Memory Management on Mobile Devices

Kunal Sareen*
kunal.sareen@anu.edu.au
Australian National University
Australia

Stephen M. Blackburn
steveblackburn@google.com
Google and Australian National University
Australia

Sara S. Hamouda
hamos@google.com
Google
Australia

Lokesh Gidra
lokeshgidra@google.com
Google
United States

Abstract

The performance of mobile devices directly affects billions of people worldwide. Yet, despite memory management being key to their responsiveness, energy efficiency, and cost, mobile devices are understudied in the literature. A paucity of suitable methodologies and benchmarks is likely both a cause and a consequence of this gap. It also reflects the challenges of evaluating mobile devices due to: i) their inherently multi-tenanted nature, ii) the scarcity of widely-used open source workloads suitable as benchmarks, iii) the challenge of determinism and reproducibility given mobile devices' extensive use of GPS and network services, iv) the complexity of mobile performance criteria.

We study this problem using the Android Runtime (ART), which is particularly interesting because it is open sourced, garbage collected, and its market extends from the most advanced to the most basic mobile devices available, with a commensurate diversity of performance expectations. Our study makes the following contributions: i) we identify pitfalls and challenges to the sound evaluation of garbage collection in ART, ii) we describe a framework for the principled performance evaluation of overheads in ART, iii) we curate a small benchmark suite comprised of widely-used real-world applications, and iv) we conduct an evaluation of these real-world workloads as well as some DaCapo benchmarks and a micro-benchmark. For a modestly sized heap, we find that the lower bound on garbage collection overheads vary considerably among the benchmarks we evaluate, from 2 % to 51 %, and that overall, overheads are similar to those identified in recent studies of Java workloads running on OpenJDK. We hope that this work will demystify the challenges of studying memory management in the Android Runtime. By doing so, we hope to open up research and lead to more innovation in this highly impactful and memory-sensitive domain.

CCS Concepts: • **Software and its engineering → Garbage collection**; • **Human-centered computing → Ubiquitous and mobile devices**.

Keywords: Garbage Collection, Memory Management, Android, Android Runtime

ACM Reference Format:
Kunal Sareen, Stephen M. Blackburn, Sara S. Hamouda, and Lokesh Gidra. 2024. Memory Management on Mobile Devices. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on Memory Management (ISMM '24), June 25, 2024, Copenhagen, Denmark.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3652024.3665510

1 Introduction

Despite there being over 15 billion mobile devices worldwide [49], there is a paucity of studies of the performance of mobile systems, and in particular garbage collection for mobile systems. The explanation is not a lack of pressing challenges. To the contrary, memory management is a primary performance concern for mobile runtimes [12, 13]. Rather, we suggest that the lack of studies reflects the deep challenges mobile devices present to sound performance evaluations.

Unfortunately, well-developed server performance evaluation methodologies [21, 32] do not readily translate to mobile devices. Mobile devices are increasingly heterogenous, with multi-tiered asymmetric cores as well as hardware accelerators for common tasks such as video decoding [34]. This heterogeneity complicates performance evaluation. Mobile devices are inherently multi-tenanted with multiple applications, hardware, etc. all vying for the same set of constrained resources. Furthermore, applications typically connect to and interact with the network to provide core functionality: be it GPS services or a server on the internet. Such a complex and non-deterministic environment is not conducive to principled performance evaluation given such sources of experimental noise. These issues are further compounded by a lack of widely-used open source benchmarks. Without a standard benchmarking suite, researchers often use

*This work was, in part, done during an internship at Google DeepMind.

ISMM '24, June 25, 2024, Copenhagen, Denmark

ACM ISBN 979-8-4007-0615-8/24/06
https://doi.org/10.1145/3652024.3665510

microbenchmarks or create ad hoc benchmarks to evaluate performance, leading to inconsistent results and difficulty in comparing results across different studies.

We delve into this problem in the context of Android, the most popular mobile operating system (as of Q4 2023) [61], focusing on the Android Runtime (ART), its language virtual machine (VM). ART is particularly interesting given it is open-source, garbage collected, and widely used over a variety of different hardware configurations.

In this paper we make the following key contributions:

- We identify challenges and pitfalls associated with performance evaluation of garbage collection on mobile devices in general and in ART in particular and describe how we overcame them.
- We create a framework for principled performance evaluation of overheads in ART for both vanilla Java workloads such as DaCapo [20] as well as real-world Android applications. We open-source this framework.
- We curate a small set of popular real-world applications and mock user interactions with them to serve as representative workloads for mobile devices.
- We evaluate the overheads of production garbage collectors in ART by using the methodologies and framework we developed. For a modestly sized heap, we find that the lower bounds on garbage collection overheads for the production GCs vary from 2 % to 51 %.

2 Background and Related Work

To provide context for the remainder of the paper, we first give an overview of relevant aspects of the Android Runtime (ART) which we use throughout the paper. We then discuss the state of the art in performance evaluation of garbage collectors and the design of benchmark suites, which provide background to Section 3 and Section 4 respectively. We conclude this section with a discussion of related work.

2.1 Android Runtime (ART)

The Android Runtime (ART) is the language virtual machine (VM) on which Android applications run. Android applications are written in Java or Kotlin and compiled to Dalvik Executable (DEX) bytecode which is executed on ART [9]. Applications may call native code, and are sometimes largely native with only a minimal Java or Kotlin wrapper. Android supports a subset of Java features and standard library.

Applications and Vanilla Java and Kotlin. As well as running Android applications, vanilla Java (or Kotlin) programs can be run on ART so long as ART's language features and standard libraries support them. Vanilla Java programs do not require the Android framework, so can be run on a more minimalist environment which, as we shall explain, makes them methodologically more straightforward. By contrast, Android applications depend on the Android framework running which complicates performance evaluation.

Application Startup. When a user starts an application, rather than directly creating a new ART instance, an application startup request is sent to a system process called the *Zygote* [16]. The Zygote is an ART instance with important libraries and classes preloaded. In order to service the application startup request, the Zygote forks itself (using copy-on-write semantics) and then starts the user-requested application in the new process. Hence, the Zygote is the parent of all Android application processes. The benefits of such a mechanism are twofold: i) application startup is fast and efficient: each application starts with a set of common preloaded libraries and classes instead of having to load them at startup; and ii) the Zygote enables resource sharing of seldom written-to data and greatly reduces the memory footprint due to its copy-on-write nature. The use of the Zygote presents methodological challenges since all ART instances inherit the same bootstrap arguments, and thus use the same garbage collector and the same heap size. Since all ART instances inherit the Zygote's heap size arguments, the Zygote must be started with a sufficiently generous heap to accommodate the most demanding of the ART instances that will run. We discuss this further in Section 3.

Compiler. ART uses a tiered interpreter and just-in-time (JIT) compiler as well as an ahead-of-time (AOT) compiler [11]. Applications can be AOT-compiled to reduce startup and memory costs. The AOT compiler can use profile data that is either generated locally by the JIT compiler, or downloaded from the Google Play Store [8]. AOT-compiled code is preferred whenever available [11].

Garbage Collectors. Responsiveness is critical for mobile devices. Android has two production garbage collectors (as of Android 14 QPR1): a generational concurrent copying collector [4] and a new concurrent mark compact collector [25, 33]. All GCs in ART use an unconditional card-marking write barrier to track pointers from the bootimage into the moving spaces, pointer updates during concurrent marking, and old-to-young pointers in the concurrent copying collector. The concurrent copying collector uses a Baker-style read barrier [45] to ensure the mutator sees a consistent heap-state. The new concurrent mark compact collector is in the style of the Compressor [46]. Android also has a simple semi-space collector which is invoked over the Zygote space before new child processes are forked, to reduce fragmentation.

Collection is singled threaded. A dedicated GC thread (the *HeapTaskDaemon*) performs concurrent collection. However, if the heap becomes exhausted because the concurrent collector is unable to keep up with allocation, the first mutator that is unable to allocate will perform collection.

Operating System and Multi-Tenancy. Android runs on a modified Linux kernel, with each application forked from the Zygote process. Android applications can exist in the foreground or background. A background application may

be frozen with its heap compressed to free up RAM, or it may be killed if necessary to provide resources for another higher priority application [12].

One of the hallmarks of rigorous performance evaluation is that the system being evaluated is measured in *isolation*. This is a challenge in the mobile setting since mobile devices are deeply multi-tenanted. For example, it is not possible to evaluate an Android application in complete isolation since the Zygote and essential services must always be running.

Hardware. Mobile devices are by nature relatively resource-constrained in terms of memory, available CPU cores, and power consumption. This amplifies the importance of the time-space tradeoff made by garbage collectors. It can also lead to application developers aggressively managing resources, for example using introspection to assess resource utilization and then responding by resizing software caches maintained by the application. The use of such introspection to implement bespoke resource management can introduce feedback loops and complicates performance analysis and debugging. Mobile devices are increasingly heterogenous, with many modern phones having three-tiered asymmetrical cores (i.e. big-middle-little cores) and have specialized hardware for certain workloads such as video encoding/decoding [34]. Mobile devices come in many diverse hardware configurations. Controlling for heterogeneity adds an additional challenge to developing sound methodology.

