Profiling Android Applications with Nanoscope

Lun Liu
University of California
Los Angeles, USA

Leland Takamine
Uber Technologies
San Francisco, USA

Adam Welc
Uber Technologies
San Francisco, USA

Abstract

User-level tooling support for profiling Java applications executing on modern JVMs for desktop and server is quite mature – from OpenJDK's Java Flight Recorder [5] enabling low-overhead CPU and heap profiling, through third-party async profilers (e.g. async-profiler [7], honest-profiler [11]), to OpenJDK's support for low-overhead tracking of allocation call sites [1].

On the other hand, despite architectural similarities between language execution environments, tooling support for analyzing performance of Java code on Android lags significantly behind. Arguably the most popular, but also virtually the only tool in this category is Android Profiler [3] – it provides process-level information about CPU, Java heap, and network activities. It also provides support for method tracing and sampling. However, the former exhibits high overhead and the latter, while providing better performance, sacrifices precision. Information provided by Android Profiler can be augmented with data collected by other tools, particularly systrace [4] which collects data at the operating system level. Unsurprisingly, third-party tools have also started emerging recently, such as Facebook's Profilo [6] framework. However, using these additional tools requires working with multiple tool-chains, may require a certain amount of manual effort (e.g., building application in a special way, inserting profiling framework API calls into the application), and it is often non-trivial to infer interesting information from the data being collected (e.g. because only part of the tooling framework is publicly available).

In this paper we describe our work on Nanoscope, a single open source, low-overhead, and extensible tool that not only works on unmodified Android applications and provides precise method traces with low overhead, but also aims to present programmers with selected relevant information within the same framework to help them develop an intuition behind a concrete non-performant application behavior. The tradeoff here is that Nanoscope requires a custom Android build as it relies on additional support from ART. In this paper, we will describe Nanoscope's design and implementation, present examples of performance-related information that can be obtained from the tool, and discuss Nanoscope-related overheads.

VMIL'18, November 04, 2018, Boston, MA, USA
2018.

1 Introduction

Compared to "regular" Java programs (i.e., executing on a desktop and on a server), tooling for Android programs (i.e., executing on mobile devices) lags significantly behind. Arguably, the most popular analysis tool for Android to locally identify performance problems comes from Google itself in the form of Android Profiler [3], providing information about an executing program and its threads, such as per-process heap memory and CPU usage, network traffic, and per-thread execution data in the form of method traces. In our experience, for a reasonably tuned applications, the most interesting piece of information provided by Android Profiler are method traces, as such applications do not exhibit sudden or large spikes in object allocation rates or per-process CPU usage. Android Profiler collects these traces either by instrumenting all application Java methods or by sampling all application threads - the former sacrifices performance (as not only all methods of all threads are traced, but tracing requires inter-thread coordination) and the latter sacrifices precision (as not every method execution is recorded). Other tool-chains (e.g., systrace [4]) exist to augment information provided by Android Profiler, but they either cannot provide VM-level information (e.g., lock contention) or they cannot operate on unmodified production application files (e.g. require special compilation process or in-application API calls), and in general cannot provide uniform user experience within a single framework. One exception to this rule is the Profilo [6] framework developed by Facebook, but this toolkit has been only partially open-sourced (e.g., without visualization tools) and is also geared towards aggregating performance-related information from applications running in production rather than providing the ability to interactively deep dive into specific performance problems of a given single application execution.

In order to overcome limitations of the existing Android performance-focused tools, we started developing our own framework called Nanoscope. In designing and implementing Nanoscope we were guided by the following principles:

- as the starting point, we would like to get similar types of information to that provided by Android Profiler with precise execution traces, but we are

not willing to pay the price of using Android Profiler tracing as it can slow down execution of our applications by an order of magnitude
- we would like to overlay additional important metrics at the OS level (e.g., CPU utilization) and at the VM level (e.g. lock contention) directly on the method execution traces
- we would like to be able to analyze performance of unmodified production applications
- when analyzing performance of our mobile applications we are mostly interested in the behavior of the main thread

In the following section we describe how these principles have been reflected in the design and implementation of Nanoscope. Please note that this is very much work in progress!

