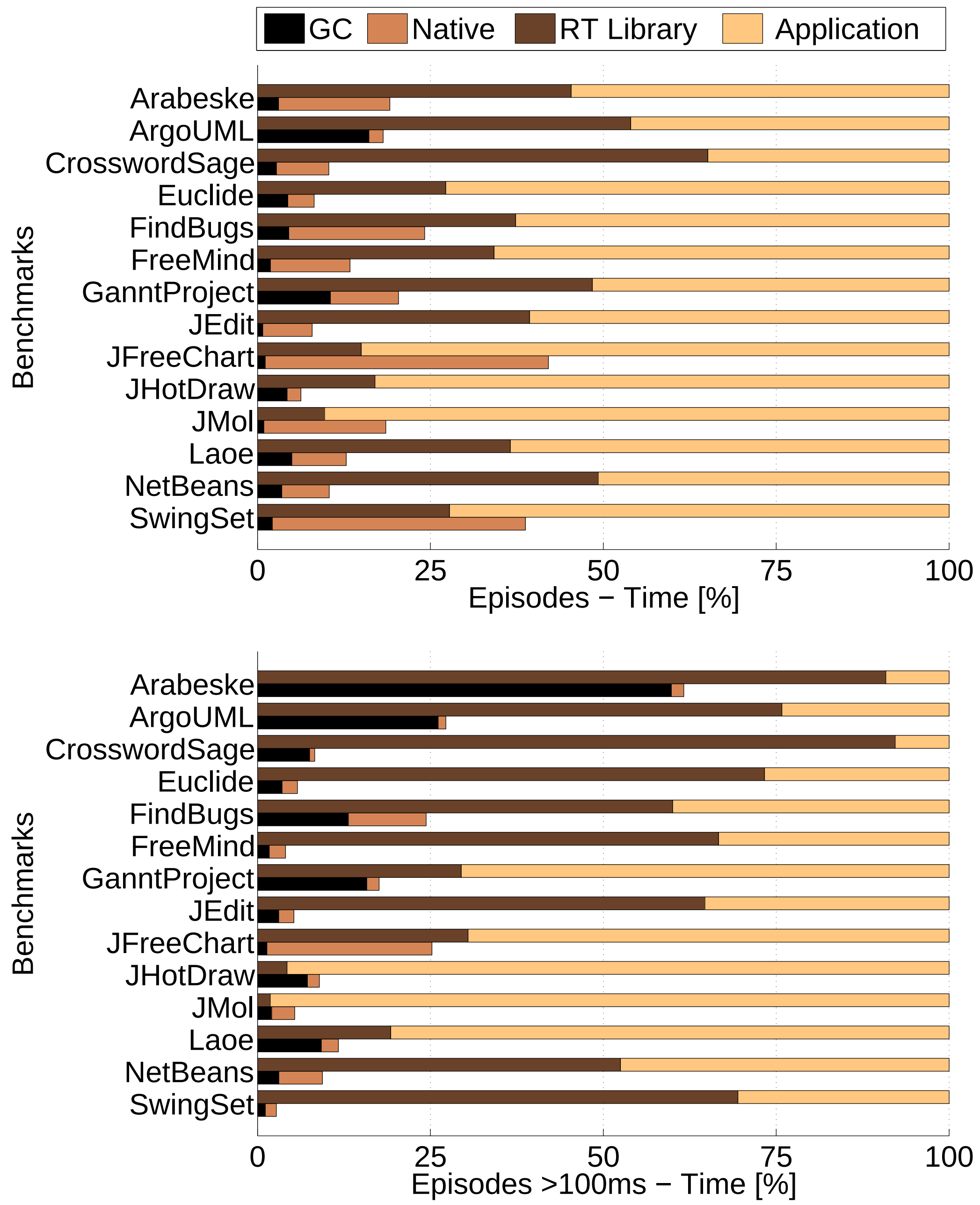


Figure 4.2. Location where time was spent during (perceptible) episodes

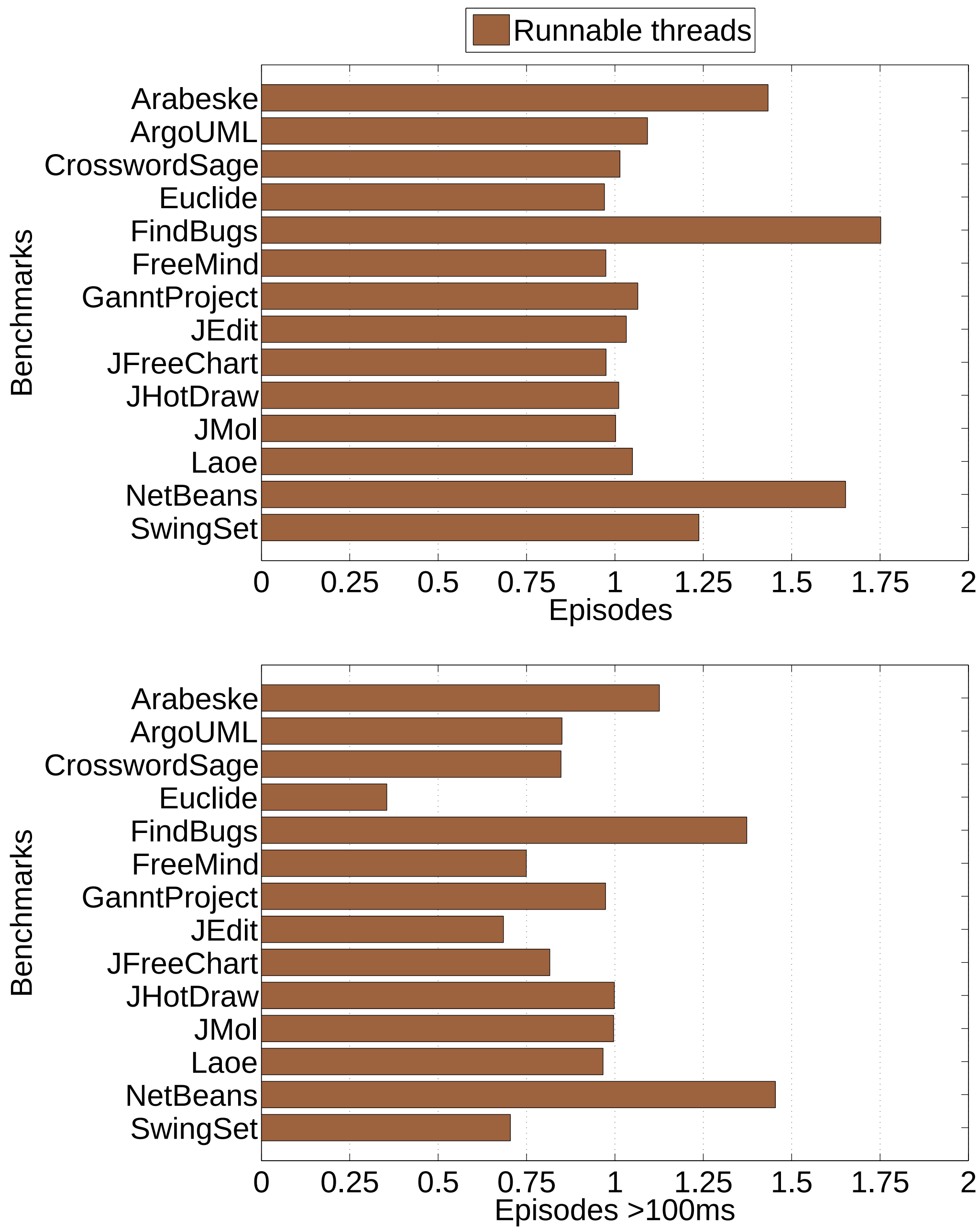


Figure 4.3. Concurrency in episodes (average # of runnable threads)

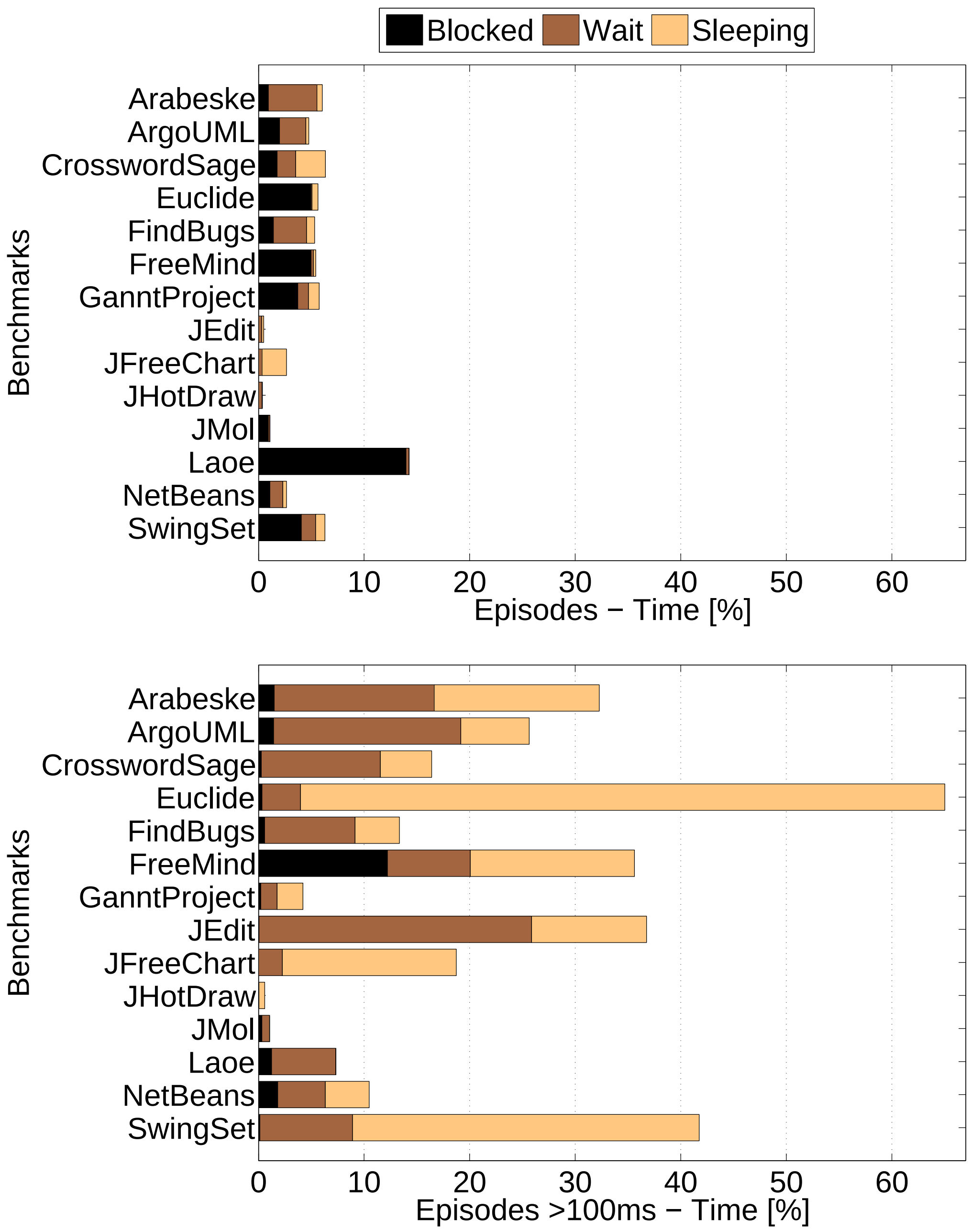


Figure 4.4. Synchronization and sleep during episodes

Chapter 5

Lag Hunting

The main goal of our approach is to identify latency bugs in interactive applications and pinpoints their causes. The implementation of the approach shown in Chapter 3 has three key drawbacks:

- **Requires manual instrumentation of the SWT or AWT event dispatch code.**

  It means that a profiler based on that approach can only be used by developers with a detailed understanding of the runtime environment, as they have to manually patch the runtime libraries of their virtual machine. While this patch is small, it can be different for different versions of the libraries.

- **Requires static instrumentation of all program and library classes.**

  This means that in order to profile an application, a developer has to locate, instrument, and re-deploy all its class files. This is not an easy feat for applications such as Eclipse, with dozens of plug-ins packaged in different ways, and it is impossible for applications with dynamically generated code. Moreover, the instrumentation of the entire code base of an application like Eclipse can take up to an hour, and the instrumentation has to be repeated every time a developer changes some code (e.g. to fix a performance issue just discovered).

- **Generates unreasonably large traces.**

  It leads to traces that are difficult to store for longer times and difficult to ship from a tester to a developer, as they can grow to hundreds of MB in size (at rates of up to 1.4MB/s, a one hour session will produce a 5GB trace). Moreover, the massive amount of data generated (and written to disk) at runtime may perturb the actual measurement results.

Not only these three drawbacks prevent effective identification of the performance problems. Many performance problems occur only in the field where an application may run on different platforms, and under different contexts which we could not anticipate in the lab. Thus, real performance testing of interactive applications should involve multiple profiles from multiple end-users.

5.1 Post Deployment

As we discussed in Chapter 2 developers employ a host of powerful approaches and tools to ensure and improve the quality of their software. They perform static analyses to identify problems even before applications are run. Then they perform testing sessions to find errors at runtime before shipping applications. They employ approaches to find errors in applications already running in the field. The former two approaches, static checking and testing, are invaluable for improving software quality. However, the latter approach, *post-deployment problem detection*, is increasingly receiving attention in research and practice. For example, Liblit et al. [80] introduce a bug finding approach that reduces the cost of evaluating runtime assertions using sampling, Hauswirth and Chilimbi [61] present an approach for low-overhead memory leak detection, or Glerum et al. [56] describe Windows Error Reporting (WER), a system for collecting and classifying crash reports.

Benefits of post-deployment problem detection. The success of WER, which has been in operation for over ten years and is running on over one billion client computers, demonstrates the two main *benefits* of post-deployment problem detection: (1) gathering information about problems in the context in which they occur, and (2) combining the collective information from a large user base. The first benefit is important because some problems only manifest themselves in a specific context, and that context may differ from the artificial environment used during testing. By different contexts we mean different hardware platforms, operating systems, execution environments, different libraries or a different set of enabled plug-ins, or different storage and network configurations. The second benefit is important because the large number of users running an application enables the use of sampling. Potentially costly dynamic analyses can be partitioned into small subsets (samples), and each application session may run only a small, low-overhead subset of the analysis. The aggregation of the partial results from a large number of sessions can then provide insights that would be difficult to gain from a limited number of testing runs.

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

Widely deployed software. Post-deployment problem detection is particularly relevant for widely deployed software, because the larger the number of deployments, the larger the number of different contexts. We group software into three classes, with increasing number of deployments: (1) hosted applications (e.g. cloud-based web mail), (2) installable services (e.g. web or database servers), (3) installable client applications (e.g. web browsers or IDEs). While all three classes may (directly or indirectly) be used by large numbers of users, the last class has the largest number of deployments and their developers have the least control or knowledge over the context in which the applications run. Moreover, that class is rapidly expanding from the traditional applications running on powerful desktop computers to apps running on resource-constrained notebooks, tablets, and smart phones, where performance grows in importance.

Perceptible performance. Client applications directly interact with users. Their performance is thus defined by the users' perception. As a significant body of research in human-computer interaction has shown, the key determinant of *perceptible* performance is the system's lag in handling user events [104; 103; 37; 41]. Thus, to detect perceptible performance bugs, one must detect perceptible lag.

5.2 Advanced profiling

As discussed in Section 5.1 our approach targets *widely-deployed* applications by gathering runtime information in the field. It targets *interactive* applications by gathering information about perceptible lag.

We now present the improvements over the original profiling approach that were necessary to realize our idea of lag hunting in the field.

- *Data Size Reduction*

  Traces produced in an hour-long profiling session can reach several GB in size. This is impractical. The simple finding that the latency is roughly exponentially distributed, that is, that most intervals (e.g. most observer calls) are short, indicates an effective optimization: We are only interested in perceptible intervals, and they have a long latency; thus by filtering out short intervals, we can drastically reduce trace size while keeping all relevant information. Moreover, on-the-fly data compression would additionally decrease trace size by an order of magnitude.

- *Instrumentation Caching*

  Interactive applications consist of hundreds or thousands of classes, and the Java class libraries, which themselves make use of observers, add several thousand classes more. Starting an interactive application means loading many of its classes. With a dynamic approach to latency profiling, class loading entails instrumentation, which can lead to a significant overhead. In most scenarios, the vast majority of classes of an interactive application do not change between runs. If we thus could cache the instrumented version of these classes, analog to existing approaches to persistent code caching [97; 39], we would be able to reduce the instrumentation effort to a minimum.

- *Incremental Analysis*

  A crucial parameter in our approach is the choice of landmark methods. Instrumentation of their calls during run-time is challenging in terms of overhead. Performing the same analysis as with the static approach to distinguish method calls might significantly increase latencies. Given that many application classes inherit from other classes. Thus, when a class is analyzed the results could be cached. If another class inherits from the one already analyzed then redundant analysis could be avoided by reuse of the cached result. Thus, we could perform an incremental analysis to avoid unnecessary repetitions.

- *Profiling Extension*

  Previously, we focused on certain landmarks as responsible for perceptible performance of interactive applications. However, the causes of perceptiveness might not be only there. Thread synchronization is another potential cause of slow performance because many modern interactive applications are multithreaded. A thread may spend time in the blocked state due to monitor contention, it could wait in synchronization, or sleep. Further, modern virtual machines benefit from garbage collection which also may cause perceptible lag during runtime. As mentioned in the approach section, stack sampling of all the threads would allow construction of the calling context tree. The cost of making such a tree is high. However, collaborative profiling, which is discussed below, would allow the use of sampling.

- *Collaborative Profiling*

  Having a lightweight profiler that dynamically instruments the application code enables one of the main goals: to have it deployed in the field. The collection of profiling data is particularly relevant for widely deployed software, because the larger the number of deployments, the larger the number of different contexts. Gathering information about problems in the context in which they occur is important because some problems only manifest themselves in a specific context, and that context may differ from the artificial environment used during testing. By different contexts we mean different hardware platforms, operating systems, execution environments, different libraries or a different set of enabled plug-ins, or different storage and network configurations. Gathering information from a large user base allows aggregation of the partial results from a large number of sessions to provide insights that would be difficult to gain from a limited number of testing runs. Moreover, combining the collective information from a large user base is important because the large number of users running an application enables the use of sampling. Potentially costly dynamic analysis can be partitioned into small subsets (samples) and each application session may run only a small, low-overhead subset of the analysis.

Figure 5.1 shows an overview of the redesigned landmark latency profiling approach. Applications are deployed with a small Java agent that collects performance information about each session. At the end of a session, the agent sends a session report to an analysis engine running on a central server. The server collects, combines, and analyzes the session reports to determine a list of performance issues. To the developers, the analysis engine appears like a traditional bug repository. The only difference is that the "bug reports" are generated automatically, without any initiative from the users, that all equivalent reports have been combined into a single issue, that related issues are automatically linked together, and that issues contain rich information that points towards the cause of the bug.

