Reclaiming Energy: Quantifying the Energy Overheads of Garbage Collection on Mobile Devices

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

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

***Abstract*—Energy efficiency is a first-order concern for mobile devices. Although garbage collection is central to the Android platform, the energy costs of garbage collection are not well understood. This is likely due to major methodological hurdles: first, heterogeneous hardware and inherently multi-tenanted software; second, UI-induced waits confound classic performance analysis methodologies; and third, difficulty in measuring energy, which is often not supported by the device, and if it is, has low temporal resolution. These hurdles have clouded our understanding of the energy costs of GC, and may have led to missed opportunities for more energy efficient mobile software.**

The principal contributions of this paper are new methodologies and a study of the energy costs of garbage collection that uses them. We introduce: i) new insight into the deep problems that UI interactions bring to the performance analysis of Android applications; ii) methodologies that address the problem of cost attribution when the resolution of the measure is coarser than the events being studied; iii) methodologies that control for the hardware heterogeneity found on modern devices; iv) a methodology that emulates the multi-tenanted environment found on mobile devices; and v) an implementation that allows us to quantify the tradeoff due to parallelism in GC.

We perform these experiments using a suite of real-world applications, a subset of the DaCapo benchmarks, and a microbenchmark. We show: i) that waits introduced by UI interactions make performance analysis of Android applications deeply challenging; ii) that single-threaded GC typically outperforms parallel GC on mobile devices; iii) that garbage collection has a significant energy overhead on Android applications and is sensitive to GC algorithm and configuration; iv) that different subsystems' energy consumption is affected by GC choice and configuration. This work sheds new light on energy overheads on mobile devices and we hope may motivate new designs in this impactful domain.

I. Introduction

The literature on garbage collection is mature, extending back to the 1960s. While methodology has advanced considerably over that time, empirical evaluations focus squarely on throughput, latency, and memory efficiency, largely overlooking energy. However, energy use is a first-order concern for mobile devices and data centers [7], [51]. The literature also largely overlooks mobile devices, despite their ubiquity. The lack of studies can likely be attributed to the complexity of conducting sound, principled evaluation on these devices, especially concerning energy measurement [48].

We identify a number of major challenges to sound evaluation on mobile devices. i) A paucity of *suitable benchmarks*, which is at least in part due to the difficulty of creating realistic workloads. ii) The confounding impact of **UI-induced waits** in Android applications which undercuts assumptions that are key to classic performance analysis. iii) A difficulty of **cost attribution**, particularly when measuring energy, which may only be observable at a coarse temporal grain. iv) The difficulty of performing sound measures on the **complex, heterogeneous hardware** common on mobile devices. v) The necessity of modelling a realistic **multi-tenanted** environment. The first of these has been addressed in recent work [48], and though an ongoing challenge, it is not our focus. We address the others.

Most performance analysis is predicated on benchmarks that *hold work constant*—the workload is invariant with respect to the parameters being evaluated. We find that UI-induced waits within Android applications undermine this premise, significantly complicating performance analysis.

Attributing costs such as time, CPU cycles, or cache misses is straightforward for relatively coarse-grained events like GC pauses given the availability of low-overhead counters with fine temporal precision. However, other measures such as energy may, due to hardware limitations, only be available at temporal resolutions coarser than the studied event period. This appears to be a show-stopper, ruling out a study of energy costs or any other measure that suffers a similar challenge. In this paper we present two methodological insights that address this problem of cost attribution in the face of coarse-grained measures.

Mobile devices utilize heterogeneous hardware, including multi-tier asymmetric cores and accelerators [25]. Benchmarking is further complicated by non-determinism from frequent network requests and background system services. We develop methodologies to manage this complex environment and control for hardware asymmetry.

Finally, mobile devices are inherently multi-tenanted, with multiple applications running concurrently. In many domains, including much of the garbage collection literature, conventional methodology takes pains to ensure the subject of the evaluation runs in near-complete isolation. However, in the mobile setting, isolation is such a departure from real usage that it may undermine validity of the study. We present methodologies that model the effects of multi-tenancy.

We investigate these challenges in the context of Android, the most popular mobile operating system [53]. Since we are interested in GC energy overheads, we focus on the Android Runtime (ART) which is the language virtual machine (VM) that powers most applications on Android.

This paper makes the following key contributions:

i) We identify waits induced by UI interactions as a fundamental hurdle to performance analysis for Android applications (§ III-A).
ii) We introduce two methodologies for addressing the problem of cost attribution when the resolution of the measure is coarser than the events being studied, and use these new methodologies to measure the energy consumption of garbage collection (§ III-A, § III-B).
iii) We introduce methodologies that control for heterogeneous hardware environments (§ III-C).
iv) We introduce a methodology which allows us to model a realistic multi-tenanted environment (§ III-D).
v) We implement an optimized single-threaded GC and evaluate it w.r.t. highly-tuned parallel GC (§ III-E).
vi) We use these methodologies to conduct a detailed study of the costs of garbage collection, including energy costs.
vii) We use energy counters to understand energy costs at a per-subsystem level.

This work is a promising step in understanding the energy costs of memory management. We hope it motivates new designs in this highly impactful domain.

II. Background and Related Work

To provide context, we start with an overview of the state-of-the-art methodologies for GC performance evaluation. We then discuss the energy measurement system present on modern Android devices. Finally, we discuss related work, focusing on performance evaluation on mobile devices.

A. Performance Evaluation

There is an extensive literature on performance evaluation for garbage collection [15], [26], [28], [50], [59], [60], [63], and performance evaluation more generally [9], [12], [22], [42]. Relevant to our work is the problem of *cost attribution* and the LBO methodology introduced by Cai et al. [13]. A longstanding challenge has been to evaluate the cost of a phenomena, such as GC, which cannot (entirely) be measured directly. While standard methodologies allow us to precisely measure stop-the-world GC pauses that take O(100 µs), other aspects of the GC such as allocation sequences and write barriers may take just a few cycles and occur millions of time, so defy direct measurement. Costs associated with second-order effects such as cache displacement or changes in locality due to allocation policies are similarly difficult to measure. These difficulties make it challenging to accurately attribute all costs of GC.

The key insight of Cai et al.'s work is that if a hypothetical ideal GC existed, then a comparison between a system using that and a system using a real GC would reveal the cost of the real GC. Furthermore, they observe that it is practical to *approximate* the ideal, and a comparison against this baseline will yield a *lower bound* on the overhead of the real system.

Cai et al. noted that there were often multiple ways to approximate the ideal, and that by taking the lowest of these they could attain a tighter bound on the real cost of the GC being studied. Two ways they approximated the ideal were to use a NoGC configuration, that simply did not GC at all, and to subtract the easily measurable costs from executions using simple GCs. Interestingly, the poor locality of the NoGC configuration meant that it was often not the best baseline. When taking the subtractive approach, Cai et al. systematically evaluated all available GCs and used the lowest (best) cost one as the baseline. Simple GCs such as semi-space [17] tend to yield the lowest cost estimate because such collectors do not have write barriers etc. whose costs are hard to attribute. Using this approach, they found that even in favorable circumstances, the state of the art GC in OpenJDK was spending 14.8% of cycles performing GC and other production GCs were costing substantially more [13].

Unfortunately, when we applied the LBO methodology to the task of measuring the energy cost of GC in mobile devices, we were blocked due to significant challenges in establishing a good baseline. First, we found that the temporal resolution of energy performance counters was too coarse to apply the subtractive approach used by Cai et al. (§ III-A). Second, we found that due to their UI interactions, Android applications *do not exhibit constant work*, which undermines the most basic premise of such performance analyses (§ III-A). Third, we found that we typically could not use the NoGC approach due to insufficient memory on the mobile platform (§ III-B). Two of the contributions of this work are methodologies that allow us to sidestep these limitations, opening the way to measuring the real energy cost of GC in mobile devices.

B. Energy Measurement

Early work on mobile device energy analysis was limited to observing battery drain as a measure of energy consumption [16], [43] or using specialized external hardware measuring power draw [19], [29], [37]. However, modern Android devices come with a rich set of sensors called the *On-Device Power Rails Monitor* (ODPM) [2], [45] which give us a relatively detailed window into energy consumption. The ODPM system adds dedicated hardware to measure each power rail on the device. An important feature of the ODPM system is that the power rail sensors are downstream of the battery so they are unaffected by the charging or discharging of the battery [45]. This means that we can use the phone while externally powered and still get accurate energy measurements.

The ODPM system exposes a variety of energy measures to the user, including the display, CPUs, DRAM, system-level cache (SLC), etc. This gives us a never-before seen look into the energy use of managed language runtimes. The power rail counters are exposed as a `sysfs` file [34]. Each power rail counter measures the energy consumption of their subsystem from when the counters were started, which in our work is usually boot time. The energy measurements are provided as a monotonically increasing count of micro watt seconds. The difference between two consecutive readings yields the energy consumed during that period.

Unfortunately, the temporal resolution of the ODPM system is just O(20 ms), so it is unable to resolve the energy impact of events like a garbage collection, which typically take O(1 ms).

