Hubble: Performance Debugging with In-Production, Just-In-Time Method Tracing on Android

Yu Luo and Kirk Rodrigues, *University of Toronto;* Cuiqin Li, Feng Zhang, Lijin Jiang, and Bing Xia, *Huawei Technologies Co., Ltd.;* David Lion and Ding Yuan, *University of Toronto*

https://www.usenix.org/conference/osdi22/presentation/luo

This paper is included in the Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-28-1

Open access to the Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

Hubble: Performance Debugging with In-Production, Just-In-Time Method Tracing on Android

Yu Luo
University of Toronto

Kirk Rodrigues
University of Toronto

Cuiqin Li
Huawei Technologies Co., Ltd.

Feng Zhang
Huawei Technologies Co., Ltd.

Lijin Jiang
Huawei Technologies Co., Ltd.

Bing Xia
Huawei Technologies Co., Ltd.

David Lion
University of Toronto

Ding Yuan
University of Toronto

Abstract

Hubble is a method-tracing system shipped on all supported and upcoming Android devices manufactured by Huawei, in order to aid in debugging performance problems. Hubble instruments every non-inlined bytecode method's entry and exit to record the method's name and a timestamp. Instead of persisting all data, trace points are recorded into an in-memory ring buffer where older data is constantly overwritten. This data is only persisted when a performance problem is detected, giving engineers access to invaluable, detailed runtime data Just-In-Time before the detected anomaly. Hubble is highly efficient, with its tracing inducing negligible overhead in real-world usage and each trace point taking less than one nanosecond in our microbenchmark. Hubble significantly eases the debugging of user-experienced performance problems and has enabled engineers to quickly resolve many bug tickets that were open for months before Hubble was available.

1 Introduction

Today, Android devices are pervasive and tightly integrated into people's daily lives, yet users still experience performance problems when using these devices. Unlike Apple's iOS and iPhone, the Android platform is far from a tightly-coupled monolithic ecosystem—the hardware (manufactured by OEMs), infrastructure system software (maintained by Google and customized by OEMs), and applications are provided by different parties, and all layers are released in a rapid yet uncoordinated development cycle. This open platform makes testing enough combinations of hardware, systems software, and applications particularly challenging. Thus, many of the performance bugs that escape current testing practices are intermittent, manifesting across multiple components maintained by different entities.

When end users experience an issue, it is often *systems* vendors that shoulder the blame, before the root cause is exposed [49]. This is particularly true for Android given its huge user base, many of whom are not tech-savvy. When such users experience an intermittent performance problem, they quickly assume that their device is at fault, simply because they could not immediately reproduce the issue on another device. However, the root cause could be in the application itself, only triggered under specific conditions or inputs. To combat these assumptions, device vendors are forced to devote ample engineering and support resources to these issues.

Yet, diagnosing performance problems that occur on a user's device is extremely challenging, owing to a lack of sufficient runtime information. While approaches like Windows Error Reporting (WER) [21] are widely adopted, they can only record runtime information *after* a problem is detected. Oftentimes this is too late, as it misses crucial information just before and during the problem. This is exacerbated for performance problems, especially intermittent ones, because the issue may vanish after being detected, before recording starts. Indeed, the primary use of WER is not to record enough information to debug an issue, but to collect error statistics that are then used to prioritize debugging effort.

Recording debugging information *before* the problem occurs is challenging. We cannot accurately predict when a problem will occur, so the only option is to continuously trace the system during *normal* execution. However, overhead is a concern. Unlike servers, mobile devices are heavily resource-constrained and their workloads are overwhelmingly interactive. Sampling-based profiling tools are available, but their trade-off between informativeness and performance is poor. Non-sampling-based profiling tools, on the other hand, are too heavyweight for continuous tracing. For example, existing profiling tools on Android like Systrace [25] and Android Studio's CPU Profiler [24] can trace every method call of an application. However, enabling this type of tracing noticeably slows down an application, sometimes by more than 10×, which is unsuitable for continuous use in production. Individual applications may implement their own in-app tracing [18,23,46], but such traces are typically only available to those applications themselves.

As a result, problems reported to Android device vendors typically only include system logs, sampled statistical metrics,

Design	Unnoticeable	No src.	Maintain	big. LITTLE
Instrumentation via JIT	✓	✓		
Ring-buffer & encoding	✓			
Hand-optimized asm	✓		✓	✓
Lock-free control	✓			

Table 1: Hubble's designs and the requirements they satisfy. The headings for the requirements are truncated as follows: "Unnoticeable" refers to having unnoticeable overhead. "No src." refers to not requiring source code. "Maintain" refers to being maintainable. "big.LITTLE" refers to supporting both big and little cores.

sparse Systrace traces, and details recorded *after* a problem has occurred, like the device model, application name, and symptom. Most times, this is not enough to be useful in debugging intermittent performance problems, and engineers are left "debugging in the dark." Consequently, many bug tickets are left open for months without any hope of resolution. Worse yet, many bugs cannot even be properly *triaged*, and after rounds of finger-pointing, it is often the low-level system engineers that bite the bullet.

1.1 Challenges and Opportunities

Therefore, a *production* tracing system that can provide fine-grained observability is desperately needed. However, continuous tracing in production is challenging; it needs to satisfy a number of stringent requirements. First, the *worst-case* overhead must be unnoticeable (it cannot exceed 3% or increase the number of performance regressions throughout the deployment cycle), regardless of whether the application is running on the powerful (big) or weaker (little) cores in ARM's big.LITTLE architecture. In addition, the tool should trace applications without access to their source. Finally, it needs to be easy to maintain, and easy to merge with every new (and often feature-breaking) Android release.

These goals and constraints are stricter than what is offered by existing solutions. For instance, while record and replay (R&R) can faithfully replay the entire execution, we are not aware of any R&R system that can achieve *worst-case* overhead below 3%. In fact, most literature [30,33,35,57] emphasizes the *average* overhead; for *production* tracing tools on Android devices, engineers are primarily concerned with the worst-case instead of the average. In addition, R&R techniques typically require deep integration with the Android runtime which means that they cannot be easily maintained.

Another challenge offered by the Android runtime environment is the semantic gap between an application written in a high-level language (Java) and its native execution, which renders a rich set of system profiling tools such as gprof [27] ineffective without the runtime's support. When applied to runtime workloads, these profilers only profile the runtime's execution instead of the applications running on top of it. For example, applying gprof to a runtime workload only provides the call graph of the runtime itself (including the interpreter, GC, and JIT-compiled code), instead of the call graph of the Java application.

Android [26] and other runtimes [7] can output symbol information during execution so that system profiling tools can be applied to profile language-level executions. This approach does not completely close the semantic gap for a few reasons. First, each profiling tool must support using these symbols; currently only the sampling-based perf [39] tool supports using the symbols, and only for JIT-compiled code. Android extended and integrated perf such that it can also profile the interpreter's execution at the language-level [26]. In addition, perf expects every symbol to have a unique memory address, which is not always true; for instance, the runtime may update JIT-compiled code with application hot-patching or recompilation based on new profiling information, thus unloading old mapped code and reusing the page [26].

Yet, the runtime environment also presents a unique opportunity: trace points can be embedded and removed transparently by the runtime without modifying the application's source. This opportunity remains under-exploited despite the popularity of managed languages (the five most popular languages on GitHub in 2021 were runtime languages). To the best of our knowledge, none of the existing language runtimes offer detailed tracing tools that can be used continuously in production. For example, the OpenJDK JVM provides a powerful JVMTI debugging interface that can embed breakpoints in applications. However, this means that execution has to be deoptimized and run in the interpreter (rather than JIT compiled). Therefore, it is mostly suitable for use in development environments. Many runtimes also provide sampling-based profiling features that show "hot" code paths, but none provide continuous method-level tracing suitable for production.

1.2 Contributions

This paper presents the design and implementation of Hubble that satisfies the aforementioned goals. Hubble can capture most method entry and exit points of any application's threads, just-in-time before a failure. We designed Hubble by combining several well-known techniques in a novel way that takes advantage of the Android platform. Table 1 shows Hubble's major designs and the requirements they satisfy.

First, Android applications are typically downloaded as bytecode and then either compiled or interpreted on the device; Hubble leverages this runtime environment to automatically embed its tracing logic into the compiled binary or interpreted logic. This enables efficient tracing, as the tracing logic can be inlined into the application, avoiding more expensive trampolines (i.e., jumps in control flow) that are common in other tracing tools. In addition, this means that Hubble is a purely black-box approach that does not depend on the application's source code.

In addition, Hubble writes trace points to an in-memory ring buffer that is only flushed when a problem is detected. This allows it to run continuously and capture information just-in-time leading up to a failure. By designing a concise, variable-length encoding, such that most trace points occupy eight bytes, a small (32MB) ring buffer is enough to capture sufficient debugging information.

Third, Hubble's performance-sensitive instrumentation logic is written in assembly. This ensures that performance is optimal even on a device's low-power (little) cores, which cannot perform out-of-order execution or have small instruction reordering buffers. In addition, this decouples Hubble from the Android compiler's compilation flow, so it avoids having the compiler affect the correctness of the tracing logic, and eases maintainability.

Finally, Hubble avoids using expensive synchronization primitives [14] in two ways: threads write trace points to thread-local buffers, avoiding inter-thread synchronization; and, Hubble communicates with these threads by using a purpose-built lock-free synchronization protocol.