2.2 Performance Evaluation in Java

Performance evaluation for Java is mature, with a sub-literature of methodology papers and four major benchmark suites [20, 59, 63, 64]. Core elements of Java performance evaluation are summarized by Blackburn et al. [21] and Georges et al. [32]. By contrast, performance evaluation for Android is the subject of few papers and there are no standard benchmark suites consisting of real-world applications. Closing those gaps are two of the key contributions of this paper.

Reproducibility and Statistical Significance. The most basic methodological technique is to control for variance. This is more difficult with Java than it was for C and Fortran workloads that came before them due to Java's dynamic compilation and garbage collection [21]. A variety of techniques exists to control for the effects of dynamic compilation, including warming the workload up and using replay compilation. Controlling for heap size is also essential since a program given a larger heap will have less garbage collection work. Above all, experiments must be repeated and the statistical significance of any result needs to be reported and accounted for in any conclusions drawn. As we will discuss, this is more difficult in a mobile setting such as Android.

Time-Space Tradeoff. Garbage collection makes a time-space tradeoff. This makes it necessary to evaluate garbage collected systems at different heap sizes. Although this is fairly straightforward for Java, it is significantly more challenging for Android, mainly because the heap size limits are determined by the Zygote and inherited by all applications.

Lower Bound Overheads. We use the LBO (lower bound overhead) methodology described by Cai et al. [22] to estimate garbage collection overheads in ART. The idea is that if an ideal collector were to exist, then the cost of any real collector could be established by comparing it to the ideal. Although the ideal collector does not exist, an upper bound on ideal performance can be established by subtracting easily-attributable collector costs from the performance of a real collector, which Cai et al. call the *distilled* cost. The lowest distilled cost among different GCs is then the closest approximation to the ideal, so is used as a baseline to estimate an empirical lower bound on the absolute costs of different GCs.

The LBO methodology has four key requirements: i) workloads that can be reliably and repeatably measured; ii) a means of cheaply and reliably measuring metrics of interest, such as wall clock time or performance counters; iii) the ability to accurately attribute costs to the garbage collector; and iv) a baseline collector that has few collector costs that are not easily attributable.

2.3 Benchmarks and Benchmark Suites

Benchmark suites are an essential research tool, facilitating systematic and controlled performance analysis. Java is supported by a relatively mature benchmark ecosystem, including the DaCapo and Renaissance open source suites [20, 59]. Android has many benchmark suites [19, 57, 58, 66, 67] but they primarily aim to test hardware performance or are microbenchmarks for tasks such as image decoding. Notably, all these benchmark suites: do not use real-world applications, are closed-source, and have opaque testing criteria.

Harnessing. A simple measure of performance might be to use the `time` command to run an application and then report its output. This will capture the application in its entirety, but the approach is limited since it does not allow the user to isolate initialization and setup from the substantive workload. For example, if we desire to measure transaction processing throughput, a server and database typically have to be initialized before the workload can commence and if we conflate initialization with transaction processing we will obscure the objective of our measurement. Suites like DaCapo *harness* the portion of the workload to be measured and allow the workload to be run multiple times within a single invocation, allowing the user to control for warmup.

While DaCapo reports the time for each iteration of the benchmark, it also provides a callback which allows the user to capture and report other metrics with respect to each benchmark iteration. The callback is invoked at the start and end of each benchmark iteration, allowing the user to start, stop, and report metrics of their choosing.

GC Cost Attribution. If the runtime is able to attribute costs to garbage collection, then by combining this with a benchmark callback, the user can attribute costs to the garbage collector on a per-iteration basis.

2.4 Related work

Most studies of application performance on Android either target application-level inefficiencies [1, 27, 29, 31, 36, 55], or the Linux kernel and associated services [35, 43, 47, 48, 51–53, 62, 69]. Surprisingly few studies look at understanding or improving the Android Runtime itself.

Hussein et al. [40, 42] propose a global memory management service for Android 4.4 that optimizes memory, performance, and power by collecting system-wide statistics and coordinating with the power manager to tune garbage collection scheduling. They model a variety of scenarios (such as sending an application into the background, sequentially executing multiple applications, etc.) in their evaluation methodology in order to best understand the effects of their global memory management service. They use a subset of the DaCapo benchmarks [20] and popular real-world applications such as Angry Birds and Spotify for their evaluations. They find that the global service can help reduce the number of applications being killed in the background, can improve the execution time and energy consumption for both foreground and background applications, and can reduce the memory consumption and number of garbage collection events significantly (10 % and 50 % respectively).

Lebeck et al. [50] propose 'Marvin', a memory management system for Android 7.1 that reduces background application kills by combining OS and language-level management. Android prefers to kill applications in the background when it is under memory pressure instead of writing the memory to flash since frequent write operations shorten the lifespan of storage devices [13]. Marvin uses a modified Linux kernel to target 'cold' objects (i.e. objects ≥ 2 KB that have not been accessed recently) in ART and checkpoints them to disk for faster memory reclamation under pressure. This requires the use of a bookmarking garbage collector [37] because otherwise swapped objects will be brought back into memory during GC time even if applications aren't using them. They find that Marvin can run more than twice as many concurrent applications than stock Android and can reclaim memory 60× faster than Android using a swap file under memory pressure for a set of microbenchmarks.

Recently, Huang et al. [38] propose 'Fleet', a memory management system for Android 10.0 to improve application (hot) launch speed and reduce background application kills. They combine OS and language-level memory management in the vein of Lebeck et al. [50]. Their core insight is that objects allocated when the application is in the foreground have a longer lifetime than objects that are allocated when the application is in the background. Using this, they optimize the background GC to avoid touching objects that were allocated in the foreground, by only collecting objects allocated in the background. Further, they group objects with similar access patterns and then place them in RAM or swap depending on their hotness akin to locality optimizations described by Huang et al. [39]. They find that Fleet achieves a 1.59× faster hot launch time and can cache 1.21× more applications than stock Android running with swap enabled.

3 Performance Evaluation on Android

As we have described in Section 2, sound evaluation of garbage collection overheads on Android is challenging. In this section we outline the major contributions of our work to the methodology of performance evaluation on Android, with a focus on understanding garbage collection overheads.

3.1 Controlling Heap Size

As discussed in Section 2.2, the ability to control the heap size is essential to evaluating performance in garbage collected languages. This is straightforward in Java (via the `-Xms` and `-Xmx` command line options), but is challenging in Android.

Because every application process is forked from the Zygote (Section 2.1), all applications share the same inherited startup parameters. Thus modifying Zygote startup parameters will change the heap size for *all* applications. If the Zygote is not given sufficient memory at startup, the phone may not boot, making it impossible to evaluate target applications in small heap sizes.

In order to control heap sizes at an application-level, we hijack a key functionality of the application specialization process. For a normal execution, after forking itself, the child Zygote process starts the application specialization process which includes "clamping" or "clearing" the *growth limit* and *capacity* of the application. The growth limit is a soft limit on the maximum heap size of an application, while the capacity is the hard limit. The growth limit is always ≤ the capacity of an application. For 'normal' applications, the capacity is clamped to the growth limit (thereby reducing the heap size), while for applications specified with a 'large' heap, the growth limit is cleared and set to the capacity (thereby increasing the heap size). We hijack this functionality.

We modify ART so that when it forks the Zygote to start a new application, it consults a metadata file listing heap sizes for applications. If an entry for the new application appears in the file, ART sets the growth limit and capacity of the application to the heap size found in the file, otherwise it uses the existing defaults. This mechanism allows us to change the heap size of a particular application without affecting the rest of the system.

As far as we know, no prior work has implemented such a system, making this the first work to systematically control for heap size in an evaluation of Android GC overheads.

3.2 GC Cost Attribution and Metrics

Attributing costs and using appropriate metrics to measure them is essential to performance analysis (Section 2.3). ART instruments its garbage collectors to collect timings of GC phases and stop-the-world GC pauses. We extend this by including hardware performance counters (such as CPU cycles, retired instruction counts, etc.) and software performance counters (such as number of page faults, task clock, etc.).

Our framework also collects *jank* and frame render timing metrics for each invocation. Jank is defined as the stuttering or choppy motion caused by skipped frames [15], a consequence of user-observed latency. We use `gfxinfo`, a command line tool, to gather these metrics [3]. These metrics reveal how GC algorithms, heap sizes, and other factors impact responsiveness, a core aspect of user experience.

We do not use metrics such as number of low-memory kills for background applications as we believe it is a proxy metric for understanding the efficiency of the memory manager. By controlling heap sizes, we can induce these low-memory situations directly to get a better picture of how each memory manager actually performs in such situations.

We do not use a framework such as ART TI [7] as it only allows agents to be attached to debuggable processes. Applications published on the Google Play Store are not marked as debuggable, hence, we cannot use ART TI agents for benchmarking real-world applications.

3.3 Feedback Loops due to Heap Introspection

A major challenge in benchmarking Android applications is their tendency to introspect their own heap usage. This introspection can dynamically alter the application's behavior, impacting the sound measurement of metrics as well as breaking the basic assumption that a benchmark be deterministic and perform the same amount of work on each invocation. Popular applications such as X (Twitter), Instagram, and Google Maps adjust their behavior based on available memory. This creates a problematic feedback loop: the application's workload becomes dependent on the heap size, which is an independent variable that we wish to control.