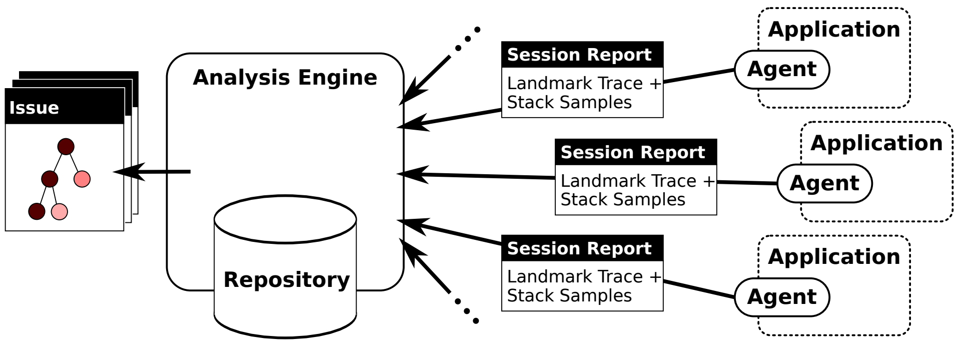


Figure 5.1. Lag Hunting

5.3 Landmark methods

As we discussed in Chapter 3, a chosen set of landmarks should fulfill three properties: First, they need to *cover most of the execution* of the program. Given that we only measure the latency of landmarks, we will be unaware of any long activity happening *outside* a landmark. Second, they need to be *called during the handling of an individual user event*. We want to exploit the human perceptibility threshold (of roughly 100 ms) for determining whether a landmark method took too long, and thus such a method needs to correspond to a major activity that is part of handling a user event. A method executed in a background thread may take much longer than 100 ms, but it will not affect the responsiveness of the application, 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 3.1), may have a very long "latency", but it is not responsible for delaying the handling of individual user events. Third, landmarks should be *called infrequently*. Tracing landmarks that are called large numbers of times for each user event would significantly increase the overhead.

5.3.1 Selecting good landmarks

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 [53] 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 plug-ins. However, the publicly callable API of plug-ins in frameworks like Eclipse is so fine grained, that the overhead for this kind of landmarks might be too large.

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

5.4 Profiling Optimizations

Our original profiling approach was impractical for use in the field. The new approach makes in-the-field-profiling practical by using dynamic instrumentation and by employing several optimizations.

5.4.1 Persistent Instrumentation Caching

Interactive applications consist of hundreds or thousands of classes, and the Java class libraries, which themselves make use of listeners, add several thousand classes more. Starting an interactive application means loading many of its classes. With our dynamic approach to latency profiling, class loading entails instrumentation, which can lead to a significant overhead.

In most scenarios, the vast majority of classes of an interactive application do not change between runs. If we thus could cache the instrumented version of these classes, analog to existing approaches to persistent code caching [97; 39], we would be able to reduce the instrumentation effort to a minimum.

We maintain a persistent code cache in a straightforward Java data structure, which we deserialize at the start of the application, and serialize when the application terminates. The cache basically contains a map from class names to class bytes (the instrumented version of the class file).

Given that a class, at runtime, is not uniquely identified by its name, but by its name and its class loader instance, we cannot be sure that the mapping from class name to instrumented code is correct in all cases. For this purpose we also store the uninstrumented class bytes in the cache. This allows us to check whether the loaded class, that should now be instrumented, truly corresponds to the class stored in the cache.

Listing 5.1 shows our caching approach. When we are asked for transforming (instrumenting) a class, we lookup information in the class cache by class name. If the class cache contains information about our class (line 4), we retrieve the instrumented bytes. If we use the safe approach (versionCheckEnabled), we also retrieve the uninstrumented bytes from the cache. We compare the uninstrumented cached bytes with the loaded bytes we are supposed to instrument (line 10). If they are identical, we know that we can safely use the instrumented bytes from the cache. If they differ, we reinstrument the loaded bytes, and we update the cache.

We also support an unsafe but potentially faster approach, where we omit the version check. We have noticed that in most cases, the version check is not necessary. It is rare that two classes with the same name are loaded by different class loaders, and that the two classes actually differ. However, we have noticed cases where classes were dynamically generated on the fly, and where they used artificially generated names. In these cases, the same names were often reused for different classes in different runs. Moreover, one application of our profiling approach is the profiling of an application during development. In that situation, the code of some classes will change between runs, and it thus is important to not use stale code from the cache. Our version check detects these changes and reinstruments classes that have changed.

There is one limitation to our persistent instrumentation caching approach. We do not reinstrument a class A, if a different class, say B, called by A, has changed. This means that there could be cases where we trace presumed listener calls from class A to class B, but where, due to changes in class B, the callee method no longer is a listener. We could update our caching mechanism to detect this situation by merging information from the decision cache into the instrumentation cache. However, this case should rarely occur in practice (a developer turning a listener method into a normal method, or vice versa), the consequence of not detecting it are benign (the application still runs correctly), and the remedy is straightforward (delete the cache after such a change), and thus we omitted this check from our implementation. Note that this omission does not lead to incorrect executions, because if class B changed in a way that broke class A, then class A would also have been changed by the developer (and we thus would have reinstrumented the new version).

Listing 5.1. Persistent Instrumentation Cache

```
byte[] transform(String className,
                 byte[] loadedBytes) {
  if (contains(className)) {
    // class (hopefully the correct one) in cache
    byte[] instrumentedBytes =
        lookupInstrumented(className);
    if (versionCheckEnabled) {
      uninstrumentedBytes =
          lookupUninstrumented(className);
      if (equals(uninstrumentedBytes,
                 loadedBytes)) {
        return instrumentedBytes;
      } else {
        instrumentedBytes =
            instrument(loadedBytes);
        put(className, loadedBytes,
                       instrumentedBytes);
        return instrumentedBytes;
      }
    } else {
      return instrumentedBytes;
    }
  } else {
    // class not in cache
    instrumentedBytes = instrument(loadedBytes);
    put(className, loadedBytes,
                   instrumentedBytes);
    return instrumentedBytes;
  }
}
```

5.4.2 Incremental Observer Method Identification

Our profiler has to instrument two types of code: (I) the event dispatch method in the GUI toolkit, and (II) any call site corresponding to a listener notification. With our dynamic instrumentation approach, we need to decide where to place instrumentations in a class A at the time class A is being loaded.

The decision for case (I) is simple, we always instrument entry and exit of the event dispatch method, which is a well known method in a well known class: dispatchEvent() in java.awt.EventQueue for AWT and readAndDispatch() in org.eclipse.swt.widgets.Display for SWT.

The decision for case (II) is more complex. Class A may contain many methods, and each method may contain many call sites. Only some of these call sites need to be instrumented. A listener notification corresponds to a call site in class A, which calls a method B.m, where B.m is a *observer method*. The decision for whether B.m is a listener method is based on the fact that listeners in AWT as well as SWT implement the java.util.EventListener interface. We only consider B.m a listener method if (1) the static type of the call target, B, is a subtype of EventListener, and (2) the target method, m, is declared in an *interface* that extends EventListener. The second clause prevents the instrumentation of calls to non-listener methods of classes implementing EventListener.

This decision whether B.m is a listener method can be expensive (it requires the traversal up the inheritance hierarchy involving multiple interface inheritance), and may need to be taken many times during the instrumentation of a program (e.g. there may be dozens or hundreds of call sites to a given method, located in many different classes). Thus, instead of redeciding every time we encounter a call site to B.m, we cache the decision the first time and look it up in a decision cache in subsequent cases.

As stated before, we need to take the decision about B.m at the time class A is being loaded. At that time, class B (and its supertypes) may not have been loaded yet. In this situation, there are two options to determine whether B.m is a listener method.

First, we could load class B (calling loadClass() on class A's class loader), and use reflection to query B (and B's supertypes') methods. Fortunately we know class A's class loader at the time we need to transform A, and it is exactly the class loader which would eventually be used by the virtual machine to load B.

So, finding B is not a problem. However, the need to fully load B at this early time can lead to problems: B might depend on A, or even be equal to A, and thus would recursively trigger the loading of A; and the use of reflection on B will cause B to be resolved (and not just loaded), possibly triggering the loading of other classes depending on A.

The second option, the option we chose to use, circumvents these problems. We can load the *bytes* of class B and B's supertypes by calling *getResource()* – instead of *loadClass()* – on class A's class loader. This allows us to get the bytes without causing the class to be loaded. We then use a class file parser library (in our case we use ASM, the same library we use for instrumentation) to parse the bytes, find the supertype names, and traverse the methods.

Now that we have described how we take the decision whether B.m is a listener method, we can describe how we cache that decision. We could just keep a cache mapping from "B.m" to "true" or "false". However, since a listener type may have many listener methods (e.g. java.awt.MouseAdapter has 8 such methods), we gather decisions about all methods of a class once we spend the effort to load a class' bytes. We thus cache, for each class, the list of its listener methods. We also cache the negative result: if we encounter a class without listener methods, we cache the empty list. This is important for performance, because the vast majority of call sites do *not* correspond to calls to listener classes, and determining that a method is *not* a listener method requires an exhaustive traversal of its supertypes. Moreover, given that we have to possibly parse several class files to take the decision for B.m, we cache the lists of listener methods of all supertypes of B along the way. Thus, if in the future some class C inherits from a supertype it has in common with B, it can stop the traversal up the inheritance hierarchy at that type and used the cached method list. This caching and reusing of intermediate results essentially "incrementalizes" the identification of listener methods.

5.4.3 Online Trace Filtering

Traces produced in an hour-long listener latency profiling session can reach several GB in size. This is impractical. The simple finding that the latency is roughly exponentially distributed, that is, that most intervals (e.g. most listener calls) are short, indicates an effective optimization: We are only interested in perceptible intervals, and they have a long latency; thus if we can filter out short intervals, we can drastically reduce trace size while keeping all relevant information.

Figure 5.2 shows the effect of filtering. The unfiltered trace on the left contains a large number of short intervals. After filtering, only intervals with a latency above a certain threshold remain (right). The information about the short intervals is aggregated in the remaining intervals (e.g. by keeping track of the number of their short children).

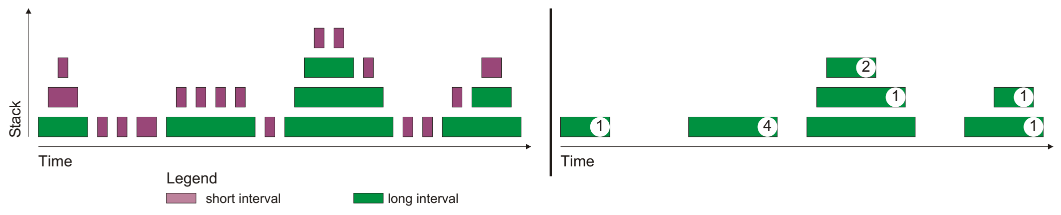


Figure 5.2. Filtering of Short Latency Intervals

How can we filter out long intervals online? Given that we write out a trace record at the begin and at the end of each interval, how can we know – at the begin of an interval – that this interval will be short and should not be traced? Obviously we have to somehow postpone the writing of a *begin* record until we know that the corresponding interval is longer than the latency threshold.

Listing 5.2 shows our approach to online trace filtering. Our profiler is called at the begin and at the end of each interval (e.g. before and after each call to a listener, or before and after each dispatch of an event). It is supposed to write a record to the trace for each begin and end. Instead of immediately writing the corresponding *begin* and *end* records, we maintain a stack of all currently open intervals (of all their *begin* records). This is similar to the call stack, which contains a stack frame for each currently active method, except that we do not have an element for each method invocation, but only for method invocations corresponding to listener calls and event dispatches. This shadow stack thus represents an abstraction of the call stack. Its purpose is to allows us to postpone writing *begin* records into the trace until we are sure that the corresponding interval is longer than a given threshold.

At the begin of an interval (method “beginInterval”), we just push a *begin* record on the stack. At the end of an interval (method “endInterval”), we pop its *begin* record from the stack, and we compute the interval’s latency. If the latency was short, the interval is to be filtered out. In that case (line 22) we do not write anything to the trace, but we aggregate the interval’s duration in a field of its parent (the directly enclosing interval). The parent will thus know the number and total duration of all its short child intervals. This information is relevant, because a parent could contain a large number of short children, and the profile should reflect that fact.

Listing 5.2. Online Trace Filtering

```
class Filter {
  Stack stack = new Stack();
  // pointer into stack
  int postponementBoundary = 0;
  // min latency of traced intervals
  long threshold;

  Filter(int threshold) {
    this.threshold = threshold;
    Event threadStart = new Event();
    stack.push(threadStart);
  }

  void beginInterval(Event start) {
    stack.push(start);
  }

  void endInterval(Event end) {
    Event start = stack.pop();
    long latency = end.timeStamp-start.timeStamp;
    if (latency < threshold) {
      // short interval,
      // filter out (keep aggregate info)
      Event parent = stack.top();
      parent.skippedIntervals++;
      parent.skippedLatency += latency;
    } else {
      // long interval,
      // all enclosing intervals are long, too
      if (postponementBoundary<stack.size()) {
        processPostponedBegins();
      } else {
        postponementBoundary = stack.size();
      }
      traceEnd(end);
    }
  }

  void processPostponedBegins() {
    while (postponementBoundary<stack.size()) {
      traceStart(stack.get(postponementBoundary));
      postponementBoundary++;
    }
  }
}
```

In case the latency was above the threshold (line 28), the *begin* and *end* records of the interval have to be written to the trace. Before we can write an interval's *begin* record, we have to ensure that the *begin* records of all its enclosing intervals have already been written. We check this in line 30. The "postponementBoundary" is an index into the shadow stack. All *begin* records on the stack below this index have already been written to the trace. The records above the index have not yet been written, because it was not yet clear whether they corresponded to long intervals. Given that the current interval is long, all enclosing intervals are long, too, and we can write their *begin* records to the trace and move the "postponementBoundary" to the current top of stack by calling "processPostponedBegins". If, on line 30, the *begin* records of all enclosing intervals have already been written, we just ensure, in line 33, that the "postponementBoundary" corresponds to the new top of stack (it might have pointed higher, to the prior top of stack, and thus might need to be moved down by one). After this postponement processing, we finally write the *end* record of the current interval to the trace (line 35).

The shadow stack always contains a sentinel at the bottom. This sentinel, represented by the "threadStart" event (line 10), corresponds to the lifetime interval of the thread. It is created when the thread is created, and it is consumed when the thread or the system shuts down. It accumulates the aggregate information about top-level short intervals (count and total latency) which otherwise would be lost.

5.5 Trace Analysis

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

5.6 Overhead and Perturbation

Our profiling approach incurs a cost: dynamic instrumentation implies that we have to spend some time, at runtime, instrumenting classes. Moreover, like with traditional landmark latency profiling, when instrumented code is executed, each instrumentation adds some cost. In our case, each individual instrumentation, placed before and after instrumented call sites, directly calls into our profiler classes, where we capture, filter, buffer, and flush trace information out to disk.

In this section we evaluate these runtime costs of our profiling approach. We first analyze the cost of *instrumenting* the program, and then we study the cost due to time spent in the *profiler*.

5.6.1 Methodology

We evaluated our profiling approach on the biggest interactive Java application we are aware of: Eclipse. In fact, Eclipse is not a single application, but it is a platform for rich client applications [84], based on a framework for which many applications and plug-ins have been built. **Figure 5.3** shows the number of classes constituting the Eclipse SDK distribution, which ranges from about 19000 classes for version 3.0 to about 33000 classes for the latest version at the time which was 3.4.1. In one of our experiments we use all the 5 different versions, all other experiments were conducted with version 3.4.1.

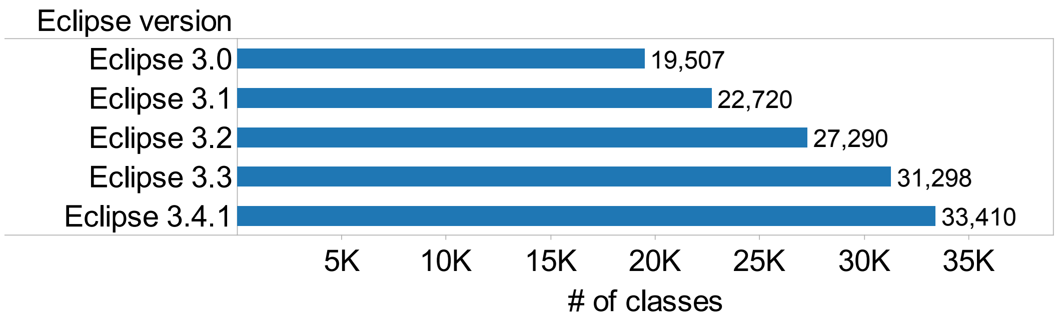


Figure 5.3. Number of Classes in Different Versions of the Eclipse SDK Distribution

We ran Eclipse on different processors (Pentium 3, Pentium 4, Pentium Dual Core, Core Duo, and Core 2 Duo) and different operating systems (Windows XP, Windows Vista, and Mac OS X). Depending on the platform we either used

Processor	GHz	RAM	OS	JVM
Pentium 3	0.8	256MB	Windows XP	Sun 1.6_07, mixed mode, sharing
Pentium 4	1.6	1GB	Windows XP	Sun 1.6_07, mixed mode, sharing
Pentium Dual Core	1.46	2GB	Windows Vista	Sun 1.6_07, mixed mode, sharing
Core Duo	2	2GB	Windows XP	Sun 1.6.0_07, mixed mode, sharing
Core 2 Duo	2.4	2GB	Windows Vista	Sun 1.6.0_07, mixed mode, sharing
Core 2 Duo	2.16	2GB	Windows Vista	Sun 1.6.0_07, mixed mode, sharing
Core 2 Duo	2.8	4GB	Mac OS X	Apple 1.5.0_16-133, mixed mode, sharing

Table 5.1. Platforms

Sun's Java virtual machine (1.6.0_07), or the one from Apple (1.5.0_16); both of them are based on the same source.

We recruited four undergraduate students who performed a prescribed set of tasks in Eclipse on their own computers under our supervision. These tasks were given to the students in writing and discussed before the start of the experiment. They consisted of two short programming problems where the students had to implement and test simple algorithms (bubble sort and minimum-cost spanning tree) in Java. They had to start with an empty workspace, create and configure a Java project, implement the algorithms in classes with a prescribed interface given the pseudo-code of the solution, write unit tests for their implementations, and run the tests and fix their code. The students repeated the same tasks for all of the experiments in this section.

5.6.2 Benefits of Incremental Observer Method Identification

The cost of dynamic instrumentation consists of two components. First, we need to decide whether or not to instrument a given call site. Second, if needed, we have to rewrite the given method to inject calls to the profiler before and after the call site. For deciding whether a given call site represents a observer notification, we have to analyze the callee to determine whether it corresponds to a observer method. This analysis requires the traversal of the inheritance hierarchy, causing the lookup and analysis of a possibly considerable number of classes. The prior static instrumentation approach performed this step whenever it encountered a call site. Our lightweight approach needs to be fast, and thus we have to reduce the time needed to take the decision. We do this by caching and reusing past decisions.

Figure 5.4 shows that decision caching drastically reduces the instrumentation latency. We ran the same 20 minute session with and without decision caching on 4 different platforms. The x-axis represents the average latency needed to instrument one class. Along the y-axis we show four different plat-

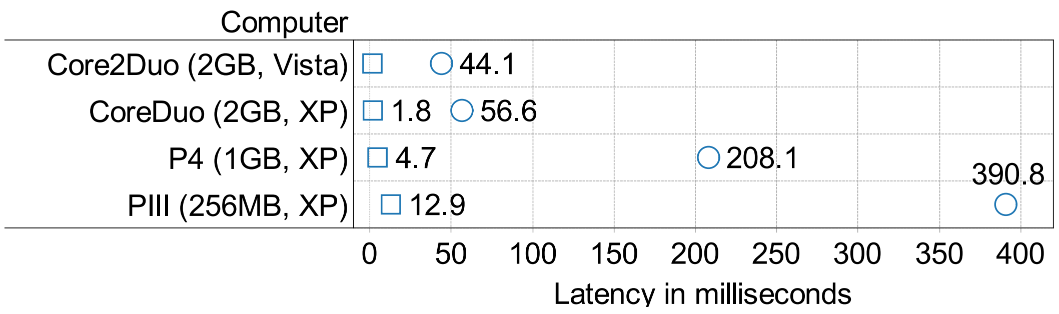


Figure 5.4. Effect of Decision Cache on Class Transformer Latencies

forms, and for each platform we show the latency with (square) and without (circle) decision caching. The figure shows that decision caching drastically speeds up instrumentation, by up to a factor of 44 (for the Pentium 4). Achieving this speedup is a major step in making dynamic latency profiling practical.