The end result is a highly efficient method-tracing system sufficient for debugging intermittent performance bugs. In our microbenchmarks, each trace point costs less than one nanosecond for nearly empty methods, and tracing overheads are quickly amortized when methods perform meaningful operations. Hubble's tracing overhead is also unnoticeable in Huawei's continuous-integration performance testing infrastructure, which includes a variety of workloads and devices. Hubble's memory overhead is approximately 64 MB by default, accounting for two 32 MB ring buffers. As of 2021, Huawei's lower-end smartphones have at least 4 GB of RAM, while higher-end ones can have up to 12 GB. Therefore, Hubble's memory overhead is less than 2%.

Hubble also strives to protect user privacy. Similar to existing error reporting systems such as WER [21], MacOS [2] and Mozilla [34] crash reports, Hubble's traces are only collected with user consent. However, these other systems collect a minidump of the memory image, whereas Hubble's traces are far less sensitive: they only consist of method names and timestamps and do not contain any variable values.

Hubble has been integrated into Huawei's core Android OS codebase, deployed across a wide range of smartphone and tablet product lines, since August, 2020. Older devices may receive Hubble's functionalities via an over-the-air OS update. Since deployment, Hubble has significantly eased the debugging of intermittent performance problems. In fact, engineers were able to quickly resolve many performance problems that remained unresolved for months.

This paper makes the following contributions:

- The design and implementation of Hubble, a highly efficient method tracing subsystem for Android, that satisfies a set of unique, practical constraints, some of which are rarely mentioned by existing literature.
- Integration of Hubble's traces with existing debugging tools, like Perfetto [40] which can show call charts. This significantly improved the trace's utility, where developers can cross-examine Hubble traces with other runtime data.
- Case studies on how Hubble diagnoses real-world performance bugs which cannot be resolved without it.

Hubble also has the following limitations. First, it can only embed tracing logic into executions that go through the Android compiler or interpreter (from bytecode); Hubble cannot trace native libraries like those invoked through the Java Native Interface (JNI). In addition, Hubble's trace buffer could pollute the CPU cache and slow down cache-optimized workloads (e.g., loop tiling [8]). However, while cache-optimization is commonplace in server workloads, it is uncommon on smartphones, especially in the interactive UI-thread. Nonetheless, we evaluate this effect in §8.

2 Related Work

Record and replay (R&R) tools [10,15,16,29,30,35,36,38,50,57] work by recording a user's input and all non-deterministic events (e.g., scheduling), so that the execution can be faithfully replayed. R&R tools do not meet our requirements for a few fundamental reasons. The first is overhead. Among all R&R tools, Reverb [35] reported the best performance, yet its overhead is still 5.5% *on average* (the worst-case is not reported). It works only on JavaScript web applications, where threads communicate using a message-passing interface. When threads share memory, R&R incurs even higher overhead. For instance, DoublePlay [57] reported a worst-case overhead of 11% for network-bound workloads (Apache webserver), 19% for disk-bound workloads (MySQL), and 278% for CPU-bound workloads (SPLASH-2 ocean). To achieve low overhead, some tools [33, 38, 45] do not record all non-determinism which prevents accurate replay. Second, since intermittent performance bugs may take days to occur, R&R traces will grow untenably large. While checkpointing could allow replay from a partial trace, the checkpointing operation itself is expensive [50]. Compared to a call chart, an R&R trace also imposes much larger privacy concerns. Finally, R&R tools require deep integration with the Android runtime and compiler. For instance, applying DoublePlay's approach to Android would require the runtime to run a parallel execution of the application, checkpoint and compare state between the two processes, and so on. Hence, R&R tools would be difficult to maintain within Android.

An attractive alternative is to use hardware-support, like Intel PT or ARM ETM, to record branch-level traces [12,28,60]. These tools have a *worst-case* runtime overhead of 1–2%. However, there are two challenges on ARM devices. First, the semantic gap on Android's runtime complicates the decoding of the branch-level trace, as it only provides the traces of the runtime's execution instead of the application. Second,

hardware support for tracing is restricted to development platforms (most ARM processors on production Android devices do not support the feature) [4].

Only a limited set of bytecode method tracing tools are available on the Android platform. Android Studio's CPU Profiler can trace every method call, but its overhead is incredibly high (a worst-case of 921× in our evaluation), because instead of embedding the tracing logic into the compiled binary, it jumps into the Android runtime after every method call. Internal tracing utilities within Android mostly leverage Java Agent, JVMTI, or equivalent ART instrumentation interfaces to perform method tracing. These mechanisms are also expensive as they force applications to be interpreted only. Aspect-oriented frameworks such as Tai Chi [53] and Logan [55] are also available to intercept method calls at runtime to execute arbitrary tracing code. However, they either require modifications to the application's source code or root access. The fastest available method tracing utility that we are aware of, Nanoscope [54], primarily targets method tracing inside an x86 Android emulator, costing up to 10× higher memory usage and performance overhead, so it is mostly useful in an application development environment.

Some tools are able to perform in-application tracing with low overhead in production. For instance, Firebase performance monitoring [23] collects various metrics (e.g., startup time) and allows developers to insert additional trace points. AppInsight [41] instruments Windows Phone application binaries to log whenever the runtime calls into and returns from application methods. The instrumentation has sufficient detail to allow a server to reconstruct how a user request was processed across different application threads and what the critical path is. These tools typically trace the entire run of an application, but at a low enough granularity that the trace does not grow untenably large. As a result, they are useful for application developers to locate bottlenecks in their application; but the coarseness of the trace may necessitate additional debugging information to locate the exact root cause, especially if the bug is in the underlying systems which are not traced. Timecard [42] goes beyond tracing by using AppInsight's traces to adjust the server's computation quality (in real-time) to meet an end-to-end response deadline.

There are also a few high-performance logging solutions like NanoLog [58] and Log20 [59] that can provide nanosecond-level logging. Both write data to thread-local ring buffers and NanoLog uses a specialized encoding to save space. NanoLog uses only the existing log statements in the application while Log20 can be used to determine where best to place log statements based on profiling the application's usage pattern.[1] In any case, the generated trace is only as detailed as the developers' instrumentation.

Outside of the Android platform, there are many call profiling tools like gprof [27], Fay [17], ftrace [51], perf [39], DTrace [9], and SystemTap [52]. These tools support various degrees of tracing from periodically sampling the call stack to calling user-defined methods using dynamic instrumentation. However, to capture traces that are detailed enough to diagnose intermittent bugs, these tools incur overhead that prevents them from tracing continuously in production systems. These tools typically require calling a method in their instrumentation, whereas Hubble directly inlines the tracing code into each method.

There are a large number of tools designed to trace each request in a distributed system. Examples include Project5 [1], MagPie [5], X-Trace [20], Dapper [47], ÜberTrace [11], and Pivot Tracing [31], as well as commercial tools like Datadog [13] and New Relic [44]. These tools typically embed trace points in critical system or network events, such as RPCs, and record an ID that is unique to each request.

3 Case Studies

We present two case studies to showcase how Hubble helped in diagnosing real-world intermittent performance problems. The first issue was within AppX, a third-party multipurpose messaging, social media, and mobile payment application with over a billion monthly active users. Occasionally, AppX users experienced intermittent UI freezes (janks) of up to two seconds. Engineers detected this problem by monitoring the traces that Systrace continuously collects—namely, performance alerts, sparse trace points, and metrics sampled at low frequencies. Figure 1 (A) shows the available trace points rendered as a method call chart in the Perfetto trace-visualization tool. For the UI thread, this consists of only a few high-level methods within the Android framework. The only conclusion engineers can infer from this data is that the UI thread was blocked for about two seconds during which it was supposed to prepare the layout and content for rendering.

In contrast, the call chart based on Hubble's trace, shown in Figure 1 (B), accurately captures every method call in both the application and the Android runtime. From the canonical method names displayed in the chart, engineers were able to quickly reconstruct the events that occurred before, during, and after the UI jank. First, the user swiped back on the device's screen within AppX (①). Then, AppX initialized a software keyboard to respond to the user's action (②). However, to display the keyboard, the scrollable chat component must be resized (③), and this became the bottleneck. Drilling down further, we can observe the series of method calls responsible for generating the list of on-screen content (④). Specifically, we can see that the UI thread is primarily blocked by various long-running methods belonging to AppX. Now with concrete evidence, our engineers concluded that the root cause was within AppX, and initiated a meaningful collaboration with AppX's developers.

The second issue was a longstanding performance bug

[1] In fact, the initial goal of this project was to integrate Log20 into Huawei's Android platforms.