The reason for the coarse temporal resolution of ODPM is two-fold. First, the underlying implementation depends on sampling which is limited to 50 Hz. Second, we established empirically that each read of the counters takes 2–4 ms.

As mentioned above, Cai et al. use a subtractive approach to creating baselines for the LBO analysis. However, this approach is dependent on being able to accurately measure stop-the-world GC pauses, which while trivial when measuring cycles or wall clock time, is not possible when measuring energy due to the coarse resolution of ODPM. One of the contributions of this work is the use of linear regression to establish an LBO baseline when the subtractive approach is unavailable.

A further limitation of ODPM is that, like RAPL [18] and some other hardware performance counters, ODPM does not include kernel support for per-process measures, so all measures are system-wide [2]. This distinction is analogous to the difference between a measure of cycles (which is process-specific) and the wall clock (which is intrinsically a system-wide measure). Consequently, any energy measurement taken with ODPM will capture all energy use by all running processes over the period of measurement, complicating analysis. Mitigating this limitation is easier when the workload (such as a Java benchmark) can be run in isolation in a pared-back environment [3], [48], but is difficult when running a full-stack real-world workload such as an Android application.

C. Related Work

Principled performance evaluation on mobile devices is fraught with complexities such as their inherent multi-tenanted nature, hardware asymmetry, lack of benchmarks, etc. The lack of studies in the literature is not surprising.

We build upon recent work by Sareen et al. [48]. They identified pitfalls and challenges involved in conducting sound and principled performance evaluations on Android 14 and outline a set of recommendations and workarounds for each problem. We use their methodological recommendations and their set of benchmarks in our evaluations. While they focused on performance of GC, we extend this by evaluating the energy overheads of GC.

Hussein et al. [29], [31] measure the end-to-end energy use of benchmarks on Android 4.4 using a hall-effect sensor [14]. They find that changing the GC strategy can reduce the energy use of some benchmarks by 20–30%.

Recently, Li et al. [35], [36] measure the impact of multi-tenancy on the responsiveness of foreground applications. They find that the frames per second (FPS) of foreground applications decreases as the number of background applications increase. They attribute this slowdown to frequent pagefaults induced by background applications. We directly measure the impact on the foreground application using performance counters.

III. Estimating Baseline Energy Consumption

As we discussed in § II-B, we face two methodological hurdles in applying Cai et al.'s approach. The coarse resolution of the ODPM's measurements means that we cannot directly measure energy consumed by stop-the-world GC, which in turn means we cannot use Cai et al.'s subtractive approach to creating an LBO baseline (§ II-A). Also, the memory constraints of the mobile setting means that the NoGC approach does not work in many cases. We address the first methodological roadblock by applying linear regression to the problem of generating a baseline, allowing us to conservatively approximate the energy of an idealized system where GC has no cost (§ III-A), although as we explain below, Android Applications violate a premise of constant work which this approach depends upon. We address the second by introducing 'almost' NoGC (§ III-B).

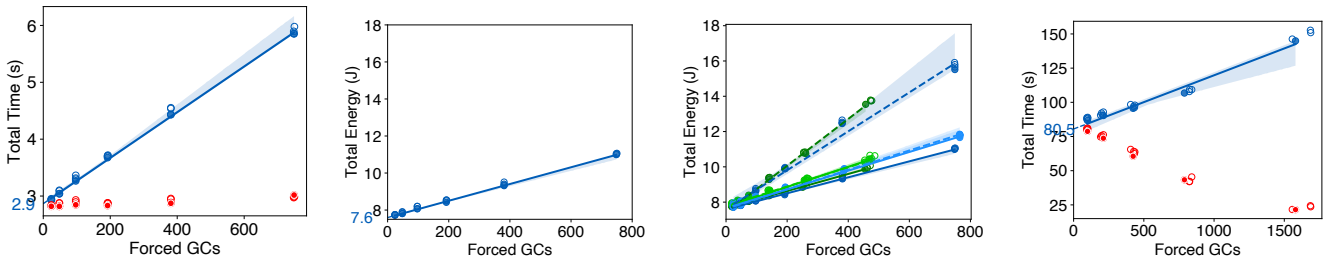
A. Baseline Estimation via Linear Regression

Our objective is to develop a methodology for approximating the total cost of a hypothetical system that uses a zero-cost GC. The most naïve approximation is simply the total time of the fastest running system. This approach may be adequate if the fastest running system has no GC overheads, like NoGC. However, we conduct this analysis for the command line applications and show that simply taking the fastest running system among the regular garbage collectors *overstates the ideal* by 11-17% compared to Cai et al.'s subtractive approach (§ II-A, [13]). Thus this simplistic approach tends to *understate overheads* by roughly 11-17%.

In trying to find a better approximation to the ideal, our constraints when measuring energy are that we can only take temporally coarse-grained measures, such as the total cost for the execution of a workload, and we cannot simply turn off the GC (§ III-B). Our insight is that we have a means of controlling the amount of GC work the system performs. For example, if we can measure the total cost of two systems, one with exactly twice as much GC as the other, we can calculate the difference in cost and thus infer the cost of GC. More generally, if we can perform the above for multiple different controlled amounts of GC, we could perform linear regression to infer the cost of a system with no GC, which would solve our problem.

Normally, a GC will be triggered according to a heuristic such as when the heap is full. However, the MMTk framework allows us to artificially trigger GCs using a 'stress factor', which will force a GC after 2^N bytes of allocation. When tracing GCs are triggered close together they will tend to perform the same amount of work because the amount of live data in the heap is approximately the same. Thus, making the simplifying assumption that the cost of each GC is approximately the same, we expect a linear relationship between the number of GCs triggered and the total cost of the GC work. This allows us to straightforwardly conduct a linear regression and reach an approximation to the hypothetical zero-cost GC.

Figure 1(a) illustrates the linear regression method with respect to *time*. Although our goal is to measure energy (Figure 1(b)), time is useful for the sake of exposition because we *can* measure time with high fidelity, so we can apply Cai et al.'s subtractive approach (§ II-A), allowing us to ground the result we derive from linear regression. In this example, we force periodic GCs at each of six GC stress factors (x-axis). For each of these we perform five executions, measuring total time (blue) and total time minus GC time (red). Because our



(a) xalan Time (b) xalan Energy (c) Multiple GC configurations (d) Google Maps Time

Fig. 1: **We use linear regression to estimate the cost of a system with a hypothetical zero-cost GC.** Each circle in the plots indicates the cost for a given number of forced GCs (*x*-axis). We perform five invocations and use the lowest value (solid circles). The other values are indicated with unfilled circles. We fit a line and take the *y*-intercept to estimate of the cost with zero GC. Since we can measure *time* with high fidelity, we can use Cai et al.'s subtractive approach [13], yielding ground truth, shown in red. The *y*-intercept in (b) indicates 7.6 J. Subfigure (c) repeats this analysis for eight GC configurations, from which we take the best (lowest), 7.4 J. Subfigure (d) repeats the analysis of xalan in (a) for Google Maps, yielding surprising results.

objective is to find the best approximation to the ideal, we discard the four least ideal (slowest) runs, and keep the fastest (solid circles) for each stress factor. We then perform linear regression and take the *y*-intercept as representative of the case when there are zero forced GCs, i.e. an approximation to the ideal GC-free system. In the example in Figure 1(a) linear regression yields a baseline of 2.87 s. The direct measures of time (red), subtracting stop-the-world GC pauses from total time, yield a best (lowest) estimate of 2.82 s. Linear regression thus yields a result that is within 2% of the direct measure.

Figure 1(b) applies the same approach to energy, where it is not possible to take a direct measure. In this case we again see that the relationship between cost (energy) and number of forced GCs is close to linear. The approach yields a baseline of 7.6 J for our hypothetical system with an ideal GC. We can repeat this for multiple GC configurations, using different GC algorithms and different core counts and affinities for the GC (Figure 1(c)). We take the lowest, yielding 7.4 J.

Unfortunately, when we apply this approach to Android applications, the results are counterintuitive (Figure 1(d)). Notice that the ground truth mutator time (red) *decreases* as the number of forced GCs goes up. This violates our premise that the mutator work will be invariant. This pattern is repeated on each of the Android applications, while all of the command line applications exhibit the expected behavior, like xalan (Figure 1(a)). We established that the source of this strange behavior is waits issued by the UI Automator scripts between activities. We can think of the total application time for these workloads as being the time actively executing the application plus the wait time. The former behaves as expected in the face of garbage collection—as GC increases, the total time increases. However, wait time by definition has the property of taking the same amount of time (regardless of GC load). Because GC time and mutator time must sum to equal the total wait time, this leads to the counterintuitive result that as the GC load increases the time attributable to the mutator shrinks. This leads to the situation we see in Figure 1(d), where mutator time decreases as GC load increases, undermining the most basic premise that the benchmark workload remains constant.

This finding introduces a profound challenge to understanding the costs of applications, like UI-driven Android applications, which naturally include the use of waits.