2 Design and implementation

The focus of the first open source release of "basic" Nanoscope [9, 10] was to provide precise method tracing similar to that offered by Android Profiler but at a much lower cost. The goal of the second "extended" version of Nanoscope[1] is to overlay additional information on method execution traces. Currently, Nanoscope is only supported in Android 7.1.2 as it requires modifications to Android Runtime [2] (ART) which are specific to a given OS version.

2.1 "Basic" Nanoscope

Currently Nanoscope focuses on analyzing a single thread (e.g., the main thread of the application) – it collects method tracing data into a thread-local data structure, where it records method entry end exit events along with their timestamps. It is therefore straightforward to extend it to analyze multiple threads – traces for multiple threads can be aligned in the UI using their timestamps in post-processing. We report overheads of collecting traces for the main thread (which in our application is the most active thread in terms of Java code execution) in Section 3 – evaluation of method trace collection for multiple threads is left for future work.

In order to implement method tracing in Nanoscope we needed to modify both the interpreter and the optimizing compiler so that the code to record a timestamp and a method name in an in-memory buffer is injected for each method prologue and epilogue – the buffer is flushed to persistent storage once tracing is done. Tracing can be enabled/disabled via system properties (no application modification required) or via in-application API calls (for increased flexibility in controlling a tracing span). Method traces for a given thread are visualized as a flame graph in a custom browser-based UI written in JavaScript that needed to be implemented to efficiently handle large volumes of tracing data. In addition to "regular" Java methods, "basic" Nanoscope also traces class loading and initialization time, which are represented as "virtual"[2] methods. A more detailed description of "basic" Nanoscope can be found in Nanoscope blog [8].

[1] Also open-sourced recently into the same Nanoscope repositories

2.2 "Extended" Nanoscope

The idea for extending Nanoscope stemmed from the observation that analyzing flame graphs themselves is not the most intuitive method of gaining insight into a given thread's execution anomalies. Even if certain methods take a long time to execute, it is not always immediately clear why that is, particularly if the methods do not contain obvious "hot spots" (e.g., hot loops). Consequently, we decided to extend Nanoscope to overlay additional metrics on the flame graph to help analyze different aspects of thread's execution at the same time. In particular, "extended" Nanoscope supports a sampling framework described in Section 2.2.1 where various interesting per-thread events (e.g. CPU utilization) can be periodically recorded for further analysis – the framework has been implemented in such a way that it is relatively easy to both add additional metrics to be tracked (as all of them are collected within the same function called at the time of collecting a sample) and to visualize them in our UI (as each one is visualized on a separate canvas whose implementation can be easily derived from the existing ones). We have also made certain additional "surgical" modifications to ART to track other interesting events in the thread's lifetime, such as thread state transitions that describe when a thread is blocked waiting for a lock, sleeping, performing GC-related activities, etc.

2.2.1 Sampling framework

Design and implementation of Nanoscope's sampling framework has been inspired by the profilers implemented for "regular" (server or desktop) Java applications, such as async-profiler [7] or honest-profiler [11]. The main idea is to use a system call to schedule a periodic signal generation by the OS that will be delivered to a given process, or in our case, a given thread. In terms of implementation, this has been achieved by modifying relevant parts of ART.

There are multiple ways to have OS to periodically send a signal to a given thread. One method is to use the `perf_event_open` system call – an apparent risk of using this method is that a thread receiving a signal can be interrupted during another system call which could

[2] In the sense of methods that do not exist in reality rather then in the sense of virtual method dispatch.