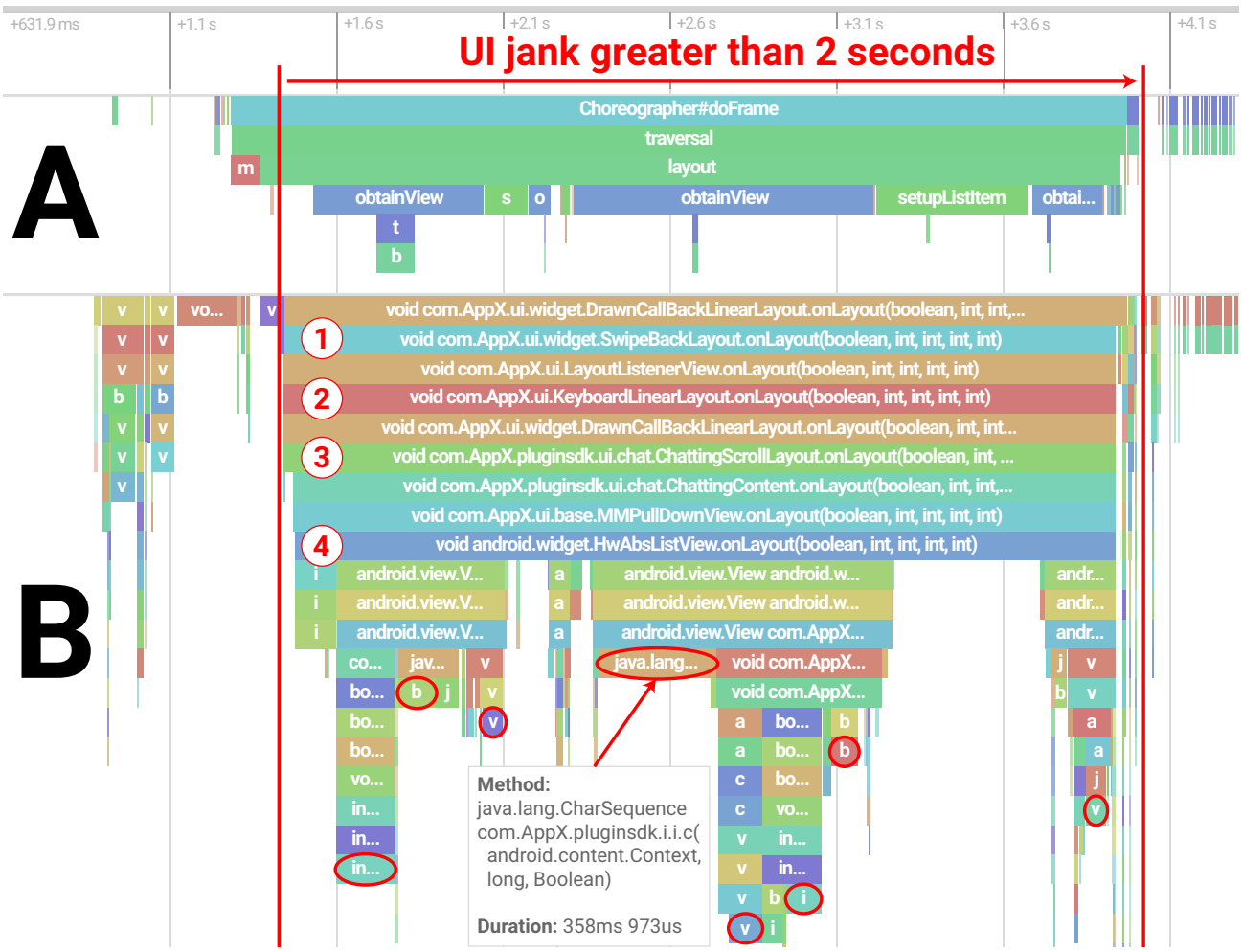

Figure 1: Screenshot of method call charts in Perfetto for the UI thread, which performs all UI and Android framework operations. (A) Traces generated by Systrace, (B) Traces with Hubble. Circled in red are 3rd-party application methods with long execution time. (A) includes *all* of Systrace's trace points recorded during this time period, whereas (B) is filtered to render only approximately 10% of all available methods.

within an internal business teleconferencing application. After the end of a teleconference, the application occasionally froze for up to a second on a small number of user devices. This annoyed users but it was not until months later that a particularly vocal employee reported the issue to management, who then opened a support ticket requesting that the issue be resolved. Our device support engineers attempted to reproduce the problem on their own, but all attempts were unsuccessful. The only method call captured by Systrace was `binder_transaction()`, which does not explain why the issue occurred. Further efforts to collaborate with the disgruntled users were also ineffective as most users were either too busy or otherwise unable to provide more detailed reproduction instructions. A few users were even invited to collaborate with an engineer to reproduce the problem, but the intermittent issue could not be reproduced after multiple attempts.

Several months later, Hubble, in pre-beta at the time, was available for internal use. The disgruntled employees happily consented to deploying Hubble onto their mobile device via an over-the-air Android OS update. Within a few days, performance anomalies were detected and their associated trace data was automatically collected. After a quick glance at Hubble's call chart, the support engineers identified that the teleconferencing software was calling `Thread.sleep()` from the UI thread after sending an Android Binder (IPC) system service call. Closer inspection revealed that immediately after a conference call ended, a series of method calls related to the Audio Manager were performed, prior to the `Thread.sleep()`. This behavior was unexpected and if not for the complete method call trace, which contained both the application and Android framework layers, we would still be stuck with many of our initial theories; e.g., the application could be collecting and sending meeting summary data back to the teleconference service or an unexplained scheduling issue.

With this new information, we brought in a developer with expertise in the Android audio stack. After examining Hub-

ble's call chart, the developer immediately identified the root cause. The problem can only be reproduced under very specific conditions where users must be connected to Bluetooth headsets using a special mode prior to ending the meeting. After the meeting ended, the application immediately rerouted audio to Bluetooth devices connected over the A2DP streaming protocol. This rerouting process requires re-initialization of Bluetooth's SCO (synchronous connection-oriented) link where the `Thread.sleep()` was invoked to wait for the link to be established. We were unfamiliar with these details, but with the help of a developer with the necessary domain knowledge, the issue was promptly fixed by moving the connection and rerouting logic into an asynchronous event handler.

4 Background and Overview

This section first discusses Hubble's design goals and the role it plays in the failure diagnosis process, which helps to understand Hubble's design. We then provide an overview of Hubble, leaving the details to the subsequent sections.

4.1 Goals and Requirements

Hubble's performance overhead and resource usage must be *undetectable* in all real-world usage scenarios. In practice, this translates to two requirements: Hubble's worst-case overhead in real-world scenarios, in terms of both latency and memory usage, should be less than 3%. This target was set by our quality assurance team since they cannot reliably measure overhead below 2–3% on mobile platforms, even under ideal conditions. Nonetheless, this is similar to the target set by other practitioners; Google, for example, reported a 2% overhead budget to deploy tracing tools in production server workloads [32, 48]. The second requirement is that the overhead budget should be respected regardless of whether Hubble is tracing workloads on big or little cores. Besides not being as fast as big cores, little cores also tend to lack advanced features like out-of-order execution. Thus, they enforce stricter restrictions on the tolerable overhead for Hubble. In any case, satisfying the target overhead only allows a tool to pass the deployment planning review. To be deployed in production, the tool needs to go through a systematic procedure consisting of three phases:

1. **Internal testing**. We simulate users using our devices by sending a stream of pseudo-random inputs to a large fleet of physical devices. Each device collects various metrics like application startup times, the number of dropped frames, and so on. Each metric forms a statistical distribution over a large number of trials. We compare the distribution before Hubble was added with the one after. If the differences are statistically insignificant, Hubble has not caused a noticeable change, and we move to phase 2.

2. **Internal beta release**. We push engineering builds to a small group of internal beta testers on their daily-use devices. We ask these users to report any performance regressions they notice and any new performance issues will need to be resolved before moving to the next phase.

3. **Public beta release**. A build is pushed to all beta users (tens of thousands of users), and we monitor all new performance anomaly reports. We only consider Hubble's overhead as undetectable when the beta build does not show a statistically significant increase in reports. Only then can the tool be further released to the entire public.

Android applications are typically distributed as bytecode compiled from high-level languages like Java. Once downloaded, this bytecode is either ahead-of-time (AOT) compiled, or executed within the Android runtime (similar to the Java Virtual Machine). The Android runtime compiles frequently executed code Just-in-Time (JIT), using the same AOT compiler. An already-compiled application could also be re-compiled, if runtime profiling reveals new optimization opportunities. Applications could also contain native libraries, i.e., code that was already compiled into native instructions. Thus, Hubble must be able to operate with only access to the downloaded or generated bytecode or native instructions.

Easy maintainability across Android versions is required. Android is typically updated every six to twelve months, with each new release potentially breaking features or making large-scale changes internally. Thus, Hubble should be modularized and decoupled from the upstream source.

Finally, as the case studies highlight, Hubble needs to be able to trace both the executions of the application and the Android framework to be useful. Ideally, device vendors would only be responsible for analyzing and debugging bugs within Android, and application developers would only be responsible for bugs within the application. The reality, however, is that bugs in the Android framework may manifest themselves in the application and vice versa. Furthermore, system traces are not available to application developers (in order to maintain users' security) and in-application traces may not be available to or easily understandable by device vendors. Exacerbating the issue, application developers and even internal developers at Huawei are reluctant to investigate bug reports without clear evidence that the bug is in their code. Whole-system method traces allow engineers to infer roughly what the application and framework are doing, together, so that the problem scope can be narrowed down to specific call chains and system services. Essentially, Hubble needs to bridge the gap between system and application developers, which in turn, will significantly ease triaging and debugging for both parties.

Overall, these requirements highlight the practical challenges of designing and deploying tracing tools onto a complex user-device platform such as Android.

4.2 The Failure Diagnosis Process

To understand Hubble's utility, we first need to overview the failure diagnosis process. Android devices ship with a set of anomaly detectors to detect common issues like lags in the UI. When an anomaly detector fires, the system saves several pieces of data such as logs, metrics, and traces. At an appropriate time, these data will be uploaded to the device vendor for analysis.