To address this issue, we modify ART to *mock* specific Java standard library APIs commonly used for heap introspection[1] By intercepting these calls, we present the application with a false (but consistent) view of the heap to avoid the application using these APIs to witness our changing of the actual heap size. That is to say, any tuning or resizing of caches that an application may perform is *invariant* to the actual heap size. We return pre-defined, constant values for the `maxMemory` and `totalMemory` APIs (512 MB and 128 MB respectively). For `freeMemory`, we return the minimum value between the actual free memory and the mocked `totalMemory` value. This ensures the reported free memory remains realistic from the perspective of the application.

We ran a sensitivity study for the applications that we mock heap usage for by measuring the execution times and GC overheads of eight different configurations of returned API values (including one where we do not mock the APIs) over nine different *actual* heap sizes. We found that the values we chose do not significantly impact the application's execution or increase GC overheads.

However, intercepting these APIs is not a complete fix since applications can still introspect heap sizes through alternative means. Even if we exhaustively go through all APIs and libraries to mock every instance of such APIs, applications can still directly obtain memory usage statistics themselves by using JNI. The best fix would be to patch application source code to remove their use of heap introspection. Unfortunately, this is not possible for the overwhelming majority of Android applications owing to their closed-source nature. To be pragmatic, we decided to use our pragmatic scheme instead of trying to conclusively fix this issue.

3.4 Evaluating Android Applications

Having addressed major challenges of benchmarking real-world applications, we now describe various other challenges we encountered when benchmarking Android applications.

Cached VM State. Closing an application on Android does not ensure that its process has exited. The application process is often kept around in memory to reduce startup times if the user relaunches the application [13]. This is problematic for us as a benchmark may start with incorrect heap size values due to the use of a previous application instance[2]. We ensure a cold start for each benchmark invocation [15] by killing any previous instance of the application[3] before starting the benchmark setup and workload. This guarantees that benchmarks do not inherit any cached state such as heap sizes from previous instances.

AOT- vs JIT-compilation. When installing an ART build that changes the GC algorithm, previously compiled AOT-code is discarded.[4] Following a system reboot, applications heavily rely on the JIT compiler for their execution. This can cause a significant increase in memory allocations as well as increasing experimental noise due to the JIT compiler. However, in steady-state, ART prefers using AOT-compiled code (Section 2.1). We gather JIT profiling data by running each benchmark 5 times with default parameters at the start of the experiment and then force a compilation of all the benchmark applications using the generated profile data. This ensures that AOT-compiled code is available and used, and hence, reduces allocations and experimental noise due to

[1] Namely the `Runtime.{max,free,total}Memory` APIs.

[2] For example, in the case where we have changed the heap size to a different value from what was previously specified in our metadata file.

[3] Using `SIGKILL`.

[4] For example, the semi-space GC does not use the read barrier and so previously compiled code with inlined read barriers must be discarded.

JIT compilation. Since we are primarily using AOT-compiled code, this obviates the need to run warmup iterations.

Network Traffic. Mobile applications often connect to remote servers to provide functionality to users.[5] However, this presents a challenge to sound performance analysis since the network traffic introduces a major source of non-determinism to the benchmarks. We tackle this in two ways. First, wherever possible, we try to limit the sources from which we fetch data. For example, in the case of a social media application such as TikTok, we only follow a select few accounts to ensure our feed does not change often.

Second, we run the benchmark with default settings once for each GC for every heap size we evaluate before starting the actual experiment. This allows the application to fetch and, ideally, cache up-to-date data, leading to more consistent performance measurements in subsequent runs. There is a tradeoff to make here between reducing the effects of non-deterministic network traffic and obtaining results within a reasonable time frame. Doing the above for every invocation ensures that each benchmarking run does not have to fetch a lot of data, but dramatically increases the time to obtain results. To achieve a balance, we chose to do this once per experiment. This allows for timely results, while slightly compromising on data freshness. Both approaches help us reduce non-determinism due to network traffic.

Deterministic record-and-replay of network traffic is generally not feasible for real-world applications. Most applications restrict the set of trusted CA certificates for security [14, 17], so it is not possible to use a man-in-the-middle proxy such as `mitmproxy` [26] to record-and-replay traffic. Further, server connections are likely to use timestamped messages and nonces for security against replay attacks [56, 70]. Overcoming such security measures requires non-trivial effort [60] or directly integrating into the application's logic during the record-and-replay process [54].

3.5 Evaluating Vanilla Java Workloads

For evaluating the performance of vanilla Java workloads, we adapt standard Java benchmarking methodologies [21] for Android. We use a standalone version of ART and utilize the `chroot`-based on-device testing [6] to run multiple ART instances on the same device without rebooting. Using a standalone ART instance allows us to control the heap size directly by using command line options. Since the vanilla Java workloads we evaluate do not require network access, we further reduce sources of experimental noise by running them with airplane-mode turned on.

Simple Baselines. A key requirement of the LBO methodology is that the baseline used to approximate the ideal application cost has few non-attributable GC costs (Section 2.2, [22]). Android's default production collectors are concurrent copying and compacting collectors which embody significant unattributable GC costs, such as concurrently marking and moving objects. We thus use ART's semi-space collector and implement a NoGC collector that simply allocates but does not collect anything to ensure a better baseline.

We apply three optimizations to the semi-space and NoGC collectors: i) we remove the need for remembered sets ('NR'), ii) we remove the write barrier ('NW'), and iii) we avoid eager return of pages to the OS ('NE'). The NR optimization already exists as a flag and simply involves shifting work to the GC. The NW optimization requires modifications to ART. This can affect correctness for the semi-space GC as it may miss pointer updates in the bootimage otherwise. However, we note that the simple vanilla Java workloads do not often write to the bootimage, and hence do not crash due to the NW optimization. The NE optimization clears unused and free pages instead of returning them back to the operating system. This does not affect correctness since Java workloads are short-lived and will return pages back to the OS when they exit. This can also dramatically improve mutator performance as each page has been touched at least once (due to the clear), reducing the number of page faults.

Unfortunately, the NE and NW optimizations do not allow us to run real-world applications. Real-world applications may update pointers in the bootimage, and hence we can run into correctness issues for the semi-space GC. Not returning pages back to the OS with semi-space makes the device restart frequently due to low memory. NoGC is also not a practical scheme on which to run the full Android OS.

4 A Suite of Android Benchmarks

Building and maintaining benchmarking suites is tedious and time-consuming and yet essential to making progress in our field (Section 2.3). While Java has major benchmark suites [20, 59, 63, 64] used extensively by both industry professionals and researchers, Android has no such suite. Due to the lack of a standard benchmark suite, researchers often tend to use ad hoc benchmarks which can introduce biases and reduce reproducibility. We now describe the design of a small benchmark suite and the challenges we faced in building it. The benchmark suite consists of popular real-world applications aimed at understanding and characterizing memory management performance on Android.

Using what we learned, we implement an extensible and modular framework for sound performance evaluation of Android applications. To the best of our knowledge, our work is the first open-source framework curating a set of real-world applications aimed at understanding application performance. Our benchmark framework and associated scripts are fully open-source and can be found at https://github.com/k-sareen/android-benchmark-runner.

[5] Only one application in our benchmark suite does not require a network connection.

Table 1. List of applications evaluated with their Google Play category, workload description, and version. Benchmarks marked with asterisks are the ones for which we had to mock APIs that queried heap sizes (see Section 3.3 for details). While TikTok also introspects heap sizes, we were able to disable this feature, and hence did not need to mock APIs for it.

Benchmark	Category	Description	Version
Adobe Acrobat	Productivity	Open PDF and jump to a chapter. Scroll 2 times and then search for some text. Scroll 3 times before returning to the start of the PDF.	24.1.0.30990
Airbnb	Travel & Local	Scroll and swipe pictures of listings 10 times.	24.06
Discord	Communication	Send messages (including one with media), react to messages, and then join a voice call for 7 seconds.	217.13
Gmail	Communication	Open newsletter and scroll to the end.	2024.01.28.605458019
Google News	News & Magazines	Scroll the "World" headlines tab 10 times.	5.100.0.604907407
Google Maps*	Travel & Local	Preview a route between two cities and go through each step.	11.115.0103
Instagram*	Social	Open Reels tab and scroll through 14 reels.	320.0.0.0.31
TikTok	Social	Open Following tab and scroll through 12 videos.	33.4.3
Twitch	Entertainment	Open Creator mode and stream for 30 seconds.	18.3.0
X (Twitter)*	Social	Open Following tab and scroll 15 times.	10.28.0

4.1 Scripting User Interactions

Most mobile applications are highly interactive. Therefore to deterministically execute representative Android workloads, we need to script UI interactions.

We use UI Automator [5, 18], an open-source library developed by the Android Open Source Project, to drive user interactions for our benchmark applications. It abstracts over device state such as the current foreground application, orientation of the device, etc. and was designed for end-to-end and cross-app testing. It allows us to search and interact with UI elements directly, for example, pressing buttons or scrolling posts. It also allows us to wait for UI animations to finish or until a particular UI element appears or disappears before sending subsequent user inputs.