5.6.3 Benefits of Instrumentation Caching

While decision caching significantly reduced the cost of instrumentation, we still observe latencies of several milliseconds for instrumenting a single class. Given that classes are often loaded in bursts (e.g. a class with its superclasses), these latencies can add up and lead to perceptible delays. We thus developed a further optimization: we cache the instrumented code in a persistent instrumentation cache.

Figure 5.5 shows the benefits of the instrumentation cache. The y-axis corresponds to the latency due to instrumenting a single class. The figure consists of three box plots, summarizing the latencies of close to 74000 class transformations in 12 different interactive sessions. This data was collected with decision caching enabled. There are three choices for how a class may be transformed: (1) it is instrumented, (2) it is taken from the instrumentation cache, without version check, (3) it is taken from the instrumentation cache, with version check. Each of these choices is represented by a box plot.

The figure shows that the first approach, instrumenting the class, is the slowest. As we have seen previously, it can lead to latencies of a few milliseconds (mean, per class, with 95% confidence interval: 1.76±0.07ms). The second approach, caching without version check, is potentially unsound, but it is the fastest (0.09±0.02ms). The third box plot shows that the version check slows down transformation somewhat: the mean of 0.16±0.02ms is statistically signif-

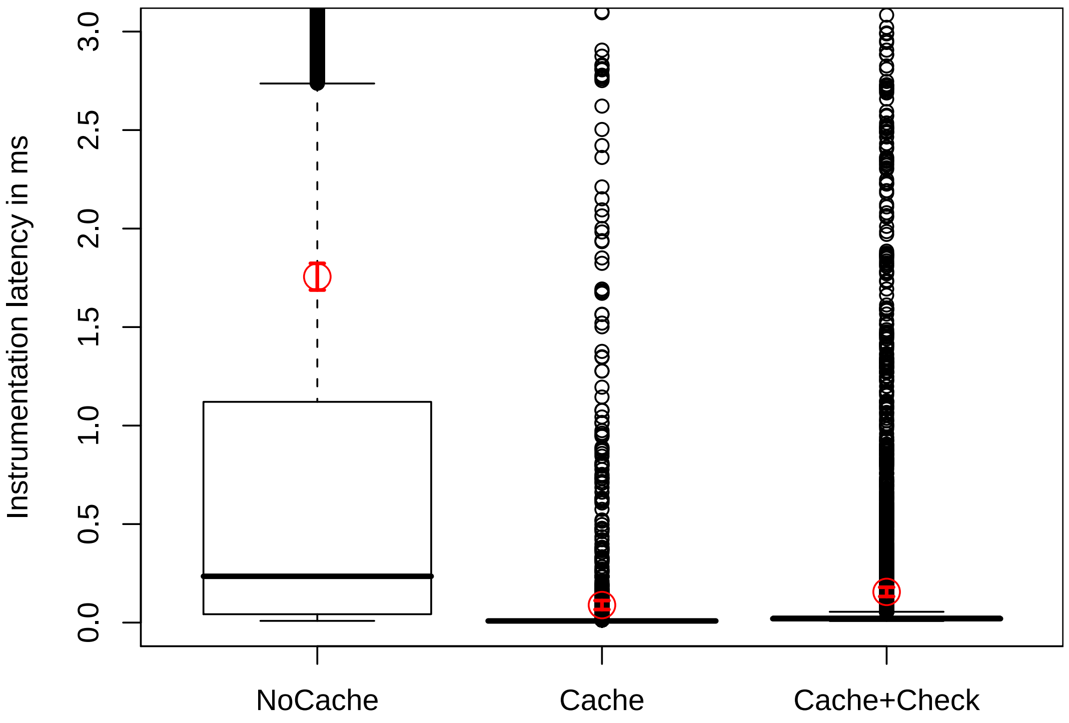


Figure 5.5. Effect of Instrumentation Cache on Class Transformer Latencies

icantly different from caching without version check, as the confidence intervals do not overlap. However, it still is an order of magnitude faster than the approach without caching. While we so far have evaluated the cost of instrumentation in isolation, **Figure 5.6** shows which percentage of the episode latency corresponds to instrumentation cost.

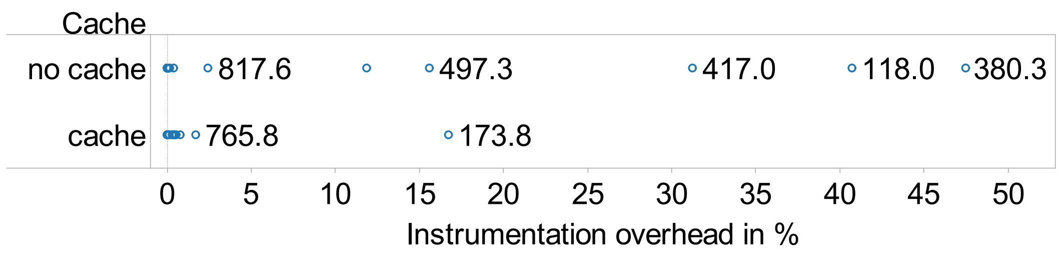


Figure 5.6. Percentage of Episode Latency Caused by Instrumentation Activity

The top of the figure represents episodes of 6 sessions where instrumentation caching was disabled, the bottom shows the episodes of 6 sessions with enabled instrumentation cache. Each circle in the figure represents an individual episode. Overall in these 12 sessions, only 180 episodes were longer than 100ms, and only those episodes are shown. The numbers next to some episodes show their duration in milliseconds. The x-axis shows the percentage of an episode's duration that is spent transforming classes.

The figure shows that only very few perceptible episodes (7 of 72) spend more than 2% in our instrumentation framework. Moreover, the figure shows that with instrumentation caching, that number is further reduced (1 of 108). Finally, in all our traces, only one episode (the one taking 118ms) became perceptible *due* to instrumentation (that is, it would have been shorter than 100ms, had we not spent 40% of its time instrumenting classes). Overall, the percentage of time, in perceptible episodes, spent instrumenting classes is 2.08% without instrumentation cache (average over top half of the figure), whereas with a cache, that value significantly drops to 0.2%.

Besides affecting the latency of episodes, instrumentation also affects application startup time. **Figure 5.7** shows the startup time for four different configurations. We define startup time as the time from the initialization of our instrumentation agent to the first dispatched event. Since the first event is dispatched very early, when the progress bar of the splash screen appears, our startup time only includes the first part of application startup. This phase does, however, include the deserialization of the persistent instrumentation cache, which is a major cost factor we want to evaluate.

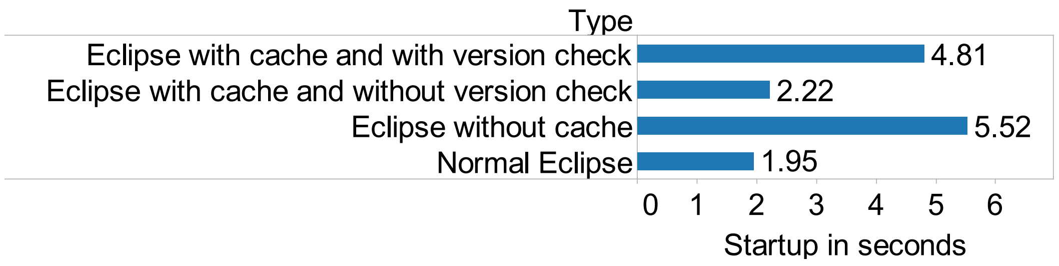


Figure 5.7. Effect of Instrumentation Cache and Version Check on Startup Time

The figure shows, for each configuration, the average startup time over 5 different runs. All runs were performed with Eclipse 3.4.1 on Windows XP on a Core Duo. The bottom bar shows the startup time of Eclipse without our profiler. The top bar shows that our recommended configuration, enabled class cache plus version check, considerably slows down startup (4.81s instead of 1.95s).

The second bar shows that this slowdown is mostly due to the version check: without version check the startup time is reduced to 2.22s. The third bar shows that at 5.52s, the startup time without cache is slower than the time with cache and version check. We conclude that caching is beneficial, but the version check, necessary for correctness, has a significant impact on startup time.

5.6.4 Benefits of Online Trace Filtering

We now focus on the cost of profiling. Before and after each instrumented call, the instrumentation invokes the profiler, and the profiler gathers the necessary information. For a single observer method call, for example, we thus perform two calls to the profiler, and each of these two calls might append information to the trace.

Given that most episodes take a short time (a few microseconds), and given that we are only interested in perceptible episodes, it makes no sense to trace information about short episodes. For this reason our online trace filtering approach allows the specification of a threshold. Episodes shorter than the threshold are not traced, and we only keep aggregate information about them.

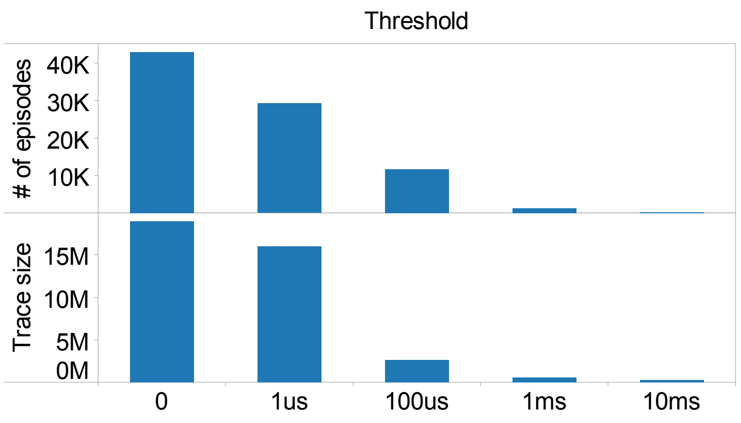


Figure 5.8. Trace Size vs. Latency Threshold

Figure 5.8 shows how the number of traced episodes (top) and the trace size (bottom) changes depending on the threshold (x-axis). Each bar represents the average over four traces, collected on four different platforms. The leftmost bar, with a threshold of 0s, shows the amount of data generated without filtering. The figure shows that setting the filter threshold to 10ms drastically reduces the amount of generated data, and thus the overhead. Given this reduction in the amount of data that needs to be written to disk by the profiler, we expect that online trace filtering also will reduce the perturbation due to the time spent in the profiler.

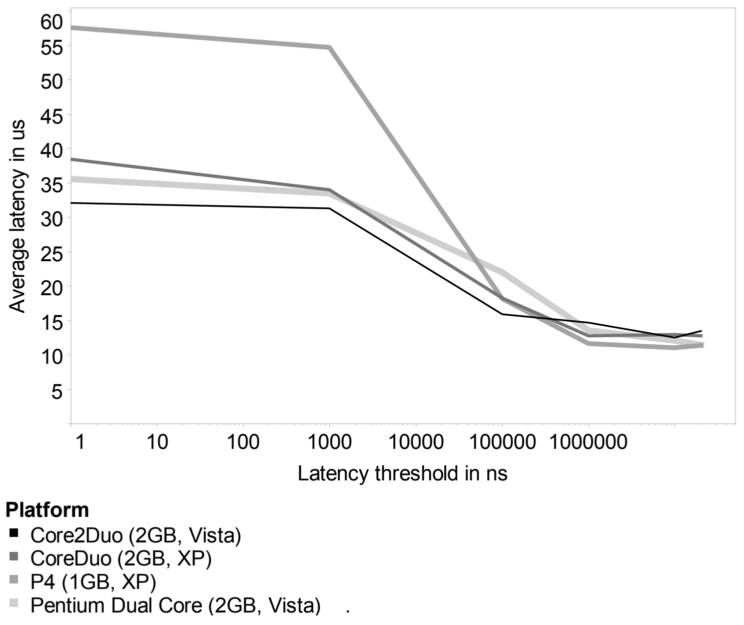


Figure 5.9. Latency of Individual Profiler Calls vs. Latency Threshold

We thus measured the time spent in the profiler for each of its invocations (at the begin and at the end of each interval) by measuring the time at the entry to and exit from the profiler, and by maintaining the average latencies of profiler calls in a session. **Figure 5.9** shows how the average profiler latency depends on the threshold. The y-axis shows the average latency of calls to the profiler. The x-axis shows the online trace filtering threshold, on a logarithmic scale from 1ns to 100ms. The figure shows four lines. Each line corresponds to a different platform on which we repeated the same 5-minute interactive session 14 times, once for each threshold (0ns, 1us, 100us, 1ms, 10ms, 20ms, ..., 100ms).

It is interesting to see that Figure 5.9 clearly follows Figure 5.8: at thresholds over 10ms, the trace size as well as the time spent in the profiler remain at a low value, while thresholds below 100us lead to orders of magnitude larger traces and several times more profiler overhead. Setting the threshold somewhere between 10ms and 100ms thus not only allows us to still capture all perceptible episodes, it also eliminates the majority of the profiling overhead.

5.6.5 Maximum Case Profiling Cost

Based on the previous findings, we have chosen two reasonable filter thresholds: 10 ms (more information) and 70 ms (lower overhead). We ran 10 sessions with a 70 ms threshold and 15 sessions with a 10 ms threshold, and again measured the time spent in the profiler by measuring the time at the entry to and exit from the profiler, and we maintained the average and maximum times of profiler calls in a session. **Figure 5.10** gives an overview of this information.

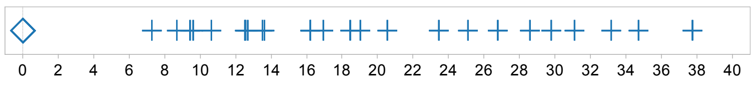


Figure 5.10. Average (Diamond) and Maximum (Cross) Profiler Latency [ms]

The x-axis shows the latency of calls to the profiler, in milliseconds. The figure contains two marks for each of 25 interactive sessions: a diamond represents the average time needed for each call to the profiler, while a cross represents the longest profiler call observed in that session.

On average, a call to the profiler takes only a few microseconds. The maximum latencies of profiler activity go up to 38 ms. On its own, such a long-duration profiler call is not perceptible. However, adding 38 ms to a previously imperceptible episode might make it perceptible. It is thus possible that, in the extreme case, our profiler perturbs our measurements.

Part III

Evaluation

This part contains three chapters. The chapters discuss the evaluation of our profiling approach. We evaluate our approach in the lab but also with real developers outside of the lab. Chapter 6 and Chapter 7 discuss evaluation with real-world applications in the lab. Chapter 6 discusses the evaluation on two real-world applications which we used in the lab. From the collected profiling data we identify and resolve performance problems. On the other hand, Chapter 7 discusses the evaluation with CodeBubbles. Profiling data used in this evaluation is collected both from the authors of this application and from the lab. In this chapter we identify and resolve performance problems in CodeBubbles, confirming with the authors that the problems and places in their code we found with our profiling approach are relevant. Chapter 8 discusses the evaluation we did out of our lab. In this evaluation, the authors used their own applications developed in Java on top of our profiling tool. Further, they analyzed the collected profiling data to identify performance problems in their applications.

Chapter 6

Evaluation with Informa and Eclipse

Initially, we evaluate our approach ourselves. We divide this part of the evaluation into two sub-parts. First, we performed an initial case study using the profiler ourselves in our research group. We tried to fix performance problems with applications such as our Informa, a web-based classroom response system [60], and afterward to validate if we successfully resolved them. In general, a resolution means that the performance problem pointed out by the profiler is perceptible and after the intervention becomes imperceptible and thus it is not reported anymore by the profiler. Second, we implicitly involved students into another case study. They implemented their course projects in the Eclipse environment. We provided them with Eclipse modified to run with our profiler. At the end of each runtime a new trace was uploaded into the central repository. It allowed us to use the profiler with lower sampling rate and thus to minimize unnecessary overhead. That way we also gathered more than one context in which landmarks execute.

6.1 Informa

The Informa clicker tool consists of two interactive applications: The instructor runs the instructor application on her computer to direct the learning activity, while the students run instances of the clicker application on their computers. The clicker applications communicate with the instructor over Java RMI. Our tool is a non-trivial interactive application, consisting of 77000 lines of code in 387 classes. In this experiment we ran the clicker and the instructor applications on top of LagHunter. While the applications communicate over Java RMI, each application runs in a separate virtual machine, and thus also produces a separate latency profile. We conducted clicker sessions where we used the instructor application to post problems that had to be solved with the clicker applications. Given that both applications are based on the same code base, we performed a single analysis combining latency profiles gathered on the instructor application and on the clicker applications. The analysis was based on twenty latency profiles.

List of bug reports. LagHunter produced 35 reports of observers that at least once took longer than 100 ms, summarizing a total of 7251 captured observer notifications. The number of notifications for any given bug ranged from 1580 to just 1 (four reported observers were notified only once, and two of those took over 600 ms to respond). Given that this analysis is based on multiple traces, a latency bug that occurs only once overall is most probably due to some background activity outside the profiled application, or due to a garbage collection. The maximum exclusive latency of the reported observers ranges from 109 ms to 3094 ms, with a total of four observers exhibiting a worst-case response time of over 1 second. The total exclusive latency, which corresponds to the overall effect a given observer has on the performance perceived by the entire community of users, ranges from 109 ms all the way to 195 seconds. The number of stack samples, which correlates with the total exclusive latency, ranges from 0 (for one observer) to 14709. We believe that the total exclusive latency of 550 ms for the one reported observer for which we have no stack samples was due to garbage collection (during garbage collection, even the JVMTI stack sampling thread is stopped). The observer CCTs, which are important for identifying the cause of a bug, range in size from 15 nodes to 6978 nodes. This largest CCT belongs to the observer with the largest number of samples and the largest total exclusive latency. Fortunately, the size of the CCT does not reflect the difficulty of identifying the cause of the long latency.

Case study. We now discuss four of the longest latency bugs in more detail.

6.1.1 Slow painting

The bug with the largest total exclusive time (195 seconds) manifests itself only in the instructor application. This application uses hierarchical agglomerative clustering to compare the solutions submitted by the different students. The result of that comparison is visualized in a dendrogram.

This bug report indicated that the painting of that dendrogram was slow. The report showed 788 requests that took up to 2284 ms. We first looked at the source code of the observer method, ignoring the CCT available in the bug report. This inspection of the observer's source code showed that it uses an inefficient implementation based on nested recursive traversals of the dendrogram.

Fix: A simple refactoring that computes a result when the dendrogram is created, instead of each time the dendrogram is traversed, allowed the elimination of the inner recursion. However, when validating the fix, we found that it did *not* significantly reduce the latency.

Root cause identification: Our failure to initially find the root cause of this latency bug is what prompted us to combine latency profiling with call stack samples. The observer's CCT for this bug highlights its root cause. The time is spent computing the size of the dendrogram nodes. Each node represents a student's solution, and computing the visual size of that node required the evaluation of the solution, which, for the given type of clicker question, entailed the compilation of the student's submitted Java code. The CCT immediately shows this non-intuitive call chain.

Fix #2. We fixed the problem by caching the result of the compiler invocation. This simple fix drastically reduced the perceptible latency.

6.1.2 Creating an RMI registry

The instructor application launches an RMI registry in which it registers its remote objects. The clicker application then looks up these objects in that registry.

LagHunter produces a bug report for an observer in the session setup dialog, with one occurrence for each run of the instructor application, and a maximum latency of 1160 ms. This observer is triggered when the user clicks the OK button of that dialog to start a session. A look at the report's CCT immediately points to a hot method that creates a local RMI registry.

Fix: Because there was no reason for waiting with creating an RMI registry until the user confirmed the session setup dialog, we moved that computation into a background task which we launch as soon as the session setup dialog is shown. At the time the user confirms the dialog, we retrieve the registry created in the background task. This relatively local change eliminated the majority of the long latency and noticeably improved the perceptible performance.

```
table.changeSelection(startRow, 0, false, false);
table.changeSelection(endRow, 0, false, true);
```

Fix: We quickly fixed the problem by replacing the two calls with one call to set the selection, followed by a call to scroll to the last selected row.

```
table.getSelectionModel().setSelectionInterval(startRow, endRow);
table.scrollRectToVisible(table.getCellRect(endRow, 0, false));
```

This simple fix sped up this action considerably.

