Lower Bound Overheads of Android Garbage Collection

Kunal Sareen (ANU)

Dec 1, 2023

Joint Work: Sara Hamouda (Google DeepMind), Steve Blackburn (Google DeepMind), Hans Boehm (Android), Lokesh Gidra (Android), Martin Maas (Google DeepMind)

Motivation

12 GB RAM

Pixel 8 Pro (2023)

Pixel 8 Pro Illustration by Mliu92 - Own work, CC BY-SA 4.0

Problem Statement

What are the costs of memory management in Android?

Is there a sound methodology for measuring costs?

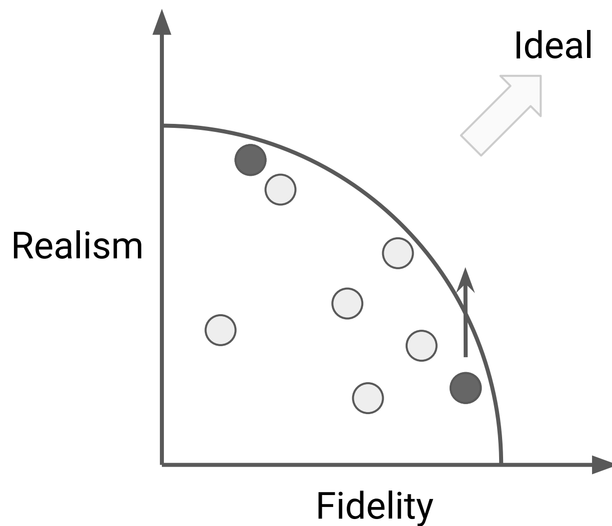
Challenges

Understanding Android Garbage Collection (GC) overhead is hard

- No simple baseline
- No proper experimental methodology
- Realistic applications are hard to control

Benchmarking: Trade-off Space

Ideal is high realism, high fidelity



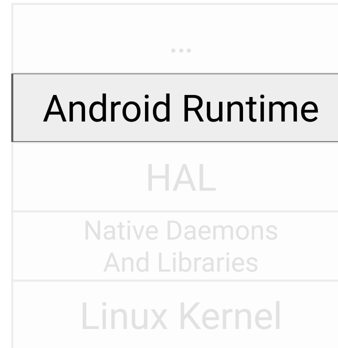
Background: Garbage Collection

Garbage collection allows programmers to forget about managing memory themselves

- Less error-prone
- Memory safe!