Since we cannot use linear regression or the subtractive approach, we must resort to the simplistic and overly conservative option of taking the lowest energy GC configuration as our best approximation to the ideal when evaluating energy on Android applications. **The findings at the start of this section indicate that by doing so we will likely understate energy consumption for the Android applications by 10-15%.**

B. Baseline Estimation via 'Almost' NoGC

In addition to the subtractive method (§ II-A), Cai et al. also used 'NoGC' to approximate the ideal system. NoGC is a GC configuration supported by MMTk (similar to Epsilon in OpenJDK [52]) that only performs fast bump pointer allocation, and no collection at all. Once the heap is exhausted, the program terminates. Although NoGC has no direct GC costs, it uses memory sparsely, leading to locality overheads which on some workloads dominate. Because there are no GCs and no subtraction is necessary, this approach works even with measures like energy which are unable to resolve the cost of individual collections. Unfortunately, NoGC requires a large heap to be useful, limiting its use in the mobile setting.

We note that many benchmarks execute multiple iterations of their workload before timing a final iteration, and that the memory demands of the last iteration may be significantly less than the memory requirement of the entire benchmark execution. With this in mind, we introduce 'almost' NoGC, which is a contrived use of the canonical semi-space algorithm. Before each benchmark iteration we force a GC, yielding an optimally compact heap. If the heap is large enough, the final iteration will complete without triggering a collection, much as the NoGC collector does. Because it performs GCs before each iteration, its memory requirements are somewhat less than NoGC. Because there are no GCs in the measured benchmark

TABLE I: **Energy LBO baseline calculation.** We use thirteen configurations of five garbage collectors to calculate up to twenty-one baseline candidates for each benchmark. The candidate closest to the ideal (lowest energy, green) is selected as the LBO baseline for that benchmark. Of the candidates, eight use linear regression (§ III-A, italics). The remainder are simple measures of total energy and are therefore very conservative. Linear regression is unavailable to Android applications (bottom five), and the native collectors (rightmost four). MMTk collectors Immix (IX) and SemiSpace (SS) are each evaluated with four core topologies. Native collectors, Concurrent Mark Compact (CMC) and Concurrent Copying (CC) are evaluated with two.

Benchmark	Minimum Calculated Energy per-Configuration (J)														
		IX				SS				NoGC	CMC		CC	
	Best	1×M	1×B	2×M	2×B	1×M	1×B	2×M	2×B	-	1×M	1×B	1×M	1×B
GCBench	0.152	*0.185* 0.253	*0.186* 0.272	*0.236* 0.314	*0.295* 0.398	*0.162* 0.220	*0.162* 0.229	*0.227* 0.307	*0.277* 0.382	0.152	0.367	0.380	0.285	0.308
lusearch	*3.620*	*3.783* 4.732	*3.784* 4.926	*3.620* 6.449	*3.804* 4.997	*3.694* 5.669	*3.624* 6.281	*3.928* 7.508	*3.671* 6.448	3.681	5.129	5.199	4.246	4.398
pmd	3.167	*3.432* 4.418	*3.480* 4.814	*3.550* 4.492	*3.498* 4.681	*3.177* 3.794	*3.207* 3.981	*3.485* 3.895	*3.298* 3.970	3.167	3.955	4.121	3.680	3.788
xalan	*7.408*	*7.661* 9.709	*7.719* 10.29	*7.408* 12.36	*7.788* 10.14	*7.596* 10.62	*7.659* 11.50	*7.600* 13.34	*7.621* 11.71	7.964	12.53	13.112	11.99	12.35
Adobe Acrobat	35.54	- 36.97	- 37.27	- 37.96	- 37.88	- 36.31	- 37.11	- 37.24	- 38.06	35.54	37.79	37.44	39.15	39.49
Google News	11.41	- 11.59	- 11.65	- 11.88	- 12.28	- 11.84	- 12.16	- 11.92	- 12.14	12.18	11.43	11.41	12.16	12.00
Google Maps	56.65	- 62.18	- 62.42	- 56.65	- 56.78	- 63.55	- 64.57	- 57.09	- 58.14	77.34	61.08	61.70	62.83	62.88
TikTok	39.45	- 41.38	- 41.01	- 40.74	- 41.11	- 41.68	- 41.95	- 41.28	- 40.36	39.79	41.27	39.45	41.27	41.67
X (Twitter)	32.39	- 32.39	- 32.87	- 32.72	- 33.22	- 32.94	- 33.89	- 33.80	- 33.66	33.85	33.00	33.06	33.45	33.28

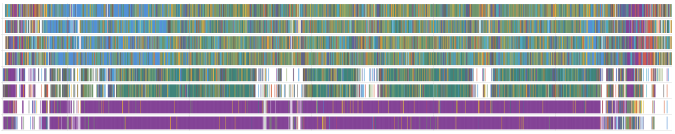
iteration, the approach works even with measures like energy which cannot resolve individual GCs.

C. Attribution and De-noising via Spatial Isolation

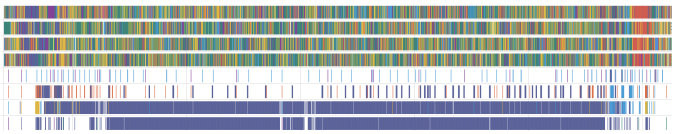
Cost attribution is complicated in multi-tenanted settings like Android. In normal use, there are typically many apps and services running at any time. Android has the concept of a *background* application, and will schedule them to the *background cpuset*, which may be the small cores, for example. However, applications deemed to be *jank sensitive* (such as music streaming) may be exempted, contending with the foreground app for CPU resources. System services are similarly exempted, creating a complex and noisy environment for performance evaluation.

A vanilla Java application does not depend on Android services, so it is not difficult to run such a workload with little interference. However, Android applications tend to make use of multiple services, so even if they are the only application running, it is hard to measure without interference. This is further complicated by the heterogenous hardware common on mobile devices, since the performance of the application will vary significantly depending on which CPU it is running on.

We address both problems by carefully constraining where applications and services may run. We isolate the application we wish to measure on our device's big and medium cores, forcing all other applications and services to run on the small cores. Furthermore, we pin the application being measured onto the big cores and optionally pin the GC threads on the medium cores. This gives us spatial isolation between the application being measured, the garbage collection threads, and all other applications and services. This isolation significantly reduces measurement noise because the extraneous services no longer contend for CPU resources. Since ODPM provides CPU energy measures per *core type*, this isolation also allows us to perform better attribution. For example, if only GC threads run on the medium core, we can attribute medium core energy use to the GC threads. Like many methodological techniques, using spatial isolation trades off realism for fidelity. Later we address this by introducing controlled adversaries into this otherwise controlled measurement framework.



(a) No spatial isolation.



(b) Services isolated on small cores, application on big cores.

Fig. 2: **We use spatial isolation to improve cost attribution and reduce noise.** Perfetto traces of X (Twitter) pinned to two big cores (CPUs 6 & 7). In (a), all other applications and services are unconstrained, so contend with the benchmark for the big cores. In (b), extraneous applications are isolated on the small cores (CPUs 0–3).

We use three Linux kernel features: `cpusets` [20], `isolcpus` [57], and `setaffinity` [39] to perform spatial isolation. `cpusets` use cgroups to limit the number of cores a given process may run on [20]. Android extensively uses `cpusets` to limit background activity or improve the latency of foreground applications [4]. `isolcpus` is a kernel boot command-line option that isolates a given set of cores and prevents the kernel scheduler from considering them for scheduling decisions [57]. `setaffinity` allows us to move our benchmark applications to the isolated cores.

This approach to spatial isolation does not prevent kernel threads or other processes that are spawned by the benchmark application from being scheduled to the isolated cores. Kernel threads explicitly ignore all `cpuset` and `isolcpus` options, while other processes spawned by the benchmark application may also be considered as 'foreground' or 'top-app' processes and so will be put into the relevant `cpuset` by Android.

TABLE II: **Adversary workloads**. We use these to realistically emulate background activity in some evaluations.

Adversary	Background Behavior
Amazon Shopping	Memory and network requests
Camera	Hardware access and memory requests
Firefox	Memory and network requests
Settings	Memory requests
Spotify	Memory and network requests
Wikipedia	Memory and network requests

TABLE III: **Hardware configuration**, listing core and memory interface (MIF) frequency settings.

Name	Cores	Clock	Memory	MIF Clock
Pixel 6 Pro	2×Cortex-X1	2.40 GHz	12 GB	3200 MHz
	2×Cortex-A76	1.99 GHz	LPDDR5	
	4×Cortex-A55	1.40 GHz		
Pixel 7 Pro	2×Cortex-X1	2.40 GHz	12 GB	3200 MHz
	2×Cortex-A78	1.99 GHz	LPDDR5	
	4×Cortex-A55	1.40 GHz		

Figure 2 shows the effect of spatial isolation using Perfetto [44] traces. In both cases the benchmark, X (Twitter), is pinned to the big cores (CPUs 6–7). In Figure 2(a), other tasks may run freely on any core including the big cores, interfering with the benchmark. In Figure 2(b), other tasks are isolated on the small cores (CPUs 0–3), leaving the benchmark isolated on the big cores, and the medium cores free for the GC threads.

While performing our evaluation we ran into a kernel bug where the affinity mask of our benchmark process was cleared when it was moved to a different `cpuset`. This bug has been fixed upstream in Linux kernel v6.2 [40], however, the kernel versions on our devices are too old so we could not cleanly apply the fix. Instead, we work around the problem by registering our benchmark with Android and avoid moving it from the 'top-app' `cpuset` once the benchmark has started.