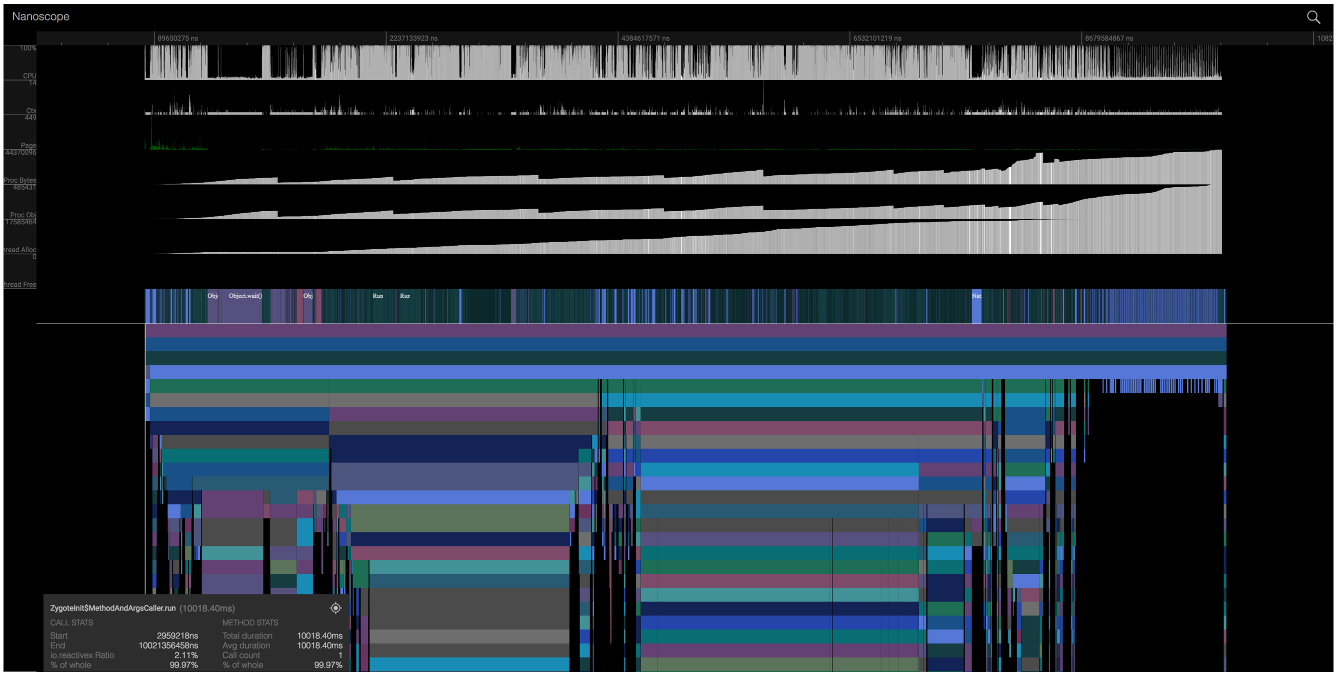

Figure 1. Nanoscope UI window

result in an application crash, but we have never experienced this situation in practice. Another method is to use `timer_settime` system call with a clock measuring a given thread's CPU time, which is supposed to only interrupt a thread in user-level code, but in our experiments we could not achieve desired fidelity of signal delivery (at least every 1ms) using this method. Consequently, we decided to support both methods in Nanoscope – a method is selected when tracing is initiated, defaulting to using `perf_event_open` system call.

Currently, the most important (in terms of insights it provided) metric collected in the sampling framework is CPU utilization - it is calculated by obtaining wall clock time (same mechanism as used for timestamps already collected by Nanoscope for method tracing) and CPU time (via `clock_gettime`), and calculating the ratio of the two. However, we also experimented with other metrics, such as the number of page faults (major and minor) and the number of context switches per sampling period (both collected by reading counters set up by the `perf_event_open` system call and subsequently available in a signal handler), as well as allocation rates per sampling period (collected by enabling ART *RuntimeStats* tracking and reading from it at every sample). We demonstrate how these metrics have been visualized in Section 2.3.

2.3 Nanoscope in action

Currently, Nanoscope focuses on analyzing performance of a single (in most cases main) thread, and our visualization tool reflects that. In Figure 1 we present a view of the entire Nanoscope UI window showing execution trace of the main thread in one of our applications. This particular execution represents a startup sequence of the application – we started the application and let it execute for ∼10s (on Nexus 6p running Nanoscope-enabled ROM based on Android 7.1.2). At the bottom of the figure is the flame graph [3] representing execution of the main thread, and at the top there are charts for various additional execution metrics (from top to bottom):

- CPU utilization
- number of context switches
- number of page faults (minor in green, major in red)
- bytes allocated by the whole process (minus freed ones)
- object allocated by the whole process (minus freed ones)
- total bytes allocated by the thread
- total bytes freed by the thread (none in this case)
- current thread state (e.g. executing native method, blocked on a lock, waiting on a condition variable, sleeping, etc.)