Background: Android

Android Architecture



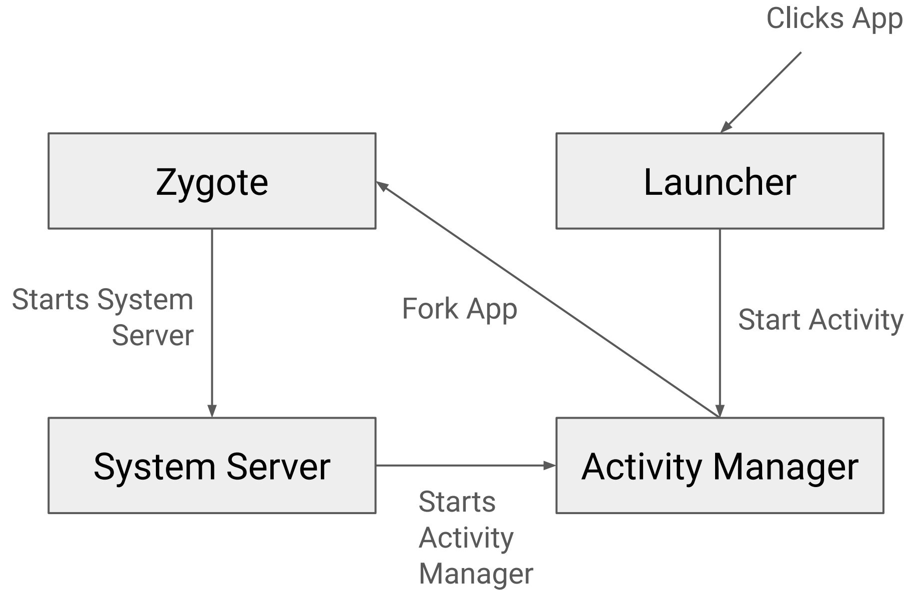
Background: Android

Zygote

- Preload common libraries and classes
- All applications are forked from Zygote

Background: Android

Zygote



Background: Lower Bound Overhead

Distilling the Real Cost of Production Garbage Collectors

Zixian Cai
Australian National University
zixian.cai@anu.edu.au

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

Michael D. Bond
Ohio State University
mikebond@cse.ohio-state.edu

Martin Maas
Google
mmaas@google.com

Abstract—Despite the long history of garbage collection (GC) and its prevalence in modern programming languages, there is surprisingly little clarity about its true cost. Without understanding their cost, crucial tradeoffs made by garbage collectors (GCs) go unnoticed. This can lead to misguided design constraints and evaluation criteria used by GC researchers and users, hindering the development of high-performance, low-cost GCs.

In this paper, we develop a methodology that allows us to empirically estimate the cost of GC for any given set of metrics. This fundamental quantification has eluded the research community, even when using modern, well-established methodologies. By distilling out the explicitly identifiable GC cost, we estimate the intrinsic application execution cost using different GCs. The minimum distilled cost forms a baseline. Subtracting this baseline from the total execution costs, we can then place an empirical lower bound on the absolute costs of different GCs. Using this methodology, we study five production GCs in OpenJDK 17, a high-performance Java runtime. We measure the cost of these collectors, and expose their respective key performance tradeoffs.

We find that with a modestly sized heap, production GCs incur substantial overheads across a diverse suite of modern benchmarks, spending _at least_ 7–82 % more wall-clock time and 6–92 % more CPU cycles relative to the baseline cost. We show that these costs can be masked by concurrency and generous provisioning of memory/compute. In addition, we find that newer low-pause GCs are significantly more expensive than older GCs, and, surprisingly, sometimes deliver _worse_ application latency than stop-the-world GCs.

Our findings reaffirm that GC is by no means a solved problem and that a low-cost, low-latency GC remains elusive. We recommend adopting the distillation methodology together with a wider range of cost metrics for future GC evaluations. This will not only help the community more comprehensively understand

and Alibaba, make extensive use of such languages. On clients, JavaScript engines are embedded in every web browser, and Java runtimes are embedded in every Android phone.

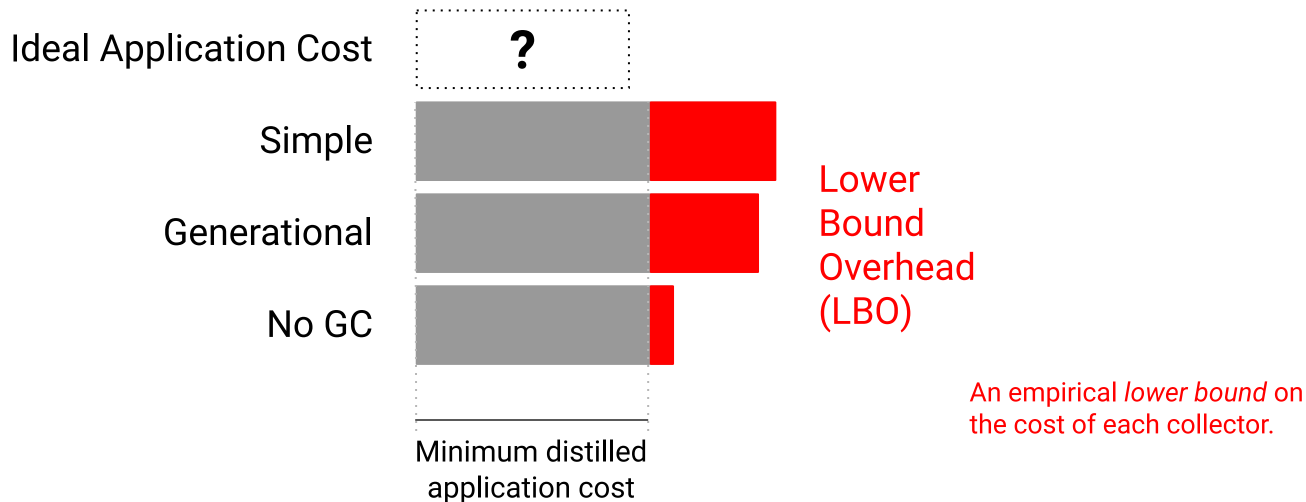
Because of the ubiquity of GC, the research community has extensively studied GC performance. The approaches include characterizing specific elements of GC behavior, performing comparative evaluation among garbage collectors (GCs), and deconstructing the performance of specific GCs. These aspects are addressed by a substantial literature, including [1]–[10]. They are explicitly non-goals of our work.