D. Controlled Realism via Adversaries

The dominant methodology for evaluating Java performance is to execute a benchmark in isolation, taking great care to create a minimally disrupted environment [12], [42]. This is an important method for gaining a minimally disturbed view of the workload under study, however in an environment where multi-tenancy is the norm, it is important to study the impact of other applications on the workload under study.

We address this with a methodology that allows us to run applications in the background while the benchmark being measured runs in the foreground. To do this, we curated a set of six adversary workloads, listed in Table II. Each is initialized via a scripted UI interaction, and then left in the background while the benchmark under study is executed. While in the background the applications will conduct minimal activities and (by definition) will not use the display. One of the applications, Firefox, will perform a search and scroll the results before being put in the background, causing memory and network requests in the background. Because they run in the background, they will not directly contend for CPU resources with the benchmark under study, due to the thread affinity of these process groups. However, they will contend for other shared resources such as lower level caches, memory, memory bandwidth, etc. In § V-D we analyze the performance impact of different adversary loads.

E. Understanding Synchronization Costs

Since multicore hardware has become mainstream, most high performance garbage collectors are carefully parallelized to maximize throughput on the available hardware. However, the Android native garbage collectors are single-threaded. Although mobile devices are typically multicore, they often have heterogenous core configurations, and multi-tenancy means that the opportunity cost associated with parallel garbage collection is more pronounced—the additional GC threads are likely to contend with other applications for CPU resources. Parallel GCs are not perfectly scalable [8] and they incur synchronization overheads.

In order to better understand these tradeoffs, we implemented a single-threaded tracing loop for one of our garbage collectors, avoiding all costs associated with synchronization such as atomic metadata operations and shared work queues. We use this to analyze the costs of synchronization and value of single-threaded collection in a mobile environment in § V-B.

IV. Methodology

We now describe the experimental methodology we used to conduct our energy and performance analysis.

A. Hardware Platform

We use two Google Pixel 6 Pros and Pixel 7 Pros with ODPM support. These devices run a Lineage OS 21 build [32], [56] based on Android 14 QPR1 and Linux kernel version 5.10.177. Lineage OS enables a more realistic environment than AOSP [6] as it allows for installation of Google Play Services—a key requirement for many real-world applications. To control for variance from frequency scaling and thermal throttling, we fix CPU and memory interface (MIF) frequencies [47] using the 'performance' governors (Table III).

We pin our benchmark's mutator threads to the two big cores. This controls for variance due to thread scheduling on asymmetric cores. To ensure pinning is respected, we use the fix to the `Runtime.availableProcessors` API from [48] for correct CPU mask checking. We borrow changes from Sareen et al.'s errata [49] which prevent real-world applications from manually scheduling their threads via `setaffinity`.

B. Software

We use a fork of the Android Runtime (ART) based on commit `451cfcf`[1] with support for the Memory Management Toolkit (MMTk) [41]. We backport changes to object scanning to simplify the MMTk binding implementation [23], [24]. However, backporting these patches to stock ART GCs was not straightforward, so they remained unpatched. We patch our

[1] https://android.googlesource.com/platform/art/+/451cfcf9d09515ef60d76bd8551fc68c6e3bf621

TABLE IV: **Benchmarks evaluated.** Minimum heap sizes were measured by performing a bisection search with CMC.

Benchmark	Description	Heap Size (MB)
Adobe Acrobat	PDF viewer	32
Google News	News feed	25
Google Maps	Navigation	57
TikTok	Social media	98
X (Twitter)	Social media	42
GCBench	Microbenchmark stressing GC [21]	13
lusearch	Text search over corpus [10]	3
pmd	Source-code analyzer [10]	24
xalan	XSLT processor [10]	3

Android build to fix upstream bugs due to finalization [46]. We patch the `system_server` service to work around upstream Linux kernel bugs in `setaffinity` and `cpuset`, as discussed in § III-C. We patch ART to ensure that GC always runs in a dedicated GC thread. Without this change, ART can perform GC in the application thread that triggered it. This allows us to reliably pin GC to specific cores, which is crucial for directly measuring and spatially isolating the collector. We measured this change's overhead, finding a 1–2 % degradation for vanilla Java workloads for a modest (2×) heap size.

We use version 0.30.0 of MMTk. We use PGO [58] when building MMTk, using profiles from Adobe Acrobat.

C. Benchmarks

Table IV lists the benchmarks we evaluate, taken from the set curated by Sareen et al. [48]. The first five are Android applications, and the remaining are vanilla Java benchmarks. We update the Android applications to recent versions and make minor modifications to their benchmarking scripts. We exclude the Airbnb benchmark because the application has changed significantly, rendering its scripts non-functional. We also exclude Instagram due to its high noise, and Discord, Gmail, and Twitch due to their low allocation rates.

MMTk and stock GCs differ in their handling of non-moving objects, so we remove the small number of non-moving objects from our heap size calculations. Heap size measurements are complicated by ART's convention of measuring only the from-space, leading to a factor-of-two difference with MMTk for copying collectors. We adjust for this discrepancy, normalizing to include the entire heap when measuring heap size.

D. JIT and AOT Compilation

For vanilla Java benchmarks, we follow standard methodology, allowing benchmarks to warm up before measurement. We find that timing the third iteration provides stable results.

Our methodology differs for real-world benchmarks. ART uses Ahead-of-Time (AOT) compiled code whenever available, maintaining a cache of compiled code [5]. New ART builds may invalidate cached AOT code, using the JIT compiler for executing applications.[2] We improve upon the state-of-the-art methodology [48] by gathering profiling data once for all configurations. We run all benchmarks for 60 invocations with a simple STW collector, saving the profile data. Then at the start of an experiment, for each configuration: we disable all background compilations, clear the profiling data, restore the saved profiling data, and then force an AOT compilation of the benchmarks. We then execute benchmarks using the AOT compiled code. This process ensures: i) all configurations use the same profiling data; ii) background compilations do not spuriously change compiled code during benchmark runs; iii) the resultant compiled code is more realistic due to extensive profiling data; and iv) the JIT compiler minimally interferes with benchmark runs. This process nearly eliminates JIT activity, allowing us to use a single iteration for real-world benchmarks while gathering low-noise measurements.

E. Garbage Collectors Used

We use ART's two production collectors: Concurrent Copying (CC) and Concurrent Mark Compact (CMC). From MMTk, we use two simple full-heap STW collectors: Immix (IX) [11] and SemiSpace (SS) [17]. We omit ART's SemiSpace GC due to its comparable performance to MMTk's SemiSpace. We use an 'almost' NoGC configuration of MMTk's SemiSpace collector, as described in § III-B. This involves running the SemiSpace collector in MMTk with the maximum allowable heap size.[3]

V. Results

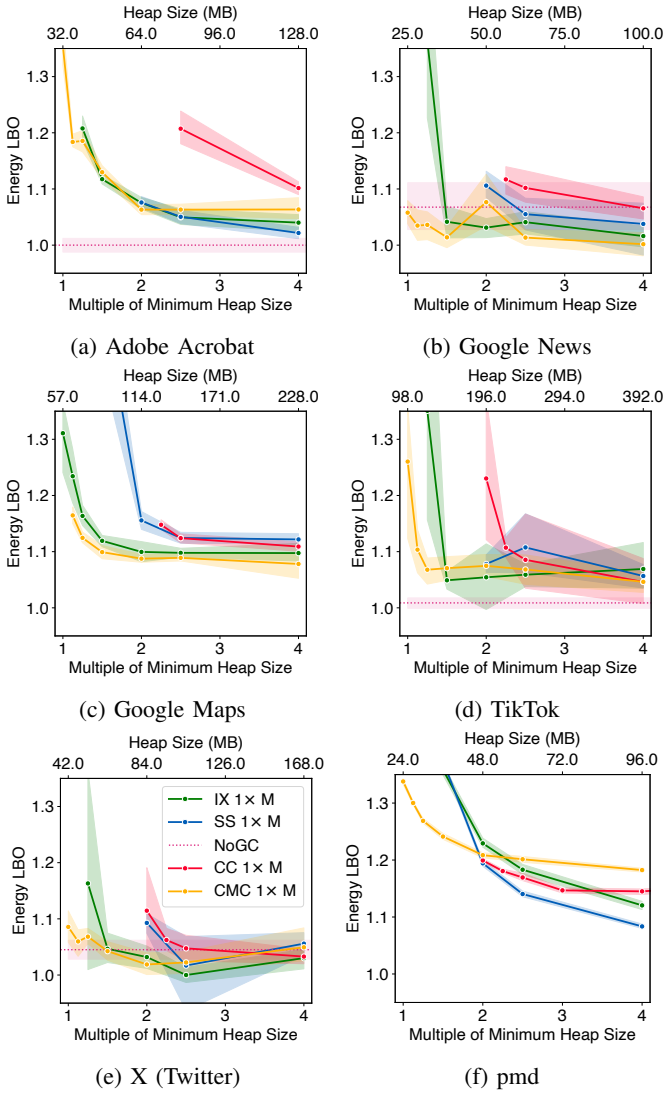
A. The Energy Cost of GC

Figure 3 shows the *lower bound* of energy overheads due to garbage collection for the five Android applications and one command line workload (pmd). The other command line workloads show curves that are similar to pmd. We evaluate two MMTk collectors, Immix (IX) and SemiSpace (SS), and the two native collectors, Concurrent Mark Compact (CMC) and Concurrent Copying (CC). The results in Figure 3 are all for single-threaded collectors (§ III-E, § V-B), and are all run on a single medium-sized core. We established that this was the most energy efficient configuration after evaluating multithreading (§ V-B) and different core topologies. Table I shows the LBO baseline calculations, highlighting that the Android applications all use a very conservative baseline. Our previous analysis (§ III-A) suggests that this conservatism likely understate total energy consumption by as much as 10-15%. We focus on the subsystems that we believe that GC can (directly) influence, namely: CPUs (big and mid cores), DRAM, memory interface, and system-level cache energy counters.