4.2.1 Anomaly Detection

Since Hubble's utility depends on an anomaly detector firing, we first provide background on the detectors available in the Android Open Source Project (AOSP) and our version of Android. There are two branches of anomaly detection mechanisms that device vendors can use in the production environment: Those implemented by Android itself and those implemented by in-house engineering teams. Both branches use information gathered either from the Android runtime layer or from the Linux kernel. In addition, both branches are generally tuned to be conservative to reduce the number of false positives. However, if a severe performance issue occurs, a signal will most likely be raised.

The anomaly detectors implemented in the AOSP have been continuously developed for over a decade. For example, the most frequently used anomaly detector is the UI jank (lag) detector, which has an extremely close correlation to user-observable performance issues. It will alert if a number of consecutive display frames are delayed longer than a predefined threshold. Android officially groups all its tracing, profiling, and anomaly detectors under one umbrella term known as systrace. In production environments, most of these anomaly detection signals and alerts are continuously captured and analyzed in real time.

Internally, we utilize a number of additional black-box anomaly detectors which monitor for a number of kernel level indicators and hardware events. For example, we implemented a system-level, HCI-based detector: Studies show users start to perceive a delay after 400-600ms. So by instrumenting the runtime where (1) a touch is detected by the screen, (2) when the signal is delivered to the application, and (3) the application generates a response, we can accurately measure the delay between (1) and (3) and fire an alert when the delay is longer than 400ms. Furthermore, we can attribute the delay to either signal delivery in the runtime or within the application.

Other black-box anomaly detectors could be as simple as monitoring whether the device has entered the thermal throttling mode. Most detectors, however, don't rely on a single metric. Instead, they correlate multiple metrics. For example, if a detector detects that the current GPU memory bandwidth utilization is high, it then checks other metrics such as the rendering queue backlog length; only if multiple of them suggest an anomaly does the detector fire a warning. Experimental anomaly detectors may further leverage real-time machine-learning monitoring Android runtime metrics like the number of locks held, memory allocation and garbage collection frequency, and so on.

4.2.2 The Utility of Hubble

When Hubble's traces are collected, they are integrated into systrace and Perfetto when presented to engineers with other runtime data. Perfetto and systrace are powerful debugging tools that can visualize a variety of runtime data, including visualizing the method trace as a call chart or flame graph. The tools also have search and analytics (e.g., using SQL) capabilities that allow developers to correlate data from different sources. For instance, developers can cross-examine traces with logs and hardware metrics. Developers can also alert based on traces. For example, one use case of Hubble is to search for the call stack that matches a specific method invocation order, get an average runtime, and alert when it exceeds a threshold. As a result, Hubble is not a standalone tool, nor the only debugging tool. Instead, developers usually start debugging by first examining the data from existing logs and metrics, and some bugs can be resolved with these alone. However, the remaining bugs—typically hard-to-diagnose, intermittent issues—require more insight, which is where Hubble excels.[2]

Key to Hubble's success is the visibility it provides into application and framework-level behaviour, without which engineers cannot triage issues. Hubble's detailed method traces also allow developers to better understand how a bug can be reproduced; with a reproduction, developers can repeatedly reproduce the bug in a development environment (with heavyweight tracing) until the issue is understood.

Nonetheless, there are some limitations to Hubble's utility. We have found Hubble's traces are not as useful in the following cases: (1) if the bug is in the system's native code (which is not traced), (2) if the method-level trace is not fine-grained enough (e.g., an infinite loop without making any function calls), or (3) if a bug is caused by incorrect data-flow (i.e., an incorrect variable value) that does not affect the call path (otherwise it could be inferred by Hubble's trace). However, Hubble's traces can still help developers to significantly narrow down the problem scope (e.g., they can locate the method that contains the infinite loop). In theory, if the distance between the root cause and the symptom is too long, Hubble could miss the cause due to the ring buffer size. However, we have not yet encountered such a case in practice.

[2] We do not have an exact number of issues exclusively resolved by Hubble, because Hubble's traces are integrated into existing debugging tools with other traces. However, we noticed the number of bug tickets containing intermittent and difficult to reproduce bugs quickly dropped after Hubble was first made available.

4.3 Overview of Hubble

Hubble modifies the compiler and interpreter to instrument tracing logic at the entry and exit of every non-inlined bytecode method, whether it is interpreted, ahead-of-time compiled, JIT compiled, or recompiled. Portions of the Android framework itself and factory installed apps, i.e., the apps that are packaged by the OEM vendor, could be already in compiled form instead of bytecode; for these cases, the trace points are embedded at the vendor's site. Hubble can also trace calls made using the JNI (when applications calls into the native libraries and the returns). However, function calls made within native libraries cannot be traced by Hubble.

Hubble adds one system thread, the trace control thread, to each application's process that can turn tracing on or off for any thread in the same process. Although Hubble instruments all bytecode methods, by default, the control thread only turns on tracing for the UI thread, which performs all UI and Android framework operations. At every method entry and exit, Hubble's tracing code writes an entry to a fixed size in-memory ring buffer. When the buffer is full, the buffer pointer will wrap around so the oldest data will be overwritten.

When a performance anomaly detector detects a performance problem, the control thread will be notified. It then notifies the UI thread to stop tracing, preventing useful debugging data prior to the problem from being overwritten. Once tracing has stopped, the control thread flushes the ring buffer to disk, before restarting tracing. The saved trace file could be sent back to Huawei to aid postmortem debugging, or post-processed and analyzed on the device, off the critical path, if a summary needs to be sent.

Each traced thread writes to a private ring buffer local to itself. Hubble keeps at most N buffers in the system, from the N threads that most recently executed in the foreground. Older buffers will be reclaimed by the system. N is configurable and the method trace logic can be programmatically enabled and disabled for individual threads, either via the runtime or by the user application itself. This means that any background threads from almost any process, even short lived ones, can be traced. However, if there are too many concurrent threads being traced, Hubble will run into memory usage issues. To solve this, we could have a ring buffer per core rather than per thread; to differentiate trace points from different threads, we could record the thread's ID (available from a register in the runtime) in each trace point. By default, N is set to 2. This is sufficient to capture both the current foreground and most recent background application's UI threads.

5 In-memory Tracing

This section describes the design and implementation of Hubble's tracing logic. We first explain the information recorded in each trace point and its encoding. We then discuss how we integrated the tracing code into Android's optimizing compiler so that compiler optimizations do not affect our instrumentation.

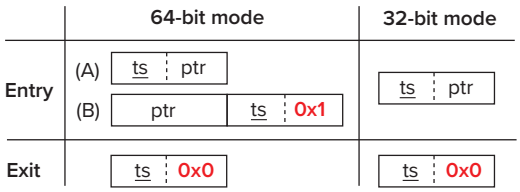

Figure 2: **Format and encoding of trace points** at method entry and exit, and in 64-bit and 32-bit execution modes. "`ts`" and "`ptr`" are timestamp (generic timer count) and method pointer. A solid bordered box represents a 64-bit slot. Underscores represent lossy encodings of timestamps.

5.1 Data Format and Encoding

Figure 2 shows the format of each trace point. As shown, method entry points have a varying encoding depending on the CPU's execution mode and other factors explained later. The CPU will change mode when executing a 32-bit or 64-bit application. Method entry trace points contain a timestamp and a method pointer, while exit points contain a timestamp and the constant `0x0`.

For timestamps, Hubble uses the Generic Timer [3] count instead of the standard system clock. A Generic Timer is a high resolution clock (nanosecond precision) and its tick value can be directly read from a register on modern ARM SoCs. It ticks at a constant frequency regardless of the CPU operation speed and the counter value starts at 0 when reset. When the trace is persisted, Hubble records the current time, which can be used to reconstruct the absolute timestamp of each trace point from the Generic Timer count.

The method pointer is the memory address of a metadata object, ArtMethod, that describes each loaded class-method and can be used to decode a method's canonical name. As part of the ClassLoader initialization process in Android's runtime (ART), an array of ArtMethods is allocated in a memory region outside the managed heap (ignored by garbage collection). ArtMethods can only be added to this array and never be modified nor removed. ART ensures that immediately after entering a method, the address of its ArtMethod is stored in register r0. Since the lifecycle of the main ClassLoader, which is responsible for loading all of the executed bytecode methods, spans the entire duration of the application, we can safely store the ArtMethod pointer in the trace buffer and reconstruct the method name after the trace data is persisted, so long as this happens before the application exits. Note that applications could use additional custom ClassLoaders with shorter lifecycles. If we persist the trace data *after* the custom ClassLoader exits, we could dereference pointers that are no longer valid. To avoid this, we install a cleanup hook for custom ClassLoaders to invalidate the trace buffer (or optionally persist the trace data).

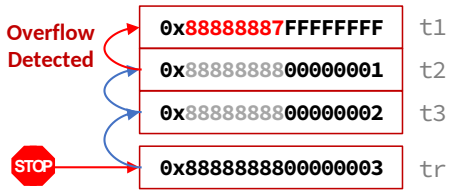

Figure 3: Iteratively Recover Truncated Timestamps.

For each thread that is traced, the control thread allocates storage for the trace in the traced thread's local storage (Java ThreadLocal [37]). This includes a ring buffer and metadata such as where content in the buffer begins and ends. The ring buffer is carved into an array of 64-bit wide integers in both 64-bit and 32-bit mode.