Because UI Automator abstracts over the device state, we do not need access to source code for target applications in order to interact with them. This is in contrast to other UI interaction libraries such as Espresso [10] which require application source code to be available in order to script UI interactions. However, as a consequence, two ART processes run concurrently now: the application we're benchmarking and the benchmark harness running UI Automator code. This is not ideal, of course, as it can increase experimental noise, however it is the only viable solution without having access to application source code.

4.2 Harnessing

Section 2.3 introduced the importance of harnessing workloads when building a benchmark suite. This is straightforward in the case of the vanilla Java workloads we use since the DaCapo workloads are already harnessed and we have the source to the other workloads.

However, the Android applications are driven by UI Automator, which runs in a separate process to the workload under test. We solve this problem by having the controller process send signals to the process running the workload, and we modified ART to handle those signals and call benchmark start and end hooks accordingly.

4.3 A Small Benchmark Suite

We chose popular real-world applications from the Google Play Store and mock typical user interactions with them. We briefly describe each benchmark and categorize them in Table 1. Our benchmark set is representative of the most common activities and use-cases of mobile devices [23, 24]. However, we do not consider two notable use-cases: web browsing and gaming. We omit browsers (and other applications based on WebView such as Amazon Shopping, Wikipedia, etc.) from our benchmark suite because they are predominantly JavaScript applications, with a thin Android Java wrapper. We omit videos games for similar reasons since most games would either allocate into the native heap directly or use a game engine such as Unity [2, 68].

Unfortunately, all of the applications in our suite are closed-source. The closed-source nature precludes the ability to deeply investigate the workloads. The few open-source applications that we mocked, such as Wikipedia[6] and Breezy Weather[7], were not suitable benchmarks for understanding ART memory management performance due to their low allocation rates.

We mocked, but do not include in our analysis, many applications that turned out not to be sensitive to memory management performance such as Spotify, Google Photos,

[6] https://github.com/wikimedia/apps-android-wikipedia
[7] https://github.com/breezy-weather/breezy-weather

Table 2. List of vanilla Java benchmarks evaluated with a short description of their workload.

Benchmark	Description
lusearch	Multi-threaded text search over an input corpus [20].
pmd	Multi-threaded source-code analyzer [20].
xalan	Multi-threaded XSLT processor for transforming XML documents [20].
GCBench	Single-threaded microbenchmark testing allocation and garbage collection performance [30].

and Medium. This is not surprising. For example, Spotify's execution time is likely dominated by decoding audio streams.

4.4 Benchmarking Challenges

We describe the challenges involved in scripting user interactions and provide illustrating examples.

Flaky Benchmarks. While UI Automator provides better control over user interactions such as waiting for certain UI elements to appear or disappear, benchmarking Android applications can still be flaky. Dynamic content displayed by applications such as Facebook complicates both benchmark completion and stable benchmark results. A shifting content layout can result in different benchmark characteristics and allocation patterns. For example, videos displayed consecutively may cause allocation spikes which may result in the benchmark failing due to out-of-memory errors. Reddit and X (Twitter) also exhibit similar problems.

Furthermore, applications sometimes behave unexpectedly at small heap sizes. For example, X (Twitter) stops displaying images if the heap size is too small when running without mocked heap introspection APIs (Section 3.3), changing the benchmark workload. On the other hand, applications such as BBC News would, paradoxically, have spikes in the number of GCs for certain heap sizes much greater than its notional minimum heap size. This behavior was still visible even when using the mocked heap introspection APIs. Unfortunately, due to the closed-source nature of these applications, we were unable to investigate the cause of this issue (and many others) further.

Idempotent Benchmarks. An important requirement for sound performance evaluation is to have idempotent benchmarks. A previous benchmark execution should not affect the correctness or performance of future executions. This means that each benchmark needs to ensure that it finishes or exits exactly where it started. This is difficult to engineer for some benchmarks. For example, if the Adobe Acrobat application fails or crashes in a state where the PDF is open at an arbitrary page, then the application will always open the PDF at this page instead of the first page.

Application Idiosyncrasies. Many applications have their own idiosyncrasies that can be challenging to overcome. For example, while creating the Google Maps benchmark, we found that its minimum heap size seemed to monotonically increase every day due to it allocating an increasingly large `byte[]` in some JNI code. This was problematic for a variety of reasons. First, it meant that the benchmark workload was not constant. Second, it meant that the minimum heap size needed to be large enough to satisfy this single allocation, resulting in the heap size being too slack for the rest of the execution. Unfortunately, due to the closed-source nature of Google Maps, we were unable to determine the cause of this behavior. We were able to come up with a workaround, however. We uninstall and reinstall the Google Maps application for every ART build we benchmark, which is not ideal.

Fixed Application Versions. There is tension between having fixed application versions for the purpose of a benchmark suite and the update cycle and distribution of real-world applications. Obtaining specific application versions can be impossible since most app stores on mobile platforms do not allow downloading older versions of applications. Even if one is able to download and install an older version of an application, the servers it connects to may refuse connections for older application versions or the application itself may display pop-ups to prompt the user to update to the latest version[8]. Furthermore, the usefulness of a benchmark suite is reduced when, due to future updates to applications moving UI elements around, the benchmark ceases to function correctly. Unfortunately, there is no clean solution to this problem. Our hope is that the applications and workloads we have chosen are relatively stable so that breakages are not frequent or are easy to fix.

5 Methodology

We implement our work on ART commit `451cfcf`[9] and make it publicly available[10]. We use two kinds of benchmarks to evaluate the overheads of ART garbage collectors: vanilla Java workloads and popular real-world applications. We describe the Java workloads we use in this section. We have already discussed the real-world applications we use.

5.1 Vanilla Java Workloads

We evaluate a subset of the DaCapo 9.12 benchmark suite [20] ported by Hussein et al. [40, 41, 42] and GCBench [30] a simple microbenchmark testing allocation and garbage collection performance. We tried to port the latest DaCapo Chopin release [28], however, it proved challenging as Android does not implement elements of the Java standard library DaCapo Chopin requires. We briefly describe each benchmark and

[8] We experienced both when we were benchmarking applications.
[9] https://android.googlesource.com/platform/art/+/451cfcf9d09515ef60d76bd8551fc68c6e3bf621
[10] Available at https://github.com/k-sareen/art/tree/lbo-rebase.

Table 3. Hardware configurations we use for our evaluation.

Name	Cores	Clock	Memory
Pixel 7 Pro	2×2.85 GHz Cortex-X1, 2×2.35 GHz Cortex-A78, 4×1.8 GHz Cortex-A55	2.40 GHz, 1.99 GHz, 1.40 GHz	12 GB LPDDR5
Pixel 4a 5G	1×2.4 GHz Kryo 475 Prime, 1×2.2 GHz Kryo 475 Gold, 6×1.8 GHz Kryo 475 Silver	2.18 GHz, 1.90 GHz, 1.51 GHz	6 GB LPDDR4X

their workload in Table 2. We make a minor modification to the lusearch benchmark parameters for the 'default' size: the benchmark now executes 52 queries instead of the usual 64. This allows the benchmark to complete within the maximum permissible heap size on Android (around 900 MB) for NoGC.

While such vanilla Java benchmarks do not reflect real-world Android usage, they provide valuable insights into core garbage collection performance as well as allow us to compare against similar results for OpenJDK GCs.

5.2 Experimental Setup

We use two Google Pixel devices as described in Table 3, allowing us to observe the impact of microarchitecture and a 'lower-end' device configuration. Unless stated otherwise, we report results from the Pixel 7 Pro.

Each device runs a Lineage OS 21 build [44, 65] based on Android 14 QPR1. We use Lineage OS instead of an AOSP build as it allows us to have a more realistic environment and is easier to work with when trying to install Google Play Services. The Pixel 7 Pro and Pixel 4a 5G are running Linux kernel versions 5.10.177 and 4.19.282 respectively.

To control for variance due to thread scheduling, we pin benchmarks to the two biggest cores of each device. To ensure that pinning is respected, we implement a fix to the `Runtime.availableProcessors` API to correctly check for CPU masks. We also force each core to use a fixed frequency by using the 'performance' CPU governor. However, we do not pick the maximum frequency, instead we pick a frequency near the maximum in order to prevent thermal throttling. The frequencies we picked are listed in Table 3.

Since we use performance counters to gather metrics, we need to run the benchmarks as `root`[11] and disable SELinux to allow the `perf_event_open` and `read` syscalls to execute. While we do not run any background applications ourselves, certain applications such as Google Maps have a background service running all the time. We run all benchmarks with location services and bluetooth turned off. We also do not install SIM cards to both devices. These measures were taken to reduce variance due to network traffic.

We use a single iteration along with a pre-compiled bootimage for the vanilla Java workloads. The pre-compiled bootimage speeds up benchmark execution considerably as well as reduces allocations due to JIT compilation. NoGC is able to complete all of the vanilla Java workloads we have within the maximum permissible heap size on Android if run with a single iteration. We run a single iteration for Android applications as described in Section 3.4.