Acrobat exhibits classic time-space tradeoff curves similar to the command line application, pmd, while the curves are a lot flatter in some other workloads, such as X, where it remains almost flat until the heap size becomes very small. We see that Android's new CMC collector (gold) tends to be more energy efficient than its predecessor, CC (red). Perhaps surprisingly, NoGC is the baseline for only one of the five Android applications (Adobe Acrobat, Table I). This suggests that the

[2] As it may contain code, e.g. barriers, that are not relevant to the new GC.

[3] 1788 MB for vanilla Java benchmarks; 1024 MB for real-world applications.



(a) Adobe Acrobat (b) Google News (c) Google Maps (d) TikTok (e) X (Twitter) (f) pmd

Fig. 3: **Lower bound energy overheads.** Despite the applications ((a)–(e)) using very conservative baselines (§ III-A), and being intrinsically noisier, they show high sensitivity to GC and significant energy overheads due to GC.

poor locality has a noticeable affect on energy consumption. We note that the multi-threaded builds (omitted) actually reduce the energy of Google Maps by around 10 % and are the baseline (Table I). The much larger error bars seen on the Android applications likely reflect the effects of waits that are intrinsic to these workloads (§ III-A). This, and the necessary conservatism of our measures, seem to underscore just how difficult it is to accurately measure the cost of garbage collection in Android applications. Even so, we can see that Android's native garbage collectors (gold and red) impose a notable energy overhead.

The Pixel 6 Pro results (not shown) are similar to the Pixel 7 Pro, with the exception of MMTk SemiSpace having more overheads than other collectors for Google Maps and X.

B. Synchronization Overheads

Most high performance garbage collectors are carefully parallelized, exploiting multicore hardware to maximize collector throughput. MMTk uses a state-of-the art work packet-based task scheduler, with excellent throughput and scaling compared to highly tuned native production collectors for Java [62]. Android's native collectors are single-threaded (§ III-E).

TABLE V: **Synchronization overheads for SS at 2× heap size.** We report average STW time and total energy over 30 runs for the multi-threaded build with two threads (MT 2T) in milliseconds and J respectively. We report relative performance of the multi-threaded build with one thread (MT 1T) and the single-threaded build (ST) by normalizing to MT 2T.

	STW Time / GC			Total Energy		
	MT 2T	MT 1T	ST	MT 2T	MT 1T	ST
Benchmark	ms	/MT 2T	/MT 2T	J	/MT 2T	/MT 2T
GCBench	7.55	0.27	0.23	0.44	0.60	0.58
lusearch	1.46	1.41	1.25	5.24	1.00	0.97
pmd	11.06	1.48	1.34	3.98	1.00	0.99
xalan	2.05	1.48	1.32	10.19	1.00	0.98
Adobe Acrobat	67.80	0.91	0.78	41.51	0.99	0.97
Google News	64.57	0.93	0.84	13.25	1.00	0.92
Google Maps	86.09	1.02	0.92	58.51	1.04	1.02
TikTok	149.10	1.13	0.95	41.94	1.01	1.00
X (Twitter)	84.99	0.97	0.90	32.56	1.00	0.98

We now explore the tradeoffs associated with parallel garbage collection in the mobile setting. To do this we implemented an *explicitly single-threaded* tracing loop for MMTk, which we compare with the highly-tuned parallel implementation, with respect to both time and energy performance.

Table V compares three variations on MMTk's SemiSpace collector at a modest 2× heap with respect to average stop-the-world (STW) garbage collector time, and total energy, with the best (lowest) results shown in green. The table depicts the absolute time and energy for the two-thread parallel implementation (MT 2T), and the *relative* time and energy for the one-thread parallel and single-threaded implementations.

There are two stand-out results in this data. First, with respect to energy, the single-threaded implementation is systematically the lowest cost, from a 42% advantage on GCBench, to at worst matching the multi-threaded build on TikTok. It is only worse than the multi-threaded build for Google Maps where it consumes 2 % more energy. Second, with respect to time, the results are strikingly different for the command line and Android applications. The former, aside from GCBench overwhelmingly favor the multi-threaded build, while the latter overwhelmingly favor the single-threaded build.

The clear take-home from this analysis is that a collector optimized for single-threaded work is likely to be preferable in a mobile setting, contrary to the design of most modern high-performance garbage collectors for servers and browsers.

C. Energy Consumption Per-Subsystem

ODPM (§ II-B) allows us to measure energy at a per-subsystem granularity across thirteen domains [2], [45]. For each workload we performed the study depicted in Figure 5, which shows the split across the 13 domains for each of 63 GC configurations. First, these measurements show how much GC affects energy consumption—each GC configuration shows a different split among subsystems. In Figure 5 we see the big

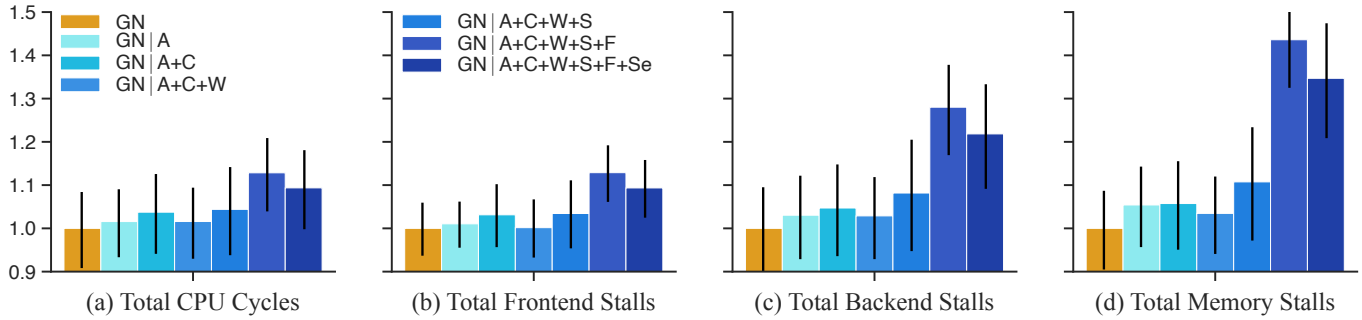


Fig. 4: **The performance impact of background applications.** Study measuring the performance of Google News when we vary the number of adversaries (0–6) running in the background (§ V-D). The results have been normalized to the baseline that is running without any adversary. We take the average over 10 runs on our Google Pixel 7 Pro. We use a simple single-threaded STW GC (Immix [11]) with a 1.5× heap size for this evaluation. The benchmark threads (including the GC thread) have been pinned to the two big cores. Total CPU cycles degrade by 10 % when we have 5–6 adversaries running in the background. Backend stalls (and specifically memory stalls) dominate the overheads.