6.1.3 Synchronous lookup of remote object

The clicker application has to connect to the instructor application in order to join a session. At startup, the clicker presents a dialog to prompt the user for the host name of the instructor's computer. Once the student clicks OK, the clicker tries to establish a connection to the instructor's remote object.

LagHunter produces a bug report for the observer responding to the OK button, with one occurrence in each trace, and an average latency of 591 ms, Because we originally had tested the application on a single machine, we had not noticed this latency ourselves. However, in the deployed setting, with significant network latencies, connecting to the instructor application turned out to be excessively slow. A quick search in the observer's CCT, starting at the root (the observer node) and following the hottest path, directly leads to the cause: a call to Naming.lookup() that is responsible for 76% of the observer's call stack samples. While designing the application, we had performed remote calls in background threads to hide their latencies, however, we had overlooked this naming service lookup, which corresponded to a remote call in our deployment.

Fix: While the identification of this problem, given the profile, took little effort, the implementation of the solution, the asynchronous execution of the lookup, including the showing of progress information to the user, took several hours. The significant effort required for turning synchronous computations into asynchronous background activities (including the necessary additions to the GUI for dealing with incomplete ongoing background activities), is one of the reasons why a profile-driven approach to eliminating long-latency activities is beneficial. Asynchronous computations in GUI applications can lead to significantly more complex code and increase the potential for concurrency-related bugs. For this reason they should only be introduced if they truly improve the perceptible performance.

6.1.4 Excessive creation of icons

The report with the second longest total exclusive latency represents calls to the *mouseDragged()* method on a mouse observer. Its 253 captured calls had an average latency of 236 ms, with a maximum of 665 ms. LagHunter produced two other, similar, bug reports about calls to *mouseReleased()* and *mousePressed()*. The source code of all three observer methods invokes other, relatively complex methods, and an initial manual study did not reveal the cause of the lag. However, the observer's CCT immediately highlighted the problem: the hottest call path from the CCT root leads directly to the constructor of the ConfidenceBar class, and continues on to a native method related to image handling. A quick look at the *ConfidenceBar* constructor showed that it loads three images files and filters the loaded images to create icons. Loading and processing images is relatively expensive. Furthermore, the resulting icons were the same for all instances of ConfidenceBar, thus creating them for each instance was unnecessary. Moreover, the CCT also revealed the caller of ConfidenceBar's constructor, and a look at its source code showed that it recreated new ConfidenceBar objects each time they were called.

Fix: We moved the image loading and processing code from the constructor into the static initializer. Our simple fix entirely eliminated all perceptible latency in the three classes of episodes.

Validation. To validate that the above fixes indeed improved the overall perceptible performance, we ran five clicker sessions with, and five without the fixes. In each session we performed the same interaction, using one instructor and four clickers, and we focused on covering the functionality affected by the bug fixes. We then computed the cumulative latency distribution over all episodes. **Figure 6.1** contains ten cumulative latency curves representing the perceptible performance of the instructor application. We omit the cumulative latency distributions of the clients, because the only bug we fixed in the client was C3, a bug that only causes a single long-latency episode for each clicker session.

The dashed curves in Figure 6.1 show the performance of the buggy version, and the solid curves show the performance after fixing the above latency bugs. The x-axis shows the latency. The y-axis shows how many episodes per second had a latency longer than X. The vertical line at 100 ms shows the perceptiblity threshold. An ideal distribution would have the shape of an "L", with the vertical line at or below 100 ms and the horizontal line at 0.

The area between the two sets of curves shows the significant improvement of the perceptible performance: the bug fixes brought the curves much closer to the ideal "L" shape. Thus, besides our subjective feeling of working with a more responsive application, and the observer latency profiles that validated the speedup due to each of the five bug fixes, also the cumulative latency distributions clearly reflect the benefit of tuning this real-world application using our approach.

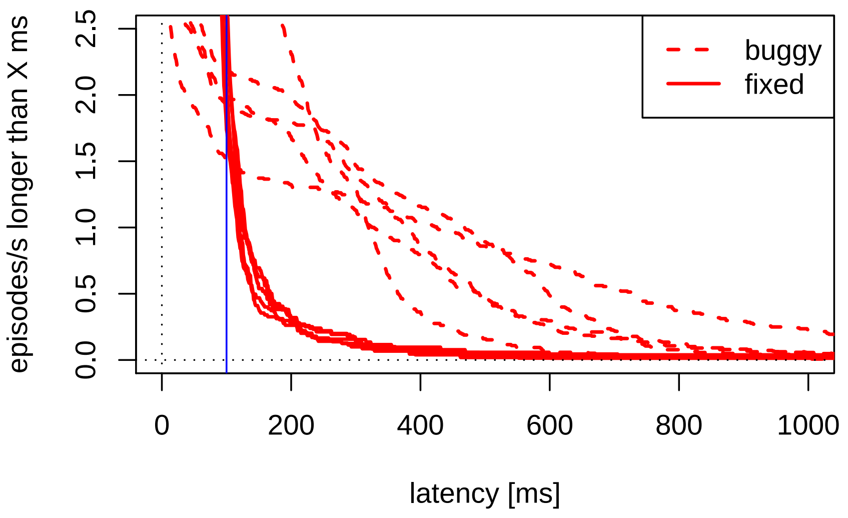


Figure 6.1. Clicker instructor application latency distribution

6.2 Eclipse

As we discussed at the beginning of this chapter, we implicitly involved students into the case study with Eclipse. Implementing their course projects, they used Eclipse modified to run on top of our profiler. At the end of each runtime a new trace was uploaded into the central repository. We collected 1108 session reports during a three-month experiment. The experiment involved 24 users working for a total of 1958 hours with Eclipse. We configured LagHunter to take a call stack sample every 500 ms on average, and to drop all samples taken during idle time intervals.

6.2.1 Parameters

LagHunter's analysis engine detected 881 issues in the given reports. **Table 6.1** characterizes five key parameters of this set of issues. For each parameter it shows the mean, first quartile, median, second quartile, 90-th percentile, and maximum value. The first parameter, the average exclusive latency, represents the severity of a given issue: the longer its latency, the more aggravating the issue. The next three parameters describe how prevalent a given issue is. From the perspective of a developer, the need to fix an issue increases when it is encountered by many users, occurs in many sessions, and occurs many times. The last parameter, the number of collected call stacks, is indicative of the chance of fixing the issue. The more call stack samples available about a given issue, the more information a developer has about the potential cause of the long latency.

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

Table 6.1. Univariate characterization

The average exclusive time of an issue varies between less than a millisecond and over 38 seconds. However, 90% of these issues take less than 119 ms. This shows that most "issues" are benign. They can easily be filtered out in the web-based front-end to our repository. We captured information about benign issues on purpose[1]. This allowed us to see whether a given landmark *always* had a long latency, or whether its long latency invocations were exceptional. The number of *users* who encounter an issue, the number of *sessions* in which an issue appears, and the number of overall *occurrences* of an issue follow equally skewed distributions. Only a small minority of issues are prevalent, which means that developer effort can be focused on fixing a small number of high-impact performance bugs. The table also shows that the *stack* sample distribution is equally slanted: half of the issues receive no stack samples, and 10% of the issues receive 42 or more stack samples.

6.2.2 Landmark location

Most of the reported landmarks correspond to listener notifications. In which parts of Eclipse are those listeners located? First, we noticed that a significant fraction of them (275 of 881) are located *outside* the standard Eclipse classes. Many of those are inside optional or 3rd party plug-ins, but some are also in the class libraries, and a few are in dynamically generated proxy classes. The majority of the 606 landmarks in the standard Eclipse distribution are located in a few dominating plug-ins. **Figure 6.2** shows the top-15 of these plug-ins.

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

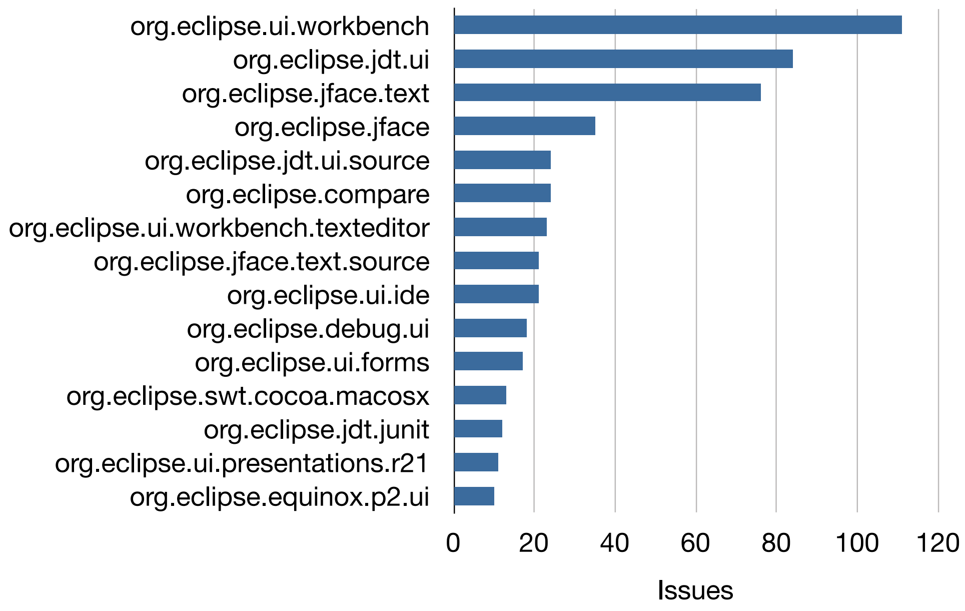


Figure 6.2. Distribution of landmarks to plug-ins

The dominant plug-in (with 111 landmarks) implements the Eclipse workbench user-interface. Moreover, given that the users in our experiment were developing software in Java, it is not surprising to notice many landmarks in the Java Development Tools (JDT) user interface (84) and the JFace text editor (76) plug-ins.

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

6.2.3 How prevalent are long-latency issues?

Figure 6.3 shows a bubble plot of issue occurrence vs. average exclusive latency. Both scales are logarithmic. Each bubble corresponds to an issue. The size of the bubble is proportional to number of users who encountered the issue.

Most issues are clustered below 100 ms. Eclipse is a mature product that has been extensively tuned for performance, and the Eclipse team, explicitly or implicitly, must have cared about the 100 ms perceptibility threshold. However, that the y-axis corresponds to the *exclusive* latency of an issue, which means that

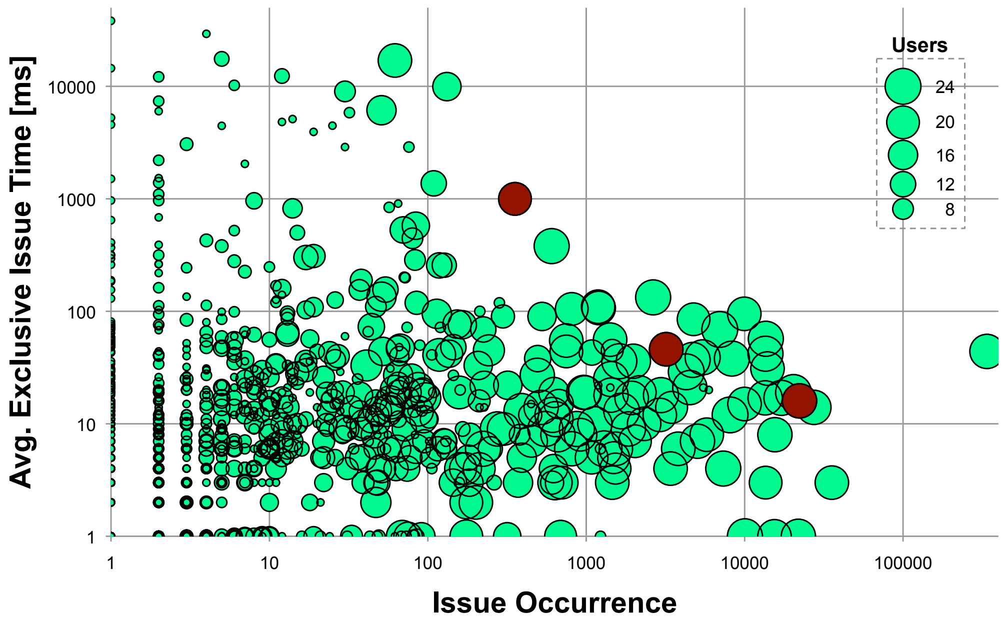


Figure 6.3. Number of users encountering an issue

the application's response time may have been significantly larger due to nested landmarks. Moreover, the axis represents the issue's *average* latency, and thus some individual occurrences of an issue might have taken significantly longer.

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

6.2.4 Availability of stack samples

Figure 6.5 has the same axes as Figure 6.3, but the bubble size is proportional to number of collected stack samples. This figure confirms that issues that occur often or exhibit a long latency have a higher chance of being encountered by the JVMTI stack sampler. These are precisely the issues developers want to fix, and thus our approach automatically produces the information where it is needed most, and omits it where it is irrelevant.

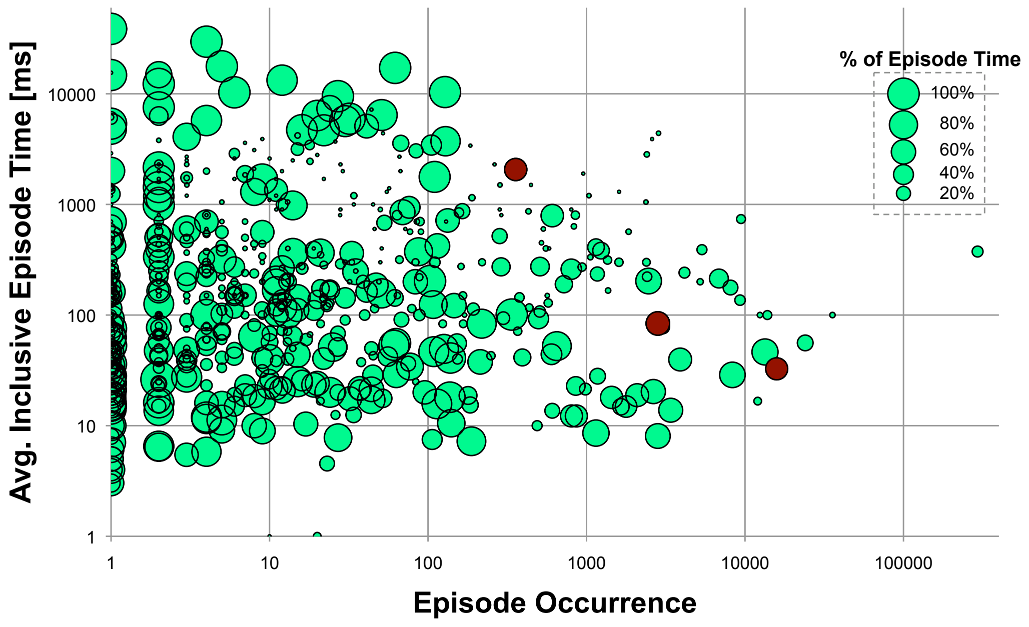


Figure 6.4. Severity of issues (x/y), and benefit of fix (size)

6.2.5 Case Studies

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

Figure 6.4 shows all issues in the repository and highlights the three case study issues. In this figure, the x-axis represents the total number of episode occurrences containing the issue. The y axis shows the average inclusive time of episodes containing the issue. The circle size represents the fraction of inclusive episode time due to the issue.

Like in Figure 6.3 and 6.5, each circle represents an issue. However, unlike in those prior figures, the x and y axes both represent the *episodes*. This corresponds to a user's view of performance: users are not interested in the exclusive time of issues, but they primarily perceive the inclusive time of complete episodes. The x/y space of Figure 6.4 thus represents the user's perspective (the severity as perceived by the user). The x-axis represents *how often* a user is annoyed, and the y-axis represents *how much* a user is annoyed each time. The third dimension, the size, represents the developer's perspective: a developer could say "I know this circle represents a frequent (x) and big (y) annoyance for the

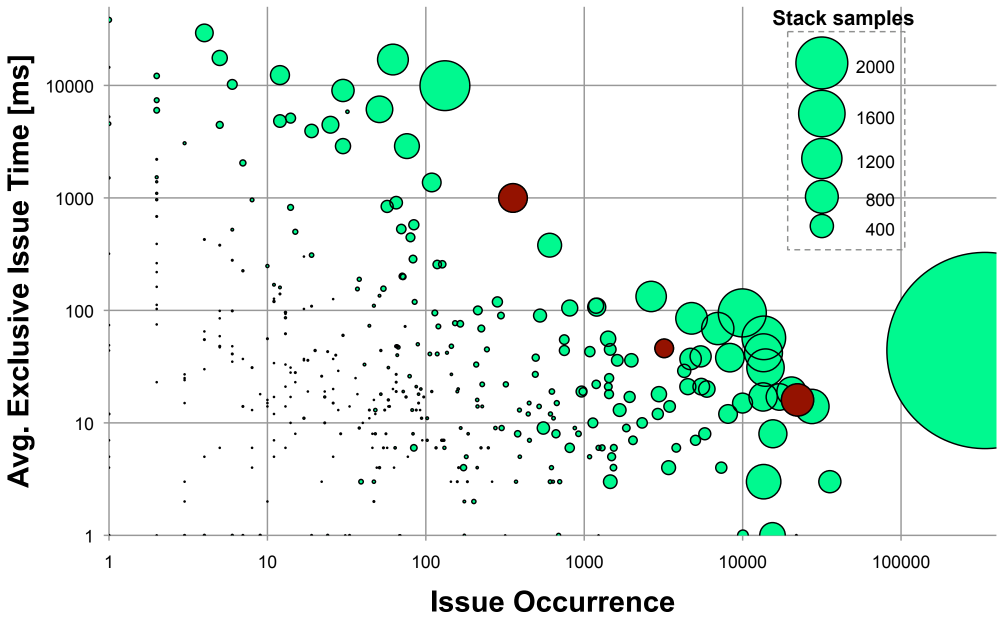


Figure 6.5. Number of stack samples per issue

user, but how much can I reduce this annoyance if I fix this issue?" The size of the circle represents the answer to that question. A circle of 50% size (like the three circles representing our three case studies) means that by fixing that issue, the latency of the episodes that contain that issue can be reduced by 50% on average.

Figure 6.4 shows that there is a relatively small number of issues a developer would want to explore. The most promising issues are those that occur frequently (to the right), lead to long latency episodes (to the top), and can significantly reduce the containing episodes' latency when fixed (large circle). The three issues we picked lie more or less on the frontier of Figure 6.4, which means that they incurred significant latency or a significant number of occurrences.

6.2.5.1 Methodology

In each case study we followed the same methodology:

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

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

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

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

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

6.2.5.2 Issues under study

Table 6.2 lists the three issues we fix in the following case studies. The first column shows the name of the landmark method (we omit the class name for brevity). The second column shows the average exclusive latency as identified

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

Table 6.2. Resolved issues

by LagHunter. The third column shows the same metric, but obtained during the reproduction step. The last column shows the latency after applying our fix. The small exclusive latency of only 16 ms for the second issue may be striking. However, this value is an average, and a significant number of this issue's occurrences were perceptibly long. More generally, the table shows a significant difference between the latencies identified in the field and the latencies observed during the reproduction in the lab. The goal of a reproduction in the lab is to confirm the presence of a problem, and to gather initial hypotheses about the causes. A reproduction in the lab thus is not necessarily a faithful reproduction of *all* of a user's activity. Moreover, it takes place in a different environment, with a potentially different application configuration, and thus necessarily leads to measurement results that can differ significantly from the average observed in the field. The important aspect of Table 6.2 is the drastic reduction in latency between the reproduced and the fixed measurements. Moreover, in the real-world use of our approach, the ultimate measure of success will be the reduction of the latencies observed *in the field*, after the deployment of the fixed version of the application.