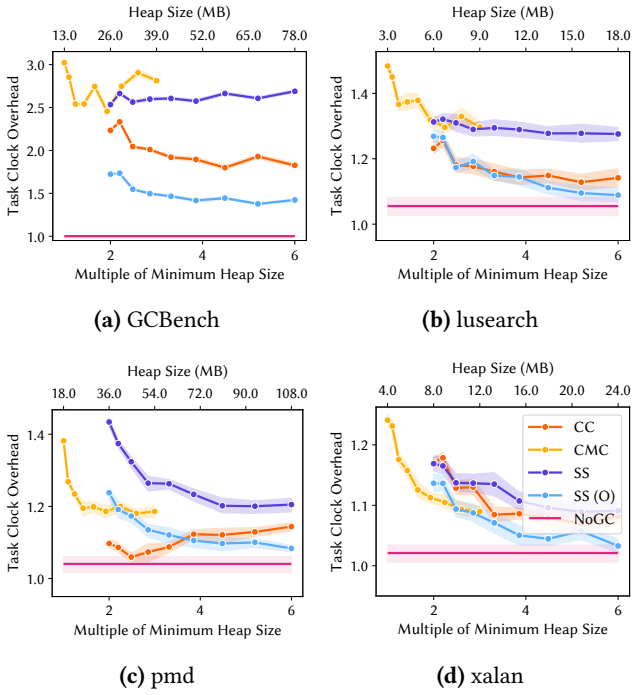
[11] Using `adb root`.

Table 4. Minimum heap values for each of the benchmarks. We gathered minimum heap values with the Concurrent-Copying collector and a bisection search.

	Heap Size (MB)	
Benchmark	Pixel 7 Pro	Pixel 4a 5G
Adobe Acrobat	18	18
Airbnb	23	23
Discord	19	19
Gmail	14	14
Google News	14	17
Google Maps	65	65
Instagram	34	39
TikTok	65	65
Twitch	17	17
X (Twitter)	24	24
GCBench	13	13
lusearch	3	3
pmd	18	18
xalan	4	4

We gather the minimum heap values for each of the benchmarks using the ConcurrentCopying collector, because it is the default production collector. Table 4 list the minimum heap values. Since we are interested in understanding the time-space tradeoff of the garbage collectors, we evaluate each GC over 9 multiples of the minimum heap size spread in a geometric progression, with values closer to the start than the end. We run each benchmark 15 times per multiple of the minimum heap size and report the average. The ConcurrentMarkCompact GC is only supported on newer devices, and hence we do not evaluate its performance on the Pixel 4a 5G. We also evaluate a SemiSpace collector without remembered sets (NR) (Section 3.5) for the real-world Android applications. However, this is not the same as the optimized SemiSpace for the vanilla Java workloads as its behavior is still similar to the stock SemiSpace.

There is an idiosyncrasy in the way ART sets heap sizes. In the JVM, the `-Xmx` command line argument sets the memory made available to the collector. This is inclusive of space the collector must hold in reserve for copying. A semi-space collector must hold in reserve half of the available memory for copying. On the other hand an in-place compacting collector does not require any copy reserve. However in ART, the command line argument dictates the maximum size of from space, not the total memory available to the collector. Consequently the memory made available to ConcurrentCopying and SemiSpace is twice the value specified on the ART command line. We could have corrected for this by modifying



(a) GCBench **(b)** lusearch

(c) pmd **(d)** xalan

Figure 1. Task clock overheads for the vanilla Java workloads on the Pixel 7 Pro averaged over 15 invocations. Each point on the x axis is a multiple of the minimum heap size for that benchmark. We plot a line for NoGC since it only runs in an 'unbounded' heap. Here, SS (O) is the SemiSpace (NR+NW+NE) build. Note how it improves over the stock SemiSpace implementation. NoGC is the baseline for GCBench, while SemiSpace (NR+NW+NE) is the baseline for the rest of the benchmarks.

the source code and made ART consistent with the JVM, but instead we correct for this by consistently reporting the available heap size, not the command line argument. The ConcurrentMarkCompact collector is unaffected since its from space is the entire heap.

6 Evaluation

6.1 Vanilla Java Workloads

Figure 1 plots the task clock overheads for the four Java workloads we evaluated. Here, *task clock* refers to the Linux performance counter `PERF_COUNT_SW_TASK_CLOCK` which sums the time spent by all threads running the application and excludes any time spent idling. This is interesting to us as it exposes the opportunity cost of concurrent GC [22]. Note how the optimized SemiSpace collector improves on the stock implementation across all benchmarks (and metrics we gathered). Overall, we see that the optimized SemiSpace collector either performs better or matches the ConcurrentCopying collector. ConcurrentMarkCompact generally has higher overheads than the two, while the stock SemiSpace

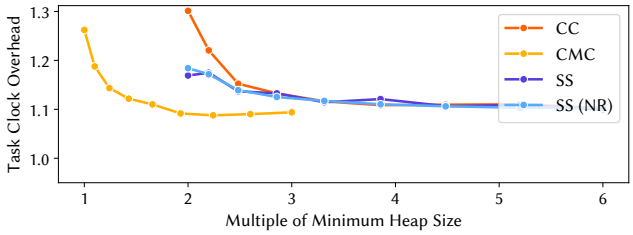


Figure 2. Geometric mean of task clock LBO across the Android application benchmarks on the Pixel 7 Pro. Each GC was ran 15 times. We remove X (Twitter) from the geometric mean calculations since it is not stable. For most of the benchmarks, the SemiSpace collectors minimize mutator overhead, with ConcurrentMarkCompact minimizing mutator overheads for Instagram.

collector has the highest overheads across all the benchmarks. The trends in the results are largely consistent with the ones from Cai et al. [22], however, the overhead values are generally lower.

Notably for pmd, ConcurrentCopying has lower task clock overheads at small heaps than for large ones (Figure 1c). We believe that this is caused by the benchmark actually benefitting from the frequent GCs which provide it better spatial locality. We confirm this by measuring CPU cache misses: the ConcurrentCopying collector minimizes cache misses even in comparison to the SemiSpace collector.

Ignoring GCBench, which is a microbenchmark, for a modest 2× heap, the optimized SemiSpace collector has overheads ranging from 14–27 %. ConcurrentCopying has overheads ranging from 10–23 % and ConcurrentMarkCompact has overheads ranging from 11–32 %.

Results for CPU cycles (not shown) are similar, while results for wall clock overheads (not shown) indicate the concurrent collectors can have lower overheads in comparison to the optimized SemiSpace (with the exception of GCBench). These trends are similar to the ones from Cai et al. [22] and further reinforces that wall clock time should not be considered on its own.

6.2 Real-World Applications

Figure 2 plots the results for the production collectors and the two SemiSpace collectors. We plot the geometric mean of the overheads across all of the benchmarks per collector and heap size. The two SemiSpace collectors are generally the ones which minimize overheads on the mutator, and hence are the baselines for LBO calculations. Wall clock time is noisy across all benchmarks. We believe this is due to the non-deterministic processing times in between user actions as well as the presence of two ART instances complicating scheduling decisions (Section 4.1). The task clock and CPU cycles are unaffected since they do not measure time the application is inactive. We now only discuss the task clock

Table 5. Task clock LBOs at a modest 2× heap for real-world applications. Best results are highlighted in green. X (Twitter) has been grayed out and excluded from summary statistics since it is too chaotic (see Figure 5 for details).

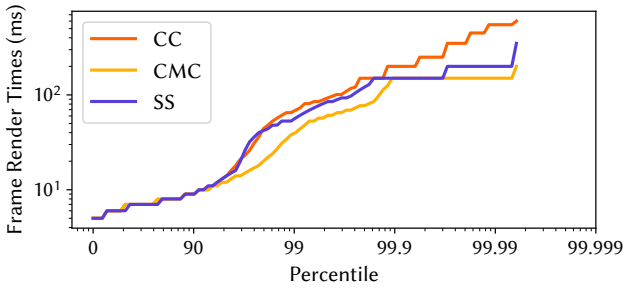
Benchmark	SS (Stock)	SS (NR)	CC	CMC
Adobe Acrobat	1.143 ±0.94 %	1.149 ±0.88 %	1.188 ±0.48 %	1.132 ±0.45 %
Airbnb	1.296 ±4.90 %	1.267 ±3.94 %	1.506 ±5.69 %	1.143 ±0.91 %
Discord	1.285 ±2.09 %	1.273 ±4.57 %	1.407 ±2.35 %	1.085 ±1.75 %
Gmail	1.169 ±2.58 %	1.157 ±1.85 %	1.370 ±4.87 %	1.061 ±1.28 %
Google News	1.159 ±4.74 %	1.172 ±3.39 %	1.309 ±5.28 %	1.097 ±1.63 %
Instagram	1.107 ±3.74 %	1.152 ±3.92 %	1.333 ±4.41 %	1.018 ±3.43 %
Maps	1.095 ±1.95 %	1.240 ±2.84 %	1.121 ±1.71 %	1.140 ±0.51 %
TikTok	1.187 ±3.48 %	1.172 ±4.05 %	1.349 ±3.99 %	1.135 ±2.40 %
Twitch	1.101 ±1.86 %	1.086 ±3.42 %	1.175 ±1.12 %	1.020 ±0.95 %
X (Twitter)	1.307 ±2.63 %	1.824 ±1.95 %	1.377 ±3.39 %	1.347 ±1.78 %
min	*1.095*	*1.086*	*1.121*	*1.018*
max	*1.296*	*1.273*	*1.506*	*1.143*
mean	*1.171*	*1.186*	*1.306*	*1.093*
geomean	**1.169**	**1.184**	**1.301**	**1.092**

results since the trends for CPU cycles are similar (CPU cycles has slightly higher overheads).