While this rich literature helps us understand how GCs compare, how they are designed, and what key sources of cost are, there is a surprising lack of clarity regarding the _real_ costs that GC brings to a programming language. In this paper, we focus on two key problems: (a) lack of clarity about the absolute cost of GC, and (b) misinterpretations of GC evaluations due to limited cost metrics. We now offer more detail regarding these two problems and outline our contributions.

a) Unclear absolute costs: The absolute cost that garbage collectors impose on modern production runtimes is an important quantification, but it has eluded the community to date. Its importance is twofold. For programming language implementers and hardware architects, understanding the absolute cost of GC and its magnitude relative to the rest of the language runtime can help them decide where to spend research and engineering resources. For language users, knowing the absolute cost of GC can help them decide whether to use a managed language or to use alternatives such as C/C++ and Rust—a decision that often cannot be easily reversed.

https://doi.org/10.1109/ISPASS55109.2022.00005

Distillation



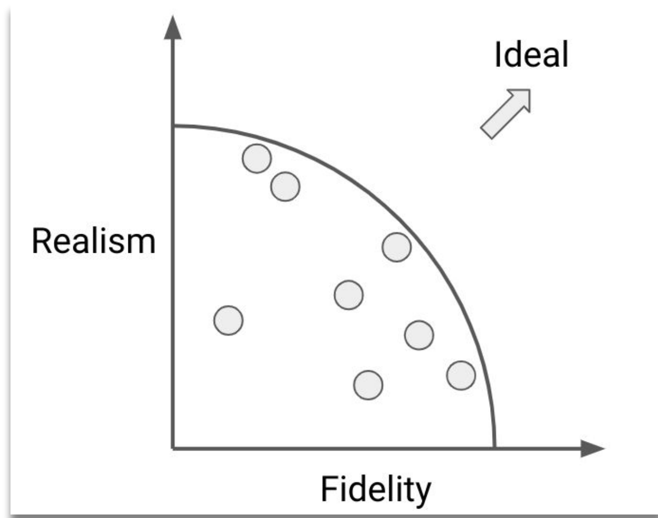
Requirements

1. Ability to cheaply, reliably gather key metrics
2. Constant application workload
3. Ability to control for time/space trade-off
4. Well-understood, simple baseline