The following three subsubsections describe each case study in detail.

6.2.5.3 Maximize and Restore

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

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

Pruning and Resolution. The calling context tree of this issue is shown in the ring chart [15] of **Figure 6.6**. The root of the tree (the main method) is rep-

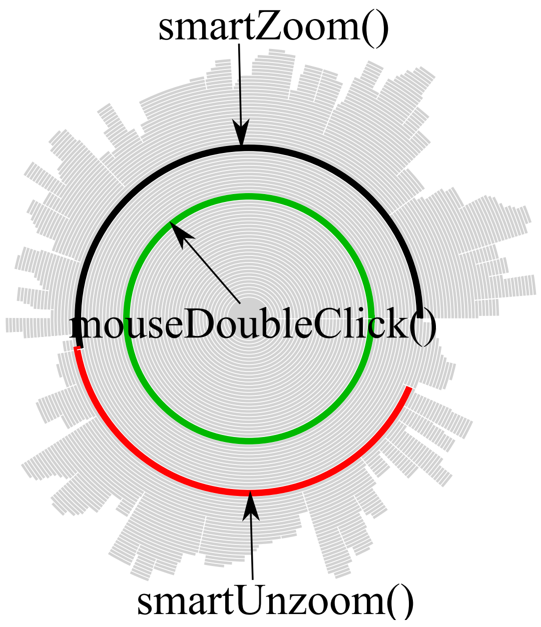


Figure 6.6. Maximize and restore

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

Validation. We reran Eclipse on our agent and repeated the experiment performing double-clicks on the page tabs. The latency decreased to only 4 ms, as shown in Table 6.2.

6.2.5.4 Rename Refactoring

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

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

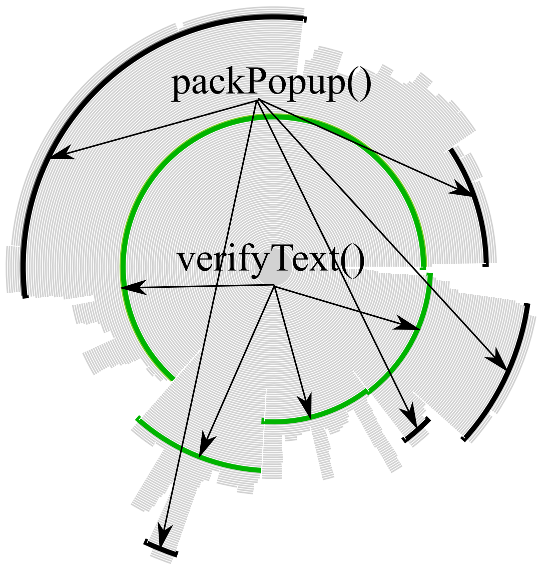


Figure 6.7. Rename refactoring

Pruning. **Figure 6.7** shows the calling context tree of this issue. The black ring segments towards the leaves of the tree represent the root cause of the problem. They correspond to calls to the `packPopup` method of the RenameInformationPopup class. As the most sampled method in this issue, it consumed most of the time. Its class implements a tooltip-style popup that appears below the identifier while in rename rafactoring mode. The popup has a special shape of a callout. The packPopup method recomputes the shape of the non-rectangular popup after each key press. We used call pruning to eliminate the expensive method calls. This significantly reduced the latency, however, the popup window did not appear anymore.

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

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

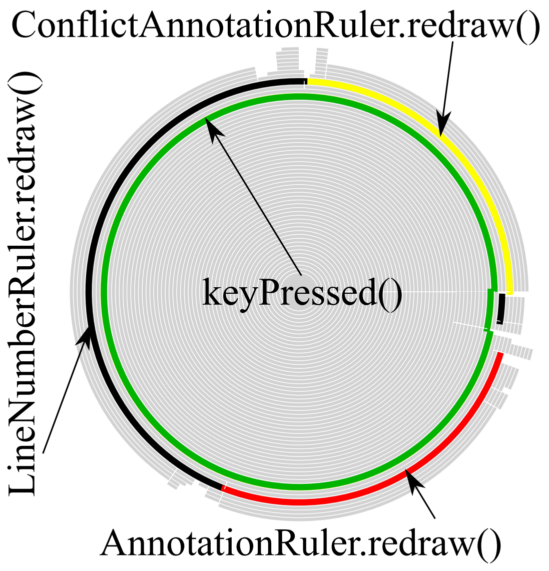


Figure 6.8. Scrolling

6.2.5.5 Scrolling

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

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

Reproduction. The calling context tree in **Figure 6.8** shows that most of the time in this issue is spent redrawing three components: LineNumberRuler, AnnotationRuler, and ConflictAnnotationRuler. Knowing about the landmark method and these three classes points us towards the action the user must have performed: scrolling the viewport of the source code editor. To reproduce this issue we open a long Java source file that can't fit on the screen. We hold down the arrow key to scroll. We notice that the scrolling is not as smooth as it could be. The bigger the opened project and the more views we open, the bigger the latency becomes.

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

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

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

Discussion. The calling context tree in Figure 6.8 shows three rulers that react to key presses. However, the calling context tree of the reproduced issue contained only `redraw` invocations for two of those ruler classes. The reason for this is that the ConflictAnnotationRuler class is not part of any standard Eclipse plugin. Sometimes, such additional plug-ins installed by users may significantly reduce the responsiveness of an application. Without the help of LagHunter, developers would not be able to discover such issues.

6.2.5.6 Limitations of Study

Issue selection. We selected three issues for our case studies. Our selection was not a blind, random process, and it was not based on strict, pre-determined criteria. While we took care in selecting severe and interesting issues, we may have been biased towards issues that are easier to understand or fix. We believe that Figure 6.4 provides a good starting-point for methodologically selecting issues to analyze in future empirical studies. Ideally, such studies would take a random sample of issues with a pre-determined minimum number of episode occurrences, average inclusive episode time, and percentage of inclusive episode time due to the issue.

Self evaluation. We evaluated the benefits of our tool by using it ourselves. Thus, our case studies may be biased by our desire to see our approach perform well. Moreover, we were not familiar with the Eclipse code base, and thus our approach to fixing a bug is not necessarily representative of the approach of an experienced Eclipse developer.

Chapter 7

Evaluation with CodeBubbles

Today's integrated development environments (IDEs) mostly work with file-based editing. A group of researchers from Brown University has implemented a different interface paradigm that is based on collections of lightweight editable fragments, called bubbles [31]. Their approach is founded on the metaphor of a bubble which is a fully editable and interactive view of a fragment such as a function, method documentation, or debugging display. In contrast to standard windows, bubbles have minimal border decoration, avoid clipping their contents by using automatic code reflow and elision, and do not overlap but instead push each other out of the way. Bubbles are placed in a virtual space represented on the screen with a strip, where a cluster of bubbles comprises a concurrently visible working set.

With the authors of CodeBubbles, we discussed their complaints. The environment was perceptibly slow. As the environment is interactive and developed in Java, we decided to try our approach with it.

In collaboration with Professor Steven P. Reiss from Brown University, we have integrated our profiling tool to be a part of CodeBubbles which enabled us to collect traces on the server as CodeBubbles is used. This gave us a chance to reveal more performance-relevant problems in the future.

During the month and a half we collected 88 traces, that is nearly 145 thousand landmark method executions of 349 distinctive landmarks, during approximately 20 hours of CodeBubbles use.

We first analyzed landmarks with perceptible average which repeat multiple times, and which are present in most of the traces. Table 7.1 shows three such examples. *ComponentMoved()*, *componentResized()*, and *componentAdded()* correspond to methods that move, rearrange, and add bubbles respectively. We discuss each of them in the remainder of this section.

Issue	Count	Average [ms]
BudaBubbleArea$BubbleManager.componentMoved	180	174
BudaBubbleArea$BubbleManager.componentResized	993	190
BudaBubbleArea$BubbleManager.componentAdded	176	159

Table 7.1. Overview of the issue

7.1 Move Bubbles

The environment works imperceptibly to a user for a small working set of bubbles. With growth of the number of bubbles and relations between them, the environment becomes sluggish as response times become perceptible. It particularly becomes perceptible when a user tries to move a group of bubbles. The *componentMoved()* method takes more than 100 ms on average. If a new bubble appears in the middle of a group of bubbles then the action of their rearrangement will occur in a burst of events to animate the movements until the bubbles fit on the screen.

Figure 7.1 shows the cumulative calling context tree built for the *componentMoved()* landmark from all the traces. The thicker the lines the more samples are taken on that branch. The *componentMoved()* method in 96% of time calls *getShape()* method. It maintains a model of the bubble group we want to move. If a new bubble should belong to a group then the AWT geometry features are used. AWT provides the Java 2D classes for defining and performing operations on objects related to two-dimensional geometry. AWT maintains our bubbles in a Vector. Each time it adds a new bubble into the vector, it creates a new vector with one more spot, copies all the elements, and adds the new one. Finally, it shows that 81% of time this action wastes in copying the elements of the vector. Moreover, this may trigger additional garbage collections and further impact the performance.

7.2 Resize Bubbles

This landmark call costs 190 ms on average, according to the traces we have collected. Resizing of a bubble leads to several actions such as group split, group merge, or bubble-add to an existing group. Figure 7.2 shows the cumulative calling context tree of the corresponding landmark.

Following the left branch, which has the thickest line along the path toward the leaf, we see that a bubble resize action may cause a bubble add action. This happens when the bubble that is being resized, touches another bubble which

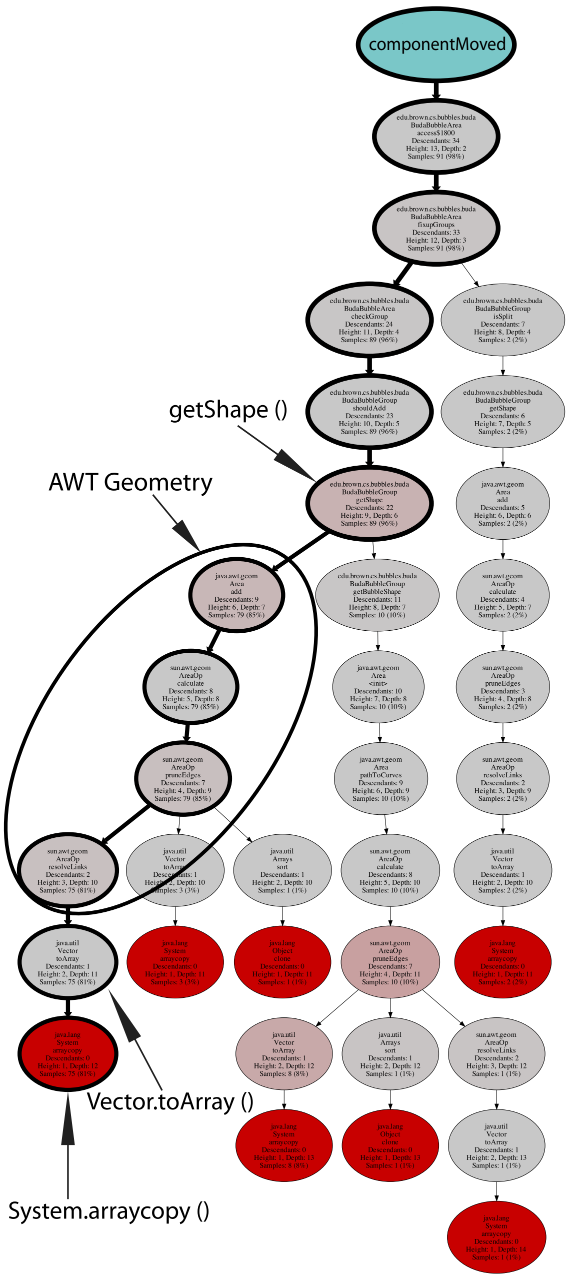


Figure 7.1. Move Bubbles CCT

is single or in a group of bubbles. Thus, they will either form a new group of two bubbles or the existing group will expand. Either way, it causes the bubble add action. The analysis from the collected traces showed that more than 95%

of samples were taken while handling this action. The *getShape()* method is on the branch of the *componentResized()* method. It maintains the bubble model using AWT geometry like in the case discussed in Section 7.1. Approximately 68% of time this branch spends in copying data array.

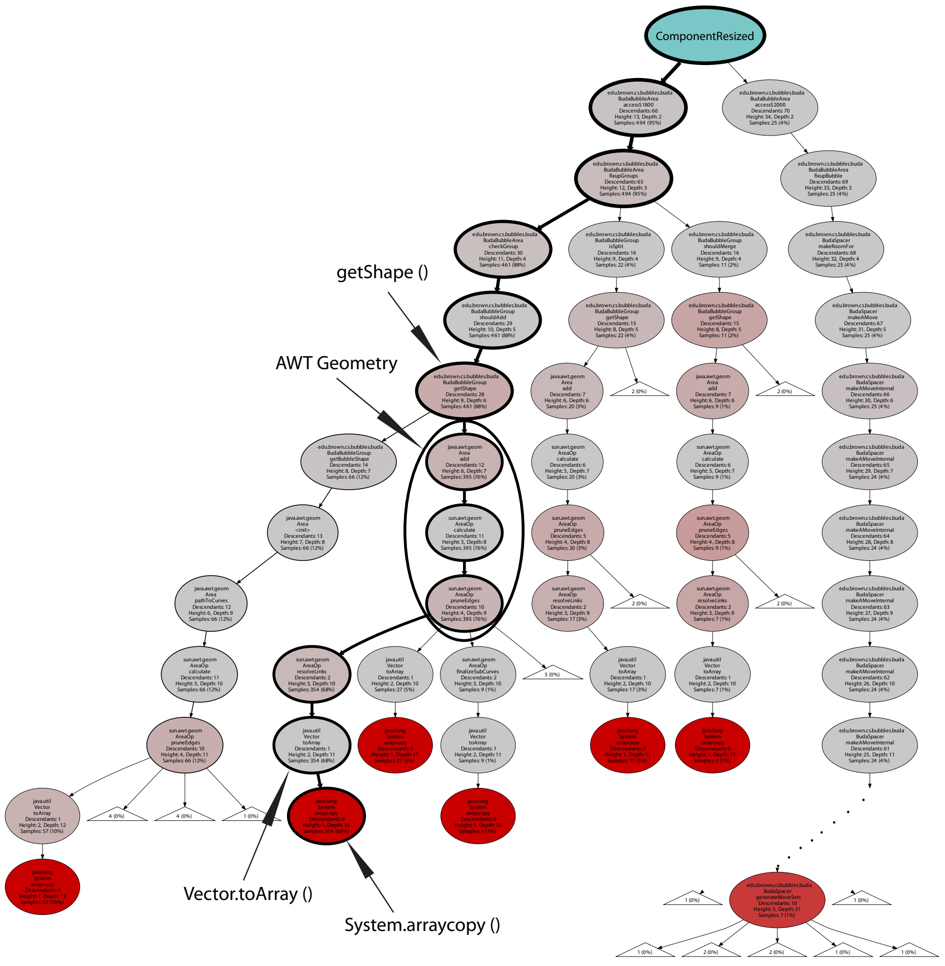


Figure 7.2. Resize Bubbles CCT

7.3 Add Bubbles

New bubble construction starts in the *componentAdded()* observer method. The cumulative calling context tree is given in Figure 7.3. As shown in the figure, this observer nests a call to the *localAddBubbles()* method which further has two branches. There is no particularly heavy branch that consumes most of the samples. The right branch, *fixupBubbles()*, contains a long chain of recursive calls to *makeAMove()*, which adjusts positions of bubbles to fit properly in the workbench. More than 25% of samples are taken while executing this method. This method unnecessarily generates a new array every time it is called for each loop step. This is not efficient especially if arrays are large. The listing below shows the original implementation:

```
private Collection<Configuration> makeAMove
   (Configuration cur,Adjustment mvd,int recurse,long when) {
   List<Configuration> rslt = new ArrayList<Configuration>();
   ...
   for (Adjustment adj : generateMoveSets(cur,mvd)) {
         rslt.addAll(makeAMove(new Configuration(cur),adj,
                     recurse+1,when));
   }
   ...
}
```

Instead, we could pass the array as one of the arguments as shown in the listing below:

```
private void makeAMove (ArrayList<Configuration> result,
     Configuration cur,Adjustment mvd,int recurse,long when) {
   ...
     for (Adjustment adj : generateMoveSets(cur,mvd)) {
            makeAMove(result, new Configuration(cur),adj,
                              recurse+1,when);
     }
   ...
}
```

We did an experiment stressing especially this functionality. We only save a few milliseconds with this small optimization. The overall latency of the landmark requires more optimizations. Looking back at Figure 7.3 we notice that almost all the leafs correspond to the *System.arraycopy()* method invoked by *Vector.toArray()*. They approximately take 75% of all taken samples. These leafs belong to sub-branches of the right *localAddBubble()* method branch and they are: *mergeGroups()*, *checkGroups()*, *addBubbleGroup()*, and *isSplit()*. Each of them maintains model of bubbles using the AWT graphics package. In this particular case it ends up copying arrays as we already have seen in Sections 7.2 and 7.1.

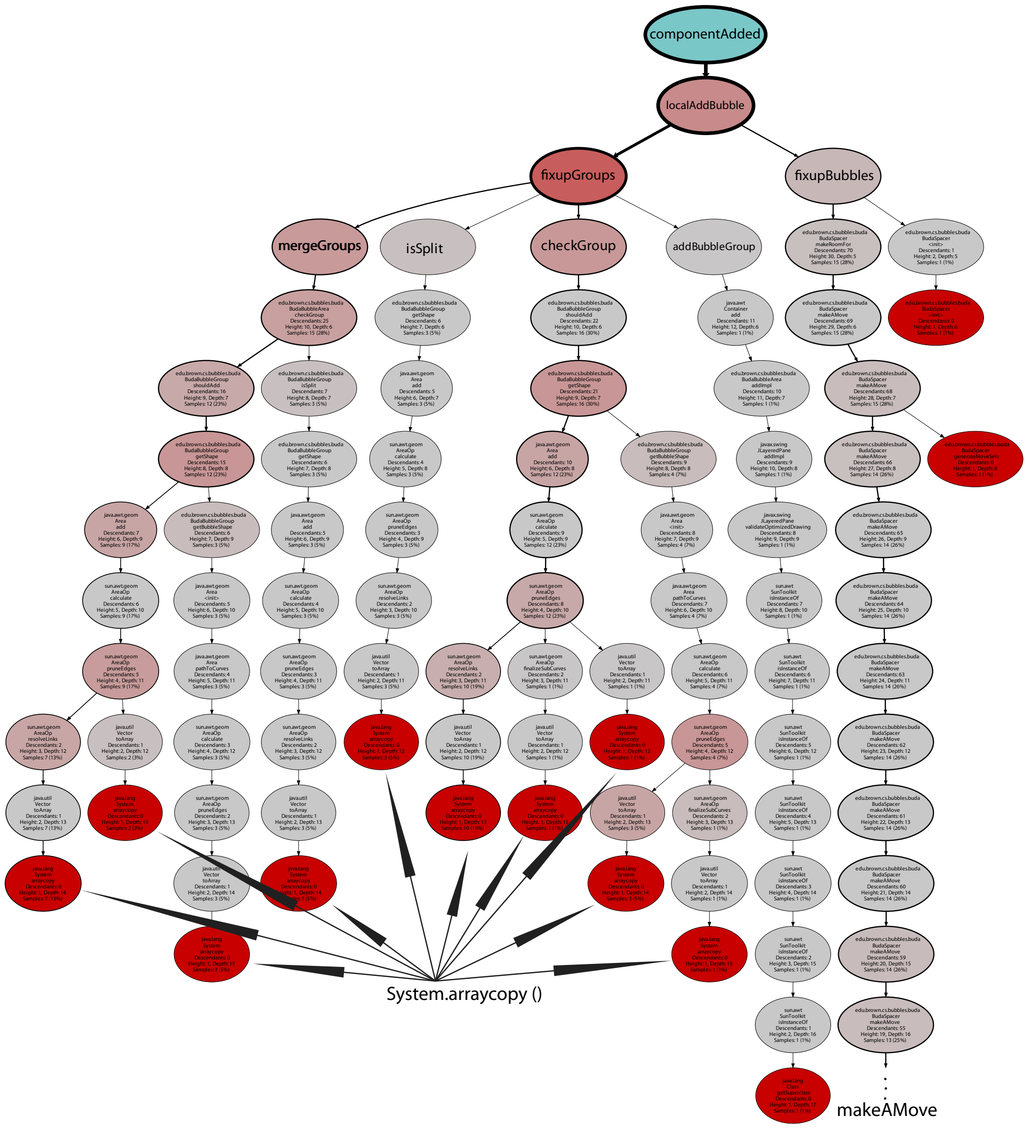


Figure 7.3. Add Bubbles CCT

7.4 Conclusion

We discussed our findings with the authors. They agreed that the places in the code identified by our profiling tool are complex and should be redesigned. They also confirmed changes such the one proposed in Section 7.3.