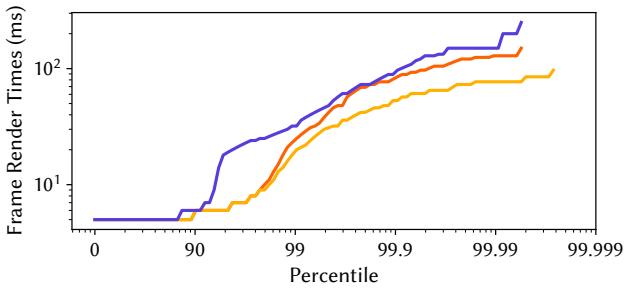
We can see in Figure 2 that the ConcurrentCopying collector has much higher overheads than the SemiSpace collector for small heaps (< 2.5×), while having similar overheads for larger heap sizes (≥ 2.5×). This is unsurprising since at tight heap sizes, the ConcurrentCopying collector would degenerate into a STW collector with the additional machinery required for concurrent copying such as read barriers, remembered sets, etc. ConcurrentMarkCompact also has higher overheads at small heap sizes, however, since it is more space-efficient than the ConcurrentCopying and SemiSpace collectors, its overhead is considerably less for any given heap size.

Table 5 lists task clock overheads for a modest 2× heap. The overheads for ConcurrentCopying range from 12–51 % and the overheads for ConcurrentMarkCompact range from 2–14 %. Since the ConcurrentMarkCompact collector is more space-efficient than ConcurrentCopying, these results are unsurprising. The SemiSpace collector without remembered sets (NR) has similar or lower overheads than stock SemiSpace, with the exception of Google Maps where it has significantly higher overheads across all heap sizes.

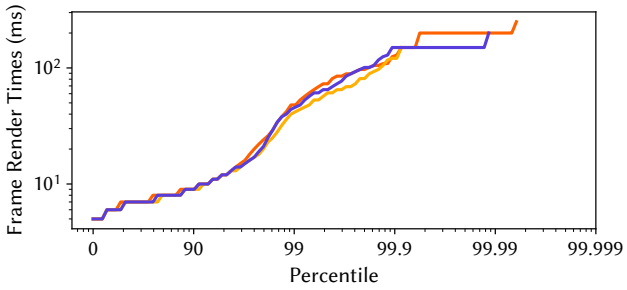
Overall, these overheads are not high, suggesting one of three things: i) the production GCs in ART are very efficient; ii) the baselines used for LBO are not close to the ideal application cost; or iii) these workloads are not particularly GC-intensive. For the vanilla Java workloads, in terms of mutator overheads, the stock SemiSpace actually had marginally better or similar results to the ConcurrentCopying collector. However, the SemiSpace collector with NR+NW+NE optimizations significantly minimized mutator overheads. This suggests that while the stock SemiSpace does reduce mutator overheads for Android applications in comparison to the concurrent collectors, it is not a good baseline. As the



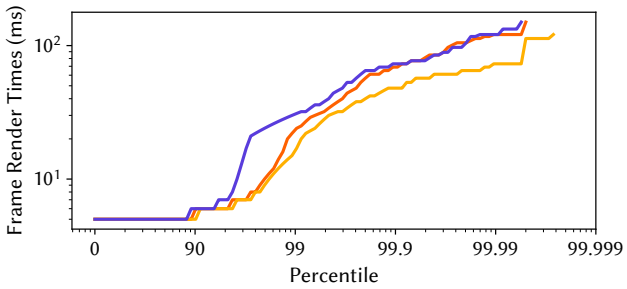
(a) Gmail, 2× heap



(b) Adobe Acrobat, 2× heap

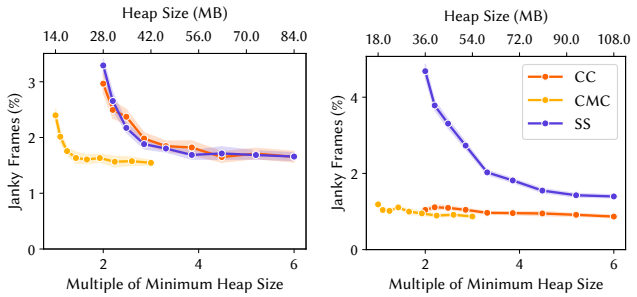


(c) Gmail, 3× heap



(d) Adobe Acrobat, 3× heap

Figure 3. Frame render times for 2× and 3× heap for Gmail and Adobe Acrobat. We omit SemiSpace (NR) for a cleaner graph as its results were similar to the stock SemiSpace. We gather all the frame render timings across 15 invocations and plot their percentile distribution. The standard deviation for the 50th and 99th percentiles across the 15 invocations for all collectors are < 1.0 ms and < 10.0 ms respectively for the two heap sizes. Note how SemiSpace and ConcurrentCopying have similar frame render times for Gmail, while ConcurrentMarkCompact generally has the lowest times.



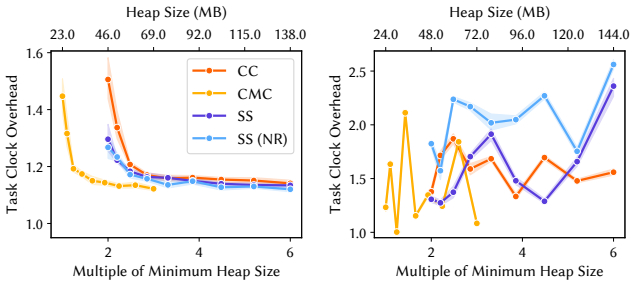
(a) Gmail **(b)** Adobe Acrobat

Figure 4. Percentage of janky frames for Gmail and Adobe Acrobat averaged over 15 invocations. Each point on the x-axis is a multiple of the minimum heap size for that benchmark. We omit SemiSpace (NR) for a cleaner graph as its results were similar to the stock SemiSpace. Note how SemiSpace and ConcurrentCopying have similar percentage of janky frames for Gmail, while ConcurrentMarkCompact improves upon the other collectors for equivalent heap sizes.

vanilla Java workload results show, it is not close to the intrinsic application cost. By definition, the LBO results are the lower-bound estimates of the true overheads.

Figure 3 plots the frame render times for a 2× and 3× heap for Gmail and Adobe Acrobat, while Figure 4 plots the percentage of janky frames. Gmail is a representative benchmark (Figures 3a and 3c). We see that with Gmail, ConcurrentCopying does not significantly improve frame render times or percentage of janky frames. On the other hand, Adobe Acrobat (Figures 3b and 3d) showcases a benchmark where ConcurrentCopying does improve on SemiSpace with respect to both measures. We believe that these results can be explained by the fact that ConcurrentCopying cannot keep up with mutator allocations for smaller heap sizes which negatively affects application responsiveness. This reinforces recent work [22, 71] on how collector pause times are not an accurate reflection of user-experienced responsiveness. ConcurrentMarkCompact generally has the lowest frame render times for similar heap sizes across all benchmarks.

Figure 5 plots the task clock overheads for Airbnb and X (Twitter). Airbnb is a representative benchmark, and behaves as expected, with smaller heap sizes having higher overheads than larger heap sizes. On the other hand, X's results are chaotic. There seems to be no visible trend for any of the collectors: larger heap sizes can have higher overheads than smaller ones. Yet the error bars are tight for all the collectors and heap sizes. This suggests that the results themselves are stable for a particular heap size, but not across different heap sizes. We believe such results are due to the effect of subtle differences in ordering of posts as well as different posts being displayed for each heap size as they execute at different moments in time. X's results underscore the difficulty in controlling and benchmarking Android applications soundly.



(a) Airbnb **(b)** X (Twitter)

Figure 5. Task clock overheads for Airbnb and X on the Pixel 7 Pro averaged over 15 invocations. Each point on the x-axis is a multiple of the minimum heap size for that benchmark. Note how Airbnb behaves as expected, while X is chaotic.

7 Conclusion

Understanding memory management performance is key to improving responsiveness, memory utilization, and costs of mobile devices. In this paper, we outline a framework for principled performance evaluation of overheads on mobile devices, with a specific focus on Android and the Android Runtime. While some of the challenges to sound performance evaluation we faced were specific to Android (such as the Zygote and AOT- vs JIT-compilation), we believe that much of our experience and many of our solutions might be applicable on other mobile platforms. In particular, the challenges we faced in curating a benchmarking suite (Section 4), feedback loops due to heap usage introspection (Section 3.3), and non-determinism due to network traffic (Section 3.4) seem applicable to any mobile platform.

We curate a small benchmark suite consisting of popular real-world applications as well as some DaCapo workloads [20] and GCBench [30]. We evaluate the lower-bound overheads of production garbage collectors in ART for our benchmark suite by applying the LBO methodology described by Cai et al. [22]. For a modestly sized heap, we find that the lower bounds on garbage collection overheads for production GCs vary considerably from 2 % to 51 % among the benchmarks we evaluate.