For the timestamp in each trace point, Hubble only stores the lower 32 bits of the Generic Timer counter, regardless of execution mode. (Even in 32-bit mode, the Generic Timer counter is 64 bits wide because the value is fetched from a co-processor that is not subject to the mode change.) Thus, the recorded timestamp may wrap around, which we handle during decoding.

Figure 3 shows how Hubble reconstructs the accurate timestamp from truncated ones. The last timestamp is a reference timestamp (`tr`), which is the complete 64-bit Generic Timer counter value recorded when the trace is persisted. Using `tr`, we can iteratively reconstruct the upper 32 bits of the previous three timestamps: if a previous timestamp has a lower value than the current one (e.g., `t3` versus `tr`), we assume it has the same upper 32 bits; if it has a higher value (e.g., `t1` versus `t2`), we assume a wrap around occurred and the upper 32 bits should be decremented by one.

Theoretically, this could lead to an error: if between two consecutive trace points more than 2^{32} ticks occur, the reconstructed timestamp will be inaccurate. However, this is unlikely to happen in reality. It takes 223.7 seconds on a Qualcomm ARM SoC and a little over 37 minutes on a Huawei-designed SoC for the lower 32-bit Generic Timer counter to tick 2^{32} times. So only if a method executes for more than 223.7 seconds, without calling another method or returning, will an inaccuracy occur.

5.1.1 Format under 64-bit Mode

Hubble uses a variable-width encoding for the ArtMethod pointer when executing in 64-bit mode. In this mode, the pointer is 64 bits; but for real-world applications, the vast majority of the pointers' upper 32 bits have the value `0x0`. We exploited this observation to increase encoding efficiency. When the upper 32 bits are `0x0`, Hubble only records the lower 32 bits of the pointer (Figure 2 (A)). Together with the lower 32 bits of the timer count, a method entry trace point occupies a single 64-bit buffer slot. If the upper 32 bits of the method pointer are not `0x0`, a method entry trace point occupies two buffer slots (Figure 2 (B)). The first 64-bit slot is used to save the complete 64-bit method pointer; in the second slot, the upper 32 bits store the timer count and the lower 32 bits store the constant `0x1`.

The method exit trace point occupies a single 64-bit slot. The upper 32 bits store the timer count, and the lower 32-bit stores `0x0`, indicating it is a method exit trace point.

Traces in this format can always be **unambiguously decoded** in reverse. To decode each trace point, Hubble first checks the lower 32 bits of the previous slot. Depending on whether its value is `0x0`, `0x1`, or another value, Hubble knows that this trace point is either a method exit, a method entry that is two slots wide (Figure 2 (B)), or a method entry that is one slot wide (Figure 2 (A)). `0x0` and `0x1` cannot be method pointers since they are invalid method pointer memory addresses. A method exit point is matched with the corresponding method entry point in a LIFO manner (implemented using a stack). Note that the decoding occurs server-side, after the persisted trace has been sent back.

5.1.2 Format under 32-bit Mode

In 32-bit mode, both method entry and exit trace points use a single buffer slot. The upper 32 bits are always the lower 32 bits of the timer count, like in 64-bit mode. For method entry points, the lower 32 bits store the method pointer, and for method exit points, the lower 32 bits store `0x0`.

5.1.3 Efficient Recording

The tracing logic can be efficiently implemented by a few assembly instructions. For example, Hubble uses only two assembly instructions to store the method entry trace point under 32-bit execution mode:

```
1 MRRC(al, scratch1, scratch0, 0b0001, 0b1111, 0b1110);
2 STRD(r0, scratch1, MemOperand(buffer, 8, PostIndex));
```

The first `MRRC` instruction is used to fetch the 64-bit Generic Timer counter value into two 32-bit CPU registers: `scratch1` and `scratch0` (readers can ignore the other operands). Then a `STRD` instruction is used to (1) store `scratch1`, which contains the lower 32-bits of the Generic Timer counter, and `r0`, which contains the ArtMethod pointer, to the memory address stored in `buffer` register, and (2) increment `buffer` by 8 bytes after the memory operation completes. So after this store instruction, `buffer` will point to the next buffer slot.

Hubble's tracing assembly is directly inlined in the basic block at each method entry and exit. Comparatively, in other profilers that use compiler instrumentations, the instrumented code will call a special tracing function. For example, `gcc -pg` instruments a call to the special function `mcount()`, which is required for tools like `gprof`. While easier to maintain and more portable, the added function call introduces overhead.

When tracing is stopped, the valid portion of the ring buffer is flushed to disk using an *fwrite* call. Three metadata files are generated. First, a complete 64-bit Generic Timer counter value (i.e., the reference timestamp) and the absolute system timestamp are collected at the same time; this facilitates the reconstruction of the actual, non-relative timestamp of each trace point if needed. Then the current buffer position and size are recorded. Finally, Hubble computes a symbol table, mapping each unique ArtMethod pointer value to the method's canonical name.

5.1.4 Alignment

Each trace point is always eight-byte (a word on 64-bit devices) aligned. Eight-byte aligned memory accesses are crucial to achieving the highest performance in both 32-bit and 64-bit mode on modern ARM SoCs. Unaligned accesses take at least one more cycle than a properly aligned memory access. In the worst case, a single unaligned access can cross a cache-line boundary and generate two cache misses or even two consecutive page faults. Worse yet, unaligned memory accesses are an unsupported operation on low-power or older ARM processors, so additional memory accesses and massaging logic are required. Accordingly, we use 32 bits to represent the constants `0x0` and `0x1`, since the performance gains of aligned accesses outweigh encoding inefficiency.

5.2 Hand-optimized Assembly

There are a few reasons to write the tracing logic in assembly. First, it decouples Hubble from the Android compiler's compilation flow. If written in C++, the compiler could move, reorder, or even remove the tracing logic (e.g., the tracing logic accesses global variables without a memory barrier (§6), which is an undefined behavior). By writing the logic in assembly, we can insert it after the compilation stage, bypassing any optimizations that are at odds with the tracing. To do so, early in the compilation stage, instead of generating the actual tracing code, we simply insert a special placeholder instruction at every method entry and exit (including exits due to exceptions); we then configure the Android compiler to exempt this instruction from its later optimization stages. After all the optimizations are performed, we replace this placeholder instruction with the actual tracing instructions. This also makes Hubble easy to maintain, as it is decoupled from any compiler changes that are not backward compatible.

Using assembly also allows us to optimize for both big and little cores. The Android compiler's optimization is heavily biased toward the big core. For example, the compiler skips the architecture-specific optimizations when they are unnecessary on big cores that support out-of-order execution. However, the little cores do not support out-of-order execution, so running the compiled code will result in poor performance. For instance, each trace point needs to check if we are at the end

```
// Initialization
start=0x0;  stop=0x0;

Control thread:                 Trace point:
1  start=buffer;                 7  if (start)
2  wait(signal);                 8    trace...
3  stop = 0x1;                   9    if (stop) {
4  while(stop==0x1)             10      stop=buffer;
5    sleep(..);                 11      start=0x0;
6  persist(..);                 12    }
```

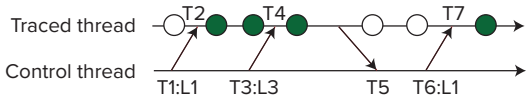

Figure 4: Lock-free Synchronization Protocol.

of the ring buffer (and if so, we need to wrap around). This check requires fetching the value of the ring buffer pointer from memory. If we manually prefetch this pointer (in assembly), it results in an approximate speedup of 35% on the little core. The compiler, however, did not perform this prefetching, because it expects the big core will perform the prefetching automatically.

Finally, because we have domain knowledge of the tracing logic and processor microarchitecture, we can perform better optimizations than the compiler, regardless of whether it is on the big or little core.

6 Tracing Control

Recall a system thread is responsible for notifying the traced thread to turn tracing on or off. The traced thread (e.g., the UI thread) is only responsible for (1) checking whether tracing is turned on, and if so, (2) writing the trace points into the trace buffer, and (3) turning tracing off if necessary. The rest of this section describes how the two threads communicate efficiently without synchronization primitives.

Figure 4 shows the communication between the control thread and the traced thread. Lines 1–6 are the control thread's logic, whereas lines 7–12 are executed at every trace point in the traced thread. Hubble uses two eventually-consistent, shared variables, `start` and `stop`. `start` is unidirectional, i.e., it is set by the control thread and read by the traced thread, and `stop` is bidirectional, as it can be set and read by both threads. Initially, both variables are set to `0x0`. To start tracing, the control thread sets `start` to the *address of the next buffer slot* (line 1 in Figure 4), and waits for a signal to stop tracing. Therefore, the value of `start` indicates two things: whether tracing is on or off, or the buffer position. At each trace point, the traced thread first checks if `start` is `0x0`, and only proceeds with tracing if it is not (line 7).