It showed cases where profiling techniques cannot help us to locate the exact line in the code responsible for bad performance. Sometimes, the closest location of the problem is to point at the group of methods or classes. As we have shown in Section 7.3, many calls to a number of methods that together accomplish some task may cost a lot in total but not individually.

Chapter 8

Evaluation with Developers

The goal of the experiments was to let real developers use our profiling approach and quantify how useful it was for them to recognize, locate and understand interactive application performance problems.

To test the usefulness of our approach, we got the support of RCUB, a computer center at the University of Belgrade in Serbia, where we had a chance to collaborate with their developers and to use our approach to search for performance problems. They are a Java-oriented group working on various projects for the state and the University. [2]

8.1 Background on Empirical Research

Looking at the existing body of research on empirical evaluation through controlled experiments, we tried to distill important aspects for the experiments we intended to do.

8.1.1 Experimental Evaluation Versus Grounded Theory

In general, experimental evaluation is a controlled evaluation of specific aspects, where the evaluator sets a hypothesis to be tested. It is required primarily to know the tasks but also to have a high enough number of participants to achieve a statistically significant interpretation of the results. Also, to avoid biased results and interpretations the tasks must be very clear, narrow, and fully controlled.

[2]http://www.rcub.bg.ac.rs/index.php?lang=en

In contrast to the idea of having a hypothesis before the experiment, with the Grounded Theory we first collect data. Then, we mark important data with a series of codes. Further, the codes are grouped by similarity. From these groups we form the categories and use them to make a hypothesis. This hypothesis is also called a reverse engineered hypothesis. Grounded Theory, as a widely accepted and systematic theory from data, has the goal to formulate hypotheses based on conceptual ideas [63] [40]. "All is data" is its fundamental property which means that everything that gets in our way when studying a certain area is data.

A number of experimental conditions are important and they differ only in the value of some controlled variable. The changes in the behavioral measure are attributed to different conditions. When designing an evaluation experiment [24] [118] [76], researchers find important aspects such as:

- **Subjects.** Who are they? Are they representative? Is the sample sufficient?
- **Variables.** What does it change and how it could reflect the measures?
- **Hypothesis.** What do we want to show?
- **Experimental design.** How are we going to do it?

8.1.2 Data Collection

The important step in the evaluation is data collection. We can acquire data by different methods such as: video, interview, questionnaire, tasks and questions, reports, etc.

Video cameras and **direct logging** of application usage have shown to be a good practice to collect valuable data. **Tasks and questions** in the experiments have to encourage cooperation. To achieve this goal we should make tasks creative but also to control them.

The **interview** is a very sensitive part of the experiment because we constantly interact with the participant. We may easily ask an incorrect question in a conversation and bias the results. Generally, while collecting data from participants in the interview, researchers can use observational methods such as:

- **Thinking aloud**. The user is asked to describe what he is doing and why, what he thinks is happening, etc.
- **Cooperative evaluation**. A variation on thinking aloud where the user collaborates in the evaluation thus both the user and evaluator can ask each other questions throughout.

- **Protocol analysis**. This includes paper, audio, and video analysis.
- **Post-task analysis**. The transcript is played back to the participant for comment immediately or delayed.

Questionnaires are a standard component of experiments that are used at every stage. They include questionnaires that collect information about participants at the beginning of the experiment, and questionnaires given at the end to gather feedback from the participants. The generation of a questionnaire is not a trivial task. Bad questionnaires can be misleading or collect meaningless data.

On the other hand, questionnaires have a number of disadvantages such as subjectiveness. Thus, questionnaires require careful design. We should ask ourselves what information is required and what the answers will look like. What researchers have found to be important is the order of the questions and their layout. If their order is not proper, people may refuse to cooperate. For example, if a survey begins with awkward or embarrassing questions, people are not likely to give honest replies to personal questions.

8.1.3 Quantitative Approach

The goal of this evaluation is to show the effectiveness of our approach in discovering performance problems. For tasks such as ours, it is extremely hard to find enough developers to participate in the experiment.

Moreover, we could not effectively use a quantitative evaluation approach for a few reasons.

First, our approach is designed to be deployed and to function with complex applications. In such environments it is hard or almost impossible to design a meaningful experiment. For example, we want to time developers and find out if there is a statistically significant speedup in finding and resolving performance bugs. For such an experiment we need a baseline to compare with. It should be another profiling tool. Whatever tool is used, solving the same problem twice with both tools is impossible, at the second time we already know where and what to look for. Another example is task size. If we decide to create a small controlled problem then the results might be completely different from those obtained with complex problems. This is due to the dramatic growth in the number of variables as problem complexity increases.

Also, trace files are filled with data that is not readable and understandable as text. Collected information from runtime has to be visualized and thus become understandable. Such a visualization requires a certain interface design and functionality to represent all data written in a trace file. Depending on these two aspects a developer might be able to navigate quicker or slower through data and understand important information. Even though we do fully controlled and focused tasks, the choice of visualization technique might bias the measurements. The reason is that we could not distinguish whether a developer's better or worse performance in solving problems comes from the usefulness of our approach or the visualization quality and understandability. Nevertheless, we could partially overcome this problem by explaining in detail how the visualization tool works before they start using it with real problems.

8.1.4 Qualitative Approach

Qualitative studies were more relevant for the evaluation we needed. In qualitative studies, we begin with decisions regarding the sample to interview, move on to data collection, and conclude with the analysis. Analysis of early data contributes to new emphasis in interviewing. The new data collected by the modified interviewing then produces new analysis results. Also, even though we have a low number of participants, it is possible to collect significant information in a structural form. The following two aspects are important:

Developing detailed descriptions.

We may want to learn as much as we want to see. We may well want to interview more than one informant and integrate their reports, but we will in any case want from our informants the fullest and the most detailed description possible.

Integrating multiple perspectives.

We may want to describe an organization, development, or event that no single person could have observed fully. Although interviews are necessary, standardized questions will not work. This is because every respondent will have different observations to contribute and we should allow her to express herself.

8.2 Study Design

In our experiment, we have evaluated how the use of LagHunter impacts developers while solving performance problems. In total, 5 developers participated in this experiment. We tried our tool on two projects they develop. Two of them were mature, with 10 years of experience of programming in Java. The other three developers have approximately 5 years of working experience in

Java. They all have graduated from the computer science department of the University of Belgrade.

Barbara Kitchenham analyzed empirical evaluations especially in the software engineering area[75]. Her work allowed us to establish the guidelines for the experiment [74]. In this experiment, we used the following protocol:

- Give developers the tutorial in written form and the video record a week before the beginning of the experiment. The video record shows how to use the profiler and the visualization tool to identify the performance problems in one of the applications that we have developed.

- Demonstrate how the profiler and the trace visualizer works one day before the experiment. Repeat the procedure from the recorded video in front of the developers. Even though they have already seen the video, this is a moment when the developers may have a question if something was not clear enough in the video.

- Provide a "kick-off" questionnaire to understand the experience and background of the developers.

- Ask the developers to run their applications on top of the profiler.

- Ask each developer to look into her own produced trace with the trace visualizer and demonstrate an understanding of the tool.

- Divide them into groups based on experience distilled from the questionnaires.

- Let them use their application on top of the profiling tool for one hour.

- Give each developer one hour to analyze traces with the visualizer and to identify problems.

- Record/write their out-loud thinking and the feedback about the problem they have been focused on.

Thus, before they used our tool, they had to read the tutorial. Also, we prepared a video where they could see, step by step, how to use our tool. Moreover, before they started we gave a presentation where we showed how our tools should be used.

Every developer was available for 2 hours to participate in the experiment, working on the project on which they are currently enrolled. During this time slot, first they had to run the tool with the Java project they work on. Then, at the end a runtime trace file was created, and they had to open it with the visualization tool we provided. In the remaining one hour slot they tried to spot a performance problem, analyze it, and tried to find a cause.

8.3 Results

Here, we present one identified problem for each project the developers were working on in the experiment.

8.3.1 SNMP Manager

The *SNMP Manager* application discovers network hosts, browses MIB trees, and analyzes network requests. The Belgrade node in the current topology is an important node not only for the Serbian University network but also for the broader region. Instead of using one of many open source applications, they have developed their own due to their special needs. The application contains nearly 400 classes.

One of the two developers with 10 years experience and one of the remaining three developers worked on this application. Our profiler identified 112 distinctive landmarks from these 2 runtimes. They both triggered the same action of tree expansion in the application. When sorting by average time the landmark that handles this action was among the top ranked.

They blamed this performance problem on slow communication with the database, because in the past they were already optimizing this part of the code. The perceptibly slow response of the application was noticeable when trying to expand the tree. This action queried the database to retrieve the children and information about every child. The amount of first-level children in the tree they have for the current topology of routers is over two thousand. Previously, they used standard approaches of measuring the performance to deal with this problem. They found multiple connections opened to the database while querying it. This contributed to the total latency with a certain cost. Thus, they optimized their code and reduced the latency but it still stayed perceptible.

The trace analysis shown the observer *treeExpanded()* method to take a long time on average. The observer has been invoked 22 times taking on average 533 ms, every time within the range from 204 ms to 918 ms. In total, the profiler took 29 samples while being inside this method inclusively. Figure 8.1 shows the cumulative calling context tree for this observer. These results were surprising.

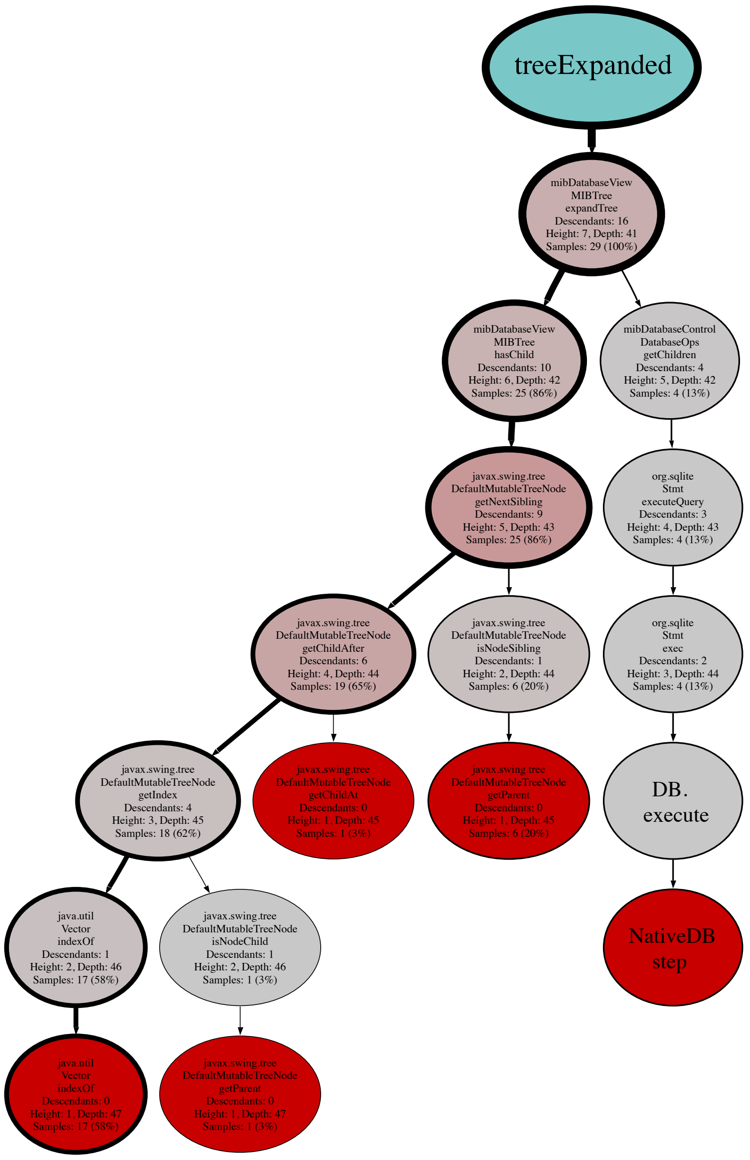

treeExpanded
mibDatabaseView MIBTree expandTree Descendants: 16 Height: 7, Depth: 41 Samples: 29 (100%)
mibDatabaseView MIBTree hasChild Descendants: 10 Height: 6, Depth: 42 Samples: 25 (86%)
mibDatabaseControl DatabaseOps getChildren Descendants: 4 Height: 5, Depth: 42 Samples: 4 (13%)
javax.swing.tree DefaultMutableTreeNode getNextSibling Descendants: 9 Height: 5, Depth: 43 Samples: 25 (86%)
org.sqlite Stmt executeQuery Descendants: 3 Height: 4, Depth: 43 Samples: 4 (13%)
javax.swing.tree DefaultMutableTreeNode getChildAfter Descendants: 6 Height: 4, Depth: 44 Samples: 19 (65%)
javax.swing.tree DefaultMutableTreeNode isNodeSibling Descendants: 1 Height: 2, Depth: 44 Samples: 6 (20%)
org.sqlite Stmt exec Descendants: 2 Height: 3, Depth: 44 Samples: 4 (13%)
javax.swing.tree DefaultMutableTreeNode getIndex Descendants: 4 Height: 3, Depth: 45 Samples: 18 (62%)
javax.swing.tree DefaultMutableTreeNode getChildAt Descendants: 0 Height: 1, Depth: 45 Samples: 1 (3%)
javax.swing.tree DefaultMutableTreeNode getParent Descendants: 0 Height: 1, Depth: 45 Samples: 6 (20%)
DB. execute
java.util Vector indexOf Descendants: 1 Height: 2, Depth: 46 Samples: 17 (58%)
javax.swing.tree DefaultMutableTreeNode isNodeChild Descendants: 1 Height: 2, Depth: 46 Samples: 1 (3%)
NativeDB step
java.util Vector indexOf Descendants: 0 Height: 1, Depth: 47 Samples: 17 (58%)
javax.swing.tree DefaultMutableTreeNode getParent Descendants: 0 Height: 1, Depth: 47 Samples: 1 (3%)

Figure 8.1. DB Query or Tree Expansion

They showed that the application spent only 13% in querying the database and approximately 60% while searching for the child nodes in the tree. Checking the source code, the developers identified that the indexing of nodes should be different, and confirmed this is the problem.

8.3.2 perfsonarUI

perfsonarUI is an infrastructure for network performance monitoring, making it easier to solve end-to-end performance problems on paths crossing networks. It is a project developed and maintained by the State Universities of Belgrade, Sofia and Zagreb. It contains nearly 2000 classes.

The remaining three developers participated in the experiment working on this project: one developer with 10 years of experience and two with 4 to 5 years of experience. They were running and using the application with our profiler for one hour. Each of them used one additional hour to look at the trace files, and tried to identify performance problems they perceived during run-time. Our profiler identified 154 distinctive landmarks from these 3 runtimes.

The developer with the most experience discovered a concrete place where the application hangs every time the action is triggered. Figure 8.2 shows the cumulative calling context tree for the *paintDirtyRegions()* method. It contains the *ChartPaint.paintComponent()* landmark in the calling context tree. This landmark appears in two different contexts. From the calling context tree, the application spent 33% and 65% in these two contexts. The landmark has been executed 18 times during the experiment. It took 1.8 sec to execute on average, having a minimum latency of 514 ms and a maximum of 10.2 sec.

Both contexts in which the landmark appeared have one heavyweight branch that goes through the *getHostName()* method that takes 20% and 44% of time respectively for each context. Almost half of the time this method internally spends comparing string values.

Looking at the source code, she confirmed that this landmark is invoked every time the table is filled with a certain router's data from the network. They fill the model of the table with data only when it should be redrawn. The same component is used in multiple places in the application. To retrieve data for each router in the table, both network activity and string comparison takes place. Referring to data from the traces, it may take up to 10 sec. We identified that this case happens when the table is filled with the list of the above 2000 routers. Instead of comparing strings, she reimplemented this method to look at the ID, which is an integer and much simpler to compare. This number also uniquely identifies every router, which made this change very simple to try in-place. We reran the application with our profiler and produced another trace triggering this action in the same contexts. The average went down to 0.8 sec, having a maximum of 6 sec of delay with above 2000 routers. The communication net-

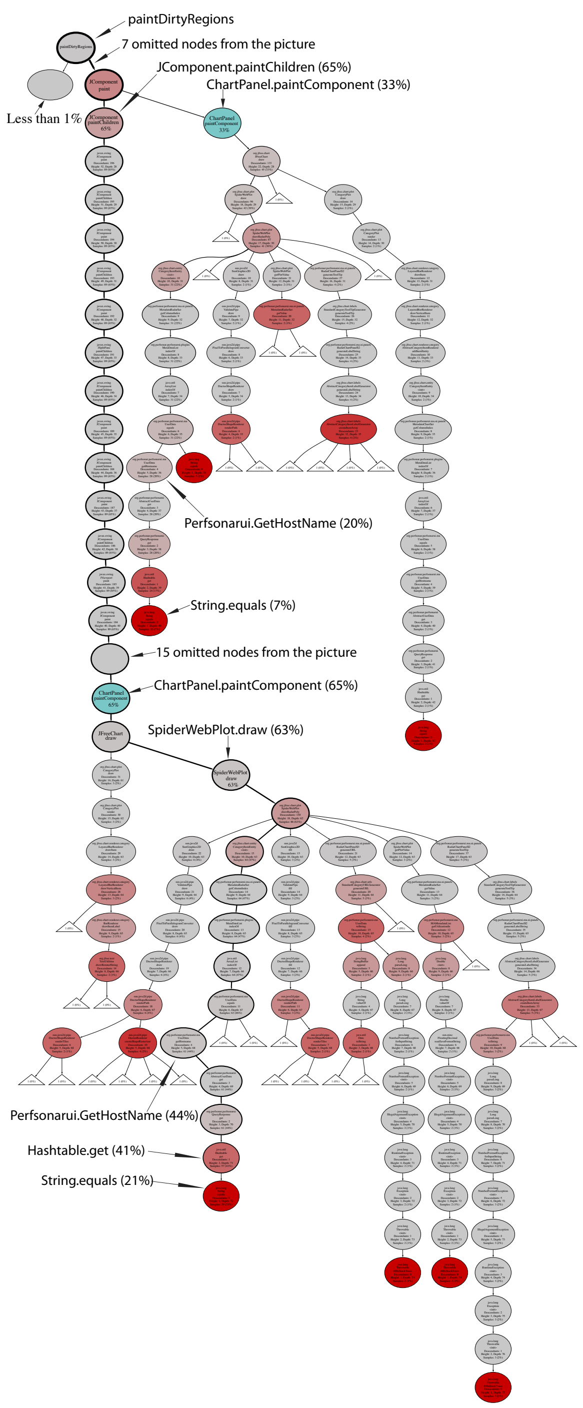

paintDirtyRegions
7 omitted nodes from the picture
JComponent.paintChildren (65%)
ChartPanel.paintComponent (33%)
Less than 1%
Perfsonarui.GetHostName (20%)
String.equals (7%)
15 omitted nodes from the picture
ChartPanel.paintComponent (65%)
SpiderWebPlot.draw (63%)
Perfsonarui.GetHostName (44%)
Hashtable.get (41%)
String.equals (21%)

Figure 8.2. PerfsonarUI

work delay they could not fix in-place as it required deeper changes in the code owned by the developers from another University. Nevertheless, the immediate idea was to cache locally these names to decrease the response time.