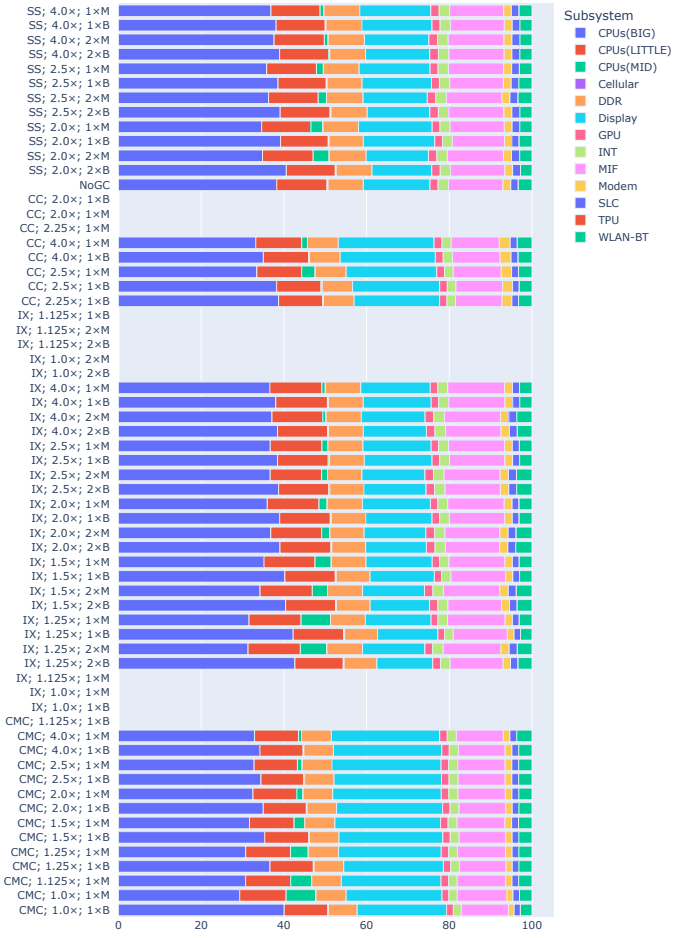


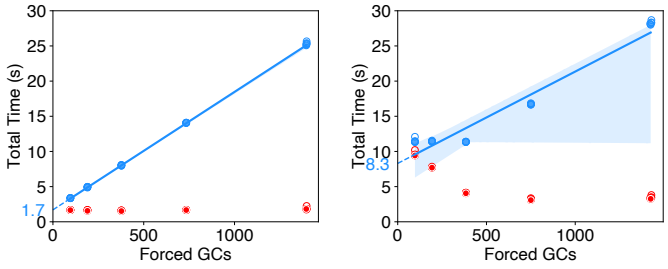
Fig. 5: **Per-subsystem energy attribution for the Adobe Acrobat Android application.** Each row shows the percentage energy consumption split among thirteen ODPM domains, for a given configuration among 63 GC configurations.

CPUs (blue) consuming as little as 30% and as much as 43% of total energy. Second (unshown), unsurprisingly, the command line applications are dominated by the big CPUs (60-80%) while the Android applications only spend 15–45% of energy on the big CPUs. This is partly explained by the display (cyan) being turned off for command line applications, but consuming 13–35% for Android applications. Third, the memory interface (pink) is important, consuming 11–19% of energy across all benchmarks. Fourth, the little CPUs (red) are more dominant on the Android applications (10–15% compared to 5.5–14% for command line applications), likely due to moving background services to the little cores (§ III-C).

D. The Impact of an Adversary on GC Performance

Mobile devices are inherently multi-tenanted while traditional performance analysis works hard to run workloads in isolation (§ II). Prior work on improving full system performance of Android devices tend to model "realistic" environments by sequentially running multiple applications [27], [31], [33], [38]. However, most work has focused on measuring and improving the number of cached applications running in the background, i.e. reducing the number of low-memory kills inflicted on background applications, rather than the performance of the foreground application. Recently Li et al. [35], [36] explored the treatment of background applications and how that affects foreground responsiveness.

In Figure 4 we execute the Google News application (GN) with increasing numbers of background adversaries (Table II), and measure its performance in terms of CPU cycles and apply a topdown analysis [61]. The first three background adversaries (Amazon, Camera, and Wikipedia) have little impact on the foreground application performance. However, once Spotify and Firefox are added, the impact becomes noticeable, slowing the foreground application by 4.4% and 12.8% respectively. These overheads are driven by memory stalls, which increase by 10.7% and 43.6% respectively.



(a) GCBench (1 Click) (b) GCBench (10 Clicks)

Fig. 6: **Linear regression to estimate mutator time for our GCBench microbenchmark.** Note how Figure 6(b) mimics the behavior of real-world applications (cf. Figure 1(d)), while Figure 6(a) behaves as we would expect (cf. Figure 1(a)).

Adversaries can impact the total energy consumption of benchmarks as well (not shown), ranging from a 2% increase for a single adversary to 10–12% when running with all of them. Most of the increase comes from memory-related counters, which is unsurprising, as these counters are system-wide. An increase in the number of background applications can lead to an increase in memory energy consumption due to increased memory activity. However, we also see an increase in the energy consumption for the big cores, likely due to the slowdown caused by memory stalls.

E. Understanding the Impact of Waits

To confirm our hypothesis of the impact of UI-induced waits (§III-A), we designed a simple Android application that runs the standard GCBench microbenchmark. The application has two buttons: A) invokes ten iterations of GCBench, while B) invokes a single iteration. We can thus run ten iterations of GCBench either by pressing A) once or B) ten times. Hence, we have two systems which have the same allocation patterns, however, one of them performs substantially more UI interactions.

Figure 6 plots the linear regression results for mutator time for the two versions of the microbenchmark. We can immediately see that although the two benchmarks perform (almost) the same number of GCs[4], the measured ground truth (red) for the mutator time in Figure 6(b) is both: i) more than the "true" ground truth in Figure 6(a), and ii) dependant on the number of forced GCs. This closely mirrors what we observed in Figure 1(d). These results suggest that the UI-induced waits are indeed the cause of the behavior we noticed in real-world applications.

F. Threats to Validity

Fundamental to most empirical evaluation is the tension between realism and fidelity. By isolating the target application (§ III-C), we have increased fidelity in CPU energy measurements at the expense of realism. To understand the impact of our change, we run an experiment where we do not isolate the target application.

[4]GCBench (10 Clicks) only allocates 2 % more than GCBench (1 Click).

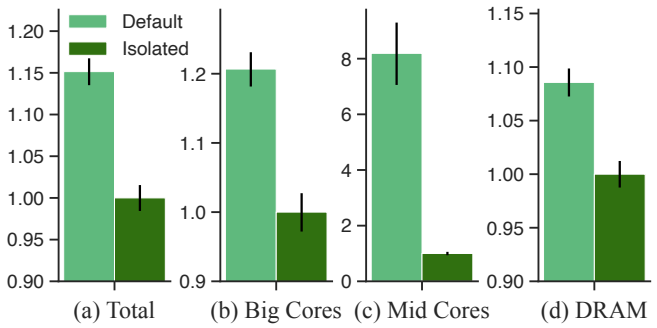


Fig. 7: **Energy measurement impact of isolating target applications.** Study measuring the energy consumption of Google Maps with and without isolation (§ III-C). We run the benchmark with a 2× heap size with the CMC collector, pinning the GC thread to the big cores. We take the average over 10 runs on our Google Pixel 7 Pro. The results have been normalized to the runs with isolation.

Figure 7 plots the measurements of various ODPM counters when controlling for isolation. We note that the performance of the benchmark (in terms of CPU cycles) between the two systems is nearly identical. We expect the CPU energy counters to have the largest differences given the isolation specifically targets the attribution of CPU energy consumption, and indeed that is what we see. Isolating the target application dramatically reduces the energy consumption of the mid cores (Figure 7(c)). This is a more accurate energy measurement because neither the GC thread nor any services should be executing there.

Surprisingly, the memory counters also differ in their energy consumption (Figure 7(d)). We believe this is a side-effect of pushing services to the little cores. This could lead to more effective caching for both the target application and background services, owing to the spatial separation. We measured the L1 data cache miss rate of the target application and found that isolating it reduced the miss rate by 10%.

VI. Conclusion

Understanding the energy overheads of garbage collection is important to improving battery life and running costs of devices. We explore the challenges of performance analysis of Android applications, introduce new methodologies that allow new analyses of energy consumption, and present a series of studies using these approaches. While the confounding effects of UI-induced waits remains an unresolved challenge for analyzing Android applications, our study has shed new light on how to evaluate Android applications, and the energy costs associated with garbage collection on mobile devices. We hope the methodological and analytical advances we present here will encourage further studies of garbage collection for mobile devices, and perhaps spur new energy-conscious designs.

VII. Acknowledgements

We thank Sara S. Hamouda for their insights and support. We thank our anonymous reviewers for their constructive and encouraging feedback.

References

[1] H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs, Second Edition*. MIT Press, 1996.

[2] Android Open Source Project. (2023) Power profiler. Archived at https://web.archive.org/web/20250501175129/https://developer.android.com/studio/profile/power-profiler. [Online]. Available: https://developer.android.com/studio/profile/power-profiler

[3] ——. (2024) ART chroot-based on-device testing. [Online]. Available: https://android.googlesource.com/platform/art/+/5937cdeeef4639e523ae3cfde3bfce3ccb252921/test/README.chroot.md

[4] ——. (2024) Identify capacity-related jank. Archived at https://web.archive.org/web/20250721050932/https://source.android.com/docs/core/tests/debug/jank_capacity. [Online]. Available: https://source.android.com/docs/core/tests/debug/jank_capacity

[5] ——. (2024) Implement ART just-in-time compiler. Archived at https://web.archive.org/web/20240306000514/https://source.android.com/docs/core/runtime/jit-compiler. [Online]. Available: https://source.android.com/docs/core/runtime/jit-compiler

[6] ——. (2025) Android open source project. [Online]. Available: https://source.android.com/

[7] ——. (2025) Power management. Archived at https://web.archive.org/web/20251112015019/https://source.android.com/docs/core/power/mgmt. [Online]. Available: https://source.android.com/docs/core/power/mgmt

[8] K. Barabash and E. Petrank, "Tracing garbage collection on highly parallel platforms," in *Proceedings of the 9th International Symposium on Memory Management, ISMM 2010, Toronto, Ontario, Canada, June 5-6, 2010*, J. Vitek and D. Lea, Eds. ACM, 2010, pp. 1–10. [Online]. Available: https://doi.org/10.1145/1806651.1806653