To turn tracing off, the control thread sets `stop` to `0x1` (line 3 in Figure 4), and then enters a polling loop until `stop` is changed to a value *greater than* `0x1` (lines 4–5). In the meantime, the traced thread performs tracing and evaluates

the value of `stop` at the end of every trace point (line 9). Once the traced thread detects that `stop` was changed to a non-zero value, it enters the logic to stop tracing. The traced thread first sets `stop` to the address of the current buffer pointer, i.e., the end position of the buffer, at line 10. So `stop` also serves dual purposes: whether tracing should stop (with value 0 or 1), or the buffer end position. (Note that `0x0` and `0x1` are invalid buffer memory addresses, so after line 10, `stop` will be greater than `0x1`.) Then the traced thread sets `start` to `0x0` at line 11, to guarantee that tracing will be disabled immediately. Finally, the control thread detects that the traced thread has stopped tracing, so it can persist the trace or clean up the ring buffer.

Figure 4 also shows an example trace control-flow. Each circle represents a trace point, with filled and blank shading indicating whether trace data is written or not. At the beginning, tracing is off. At T1, the control thread turns tracing on at line 1 (L1) by setting `start` to a non-zero value. This new value is propagated to the traced thread at time T2, as the result of eventual consistency in the memory cache coherence protocol. Then, the following three trace points are written to buffer. At T3, the control thread turns tracing off by setting `stop` to `0x1`, which is propagated to the traced thread at T4. The traced thread then executes lines 10–11, and at T5, the control thread detects that `stop` was changed to a value greater than `0x1`; so it breaks out of the polling loop and persists the trace. After the trace is persisted, the control thread restarts tracing at time T6 (line 1).

This design is highly efficient. Each trace point needs to check the values of `start` and `stop` only if the trace has been started. `start` and `stop` are regular shared variables that are almost always cached. In comparison, any alternative design that uses synchronization primitives or atomic variables would introduce much higher overhead in each trace point, which is on the critical path.

Since tracing is stopped and the current ring buffer location is written to the `stop` variable by the traced thread itself, no additional trace point will be written to the buffer afterwards and the buffer metadata will be consistent. For example, if the last trace point is a 64-bit method entry occupying two slots, it is guaranteed that both slots are written with the buffer pointer correctly incremented before tracing is stopped.

If the traced thread is executing native code, either through the JNI or a custom ClassLoader, it cannot respond to the control thread's stop tracing request, because the logic to stop tracing is only instrumented in bytecode methods. Therefore, the control thread further checks whether the traced thread is in native execution when it attempts to stop tracing. If so, the control thread will first obtain ART's state transition lock that prevents the traced thread's execution from changing state, i.e., from native execution back to the bytecode world (either the interpreter or compiled code). Then the control thread forcibly copies the buffer position to `stop`, and sets `start` to `0x0`, followed by a memory fence. Finally, the control thread can release the state transition lock. A subtle data race could occur during state transition where just before the lock is obtained, the traced thread transitions back to the bytecode world. Debugging this unfortunately took weeks, but we fixed it by rechecking the traced thread's execution state after obtaining the lock.

7 Privacy and Security

Security and privacy are some of our top priorities. Hubble does not collect personally identifying information, such as phone numbers or user IDs. Hubble's traces only contain method names and timestamps, there are no actual *data values*, not even parameter values. Widely-adopted error reporting systems like Windows Error Reporting (WER) [21], MacOS' crash report [2], or the Mozilla Crash Reporter [34], record a subset of the memory state or often collect system logs. In comparison, Hubble's traces are far less sensitive. Similar to WER and other widely-adopted error reporting systems, Hubble uses an informed consent policy.

Even when user consent is given, Hubble further strives to minimize the amount of data that leaves the device. Hubble has the capability to perform the same analyses that are performed server-side, locally on a user's device, with only a summary being sent back to the vendor. For example, Hubble can quickly scan the trace files and compute the top methods with the longest "self-execution-time", or it can automatically isolate and extract the longest method call chains from when a performance anomaly occurred. Performance bug models could be distributed to client devices, containing "signatures" of problematic method names or method call chains, and if there is a match, statistics could be sent back instead of the complete trace.

Hubble also exploits many built-in data security features in Android and the Linux kernel to protect trace data. The traces are stored inside an application-private storage area that is protected by the kernel-level application sandbox. Only the application itself with matching its UID, device vendors, and application developers—when they configure their mobile device in debug mode—have access to the trace files.

8 Evaluation

Hubble has been repeatedly tested on Huawei's performance testing framework, which included the top 100 popular applications, with workloads including startup, stress testing (simulated random screen touches at a high rate), and normal usage simulations, on all supported devices. Overall, we have found Hubble's overhead is statistically insignificant in real-world use-cases. Hubble tracing is now enabled by default in all Huawei testing frameworks.

We have designed a few experiments to stress test and study Hubble's runtime characteristics, aiming to answer four questions: (1) What is the runtime cost of Hubble's tracing?

(2) What is Hubble's effect on cache behavior and memory bandwidth? (3) What is Hubble's overhead in the most demanding real-world scenarios? (4) How long of an execution trace can be stored in the ring buffer? We did not evaluate power consumption. Despite best-effort attempts, we could not reliably observe battery overhead in any experiments. Huawei's devices are shipped with aggressively tuned power-saving profiles and thus far, we have not observed an increase in reports of battery drain.

Unless otherwise specified, experiments were performed on a Google Pixel 1 phone that is well-supported by the open-source version of Android (AOSP). The phone contains a Qualcomm Snapdragon 821 processor with two high-performance cores each with a 64 KB L1 (divided equally for instructions and data) and 1.5 MB L2 cache, and two low-power cores each with a 64 KB L1 and 512 KB L2 cache.

We compared three execution modes: (1) baseline – the phone running *unmodified* Android; (2) tracing off – Hubble is enabled and applications are instrumented, but tracing is turned off; and (3) tracing on. Baseline experiments were performed on AOSP's android-10.0.0_r2 [56] branch. We recompiled the same branch with Hubble enabled.

Hubble's overhead could only be measured reliably in CPU-intensive and unrealistic microbenchmarks. Repeatedly running the two microbenchmarks in §8.1 and §8.2 causes the CPU to quickly reduce its clock speed due to severe thermal throttling. To improve the validity and reproducibility of the experiments, we placed the phone on bags of ice water.

8.1 Trace Point Overhead

Hubble's tracing overhead is amortized by the amount of work performed by the traced method. Since Hubble's tracing logic does not impose any dependencies on the traced method, nor does it use synchronization primitives on the critical path, the amortization effect will be enlarged by the deeper CPU pipeline. We evaluated both the cost of an individual trace point as well as the overall runtime overhead as the method performs more work. For comparison, we also evaluated Android's built-in method tracing utility, typically invoked via Android Studio's CPU profiler, henceforth referred to ASMT.

Listing 1 shows the method used. The amount of work done can be controlled through the `work` parameter. To prevent the method from being inlined by the JIT compiler, we added tail-recursion on line 5. In addition, we executed the method with a `depth` of `10` since the compiler still performs inlining at lower depths. `sum` is carried across calls to ensure that the loop is not optimized away by dead code elimination.

We ran the method with `work` values of 0, 1, 10, 100, and 1,000. We measured the runtime of two billion iterations. The cost of a trace point is calculated as the overhead of the 0-work experiment divided by two, since each method call contains a method-entry and method-exit trace point. To ensure the method is compiled by the JIT compiler before evaluation, we ran the experiment until its runtime stabilized to a maximum variance of five percent. The method is then executed ten times for each experiment.

		Average Cost (ns)	Standard Deviation (ns)	Performance Overhead (%)
ASMT	**32-bit**	3,911.575	59.2450	920,587%
Tracing ON	**64-bit**	3,366.050	57.8026	748,510%
Hubble Method	**32-bit**	0.725	0.0551	171%
Tracing ON	**64-bit**	0.650	0.0023	145%
Hubble Method	**32-bit**	0.001	0.0030	0%
Tracing OFF	**64-bit**	0.008	0.0027	2%

Table 2: Cost of a Single Trace Point

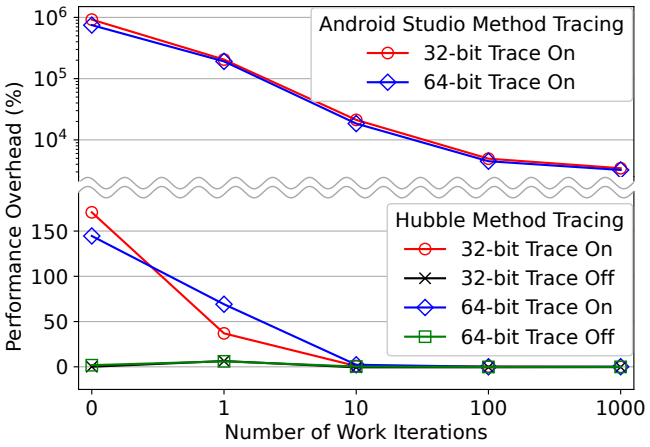

Figure 5: Performance Overhead Over Work Iterations

```
1 public long Test(int depth, int work, long sum) {
2   for (int i = 0; i < work; i++) {
3     sum *= i - 1; sum /= i + 2;
4   }
5   if (depth > 1) return Test(depth - 1, work, sum);
6   return sum;
7 }
```

Listing 1: Program used for measurement.

Table 2 shows the results of the 0-work experiment, with the other work values in Figure 5. The 0-work experiment shows that on average, each Hubble trace point costs less than one nanosecond when tracing is on, and less than 10 picoseconds when tracing is off. This is far less than ASMT's overhead which is on the order of microseconds. Figure 5 shows the amortization effect: as the amount of work done by the method is increased, Hubble's tracing overhead percentage decreases quickly. Note that in reality, small methods like this would likely be inlined, excluding them from being traced.