Part IV

Epilogue

This part contains three chapters. Chapter 9 discusses the applications of our approach in other fields. On the other hand, Chapter 10 discusses the limitations of the approach. Chapter 11 concludes this dissertation.

Chapter 9

Future Work

In this chapter we discuss the applications of our approach for the regression benchmarking and the plug-in blame assignment.

9.1 Regression Benchmarking

One additional interesting application of our approach is its use to automate regression benchmarking of interactive applications.

How does a development team of an interactive application know whether the current release improves or degrades performance over the prior version? Traditionally, organizations developing large software systems maintain automatic build and regression test infrastructures, and often they use these infrastructures to also assess performance by measuring the time needed to execute each of the regression test cases. Such a *regression benchmarking* [70] approach can also be used for interactive applications, at least for the isolated non-interactive components that are covered by regression tests. However, such measurements do not truly reflect interactive performance: they look at the speed of specific computations (whichever functionality a specific test case covers) in isolation. The perceptible performance of the overall application, however, depends on the latency of handling user requests, where each request may involve the notification of a diverse set of observers, and each observer may perform a diverse set of basic computations. To determine the perceptible performance, it is thus essential to move beyond unit tests as benchmarks. We propose to use the cumulative latency distributions over a set of test user sessions for this purpose.

To demonstrate this application, we have studied how the latency distribution has changed over different versions of Eclipse. Figure 5.3 in Section 5.6 has shown how Eclipse has grown by 71%, in terms of the number of classes, from version 3.0 to version 3.4.1. These new classes provide additional functionality. Most additional features need to observe some aspects of the application state, in order to provide their functionality. They thus probably register new observers, and those new observers contribute to additional latency. We wondered whether this effect was visible in the cumulative latency distribution.

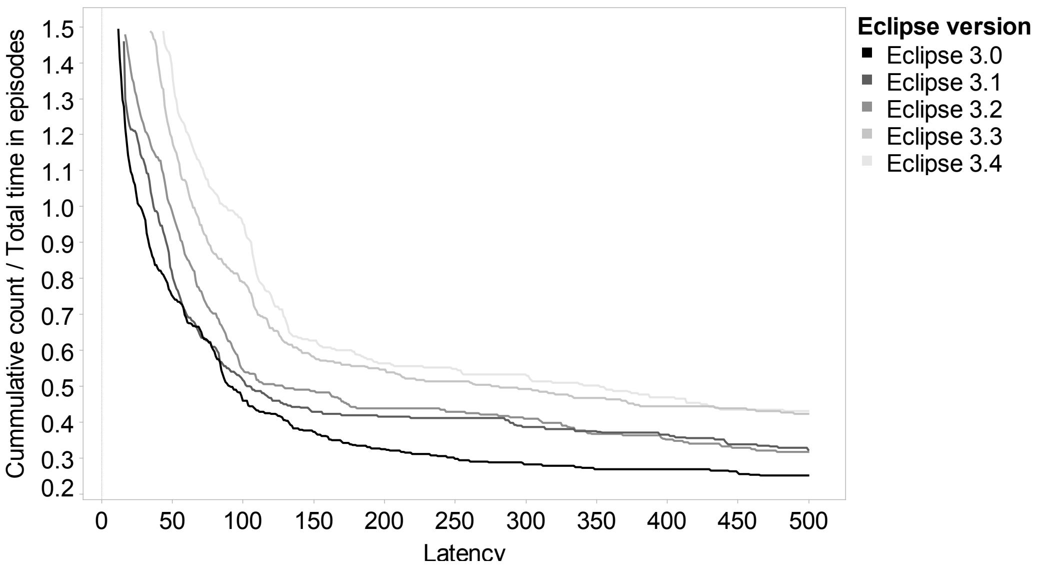


Figure 9.1. The Evolution of Eclipse's Cumulative Latency Distribution

Figure 9.1 shows that it is clearly visible. Each curve represents an aggregation of three interactive sessions with a given Eclipse version. The curves show a surprisingly clear decrease of the perceptible performance with each successive version. The most recent version, the top curve, clearly has the worst latency distribution, and the older the version, the better the performance. Moreover, the gaps between the curves show that versions 3.1 and 3.3 caused a bigger drop in performance, while versions 3.2 and 3.4 represent less pronounced performance regressions.

We can see that the curves show a significant number of perceptible latency episodes: version 3.4.1, for example, exhibits about one episode longer than 100 ms every second, and one episode longer than half a second every 0.45 seconds of in-episode time. Given that our profiler also collects – besides the cumulative latency distributions shown in the figure – information about each perceptible-latency observer notification, a developer will also receive information helping her to fix the performance regression. Our profile for version

3.4.1, for example, shows that a significant number of long-latency episodes is due to notifications of *org.eclipse.jdt.internal.ui.packageview.SelectionTransfer-DropAdapter.drop()*.

Our profiler enables the collection of these kinds of profiles in the field, for example as part of a beta testing program. Developers of interactive applications could combine our profiler with an approach to usage data collection [51] to ship the collected profiles from end users back to the developer's server. They could then use this information as part of their quality assurance process, and set specific goals for latency distributions of future versions.

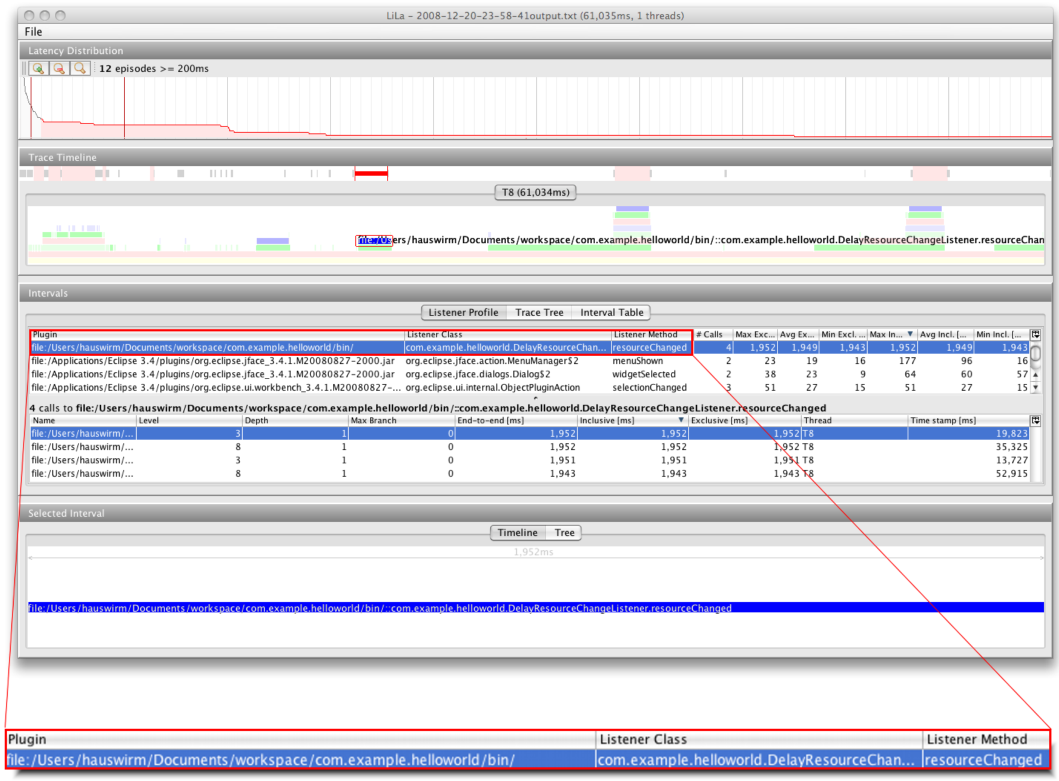
9.2 Plug-In Blame Assignment

The previous use case of latency profiling showed how developers can control performance throughout the evolution of their application. Applications like Eclipse, however, are not monolithic systems: they are frameworks that can be extended by plug-ins in a multitude of ways. Each end-user may install a different set of plug-ins, and the specific combination of plug-ins of a given installation may never have been performance-tested by a developer. Moreover, users tend to install additional plug-ins over time, so the set of plug-ins in a given installation may constantly change. For example we installed a plug-in to support the writing of LaTeX documents in Eclipse, and another set of plug-ins to access the Subversion repository of our group.

If an end-user perceives his specific installation of Eclipse as sluggish, it would be helpful for him to know what to do to fix the problem. If it was possible for an ordinary end-user to identify which of the many plug-ins was causing the perceptible slowdown, he could disable, uninstall, or upgrade that plug-in.

Our latency profiler can help in this situation. Often plug-ins register as observers to observe changes in application state. Thus, if a plug-in causes excessive latency, our profiler can identify the corresponding observer notification. We have extended our profiler to record, in addition to the observer's class and method, also the observer's class loader, and its class path. In a framework like Eclipse, each plug-in has its own class loader, and the class files of a given plug-in are stored in its own location (a JAR file or a directory). By collecting the class path, we can identify the plug-in containing the sluggish observer.

We have built a simple example of such a "rough" Eclipse plug-in to validate our idea. Our plug-in is based on Eclipse's Helloworld example plug-in. The only addition is that it observes changes in the Eclipse workspace (creation, modification, and deletion of resources) by registering *an org.eclipse.core.resources. IResourceChangeListener* with Eclipse's Workspace. We artificially introduced a delay in that observer, simulating the expensive computation a real plug-in might perform. We then ran Eclipse on top of our profiler. After we enabled our plug-in, all resource modifications became sluggish as expected. Note that the observed behavior would not tell a user where the problem lies. A very advanced user might suspect there to be a problem in the resource management plug-in, which, in fact, is not the case, as our plug-in is not responsible for resource management, it is just a presumably passive observer.


LiLa – 2008-12-20-23-58-41output.txt (61,035ms, 1 threads)
File
Latency Distribution
12 episodes >= 200ms
Trace Timeline
T8 (61,034ms)
file:/Users/hauswirm/Documents/workspace/com.example.helloworld/bin/::com.example.helloworld.DelayResourceChangeListener.resourceChan
Intervals
Listener Profile
Trace Tree
Interval Table

Plugin	Listener Class	Listener Method	# Calls	Max Exc...	Avg Ex...	Min Excl. ...	Max In...	Avg Incl. [...	Min Incl. [...
file:/Users/hauswirm/Documents/workspace/com.example.helloworld/bin/	com.example.helloworld.DelayResourceChan...	resourceChanged	4	1,952	1,949	1,943	1,952	1,949	1,943
file:/Applications/Eclipse 3.4/plugins/org.eclipse.jface_3.4.1.M20080827-2000.jar	org.eclipse.jface.action.MenuManager$2	menuShown	2	23	19	16	177	96	16
file:/Applications/Eclipse 3.4/plugins/org.eclipse.jface_3.4.1.M20080827-2000.jar	org.eclipse.jface.dialogs.Dialog$2	widgetSelected	2	38	23	9	64	60	57
file:/Applications/Eclipse 3.4/plugins/org.eclipse.ui.workbench_3.4.1.M20080827-...	org.eclipse.ui.internal.ObjectPluginAction	selectionChanged	3	51	27	15	51	27	15

4 calls to **file:/Users/hauswirm/Documents/workspace/com.example.helloworld/bin/::com.example.helloworld.DelayResourceChangeListener.resourceChanged**

Name	Level	Depth	Max Branch	End-to-end [ms]	Inclusive [ms]	Exclusive [ms]	Thread	Time stamp [ms]
file:/Users/hauswirm/...	3	1	0	1,952	1,952	1,952	T8	19,823
file:/Users/hauswirm/...	8	1	0	1,952	1,952	1,952	T8	35,325
file:/Users/hauswirm/...	3	1	0	1,951	1,951	1,951	T8	13,727
file:/Users/hauswirm/...	8	1	0	1,943	1,943	1,943	T8	52,915

Selected Interval
Timeline
Tree
1,952ms
file:/Users/hauswirm/Documents/workspace/com.example.helloworld/bin/::com.example.helloworld.DelayResourceChangeListener.resourceChanged

Plugin	Listener Class	Listener Method
file:/Users/hauswirm/Documents/workspace/com.example.helloworld/bin/	com.example.helloworld.DelayResourceChan...	resourceChanged

Figure 9.2. Identification of Responsible Plugin in our Listener Profile Visualizer

Figure 9.2 shows that our profile directly points out the observer, its class, and the plug-in that class belongs to. With an appropriate user interface, our profiler could pop up a window to notify the user whenever a observer took long, and propose which plug-in the user may want to disable or upgrade. We believe that such a feature would be helpful in many contexts, not just for Java applications like Eclipse. For example, modern desktop computers are running a multitude of “plug-ins”, from on-access virus scanners, over anti-spyware, to advertisement blockers in web browsers. In our own experience, some of these “plug-ins” indeed add perceptible latency. For a user it is often difficult or impossible to tell whom to blame when a computer becomes “sluggish”. A system based on our approach would directly assign the blame to the responsible component.

Chapter 10

Limitations

This chapter discusses the limitations of the profiling approach and the implementation presented in this dissertation.

User-relevant issues. Do the issues reported by our approach correspond to those performance problems users really care about? Our approach exploits the fact that there is a known threshold at which users start to perceive latency. This is a unique situation: we have a performance metric (event handling latency), *and* we have an absolute value for that metric (around 100 ms for discrete events), established by prior research [104; 103; 37; 41], at which performance is sufficient. However, it would be interesting to study, in the context of general interactive applications, whether users indeed would report perceptible latencies above the threshold as performance issues. One could do that by combining automatic lag hunting with the manual fly-by profiling approach [15], where users can click a button to immediately report a performance problem when it occurs.

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

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

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

Concurrency. LagHunter targets interactive applications. Prior work has shown that such applications rarely exhibit significant concurrent behavior [17]. However, our profiler does work with concurrent applications, and it tracks the behavior of all threads, so it is possible to see what other threads were doing while the user interface thread exhibited lag. Moreover, our lag hunting approach is not limited to interactive applications. For example, it could also be used to track lag in transaction-oriented server applications, where concurrency is much more prevalent. In that situation, it might be beneficial to extend LagHunter to track dependencies between threads. That way it could also highlight the reasons for why a thread blocked or waited.

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

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

However, to alleviate the above privacy concerns, we have developed a dialog that can automatically be shown to the user after a latency bug occurred. That dialog presents a screenshot and allows the user to black out arbitrary regions of the screen (e.g. to remove passwords or other private information that is visible). Moreover, we overlay the history of the mouse coordinates over that screenshot, and allow the user to navigate back in time in terms of mouse position, and thus to indicate when the bug occurred. Finally, we ask the user to point where (position/GUI component in the screenshot) the bug occurred. We did not use these features in our study, because we had not yet fully implemented them at that time. However, our hypothesis[1] is that they might help to gather useful information aiding developers in bug reproduction, while preserving user privacy.

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

Inaccuracy of stack sampling. Mytkowicz et al. [91] have shown that current Java stack sampling profilers are inaccurate due to the placement of safepoints by the virtual machine's JIT compilers. Our approach depends on the same infrastructure underlying the profilers studied in that paper, and we thus

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

are prone to the same inaccuracies in the calling context profiles collected for each issue. The fact that we collect samples in many different environments (different users, hardware, operating systems, virtual machines), over many different program runs, might mitigate part of the problem. Moreover, our landmark tracing approach is based on bytecode instrumentation, and thus the latency we report is not affected by that problem. Finally, once virtual machine implementers correct the problems pointed out by Mytkowicz et al., our sampled calling context profiles will automatically become more accurate.

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

Application selection. We evaluate LagHunter by catching latency bugs in only several interactive applications. This carries the risk that our approach might not be effective in finding bugs in other programs. However, Eclipse is an order of magnitude larger and richer than most other interactive Java programs. Moreover, it consists of a large number of independent plug-ins, and some of those plug-ins are more complex than many normal stand-alone applications. Eclipse also is the only Java application that uses two separate Java GUI toolkits: its default toolkit is SWT, but some Eclipse plugins use AWT/Swing. Besides Eclipse, LagHunter is used to catch performance bugs in a range of other applications, including application we discuss in the evaluation chapters.

Chapter 11

Conclusion

Many if not most computer systems are used by human users. The performance of such interactive systems ultimately affects users. What affects them is the perceptible lag in handling their requests. In general, the performance of interactive applications can be determined by the perceptible lag.

In this dissertation, we introduced a novel approach, landmark latency profiling, that helps developers dealing with such performance problems. The approach seeks out perceptible lags and investigates their causes. We showed how the approach helps developers to understand to improve the perceptible performance of their interactive applications.

The approach identifies methods relevant for handling user actions, which dramatically narrows the scope of the application code where a developer has to look for problems. The approach is designed to work in the field. The reason is that many problems occur only in some contexts and platforms which are impossible to reproduce in the lab. We succeed in providing an approach that gives an accurate view of the causes of performance problems and perturbs the interactive application insignificantly.

We evaluated our approach on real-world applications ourselves such as Informa, Eclipse, and CodeBubbles. Also, we evaluated the usefulness of our approach in the field by letting other developers use it and to try identify performance problems and their causes. Chapter 6 and Chapter 7 discuss how the approach helps identifying problems in the code. We showed in which cases the approach helps developers to recognize long latency problems and to navigate in their code to the places that contribute to long latencies. Nevertheless, we found limitations of the approach when the approach cannot help and we discussed that in Chapter 10.

In addition, our approach has a range of possible applications. We described how to use the approach for automatic software testing. We also showed how the approach can be used for regression benchmarking of interactive applications. Moreover, we believe that our approach could be successfully extended to the field of distributed systems where latency is a crucial parameter for handling the events.

This dissertation demonstrated that understanding the performance of modern complex systems requires more dedicated profiling approaches. We showed how our approach is more effective than the standard hotness profiling approaches in identifying performance problems for the context of interactive applications. In particular, our study showed that our approach can successfully highlight the region of code in which a performance problem occurs and identify the problem cause in many relevant situations. Finally, the experience from this work shows that having the approach deployed in the field, allows a developer to have insight into the performance problems which otherwise would be impossible to find.

Publication List

The work on this thesis resulted with a number of publications at the scientific journals, conferences, and workshops. Hereby, we provide the overview of the publications along the path toward the thesis:

- **The potential of speculative class-loading** [122], Dmitrijs Zaparanuks, Milan Jovic, and Matthias Hauswith

  Conference Proceedings at Principles and Practice of Programming in Java (PPPJ) 2007, Lisbon, Portugal

  The work presented in this paper required changes in the code of the research virtual machine JikesRVM. It helped me to acquire necessary understanding of the internal mechanisms of one virtual machine.

- **Measuring the Performance of Interactive Applications with Listener Latency Profiling** [67], Milan Jovic, and Matthias Hauswirth

  Conference Proceedings at Principles and Practice of Programming in Java (PPPJ) 2008, Modena, Italy

  This paper represents the inception of the approach presented in this dissertation. It mostly reflects the content of Chapter 3.

- **Accuracy of performance counter measurements** [123], Dmitrijs Zaparanuks, Milan Jovic, and Matthias Hauswirth

  Conference Proceedings at the International Symposium on Performance Analysis of Systems and Software (ISPASS) 2009, Boston, Massachusetts, USA

- **Listener Latency Profiling: Measuring the Perceptible Performance of Interactive Applications** [68], Milan Jovic, and Matthias Hauswirth
  Journal Science of Computer Programming, 2010

  The paper "Measuring the Performance of Interactive Applications with Listener Latency Profiling" [67] presented at PPPJ was selected to be published in the Science of Computer Programming Journal. This is an extended version of the PPPJ paper. It includes two additional parts. The first one contains novel results we got including new landmark types, and the second is the extended evaluation with NetBeans.