[3] We omitted names of methods in the flame graph for confidentiality reasons

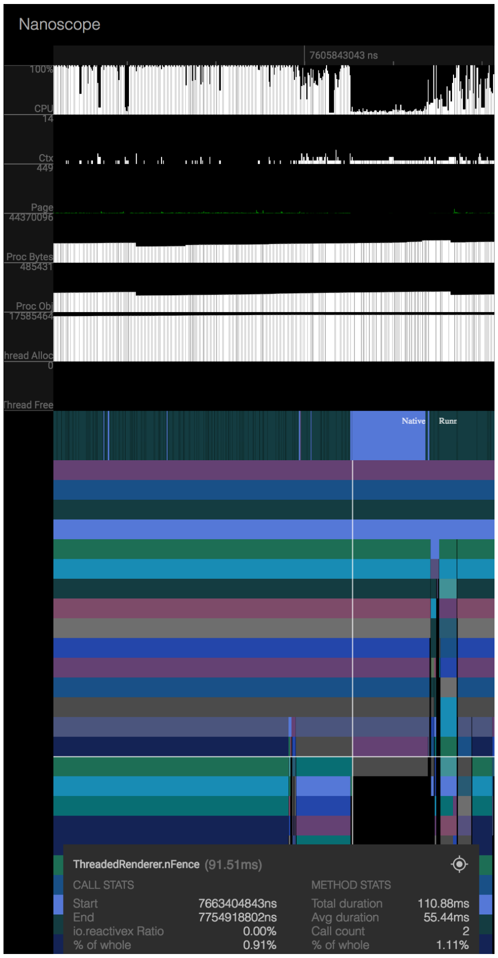

Figure 2. Renderer thread's `nFence` method

We currently display all metrics that are being collected, but in the near future we plan to make Nanoscope more configurable – have the user choose which metrics are of interest and visualize only these metrics. At the bottom left corner there is also a window containing detailed information about the method currently selected in the flame graph (in this case the main method of this thread, represented by top-most frame in the flame graph), such as duration of the selected method call, average duration of this method'a execution across all its invocations, number of times this method has been called, etc.

At this stage of the project we have not yet taken any actions on the data generated by Nanoscope, but it is evident that certain, potentially useful, observations would be difficult to make without the kind of information it provides.

For example, focusing on the part of the execution trace representing the end of the startup sequence when the application is largely idle waiting for user input (right-hand side of Figure 1) reveals that while the main thread no longer allocates any memory (as expected), the overall number of objects in the application process is still growing. It does not necessarily indicate existence of a real problem, but considering that the main thread is the one expected to execute a large portion of application's Java code, it may be worth investigating increase in object allocations on background threads.

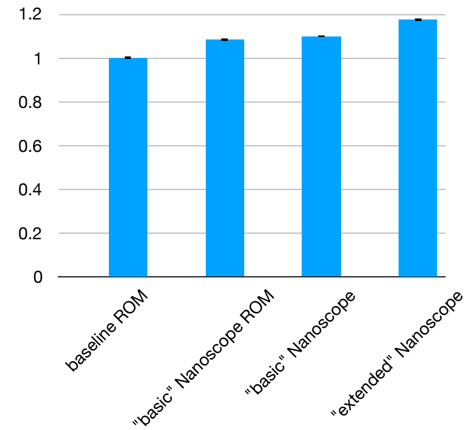

Figure 3. Nanoscope overheads

Another example is unusually long (almost 100ms) period of low CPU utilization while executing a native `nFence` method that belongs to Android's renderer thread (it can be observed as a light blue rectangle in a chart describing current thread state metric [4]) – a zoomed-in relevant portion of the execution trace is presented in Figure 2. Again, it is not necessarily an indication of an actual (and fixable) problem but may be worth a second look.