8.2 Cache Effects Microbenchmark

We used matrix-multiplication (MM) to measure Hubble's effects on the cache. MM is a classic workload that can either benefit heavily from caching or suffer ample cache misses [8]. When multiplying large matrices, a naïve implementation

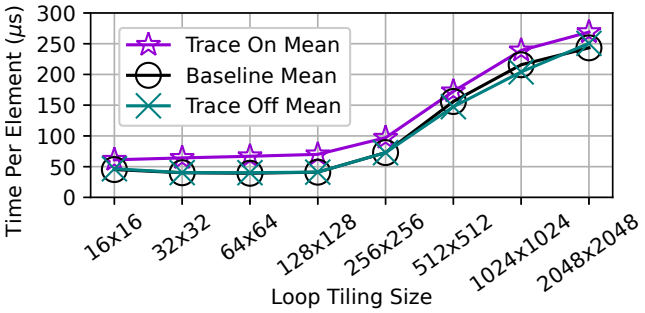

Figure 6: Cache and Memory Effects

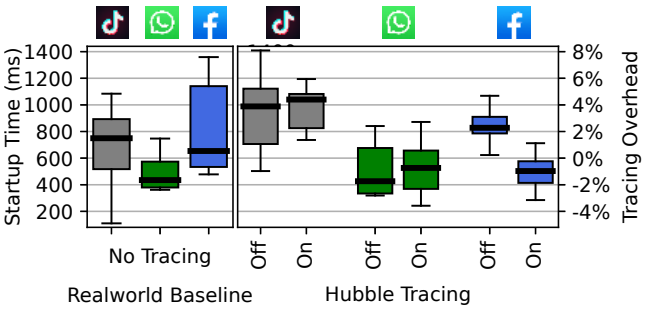

Figure 7: Application Startup Time

causes many unnecessary cache misses. However, the majority of these cache misses can be avoided using loop-tiling, i.e., partition each matrix into many small tiles, where each fits in the cache, and perform all accesses on one tile before moving on to the next. We examined Hubble's effect on each level of the cache by gradually increasing the tile size.

We evaluated Hubble's effect on MM with eight different tile sizes: 16×16, .., 2048×2048. The input matrices are 2048×2048, and each element is a four-byte integer. This means tile sizes 64×64 and below fit within the L1 cache; tile sizes 256×256 and below fit within L2; and all remaining tile sizes exceed both cache levels. To evaluate the highest amount of interleaved memory-contention that Hubble may have with MM, we performed each multiply and add operation inside a method such that two trace points are produced for each step of MM. We also inserted a dummy tail recursion call so that the JIT compiler does not inline the method. For each tile size, we ran the experiment five times. We did not compare with ASMT because it was too slow.

Figure 6 shows the results. With Hubble's tracing turned off, we could not reliably observe any overhead. With Hubble's tracing turned on, for the smallest tile size that fits within the L1 cache, Hubble has a min / max / mean overhead of 41% / 70% / 54%. When the tile size still fits within the L2 cache at 128×128, the overhead increased slightly to a min / max / mean of 64% / 83% / 70%. Finally, when the tile size is much larger than the L2 cache, caching is no longer effective. In this region, the increased execution time when tracing is turned on did not deviate significantly from smaller tile sizes, but the amortized overhead decreased.

Thus, in the absolute worst case scenarios, Hubble indeed affects programs heavily optimized for caching and, to some extent, memory-bound programs. However, in practice, similar small methods invoked in a tight loop would be inlined and excluded from tracing, not to mention that such loop-tiling is unlikely to be used in an application's UI thread.

8.3 Startup Overhead Macrobenchmark

We measured Hubble's overhead on application startup, one of the most demanding but realistic workloads for a method tracing tool since it comprises hundreds of thousands of method calls in a short period of time. These methods perform data loading and processing to prepare the application's UI and are often optimized to ensure the application loads quickly [22].

Since the performance of the application startup process varies significantly in practice, we took additional measures to minimize variation across benchmark runs. Specifically, we ran all experiments while disconnected from the network, eliminating variance introduced by network connections. We launched the target application repeatedly until its startup time stabilized to within a maximum variance of 5% (without these measures, the normal variance can be as much as 100% as shown on the left hand side of Figure 7). Each application was launched programmatically, avoiding any extraneous touch input that would occur with manual interactions. The startup time was obtained from a syslog message that indicates the duration from when the application process launched to the time after the application's UI has been drawn on the screen. To force cold starts (where the application starts completely unloaded), we manually killed each application before starting it again. Furthermore, we performed tests in quick succession to encourage the scheduler to place the application process on the performance-oriented CPU core operating at the maximum clock speed.

We ran the benchmark on the three applications that had the most downloads in 2020 [6]: TikTok, WhatsApp, and Facebook. The results are presented as a box and whisker chart on the right hand size in Figure 7. As the figure shows, the measured startup times vary considerably. To determine if Hubble causes a statistically significant difference in application startup time in our tightly controlled test environment, we performed two single-tailed dependent (paired sample) t-tests with a significance level of 5%. The t-test on the results of tracing turned off produced a p-value of 14.25% and the t-test of tracing turned on produced a p-value of 33.18%, both of which exceed the 5% threshold. Thus, we cannot conclude that Hubble causes a statistically significant difference in application startup time. In contrast, ASMT increased the average startup time of the three applications by approximately 10 times.

Although application startup overhead fluctuates significantly under real world scenarios, the number of methods executed remains nearly constant. When disconnected from

the network, TikTok, WhatsApp, and Facebook filled 6.0 MB, 3.8 MB, 6.4 MB of Hubble's ring buffer respectively; this corresponds to roughly 400,000, 250,000, and 420,000 methods invocations. When connected to the internet, the ring buffer content increased to 14 MB, 5.1 MB, and 11 MB because the applications loaded the user's content. In all three applications, the 32 MB of ring buffer proved to be more than sufficient to capture the entire application startup sequence. In Huawei's Hubble deployment, the 32 MB trace buffer is able to store the duration of almost all application startup and intermittent performance anomalies that our support engineers have encountered.

The results of the macrobenchmark were also in-line with results from our automated performance-regression testing, as well as feedback from support engineers and application developers. Recall that in part one of Huawei's three-phase deployment process (§4.1), we ran automated tests across a large fleet of devices and any significant statistical deviation in the results will prevent a new build from being deployed. In the automated performance-regression tests, we measured the application startup-time (both cold and warm startup) of the 100 most-downloaded third-party applications in addition to all our own applications. We categorized startup times into increments of 500 ms and count the number of applications that fall in each increment. After Hubble's deployment, we have not recorded any statistically-significant changes in the number of applications in each bucket for both cold and warm startup times.

The choice of 500 ms may seem high; however, Farrer *et al.* showed that users do not feel any loss of control (i.e., that an application is not responding to their action) until the response times reach approximately 350 ms [19], and users feel like they have completely lost control when response times exceed approximately 750 ms. Thus, our QA teams (and others [43]) have found that 500 ms increments are a good categorization to qualitatively evaluate loading speed—response times below 500 ms are considered excellent, 500–1000 ms is considered good, and above one second is considered slow.

9 Experiences

Hubble was shipped in the production branch of Huawei's Android system in August, 2020. An early prototype was merged into the main development branch in 2019, and engineers have been using it since. Huawei also runs a beta program where users can receive new features before public release. There are currently tens of thousands of beta users, and Hubble is enabled on their daily-use devices. For other end users, Hubble can only be enabled with their express consent.

The trace collection frequencies and retention policies vary depending on the type of users, the level of consent granted, operating region and local regulations, and device model. Internal beta users may not have any data upload restrictions. However, there are often additional restrictions on public users (including those beta users that are outside of Huawei). A common policy is that each user device can upload at most three traces per week. Which three traces to upload is configurable. For instance, sometimes there is a targeted campaign to improve specific applications, so in that case, only traces of anomalies for those applications are uploaded; other times we collect traces for anomalies whose symptoms are extremely severe; or, in the default case, we collect the first three anomalies detected. Although three traces is a low threshold, with a large user base, we are usually able to collect one or a few traces for each important issue.

Besides debugging production issues, Hubble is equally useful for debugging problems discovered during automated testing. Before Hubble, developers used ASMT to debug performance regressions, but due to its overhead it could only be enabled when debugging. This is cumbersome, and many problems simply could not be reproduced while debugging or worse, new issues would appear with ASMT enabled. Now, whenever a performance regression is detected, Hubble's traces are automatically collected, helping developers quickly narrow down the root cause without reproducing the issue.

A happy accident of implementing the tracing in assembly was that we discovered a bug in ARM's reference design on an older CPU model. While optimizing and testing the tracing assembly on a large number of devices, we found that when a specific permutation of 32-bit assembly instructions is used together with the Generic Timer counter, a segmentation fault could occur on the out-of-order performance cores. The bug was confirmed by the chip design team and fixed in later CPU models. On the buggy CPU model, we work around the issue by using an `ISB` instruction to flush the CPU pipeline after fetching the Generic Timer counter.

10 Concluding Remarks

Call profilers are known to be useful in debugging, however, their use has been limited to the development environment as a result of their overhead. Hubble shows that by leveraging Android's on-device compilation process, a just-in-time flushing strategy, and together with careful system-level design and engineering, we can achieve a highly efficient tool that can collect fine-grained call traces even in production environments. Hubble has proved its usefulness by significantly easing engineers' postmortem debugging processes.