We believe our work is a promising step in demystifying memory management performance in the Android Runtime. We hope it leads to growing interest and innovation in memory management research for mobile devices.

Acknowledgments

This material is based upon work supported by the Australian Research Council under Grant No. DP190103367. We thank Hans Boehm, Zixian Cai, Nicolas Geoffrey, Roland Levillain, Martin Maas, Martin Stjernholm, and Jiakai Zhang for their insights and patience answering questions. We thank our anonymous reviewers for their constructive and encouraging feedback.

References

[1] Mohammad A. Alkandari, Ali Kelkawi, and Mahmoud O. Elish. 2021. An Empirical Investigation on the Effect of Code Smells on Resource Usage of Android Mobile Applications. *IEEE Access* 9 (2021), 61853–61863. https://doi.org/10.1109/ACCESS.2021.3075040

[2] Android Open Source Project. 2023. *Get started with game development in Unity.* Retrieved 2024-05-07 from https://developer.android.com/games/engines/unity/start-in-unity Archived at https://web.archive.org/web/20231208180248/https://developer.android.com/games/engines/unity/start-in-unity.

[3] Android Open Source Project. 2023. `dumpsys`. Retrieved 2024-03-06 from https://developer.android.com/tools/dumpsys Archived at https://web.archive.org/web/20240309134245/https://developer.android.com/tools/dumpsys.

[4] Android Open Source Project. 2024. *Android 8.0 ART improvements.* Retrieved 2024-03-07 from https://source.android.com/docs/core/runtime/improvements Archived at https://web.archive.org/web/20240307035338/https://source.android.com/docs/core/runtime/improvements.

[5] Android Open Source Project. 2024. *Android Jetpack.* Retrieved 2024-03-06 from https://android.googlesource.com/platform/frameworks/support/

[6] Android Open Source Project. 2024. *ART Chroot-Based On-Device Testing.* Retrieved 2024-03-06 from https://android.googlesource.com/platform/art/+/5937cdeeef4639e523ae3cfde3bfce3ccb252921/test/README.chroot.md

[7] Android Open Source Project. 2024. *ART TI.* Retrieved 2024-03-07 from https://source.android.com/docs/core/runtime/art-ti Archived at https://web.archive.org/web/20240307035559/https://source.android.com/docs/core/runtime/art-ti.

[8] Android Open Source Project. 2024. *Baseline Profiles overview.* Retrieved 2024-03-06 from https://developer.android.com/topic/performance/baselineprofiles/overview Archived at https://web.archive.org/web/20240306001211/https://developer.android.com/topic/performance/baselineprofiles/overview.

[9] Android Open Source Project. 2024. *Dalvik executable format.* Retrieved 2024-03-06 from https://source.android.com/docs/core/runtime/dex-format Archived at https://web.archive.org/web/20240228013516/https://source.android.com/docs/core/runtime/dex-format.

[10] Android Open Source Project. 2024. *Espresso.* Retrieved 2024-03-08 from https://developer.android.com/training/testing/espresso Archived at https://web.archive.org/web/20240308230026/https://developer.android.com/training/testing/espresso.

[11] Android Open Source Project. 2024. *Implement ART just-in-time compiler.* Retrieved 2024-03-06 from https://source.android.com/docs/core/runtime/jit-compiler Archived at https://web.archive.org/web/20240306000514/https://source.android.com/docs/core/runtime/jit-compiler.

[12] Android Open Source Project. 2024. *Low Memory Killer Daemon.* Retrieved 2024-03-06 from https://source.android.com/docs/core/perf/lmkd Archived at https://web.archive.org/web/20240306003006/https://source.android.com/docs/core/perf/lmkd.

[13] Android Open Source Project. 2024. *Memory allocation among processes.* Retrieved 2024-03-07 from https://developer.android.com/topic/performance/memory-management Archived at https://web.archive.org/web/20240307030326/https://developer.android.com/topic/performance/memory-management.

[14] Android Open Source Project. 2024. *Network security configuration.* Retrieved 2024-05-07 from https://developer.android.com/privacy-and-security/security-config Archived at https://web.archive.org/web/20240417222750/https://developer.android.com/privacy-and-security/security-config.

[15] Android Open Source Project. 2024. *Overview of measuring app performance.* Retrieved 2024-03-06 from https://developer.android.com/topic/performance/measuring-performance Archived at https://web.archive.org/web/20240305234427/https://developer.android.com/topic/performance/measuring-performance.

[16] Android Open Source Project. 2024. *Overview of memory management.* Retrieved 2024-03-06 from https://developer.android.com/topic/performance/memory-overview Archived at https://web.archive.org/web/20240306001637/https://developer.android.com/topic/performance/memory-overview.

[17] Android Open Source Project. 2024. *Security with network protocols.* Retrieved 2024-05-07 from https://developer.android.com/privacy-and-security/security-ssl Archived at https://web.archive.org/web/20240408142621/https://developer.android.com/privacy-and-security/security-ssl.

[18] Android Open Source Project. 2024. *Write automated tests with UI Automator.* Retrieved 2024-03-06 from https://developer.android.com/training/testing/other-components/ui-automator Archived at https://web.archive.org/web/20240306004012/https://developer.android.com/training/testing/other-components/ui-automator.

[19] AnTuTu. 2021. *AnTuTu Benchmark.* Retrieved 2024-03-07 from https://www.antutu.com/en/index.htm

[20] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, Peri L. Tarr and William R. Cook (Eds.). ACM, 169–190. https://doi.org/10.1145/1167473.1167488

[21] Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2008. Wake up and smell the coffee: evaluation methodology for the 21st century. *Commun. ACM* 51, 8 (2008), 83–89. https://doi.org/10.1145/1378704.1378723

[22] Zixian Cai, Stephen M. Blackburn, Michael D. Bond, and Martin Maas. 2022. Distilling the Real Cost of Production Garbage Collectors. In *International IEEE Symposium on Performance Analysis of Systems and Software, ISPASS 2022, Singapore, May 22-24, 2022.* IEEE, 46–57. https://doi.org/10.1109/ISPASS55109.2022.00005

[23] Laura Ceci. 2023. *Leading Android app categories worldwide 2019.* https://www.statista.com/statistics/200855/favourite-smartphone-app-categories-by-share-of-smartphone-users/

[24] Laura Ceci. 2024. *Most popular smartphone activities for global users 2022-2023.* https://www.statista.com/statistics/1337895/top-smartphone-activities/

[25] Seang Chau. 2022. *Android 13 is in AOSP!* Retrieved 2024-03-07 from https://android-developers.googleblog.com/2022/08/android-13-is-in-aosp.html

[26] Aldo Cortesi, Maximilian Hils, Thomas Kriechbaumer, and contributors. 2010–. mitmproxy: A free and open source interactive HTTPS proxy. https://mitmproxy.org/ [Version 10.3].

[27] Luis Cruz and Rui Abreu. 2017. Performance-Based Guidelines for Energy Efficient Mobile Applications. In *4th IEEE/ACM International Conference on Mobile Software Engineering and Systems, MOBILESoft@ICSE 2017, Buenos Aires, Argentina, May 22-23, 2017.* IEEE, 46–57. https://doi.org/10.1109/MOBILESoft.2017.19

[28] DaCapo Developers. 2023. *The DaCapo Benchmark Suite*. Retrieved 2024-03-12 from https://github.com/dacapobench/dacapobench/releases/tag/v23.11-chopin

[29] Teerath Das, Massimiliano Di Penta, and Ivano Malavolta. 2020. Characterizing the evolution of statically-detectable performance issues of Android apps. *Empir. Softw. Eng.* 25, 4 (2020), 2748–2808. https://doi.org/10.1007/S10664-019-09798-3

[30] John Ellis, Pete Kovac, Hans-J. Boehm, and Will Clinger. 1997. *An Artificial Garbage Collection Benchmark*. Retrieved 2024-03-12 from https://hboehm.info/gc/gc_bench.html

[31] Ruitao Feng, Guozhu Meng, Xiaofei Xie, Ting Su, Yang Liu, and Shang-Wei Lin. 2019. Learning Performance Optimization from Code Changes for Android Apps. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2019, Xi'an, China, April 22-23, 2019*. IEEE, 285–290. https://doi.org/10.1109/ICSTW.2019.00067

[32] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. (Eds.). ACM, 57–76. https://doi.org/10.1145/1297027.1297033

[33] Lokesh Gidra, Hans-J. Boehm, and Joel Fernandes. 2020. *Utilizing the Linux Userfaultfd System Call in a Compaction Phase of a Garbage Collection Process*. Technical Report. Google LLC. https://www.tdcommons.org/cgi/viewcontent.cgi?article=4751&context=dpubs_series

[34] Monika Gupta. 2021. *Google Tensor is a milestone for machine learning*. Google LLC. Retrieved 2024-03-06 from https://blog.google/products/pixel/introducing-google-tensor/

[35] Sangwook Shane Hahn, Sungjin Lee, Inhyuk Yee, Donguk Ryu, and Jihong Kim. 2017. Improving User Experience of Android Smartphones Using Foreground App-Aware I/O Management. In *Proceedings of the 8th Asia-Pacific Workshop on Systems, Mumbai, India, September 2, 2017*. ACM, 5:1–5:8. https://doi.org/10.1145/3124680.3124721