3 Overheads

In its current form, as compared to Android Profiler, Nanoscope strikes a certain tradeoff – the analysis is focused on a single thread, but at least some execution metrics go beyond what Android Profiler can produce, and the overhead of data collection is expected to be much lower. While we already made an argument for usefulness of execution metrics provided by Nanoscope, we have not yet discussed its overheads.

In order to measure Nanoscope-related overheads, we executed startup sequence of the same application we used for demonstrating Nanoscope features in Section 2.3 100 times and measured the time from application start to the point when it finished one of the internally defined execution spans indicating the end of the startup sequence. We measured execution times for four different configurations:

[4] All represented thread states are color-coded and light blue color indicates native method call.

- **baseline ROM** – "clean" build of unmodified Android OS 7.1.2 that is the basis for Nanoscope implementation
- **"basic" Nanoscope ROM** – a build of Android OS 7.1.2 with "basic" Nanoscope extensions but with all data collection disabled
- **"basic" Nanoscope** – a build of Android OS 7.1.2 with "basic" Nanoscope extensions and with method tracing collection enabled
- **"extended" Nanoscope** – a build of Android OS 7.1.2 with "extended" Nanoscope extensions and with both method tracing and additional metrics collection enabled

In Figure 3 we plot average execution times (along with 95% confidence intervals) for the Nanoscope-enabled configurations normalized with respect to the execution time for unmodified Android 7.1.2 ("baseline"). As we can see, the overhead of "base" Nanoscope is very reasonable, and does not exceed 10%, which compares very favorably with Android Profiler that can slow down application execution by an order of magnitude when collecting precise traces. It is somewhat surprising that even with all tracing disabled, the overhead of Nanoscope-enabled build is noticeable – we attribute that to the code that is added to both interpreter and optimizing compiler whose fast path is always executed, but we defer more thorough investigation of this issue to future work. More importantly, however, the overhead of the sampling framework and additional several metrics collection is limited to a total of 18%.

4 Future directions

Our work on Nanoscope has only just begun, and while we feel like the tool can already be useful in locally diagnosing various performance problems, we plan to continue refining and extending Nanoscope in foreseeable future, with the help of both internal customers and a larger open source community. In particular, we would like to extend Nanoscope to collect data for multiple threads. In addition to more work at the VM level that would likely be required to accomplish this, it would also require different ways of visualizing execution of, potentially, a large number of threads – flame graphs are not very practical in this kind of setting. We may also consider further refining information provided by the tool – for example to not only indicate points of lock contention or waiting on a condition variable but also to highlight which threads exactly are involved in these kinds of events. Finally, we are thinking of providing additional information about object allocations, for example via low-overhead heap sampling similar to the one proposed for the HotSpot JVM [1].

References

[1] Jean Christophe Beyler. 2018. JEP 331: Low-Overhead Heap Profiling. http://openjdk.java.net/jeps/331
[2] Google. 2018. ART and Dalvik. https://source.android.com/devices/tech/dalvik/
[3] Google. 2018. Measure app performance with Android Profiler. https://developer.android.com/studio/profile/android-profiler
[4] Google. 2018. systrace. https://developer.android.com/studio/command-line/systrace
[5] Markus Grönlund and Erik Gahlin. 2018. JEP 328: Flight Recorder. http://openjdk.java.net/jeps/328
[6] Delyan Kratunov. 2018. Profilo: Understanding app performance in the wild. https://code.fb.com/android/profilo-understanding-app-performance-in-the-wild/
[7] Andrei Pangin and Vadim Tsesko. 2018. async-profiler. https://github.com/jvm-profiling-tools/async-profiler
[8] Leland Takamine and Brian Attwell. 2018. Introducing Nanoscope: An Extremely Accurate Method Tracing Tool for Android. https://source.android.com/devices/tech/dalvik/
[9] Uber. 2018. Nanoscope. https://github.com/uber/nanoscope
[10] Uber. 2018. Nanoscope ART. https://github.com/uber/nanoscope-art
[11] Richard Warburton. 2018. honest-profiler. https://github.com/jvm-profiling-tools/honest-profiler