Acknowledgements

We thank our shepherd Jonathan Mace and the anonymous reviewers for their insightful comments. Adrian Chiu provided help for us to understand the internals of a language runtime and its JIT compiler. This research was supported by a contract between Huawei and University of Toronto.

References

[1] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the 19th Symposium on Operating Systems Principles*, SOSP '03, pages 74–89. ACM, October 2003.

[2] Apple Inc. *Diagnosing Issues Using Crash Reports and Device Logs*, May 2021. https://developer.apple.com/documentation/xcode/diagnosing-issues-using-crash-reports-and-device-logs.

[3] Arm Limited. *AArch64 Programmer's Guides: Generic Timer*, August 2019. https://documentation-service.arm.com/static/600eb3264ccc190e5e68023a.

[4] Arm Limited. *Arm® Architecture Reference Manual*, July 2021. https://documentation-service.arm.com/static/611fa684674a052ae36c7c91.

[5] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for Request Extraction and Workload Modelling. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, OSDI '04, pages 259–272. USENIX Association, December 2004.

[6] Adam Blacker. Worldwide & US Download Leaders 2020. January 2021. https://blog.apptopia.com/worldwide-us-download-leaders-2020.

[7] Brendan Gregg. *Linux perf Examples: 4.3 JIT Symbols (Java, Node.js)*, July 2020. https://www.brendangregg.com/perf.html#JIT_Symbols.

[8] Randal E. Bryant and David R. O'Hallaron. Computer Systems: A Programmer's Perspective. chapter 6, pages 615–629. Pearson, 2nd edition, 2011.

[9] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic Instrumentation of Production Systems. In *Proceedings of the 10th USENIX Annual Technical Conference*, USENIX ATC '04, pages 15–28. USENIX Association, June 2004.

[10] Jong-Deok Choi and Harini Srinivasan. Deterministic Replay of Java Multithreaded Applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, SPDT '98, pages 48–59. ACM, August 1998.

[11] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation*, OSDI '14, pages 217–231. USENIX Association, October 2014.

[12] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. REPT: Reverse Debugging of Failures in Deployed Software. In *Proceedings of the 13th Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 17–32. USENIX Association, October 2018.

[13] Datadog. Cloud Monitoring as a Service. https://www.datadoghq.com/.

[14] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything You Always Wanted to Know about Synchronization but Were Afraid to Ask. In *Proceedings of the 24th Symposium on Operating Systems Principles*, SOSP '13, pages 33–48. ACM, November 2013.

[15] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, OSDI '02, pages 211–224. USENIX Association, December 2002.

[16] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. Execution Replay of Multiprocessor Virtual Machines. In *Proceedings of the 4th International Conference on Virtual Execution Environments*, VEE '08, pages 121–130. ACM, March 2008.

[17] Úlfar Erlingsson, Marcus Peinado, Simon Peter, and Mihai Budiu. Fay: Extensible Distributed Tracing from Kernels to Clusters. In *Proceedings of the 23rd Symposium on Operating Systems Principles*, SOSP '11, pages 311–326. ACM, October 2011.

[18] Facebook, Inc. Profilo - An Android Performance Library. https://facebookincubator.github.io/profilo/.

[19] Chlöé Farrer, G Valentin, and Jean-Michel Hupé. The Time Windows of the Sense of Agency. *Consciousness and Cognition*, 22(4):1431–1441, December 2013.

[20] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-trace: A Pervasive Network Tracing Framework. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation*, NSDI '07, pages 271–284. USENIX Association, April 2007.

[21] 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 *Proceedings of the 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 103–116. ACM, October 2009.

[22] Google LLC. *App Startup Time*, April 2021. https://developer.android.com/topic/performance/vitals/launch-time.

[23] Google LLC. *Firebase Performance Monitoring*, April 2021. https://firebase.google.com/docs/perf-mon.

[24] Google LLC. *Inspect CPU activity with CPU Profiler*, May 2021. https://developer.android.com/studio/profile/cpu-profiler.

[25] Google LLC. *Overview of System Tracing*, May 2021. https://developer.android.com/topic/performance/tracing.

[26] Google LLC. *Simpleperf Profiling Tool: JIT Symbols*, September 2021. https://android.googlesource.com/platform/system/extras/+/ec8d549d4c4300dcfb4e12353eccbeba17bf7725/simpleperf/doc/jit_symbols.md.

[27] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A Call Graph Execution Profiler. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, pages 120–126. ACM, June 1982.

[28] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. Failure Sketching: A Technique for Automated Root Cause Diagnosis of In-production Failures. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 344–360. ACM, October 2015.

[29] Dongyoon Lee, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Chimera: Hybrid Program Analysis for Determinism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 463–474. ACM, June 2012.

[30] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. Respec: Efficient Online Multiprocessor Replayvia Speculation and External Determinism. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 77–90. ACM, March 2010.

[31] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 378–393. ACM, October 2015.

[32] Gabriel Marin, Alexey Alexandrov, and Tipp Moseley. Break Dancing: Low Overhead, Architecture Neutral Software Branch Tracing. In *Proceedings of the 22nd Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '21, pages 122–133. ACM, June 2021.

[33] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. Towards Practical Default-On Multi-Core Record/Replay. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 693–708. ACM, April 2017.

[34] Mozilla. Mozilla Crash Reporter, May 2021. https://support.mozilla.org/en-US/kb/mozillacrashreporter.

[35] Ravi Netravali and James Mickens. Reverb: Speculative Debugging for Web Applications. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, pages 428–440. ACM, November 2019.

[36] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering Record and Replay for Deployability. In *Proceedings of the 2017 USENIX Annual Technical Conference*, USENIX ATC '17, pages 377–389. USENIX Association, July 2017.

[37] Oracle Corporation. *ThreadLocal (Java Platform SE 7)*, December 2020. https://docs.oracle.com/javase/7/docs/api/java/lang/ThreadLocal.html.

[38] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *Proceedings of the 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 177–192. ACM, October 2009.

[39] *Perf Wiki*, June 2020. https://perf.wiki.kernel.org/index.php/Main_Page.

[40] Perfetto - System profiling, App Tracing and Trace Analysis. https://perfetto.dev/.

[41] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. AppInsight: Mobile App Performance Monitoring in the

Wild. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation*, OSDI '12, pages 107–120. USENIX Association, October 2012.

[42] Lenin Ravindranath, Jitendra Padhye, Ratul Mahajan, and Hari Balakrishnan. Timecard: Controlling User-Perceived Delays in Server-Based Mobile Applications. In *Proceedings of the 24th Symposium on Operating Systems Principles*, SOSP '13, pages 85–100. ACM, November 2013.

[43] Raygun. *Real User Monitoring Performance Metrics*, May 2022. https://raygun.com/documentation/product-guides/real-user-monitoring/for-web/performance-metrics/.

[44] New Relic. New Relic®. https://newrelic.com/.

[45] Michiel Ronsse and Koen De Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM Transactions on Computer Systems*, 17(2):133–152, May 1999.

[46] Kedar Sadekar. Netflix Engineering Blog: Scalable Logging and Tracking. June 2012. https://netflixtechblog.com/scalable-logging-and-tracking-882bde0ddca2.

[47] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical report, Google, Inc., April 2010.

[48] Richard L. Sites. Datacenter Computers - Modern Challenges in CPU Design. Video, February 2015. https://vimeo.com/121396406.

[49] Joel Spolsky. How Microsoft Lost the API War. June 2004. https://www.joelonsoftware.com/2004/06/13/how-microsoft-lost-the-api-war/.

[50] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. In *Proceedings of the 10th USENIX Annual Technical Conference*, USENIX ATC '04, pages 29–44. USENIX Association, June 2004.

[51] Steven Rostedt. *ftrace - Function Tracer*, July 2017. https://www.kernel.org/doc/Documentation/trace/ftrace.txt.

[52] SystemTap. https://sourceware.org/systemtap/.

[53] *Tai Chi*, May 2020. https://taichi.cool/doc/.

[54] Leland Takamine and Brian Attwell. Introducing Nanoscope: An Extremely Accurate Method Tracing Tool for Android. April 2018. https://eng.uber.com/nanoscope/.

[55] Jiang Tenglicheng. Logan: Meituan Open Source Mobile Terminal Basic Log Library. October 2018. https://tech.meituan.com/2018/10/11/logan-open-source.html.

[56] The Android Open Source Project. Android 10.0.0 Release 2, 2019. https://android.googlesource.com/platform/build/+/refs/tags/android-10.0.0_r2.

[57] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. DoublePlay: Parallelizing Sequential Logging and Replay. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 15–26. ACM, March 2011.

[58] Stephen Yang, Seo Jin Park, and John Ousterhout. NanoLog: A Nanosecond Scale Logging System. In *2018 USENIX Annual Technical Conference*, USENIX ATC '18, pages 335–350. USENIX Association, July 2018.

[59] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. Log20: Fully Automated Optimal Placement of Log Printing Statements under Specified Overhead Threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 565–581. ACM, October 2017.

[60] Gefei Zuo, Jiacheng Ma, Andrew Quinn, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. Execution Reconstruction: Harnessing Failure Reoccurrences for Failure Reproduction. In *Proceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2021, pages 1155–1170. ACM, June 2021.