[36] Geoffrey Hecht, Naouel Moha, and Romain Rouvoy. 2016. An empirical study of the performance impacts of Android code smells. In *Proceedings of the International Conference on Mobile Software Engineering and Systems, MOBILESoft '16, Austin, Texas, USA, May 14-22, 2016*. ACM, 59–69. https://doi.org/10.1145/2897073.2897100

[37] Matthew Hertz, Yi Feng, and Emery D. Berger. 2005. Garbage collection without paging. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, Vivek Sarkar and Mary W. Hall (Eds.). ACM, 143–153. https://doi.org/10.1145/1065010.1065028

[38] Jiacheng Huang, Yunmo Zhang, Junqiao Qiu, Yu Liang, Rachata Ausavarungnirun, Qingan Li, and Chun Jason Xue. 2024. More Apps, Faster Hot-Launch on Mobile Devices via Fore/Background-aware GC-Swap Co-design. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir (Eds.). ACM, 654–670. https://doi.org/10.1145/3620666.3651377

[39] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J. Eliot B. Moss, Zhenlin Wang, and Perry Cheng. 2004. The garbage collection advantage: improving program locality. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, John M. Vlissides and Douglas C. Schmidt (Eds.). ACM, 69–80. https://doi.org/10.1145/1028976.1028983

[40] Ahmed Hussein, Mathias Payer, Antony L. Hosking, and Christopher A. Vick. 2015. Impact of GC design on power and performance for Android. In *Proceedings of the 8th ACM International Systems and Storage Conference, SYSTOR 2015, Haifa, Israel, May 26-28, 2015*, Dalit Naor, Gernot Heiser, and Idit Keidar (Eds.). ACM, 13:1–13:12. https://doi.org/10.1145/2757667.2757674

[41] Ahmed Hussein, Mathias Payer, Antony L. Hosking, and Christopher A. Vick. 2017. *Android Etalon – DaCapo*. Retrieved 2024-03-12 from https://github.com/fizous/etalondacapo

[42] Ahmed Hussein, Mathias Payer, Antony L. Hosking, and Christopher A. Vick. 2017. One Process to Reap Them All: Garbage Collection as-a-Service. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2017, Xi'an, China, April 8-9, 2017*. ACM, 171–186. https://doi.org/10.1145/3050748.3050754

[43] Sangoh Park Jaehwan Lee. 2021. Mobile Memory Management System Based on User's Application Usage Patterns. *Computers, Materials & Continua* 68, 3 (2021), 4031–4050. https://doi.org/10.32604/cmc.2021.017872

[44] Nolen Johnson. 2024. *Changelog 28 - Fantastic Fourteen, Amazing Applications, Undeniable User-Experience*. Retrieved 2024-03-11 from https://lineageos.org/Changelog-28/

[45] Henry G. Baker Jr. 1978. List Processing in Real Time on a Serial Computer. *Commun. ACM* 21, 4 (1978), 280–294. https://doi.org/10.1145/359460.359470

[46] Haim Kermany and Erez Petrank. 2006. The Compressor: concurrent, incremental, and parallel compaction. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, Michael I. Schwartzbach and Thomas Ball (Eds.). ACM, 354–363. https://doi.org/10.1145/1133981.1134023

[47] Sang-Hoon Kim, Jinkyu Jeong, and Jin-Soo Kim. 2017. Application-Aware Swapping for Mobile Systems. *ACM Trans. Embed. Comput. Syst.* 16, 5s (2017), 182:1–182:19. https://doi.org/10.1145/3126509

[48] Sang-Hoon Kim, Jinkyu Jeong, Jin-Soo Kim, and Seungryoul Maeng. 2016. SmartLMK: A Memory Reclamation Scheme for Improving User-Perceived App Launch Time. *ACM Trans. Embed. Comput. Syst.* 15, 3 (2016), 47:1–47:25. https://doi.org/10.1145/2894755

[49] Federica Laricchia. 2023. *Number of mobile devices worldwide 2020-2025*. Retrieved 2024-03-07 from https://www.statista.com/statistics/245501/multiple-mobile-device-ownership-worldwide

[50] Niel Lebeck, Arvind Krishnamurthy, Henry M. Levy, and Irene Zhang. 2020. End the Senseless Killing: Improving Memory Management for Mobile Operating Systems. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 873–887. https://www.usenix.org/conference/atc20/presentation/lebeck

[51] Cong Li, Jia Bao, and Haitao Wang. 2017. Optimizing low memory killers for mobile devices using reinforcement learning. In *13th International Wireless Communications and Mobile Computing Conference, IWCMC 2017, Valencia, Spain, June 26-30, 2017*. IEEE, 2169–2174. https://doi.org/10.1109/IWCMC.2017.7986619

[52] Changlong Li, Yu Liang, Rachata Ausavarungnirun, Zongwei Zhu, Liang Shi, and Chun Jason Xue. 2023. ICE: Collaborating Memory and Process Management for User Experience on Resource-limited Mobile Devices. In *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*, Giuseppe Antonio Di Luna, Leonardo Querzoni, Alexandra Fedorova, and Dushyanth Narayanan (Eds.). ACM, 79–93. https://doi.org/10.1145/3552326.3567497

[53] Yu Liang, Jinheng Li, Rachata Ausavarungnirun, Riwei Pan, Liang Shi, Tei-Wei Kuo, and Chun Jason Xue. 2020. Acclaim: Adaptive Memory Reclaim to Improve User Experience in Android Systems. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 897–910. https://www.usenix.org/conference/atc20/presentation/liang-yu

[54] Felipe Lima. 2017. *Writing fast, deterministic and accurate Android Integration tests.* Retrieved 2024-05-07 from https://medium.com/airbnb-engineering/writing-fast-deterministic-and-accurate-android-integration-tests-c56811bd14e2

[55] Rodrigo Morales, Rubén Saborido, Foutse Khomh, Francisco Chicano, and Giuliano Antoniol. 2018. EARMO: An Energy-Aware Refactoring Approach for Mobile Apps. *IEEE Trans. Software Eng.* 44, 12 (2018), 1176–1206. https://doi.org/10.1109/TSE.2017.2757486

[56] Okta. 2023. *What Is a Cryptographic Nonce? Definition & Meaning.* Retrieved 2024-05-07 from https://www.okta.com/au/identity-101/nonce/

[57] PassMark Software. 2021. *PassMark PerformanceTest Mobile.* Retrieved 2024-03-07 from https://www.passmark.com/products/pt_mobile/

[58] Primate Labs Inc. 2023. *Geekbench 6 – Cross-Platform Benchmark.* Retrieved 2024-03-07 from https://www.geekbench.com

[59] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tuma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: benchmarking suite for parallel applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 31–47. https://doi.org/10.1145/3314221.3314637

[60] Onur Sahin, Assel Aliyeva, Hariharan Mathavan, Ayse K. Coskun, and Manuel Egele. 2019. RANDR: Record and Replay for Android Applications via Targeted Runtime Instrumentation. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 128–138. https://doi.org/10.1109/ASE.2019.00022

[61] Ahmed Sherif. 2024. *Market share of mobile operating systems worldwide from 2009 to 2023, by quarter.* https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/

[62] Atikant Singh, Aakriti V Agrawal, and Anuradha Kanukotla. 2016. A method to improve application launch performance in Android devices. In *2016 International Conference on Internet of Things and Applications (IOTA)*. 112–115. https://doi.org/10.1109/IOTA.2016.7562705

[63] SPEC Standard Performance Evaluation Corporation. 2015. The SPECjbb 2015 benchmark. https://www.spec.org/jbb2015/

[64] SPEC Standard Performance Evaluation Corporation. 2015. The SPECjvm 2008 benchmark. https://www.spec.org/jvm2008/

[65] The LineageOS Project. 2024. *LineageOS Android Distribution.* Retrieved 2024-03-11 from https://lineageos.org

[66] UL Solutions. 2021. *PCMark Android Benchmark.* Retrieved 2024-03-07 from https://benchmarks.ul.com/pcmark-android

[67] UL Solutions. 2023. *3DMark Android Benchmark.* Retrieved 2024-03-07 from https://benchmarks.ul.com/3dmark-android

[68] Unity Technologies. 2024. *Unity Real-Time Development Platform | 3D, 2D, VR & AR Engine.* Retrieved 2024-05-07 from https://unity.com/

[69] Yong-Xuan Wang, Chung-Hsuan Tsai, and Li-Pin Chang. 2021. Killing Processes or Killing Flash? Escaping from the Dilemma Using Lightweight, Compression-Aware Swap for Mobile Devices. *ACM Trans. Embed. Comput. Syst.* 20, 5s (2021), 90:1–90:24. https://doi.org/10.1145/3477021

[70] Wikipedia Contributors. 2024. *Replay Attack.* Retrieved 2024-05-07 from https://en.wikipedia.org/wiki/Replay_attack Archived at https://en.wikipedia.org/w/index.php?title=Replay_attack&oldid=1193515559.

[71] Wenyu Zhao, Stephen M. Blackburn, and Kathryn S. McKinley. 2022. Low-latency, high-throughput garbage collection. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 76–91. https://doi.org/10.1145/3519939.3523440

Received 2024-03-22; accepted 2024-05-10