[9] S. M. Blackburn, Z. Cai, R. Chen, X. Yang, J. Zhang, and J. N. Zigman, "Rethinking Java performance analysis," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*, L. Eeckhout, G. Smaragdakis, K. Liang, A. Sampson, M. A. Kim, and C. J. Rossbach, Eds. ACM, 2025, pp. 940–954. [Online]. Available: https://doi.org/10.1145/3669940.3707217

[10] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. B. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "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*, P. L. Tarr and W. R. Cook, Eds. ACM, 2006, pp. 169–190. [Online]. Available: https://doi.org/10.1145/1167473.1167488

[11] S. M. Blackburn and K. S. McKinley, "Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance," in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, R. Gupta and S. P. Amarasinghe, Eds. ACM, 2008, pp. 22–32. [Online]. Available: https://doi.org/10.1145/1375581.1375586

[12] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffmann, A. M. Khan, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "Wake up and smell the coffee: evaluation methodology for the 21st century," *Commun. ACM*, vol. 51, no. 8, pp. 83–89, 2008. [Online]. Available: https://doi.org/10.1145/1378704.1378723

[13] Z. Cai, S. M. Blackburn, M. D. Bond, and M. Maas, "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, 2022, pp. 46–57. [Online]. Available: https://doi.org/10.1109/ISPASS55109.2022.00005

[14] T. Cao, S. M. Blackburn, T. Gao, and K. S. McKinley, "The Yin and Yang of power and performance for asymmetric hardware and managed software," in *39th International Symposium on Computer Architecture (ISCA 2012), June 9-13, 2012, Portland, OR, USA*. IEEE Computer Society, 2012, pp. 225–236. [Online]. Available: https://doi.org/10.1109/ISCA.2012.6237020

[15] M. Carpen-Amarie, G. Vavouliotis, K. Tovletoglou, B. Grot, and R. Müller, "Concurrent GCs and modern Java workloads: A cache perspective," in *Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management, ISMM 2023, Orlando, FL, USA, 18 June 2023*, S. M. Blackburn and E. Petrank, Eds. ACM, 2023, pp. 71–84. [Online]. Available: https://doi.org/10.1145/3591195.3595269

[16] X. Chen and Z. Zong, "Android app energy efficiency: The impact of language, runtime, compiler, and implementation," in *2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom), BDCloud-SocialCom-SustainCom 2016, Atlanta, GA, USA, October 8-10, 2016*, Z. Cai, R. A. Angryk, W. Song, Y. Li, X. Cao, A. G. Bourgeois, G. Luo, L. Cheng, and B. Krishnamachari, Eds. IEEE Computer Society, 2016, pp. 485–492. [Online]. Available: https://doi.org/10.1109/BDCloud-SocialCom-SustainCom.2016.77

[17] C. J. Cheney, "A nonrecursive list compacting algorithm," *Commun. ACM*, vol. 13, no. 11, pp. 677–678, 1970. [Online]. Available: https://doi.org/10.1145/362790.362798

[18] L. Cruz, "Tools to measure software energy consumption from your computer," http://luiscruz.github.io/2021/07/20/measuring-energy.html, 2021, blog post.

[19] L. Cruz and R. Abreu, "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, 2017, pp. 46–57. [Online]. Available: https://doi.org/10.1109/MOBILESoft.2017.19

[20] S. Derr, P. Jackson, C. Lameter, P. Menage, and H. Seto. (2004) Cpusets. [Online]. Available: https://docs.kernel.org/admin-guide/cgroup-v1/cpusets.html

[21] J. Ellis, P. Kovac, H.-J. Boehm, and W. Clinger. (1997) An artificial garbage collection benchmark. [Online]. Available: https://hboehm.info/gc/gc_bench.html

[22] A. Georges, D. Buytaert, and L. Eeckhout, "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*, R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., Eds. ACM, 2007, pp. 57–76. [Online]. Available: https://doi.org/10.1145/1297027.1297033

[23] L. Gidra. (2024) Embed component-size shift in class-flags. [Online]. Available: https://android-review.googlesource.com/c/platform/art/+/3181647

[24] ——. (2024) Use variable sized ref-offset bitmap for fast `VisitReferences()`. [Online]. Available: https://android-review.googlesource.com/c/platform/art/+/3098038

[25] M. Gupta. (2021) Google Tensor is a milestone for machine learning. Google LLC. [Online]. Available: https://blog.google/products/pixel/introducing-google-tensor/

[26] M. Hertz and E. D. Berger, "Quantifying the performance of garbage collection vs. explicit memory management," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, R. E. Johnson and R. P. Gabriel, Eds. ACM, 2005, pp. 313–326. [Online]. Available: https://doi.org/10.1145/1094811.1094836

[27] J. Huang, Y. Zhang, J. Qiu, Y. Liang, R. Ausavarungnirun, Q. Li, and C. J. Xue, "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*, R. Gupta, N. B. Abu-Ghazaleh, M. Musuvathi, and D. Tsafrir, Eds. ACM, 2024, pp. 654–670. [Online]. Available: https://doi.org/10.1145/3620666.3651377

[28] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng, "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*, J. M. Vlissides and D. C. Schmidt, Eds. ACM, 2004, pp. 69–80. [Online]. Available: https://doi.org/10.1145/1028976.1028983

[29] A. Hussein, M. Payer, A. L. Hosking, and C. A. Vick, "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, D. Naor, G. Heiser, and I. Keidar, Eds. ACM, 2015, pp. 13:1–13:12. [Online]. Available: https://doi.org/10.1145/2757667.2757674

[30] ——. (2017) Android Etalon – DaCapo. [Online]. Available: https://github.com/fizous/etalondacapo

[31] ——, "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, 2017, pp. 171–186. [Online]. Available: https://doi.org/10.1145/3050748.3050754

[32] N. Johnson. (2024) Changelog 28 – Fantastic Fourteen, Amazing Applications, Undeniable User-Experience. [Online]. Available: https://lineageos.org/Changelog-28/

[33] N. Lebeck, A. Krishnamurthy, H. M. Levy, and I. Zhang, "End the senseless killing: Improving memory management for mobile operating systems," in *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, A. Gavrilovska and E. Zadok, Eds. USENIX Association, 2020, pp. 873–887. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/lebeck

[34] S. Lee. On-device power rail monitor (ODPM) driver. Google LLC. [Online]. Available: https://android.googlesource.com/kernel/gs/+/2dc8937d29949d48e8a27c5bcbe164e354918001/drivers/iio/power/odpm.c

[35] C. Li, Y. Liang, R. Ausavarungnirun, Z. Zhu, L. Shi, and C. J. Xue, "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*, G. A. D. Luna, L. Querzoni, A. Fedorova, and D. Narayanan, Eds. ACM, 2023, pp. 79–93. [Online]. Available: https://doi.org/10.1145/3552326.3567497

[36] C. Li, Z. Zhu, C. J. Xue, Y. Liang, R. Ausavarungnirun, L. Shi, and X. Zhou, "Freezing-based memory and process co-design for user experience on resource-limited mobile devices," *ACM Trans. Comput. Syst.*, vol. 43, no. 1-2, pp. 1–29, 2025. [Online]. Available: https://doi.org/10.1145/3714409

[37] D. Li, S. Hao, J. Gui, and W. G. J. Halfond, "An empirical study of the energy consumption of Android applications," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE Computer Society, 2014, pp. 121–130. [Online]. Available: https://doi.org/10.1109/ICSME.2014.34

[38] Y. Liang, A. Shen, C. J. Xue, R. Pan, H. Mao, N. Mansouri-Ghiasi, Q. Jiang, R. Nadig, L. Li, R. Ausavarungnirun, M. Sadrosadati, and O. Mutlu, "Ariadne: A hotness-aware and size-adaptive compressed swap technique for fast application relaunch and reduced CPU usage on mobile devices," in *IEEE International Symposium on High Performance Computer Architecture, HPCA 2025, Las Vegas, NV, USA, March 1-5, 2025*. IEEE, 2025, pp. 1588–1602. [Online]. Available: https://doi.org/10.1109/HPCA61900.2025.00118

[39] Linux man-pages Project Contributors. (2025) `sched_setaffinity(2)` — Linux manual page. [Online]. Available: https://man7.org/linux/man-pages/man2/sched_setaffinity.2.html

[40] W. Long. (2022) sched: Persistent user requested affinity. [Online]. Available: https://lore.kernel.org/all/20220922180041.1768141-3-longman@redhat.com/T/#m51982911c50a294c26f9f84f999081d963250a86

[41] MMTk Research Group, "The Memory Management Toolkit," 2025. [Online]. Available: https://www.mmtk.io/

[42] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!" in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009*, M. L. Soffa and M. J. Irwin, Eds. ACM, 2009, pp. 265–276. [Online]. Available: https://doi.org/10.1145/1508244.1508275

[43] W. Oliveira, R. Oliveira, and F. Castor, "A study on the energy consumption of Android app development approaches," in *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, J. M. González-Barahona, A. Hindle, and L. Tan, Eds. IEEE Computer Society, 2017, pp. 42–52. [Online]. Available: https://doi.org/10.1109/MSR.2017.66