- **LagAlyzer: A latency profile analysis and visualization tool** [17], Andrea Adamoli, Milan Jovic, and Matthias Hauswirth
  Conference Proceedings at the International Symposium on Performance Analysis of Systems and Software (ISPASS) 2010, White Plains, NewYork, USA

  In this paper we presented the first study that gives insight into why interactive Java applications are sometimes perceived as slow. Our contribution of the work presented in this paper is shown in Chapter 4.

- **Performance Testing of GUI Applications** [69], Milan Jovic and Matthias Hauswirth
  International Workshop on TESTing Techniques and Experimentation Benchmarks for Event-Driven Software (TESTBEDS) 2010, Paris, France

  In this paper we broaden the notion of GUI testing to include GUI performance testing using our approach. We outline the challenges specific to performance testing of GUI applications. We proposed a research agenda towards an automatic approach that would make GUI performance testing practical.

- **Automating Performance Testing of Interactive Java Applications** [66], Milan Jovic, Andrea Adamoli, Dmitrijs Zaparanuks, Matthias Hauswirth
  Proceedings of the Workshop on Automation of Software Test (AST) at ICSE 2010, Cape Town, South Africa

  Along the way toward our dissertation we found other areas were our approach can be applied. Automated software testing as one of the most interesting reflected in this publication. The idea of automating software testing with our approach is presented in Section 9.1 when speaking about regression test benchmarking of interactive applications.

- **Automating Performance Testing of Interactive Java Applications**, Andrea Adamoli, Dmitrijs Zaparanuks, Milan Jovic, Matthias Hauswirth
  Software Quality Journal

  The paper "Automating Performance Testing of Interactive Java Applications" presented at AST 2010 was selected for the Software Quality Journal and this publication is the extended version of the paper.

- **Catch Me If You Can: Performance Bug Detection in the Wild**, Milan Jovic, Andrea Adamoli, Matthias Hauswirth
  In Proceedings of the Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) 2011 conference, Portland, Oregon, USA

  The paper represents the final approach of our thesis and its content spans over two chapters: Chapter 5 and Chapter 6.

Bibliography

[1] BTrace, Jaroslav Bachorik, https://btrace.dev.java.net/.

[2] DJProf, Java Profiling with AspectJ, http://www.mcs.vuw.ac.nz/ djp/d-jprof/.

[3] EJP, Sebastien Vauclair, Extensible Java Profiler, http://ejp.source-forge.net/.

[4] JMemProf, Michael Clark, Java Memory Profiler, http://oss.meta-paradigm.com/jmemprof/.

[5] JProfiler, Java profiler, http://www.ej-technologies.com/products-/jprofiler/overview.html.

[6] JRat, Java Runtime Analysis Toolkit, http://jrat.sourceforge.net/.

[7] NetBeans Profiler, Sun Microsystems, http://netbeans.org/community-/magazine/html/04/profiler.html.

[8] Profiler4J, Antonio S. R. Gomes, Profiler for Java, http://profiler4j-.sourceforge.net/.

[9] Shark, performance tool, Apple Computer, http://developer.apple.com-/performance.

[10] Video Guidelines, UIST Hints for Making Good Videos, http://www.acm-.org/uist/uist2009/call/videoguide.html.

[11] Visual VM, Sun Microsystems, https://visualvm.dev-.java.net/.

[12] JVM TI, JVM Tool Interface, Sun Microsystems, http://java.sun.com-/j2se/1.5.0/docs/guide/jvmti/jvmti.html, 2004.

[13] Kelly O'Hair, A Heap/CPU Profiling Tool in J2SE 5.0, Sun Microsystems, java.sun.com, 2004.

[14] TPTP, Eclipse TPTP Profiler, http://www.eclipse.org/articles/Article-TPTP-ProfilingTool/tptpProfiling-Article.html, 2006, 2006.

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

[16] Andrea Adamoli, Milan Jovic, and Matthias Hauswirth. Lagalyzer: A latency profile analysis and visualization tool. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 13 –22, 28-30 2010.

[17] Andrea Adamoli, Milan Jovic, and Matthias Hauswirth. Lagalyzer: A latency profile analysis and visualization tool. In *ISPASS*, pages 13–22, 2010.

[18] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 361–370, New York, NY, USA, 2006. ACM.

[19] Taweesup Apiwattanapong and Mary Jean Harrold. Selective path profiling. In *PASTE '02: Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 35–42, New York, NY, USA, 2002. ACM.

[20] Matthew Arnold and David Grove. Collecting and exploiting high-accuracy call graph profiles in virtual machines. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 51–62, Washington, DC, USA, 2005. IEEE Computer Society.

[21] Aleksandar M. Bakic, Matt W. Mutka, and Diane T. Rover. Brisk: A portable and flexible distributed instrumentation system. *Parallel Processing Symposium, International*, 0:387, 1999.

[22] Thomas Ball and James R. Larus. Efficient path profiling. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[23] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. Magpie: online modelling and performance-aware systems. In *HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems*, pages 15–15, Berkeley, CA, USA, 2003. USENIX Association.

[24] Victor R. Basili. The role of experimentation in software engineering: past, current, and future. In *Proceedings of the 18th international conference on Software engineering*, ICSE '96, page 442–449, Washington, DC, USA, 1996. IEEE Computer Society, IEEE Computer Society.

[25] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiß, Rahul Premraj, and Thomas Zimmermann. Quality of bug reports in eclipse. In *eclipse '07: Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, pages 21–25, New York, NY, USA, 2007. ACM.

[26] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 308–318, New York, NY, USA, 2008. ACM.

[27] Walter Binder. Portable, efficient, and accurate sampling profiling for java-based middleware. In *SEM '05: Proceedings of the 5th international workshop on Software engineering and middleware*, pages 46–53, New York, NY, USA, 2005. ACM.

[28] Walter Binder, Jarle Hulaas, and Philippe Moret. Advanced java bytecode instrumentation. In *PPPJ '07: Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 135–144, New York, NY, USA, 2007. ACM.

[29] Michael D. Bond and Kathryn S. McKinley. Practical path profiling for dynamic optimizers. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 205–216, Washington, DC, USA, 2005. IEEE Computer Society.

[30] Michael D. Bond and Kathryn S. McKinley. Probabilistic calling context. *SIGPLAN*, 42(10):97–112, 2007.

[31] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. Laviola. Code bubbles: Rethinking the user interface

paradigm of integrated development environments. In *In Proc. ICSE*, 2010.

[32] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *In Proceedings of Supercomputing*, 2000.

[33] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[34] Gerardo Canfora and Luigi Cerulo. Fine grained indexing of software repositories to support impact analysis. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 105–111, New York, NY, USA, 2006. ACM.

[35] Gerardo Canfora and Luigi Cerulo. Supporting change request assignment in open source development. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1767–1772, New York, NY, USA, 2006. ACM.

[36] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the USENIX 2004 Annual Technical Conference*, pages 15–Ű28. USENIX, 2004.

[37] 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.

[38] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 595–604, Washington, DC, USA, 2002. IEEE Computer Society.

[39] Code Caching with BEA JRockit. http://e-docs.bea.com/wljrockit/-docs142/userguide/codecach.html.

[40] Gerry Coleman and Rory O'Connor. Using grounded theory to understand software process improvement: A study of irish software product companies. *Inf. Softw. Technol.*, 49:654–667, June 2007.

[41] James R. Dabrowski and Ethan V. Munson. Is 100 milliseconds too fast? In *CHI '01 extended abstracts on Human factors in computing systems*, pages 317–318, New York, NY, USA, 2001. ACM.

[42] Markus Dahm. Byte code engineering. In *Java-Informations-Tage*, pages 267–277. Springer-Verlag, 1999.

[43] Mikhail Dmitriev. Design of jfluid: a profiling technology and tool based on dynamic bytecode instrumentation. Technical report, Mountain View, CA, USA, 2003.

[44] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: less is more. *SIGPLAN Not.*, 35(11):202–211, 2000.

[45] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. Dynamic metrics for java. *SIGPLAN Not.*, 38(11):149–168, 2003.

[46] Antonio Espinosa, Tomás Margalef, and Emilio Luque. Automatic detection of parallel program performance problems. In *SPDT '98: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, page 149, New York, NY, USA, 1998. ACM.

[47] T. Fahringer, R. Prodan, Rubing Duan, F. Nerieri, S. Podlipnig, Jun Qin, M. Siddiqui, Hong-Linh Truong, A. Villazon, and M. Wieczorek. Askalon: A grid application development and computing environment. In *GRID '05: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pages 122–131, Washington, DC, USA, 2005. IEEE Computer Society.

[48] Michael Fischer, Martin Pinzger, and Harald Gall. Analyzing and relating bug report data for feature tracking. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, page 90, Washington, DC, USA, 2003. IEEE Computer Society.

[49] KrisztiĞn Flautner, Rich Uhlig, Steve Reinhardt, and Trevor Mudge. Thread-level parallelism and interactive performance of desktop applications. In *Proceedings of Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX*, 2000.

[50] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *In NSDI*, 2007.

[51] Eclipse Foundation. Eclipse Usage Data Collector. http://www.eclipse-.org/epp/usagedata, 2008.

[52] M. D. Penta G. Antoniol, H. Gall and M. Pinzger. Mozilla: Closing the circle. technical report, 2004.

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

[54] M. Gerndt. Automatic performance analysis tools for the grid: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(2-4):99–115, 2005.

[55] C. B. Gibbs. Controller design:interaction of controlling limbs, time-lags, and gains in positional and velocity systems. *Ergonomics*, 1962.

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

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

[58] Jan L. Guynes. Impact of system response time on state anxiety. *Commun. ACM*, 31(3):342–347, 1988.

[59] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, New York, NY, USA, 2002. ACM.

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

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

[62] Joseph L. Hellerstein, Mark M. Maccabee, W. Nathaniel Mills Iii, and John J. Turek. Ete: A customizable approach to measuring end-to-end response times and their components in distributed systems. In *ICDCS '99: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, page 152, Washington, DC, USA, 1999. IEEE Computer Society.

[63] Rashina Hoda, James Noble, and Stuart Marshall. Using grounded theory to study the human aspects of software engineering. In *Human Aspects of Software Engineering*, HAoSE '10, pages 5:1–5:2, New York, NY, USA, 2010. ACM.

[64] IBM alphaWorks: Jikes Bytecode Toolkit. http://www.alphaworks.ibm-.com/tech/jikesbt, March 2000.

[65] Shrinivas Joshi. A technique and tool for selective capture and replay of program executions, software maintenance, 2007. icsm, 2007.

[66] Milan Jovic, Andrea Adamoli, Dmitrijs Zaparanuks, and Matthias Hauswirth. Automating performance testing of interactive java applications. In *AST '10: Proceedings of the 5th Workshop on Automation of Software Test*, pages 8–15, New York, NY, USA, 2010. ACM.

[67] Milan Jovic and Matthias 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.

[68] Milan Jovic and Matthias Hauswirth. Listener latency profiling: Measuring the perceptible performance of interactive java applications. *Science of Computer Programming*, 2010.

[69] Milan Jovic and Matthias Hauswirth. Performance testing of gui applications. In *TESTBEDS '10: Second International Workshop on TESTing Techniques Experimentation Benchmarks for Event-Driven Software*, 2010.

[70] Tomas Kalibera, Jakub Lehotsky, David Majda, Branislav Repcek, Michal Tomcanyi, Antonin Tomecek, Petr Tuma, and Jaroslav Urban. Automated benchmarking and analysis tool. In *valuetools '06: Proceedings of the 1st international conference on Performance evaluation methodolgies and tools*, page 5, New York, NY, USA, 2006. ACM.

[71] Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 1–11, New York, NY, USA, 2006. ACM.

[72] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. pages 327–353. Springer-Verlag, 2001.

[73] Killian. MIPS computer systems. In *inc. MIPS language programmer's guide*, 1986.

[74] Barbara Kitchenham, Hiyam Al-Khilidar, Muhammad Ali Babar, Mike Berry, Karl Cox, Jacky Keung, Felicia Kurniawati, Mark Staples, He Zhang, and Liming Zhu. Evaluating guidelines for empirical software engineering studies. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, ISESE '06, pages 38–47, New York, NY, USA, 2006. ACM.

[75] Barbara Kitchenham and Shari Lawrence Pfleeger. Principles of survey research part 6: data analysis. *SIGSOFT Softw. Eng. Notes*, 28:24–27, March 2003.

[76] Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Softw. Eng.*, 28(8):721–734, 08/2002 2002.

[77] Andrew J. Ko, Brad A. Myers, and Duen Horng Chau. A linguistic analysis of how people describe software problems. In *VLHCC '06: Proceedings of the Visual Languages and Human-Centric Computing*, pages 127–134, Washington, DC, USA, 2006. IEEE Computer Society.

[78] Han Bok Lee and Benjamin G. Zorn. Bit: A tool for instrumenting java bytecodes. In *In The USENIX Symposium on Internet Technologies and Systems*, pages 73–82. USENIX Association, 1997.

[79] Ben Liblit and Alex Aiken. Distributed program sampling. In *In Proceedings of PLDI'03*, pages 141–154, 2003.

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

[81] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 15–26, New York, NY, USA, 2005. ACM.

[82] 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.

[83] George William Jr. McAdoo. The effects of success, mild failure, and strong failure feedback on a-state for subjects who differ in a-trait, unpublished doctoral dissertation, florida state university, 1970.

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

[85] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tool. *Computer*, 28(11):37–46, 1995.

[86] Cary Millsap. Thinking clearly about performance. *Queue*, 8(9):10–20, 2010.

[87] Philippe Moret, Walter Binder, and Éric Tanter. Polymorphic bytecode instrumentation. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 129–140, New York, NY, USA, 2011. ACM.

[88] Philippe Moret, Walter Binder, Alex VillazŮn, Danilo Ansaloni, and Abbas Heydarnoori. Visualizing and exploring profiles with calling context ring charts. *Software: Practice and Experience*, 40(9):825–847, 2010.

[89] Tipp Moseley, Alex Shye, Vijay Janapa Reddi, Dirk Grunwald, and Ramesh Peri. Shadow profiling: Hiding instrumentation costs with parallelism. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 198–208, Washington, DC, USA, 2007. IEEE Computer Society.

[90] Todd Mytkowicz, Devin Coughlin, and Amer Diwan. Inferred call path profiling. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 175–190, New York, NY, USA, 2009. ACM.

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

[92] Tia Newhall and Barton P. Miller. Performance measurement of dynamically compiled java executions. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 42–50, New York, NY, USA, 1999. ACM.

[93] Alessandro Orso, Shrinivas Joshi, Martin Burger, and Andreas Zeller. Isolating relevant component interactions with jinsi. In *WODA '06: Proceedings of the 2006 international workshop on Dynamic systems analysis*, pages 3–10, New York, NY, USA, 2006. ACM.

[94] David J. Pearce, Matthew Webster, Robert Berry, and Paul H. J. Kelly. Profiling with aspectj. *Softw. Pract. Exper.*, 37(7):747–777, 2007.

[95] PerfCtr, Performance Counters, Hardware Performance Monitor tool, http://user.it.uu.se/ mikpe/linux/perfctr/.

[96] Perfmon2, Hardware Performance Monitor tool, http://perfmon2-.sourceforge.net/.

[97] V. J. Reddi, D. Connors, R. Cohn, and M. D. Smith. Persistent Code Caching: Exploiting Code Reuse Across Executions and Applications. In *Proceedings of the international symposium on Code Generation and Optimization*, 2007.

[98] Steven P. Reiss. Visualizing the java heap. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 251–254, New York, NY, USA, 2010. ACM.

[99] Luiz De Rose. The hardware performance monitor toolkit. In *Euro-Par '01: Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, pages 122–131, London, UK, 2001. Springer-Verlag.

[100] Subhajit Roy and Y. N. Srikant. Profiling k-iteration paths: A generalization of the ball-larus profiling algorithm. In *CGO '09: Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 70–80, Washington, DC, USA, 2009. IEEE Computer Society.

[101] Per Runeson, Magnus Alexandersson, and Oskar Nyholm. Detection of duplicate defect reports using natural language processing. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 499–510, Washington, DC, USA, 2007. IEEE Computer Society.

[102] T. B. Sheridan and W. R. Ferrel. Remote manipulative control with transmission delay. *Transactions on Human Factors in Electronics*, 1968.

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

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

[105] Benjamin H. Sigelman, Luiz AndrŐ Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Google Technical Report dapper-2010-1, April 2010.

[106] Robert Snelick. S-check: a tool for tuning parallel programs. *Parallel Processing Symposium, International*, 0:107, 1997.

[107] R. Spezialetti, M. Gupta. Exploiting program semantics for efficient instrumentation of distributed event recognitions. *Reliable Distributed Systems*, pages 181–190, 1994.

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

[109] Amitabh Srivastava and Alan Eustace. Atom: a system for building customized program analysis tools. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 196–205, New York, NY, USA, 1994. ACM.

[110] Ariel Tamches and Barton P. Miller. Using dynamic kernel instrumentation for kernel and application tuning. *Int. J. High Perform. Comput. Appl.*, 13(3):263–276, 1999.

[111] Hong-Linh Truong, Thomas Fahringer, Georg Madsen, Allen D. Malony, Hans Moritsch, and Sameer Shende. On using scalea for performance analysis of distributed and parallel programs. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 34–34, New York, NY, USA, 2001. ACM.

[112] Jon A. Turner. Computer mediated work: the interplay between technology and structured jobs. *Commun. ACM*, 27(12):1210–1217, 1984.

[113] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[114] Kapil Vaswani, Aditya V. Nori, and Trishul M. Chilimbi. Preferential path profiling: compactly numbering interesting paths. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 351–362, New York, NY, USA, 2007. ACM.

[115] Alex Villazón, Walter Binder, and Philippe Moret. Aspect weaving in standard java class libraries. In *PPPJ '08: Proceedings of the 6th international symposium on Principles and practice of programming in Java*, pages 159–167, New York, NY, USA, 2008. ACM.

[116] Intel VTune performance analyzers. http://www.intel.com/software/-products/vtune, 2003.

[117] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 1, Washington, DC, USA, 2007. IEEE Computer Society.

[118] Roel Wieringa, Hans Heerkens, and Björn Regnell. How to write and read a scientific evaluation paper. In *Proceedings of the 2009 17th IEEE International Requirements Engineering Conference, RE*, RE '09, page 361–364, Washington, DC, USA, 2009. IEEE Computer Society, IEEE Computer Society.

[119] Guoqing Xu, Atanas Rountev, Yan Tang, and Feng Qin. Efficient checkpointing of java software using context-sensitive capture and replay. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 85–94, New York, NY, USA, 2007. ACM.

[120] Zhichen Xu, Barton P. Miller, and Oscar Naim. Dynamic instrumentation of threaded applications. In *PPoPP '99: Proceedings of the seventh ACM*

SIGPLAN symposium on Principles and practice of parallel programming, pages 49–59, New York, NY, USA, 1999. ACM.

[121] D. Zaparanuks and M. Hauswirth. Characterizing the design and performance of interactive java applications. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 23 –32, 28-30 2010.

[122] Dmitrijs Zaparanuks, Milan Jovic, and Matthias Hauswirth. The potential of speculative class-loading. In *PPPJ*, pages 209–214, 2007.

[123] Dmitrijs Zaparanuks, Milan Jovic, and Matthias Hauswirth. Accuracy of performance counter measurements. In *ISPASS*, pages 23–32, 2009.

[124] Xiaotong Zhuang, Mauricio J. Serrano, Harold W. Cain, and Jong-Deok Choi. Accurate, efficient, and adaptive calling context profiling. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 263–271, New York, NY, USA, 2006. ACM.