[44] Perfetto Contributors. (2017) Perfetto. Google LLC. [Online]. Available: https://perfetto.dev

[45] ——. (2024) Perfetto: Power data sources. Google LLC. [Online]. Available: https://github.com/google/perfetto/blob/2b4c0fecc1db125c09221539c573d518cdd786ad/docs/data-sources/battery-counters.md

[46] M. Phillips. (2024) Workaround bad GC of tombstone watcher. [Online]. Available: https://android-review.googlesource.com/q/I9226e4368b03bd4742fccdafde6018f145da63e6

[47] Qualcomm. (2024) Qualcomm Linux Kernel Guide | Cache and Memory DVFS. [Online]. Available: https://docs.qualcomm.com/bundle/publicresource/topics/80-70014-3/features.html#sub$cache-and-memory-dvfs

[48] K. Sareen, S. M. Blackburn, S. S. Hamouda, and L. Gidra, "Memory management on mobile devices," in *Proceedings of the 2024 ACM SIGPLAN International Symposium on Memory Management, ISMM 2024, Copenhagen, Denmark, 25 June 2024*, M. D. Bond, J. W. Lee, and H. Payer, Eds. ACM, 2024, pp. 15–29. [Online]. Available: https://doi.org/10.1145/3652024.3665510

[49] ——. (2024) Memory management on mobile devices: With errata (2024-10). [Online]. Available: https://www.steveblackburn.org/pubs/papers/android-ismm-2024-errata.pdf

[50] K. Sareen and S. M. Blackburn, "Better understanding the costs and benefits of automatic memory management," in *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes, MPLR 2022, Brussels, Belgium, September 14-15, 2022*, E. G. Boix and T. Wrigstad, Eds. ACM, 2022, pp. 29–44. [Online]. Available: https://doi.org/10.1145/3546918.3546926

[51] A. Shehabi, S. J. Smith, A. Hubbard, A. Newkirk, N. Lei, M. Abu, B. Siddik, B. Holecek, J. Koomey, E. Masanet, and D. Sartor, "2024 United States Data Center Energy Usage Report," *LBL Publications*, 2024, Report #: LBNL-2001637. [Online]. Available: http://dx.doi.org/10.71468/P1WC7Q

[52] A. Shipilev. (2017) JEP 318: Epsilon: A No-Op Garbage Collector (Experimental). [Online]. Available: https://openjdk.org/jeps/318

[53] Statista Research Department. (2025) Market share of mobile operating systems worldwide from 2009 to 2025, by quarter. [Online]. Available: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/

[54] The LineageOS Project. (2024) Install LineageOS on Google Pixel 6 Pro. [Online]. Available: https://web.archive.org/web/20241120000310/https://wiki.lineageos.org/devices/raven/install/

[55] ——. (2024) Install LineageOS on Google Pixel 7 Pro. [Online]. Available: https://web.archive.org/web/20241120044957/https://wiki.lineageos.org/devices/cheetah/install/

[56] ——. (2024) LineageOS Android distribution. [Online]. Available: https://lineageos.org

[57] The Linux Kernel Development Community. (2025) The kernel's command-line parameters. [Online]. Available: https://docs.kernel.org/admin-guide/kernel-parameters.html

[58] The Rust Project Developers. (2019) Profile-guided optimization. [Online]. Available: https://doc.rust-lang.org/rustc/profile-guided-optimization.html

[59] X. Yang, S. M. Blackburn, D. Frampton, and A. L. Hosking, "Barriers reconsidered, friendlier still!" in *International Symposium on Memory Management, ISMM '12, Beijing, China, June 15-16, 2012*, M. T. Vechev and K. S. McKinley, Eds. ACM, 2012, pp. 37–48. [Online]. Available: https://doi.org/10.1145/2258996.2259004

[60] X. Yang, S. M. Blackburn, D. Frampton, J. B. Sartor, and K. S. McKinley, "Why nothing matters: the impact of zeroing," in *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, C. V. Lopes and K. Fisher, Eds. ACM, 2011, pp. 307–324. [Online]. Available: https://doi.org/10.1145/2048066.2048092

[61] A. Yasin, "A Top-Down method for performance analysis and counters architecture," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, Monterey, CA, USA, March 23-25, 2014*. IEEE Computer Society, 2014, pp. 35–44. [Online]. Available: https://doi.org/10.1109/ISPASS.2014.6844459

[62] W. Zhao, S. M. Blackburn, and K. S. McKinley, "Work packets: A new abstraction for GC software engineering, optimization, and innovation," *Proc. ACM Program. Lang.*, vol. 9, no. OOPSLA2, Oct. 2025. [Online]. Available: https://doi.org/10.1145/3763139

[63] B. G. Zorn, "The measured cost of conservative garbage collection," *Softw. Pract. Exp.*, vol. 23, no. 7, pp. 733–756, 1993. [Online]. Available: https://doi.org/10.1002/spe.4380230704

APPENDIX

A. Abstract

In this artifact, we provide the scripts, builds, and device configurations used to understand the energy overheads of garbage collection in the paper.

B. Artifact check-list (meta-information)

- **Program:** Our fork of the Android Runtime (ART) with MMTk support. Our fork of Lineage OS 21 (based on Android 14 QPR2) with bug fixes and workarounds as per § IV. DaCapo benchmarks ported to Android by Hussein et al. [29]–[31].
- **Run-time environment:** Pixel 6 Pro or Pixel 7 Pro running our fork of Lineage OS 21. A desktop running Ubuntu 24.04 (Linux kernel 6.14.0-27-generic). Our fork of ART with support for MMTk.
- **Hardware:** Pixel 6 Pro or Pixel 7 Pro with fixed CPU and MIF frequencies as per Table III. x86_64 desktop with at least 64 GB RAM and 400 GB disk space to build Android.
- **Run-time state:** Need to sign in and set up Adobe Acrobat, X (Twitter), and TikTok benchmarks. Adobe Acrobat uses a copy of *Structure and Interpretation of Computer Programs, Second Edition* [1]. TikTok user follows one account.[5] X user follows one account.[6] A Spotify account is required to sign in to play music as an adversary (§ III-D).
- **Execution:** We require a `userdebug` build of Android to set up various parameters on device (such as CPU frequency and ODPM sampling rate). Devices need to use the `isolcpus` kernel command-line option for isolating benchmarks (§ III-C). Scripts are provided which set all necessary parameters and boot devices with the required kernel command-line options.
- **Metrics:** Energy overheads, wall-clock time, CPU cycles, stalled backend and frontend cycles.
- **Output:** Logs with performance metrics.
- **Experiments:** Experiments are automated with a fork of `running-ng` 0.4.8 which has support for running benchmarks on Android devices. We update UI Automator scripts from Sareen et al. [48] to work with newer versions of real-world applications or to make them more deterministic (§ IV-C).
- **How much disk space required (approximately)?:** 500 GB+.
- **How much time is needed to prepare workflow (approximately)?:** At least 1 day.
- **How much time is needed to complete experiments (approximately)?:** At least 3–4 days.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Apache License, Version 2.0
- **Workflow automation framework used?:** Fork of `running-ng` v0.4.8 with support for running benchmarks on Android devices.[7] UI Automator scripts from Sareen et al. [48].[8]
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.18908418

[5]Andy Cooks (`@andy_cooks`).
[6]Phil Dream (`@DreamPhil97`).
[7]https://github.com/k-sareen/running-ng/releases/tag/ispass-2026-artifact.
[8]https://github.com/k-sareen/android-benchmark-runner/releases/tag/ispass-2026-artifact.

C. Description

1) How to access: Download from the Zenodo archive.

2) Hardware dependencies: Please see above check-list.

3) Software dependencies: `adb` and `fastboot` need to be installed on host machine. `running-ng` needs to be installed and visible in the `PATH`.

Google Play Services (not provided) needs to be installed while flashing Android on devices. Our devices are running Google Play Services version 26.08.34 (190400-876566425).

D. Installation

We have provided Lineage OS 21 builds for Pixel 6 Pro and Pixel 7 Pro devices. Follow Lineage OS 21 installation instructions to set up the device [54], [55]. Google Play Services (not provided) also needs to be installed before the device is booted for the first time.

Adobe Acrobat, X (Twitter), TikTok, and Spotify all require user accounts. Adobe Acrobat uses a copy of *Structure and Interpretation of Computer Programs, Second Edition* [1] named `SICP.pdf` saved in the `Documents` folder. It also requires storage access permissions while setting up. Our X and TikTok users follow a single account (see § B).

For setting up all other scripts and benchmarks, please refer to the `README.md` of the artifact.

E. Experiment workflow

Use provided scripts to run experiments. Please consult the `README.md` of the artifact for more details.

F. Evaluation and expected results

We are applying for the "Artifact Available" badge.

G. Experiment customization

Our scripts systematically run experiments. `running-ng` experiments are composable and customizable YAML files, while our other scripts are clean and modular, providing rich customization options.

H. Notes

Sometimes the devices can crash or end up in bootloops. Unfortunately, we were unable to understand the underlying cause before the paper deadline. You may have to restart the experiments if such a case arises.

I. Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://cTuning.org/ae