Workload Selection

Start with DaCapo + GCBench

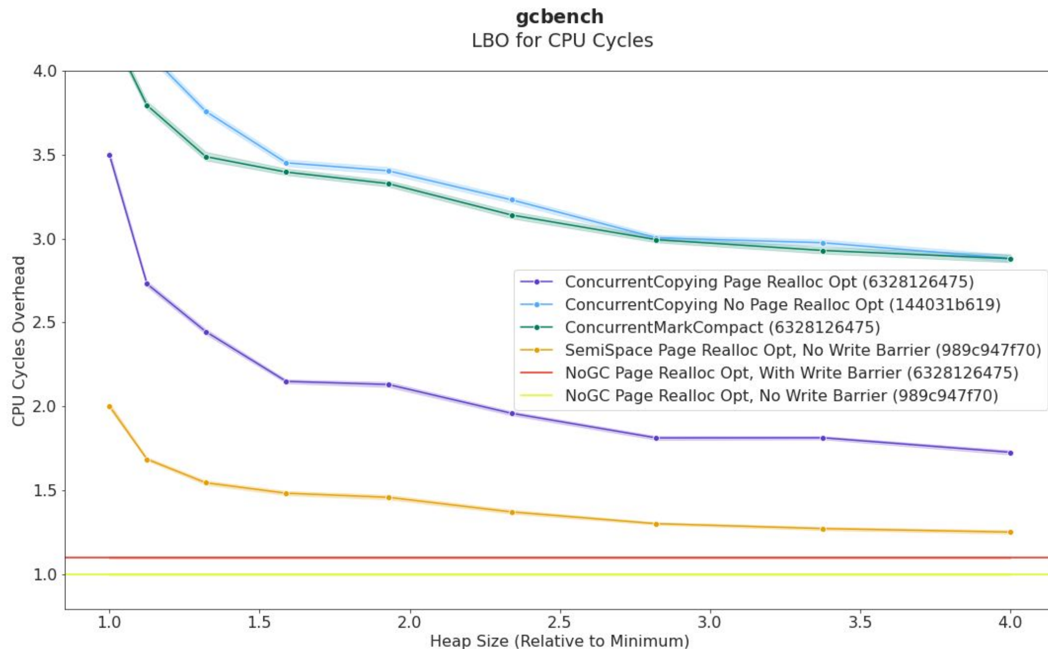
Build understanding from simple benchmarks and extending to more complex ones



Key Contribution: Obtaining LBO Graphs for Android

How did we get here?

- Attribution of costs
- Reducing sources of noise
- Refining methodology



Experimental Methodology

Attribution of costs

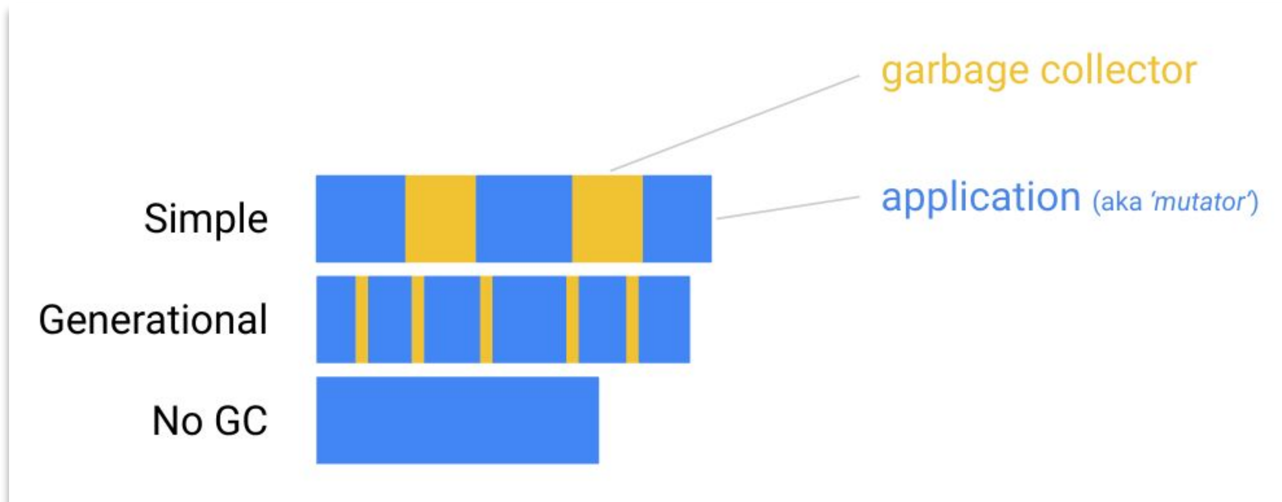


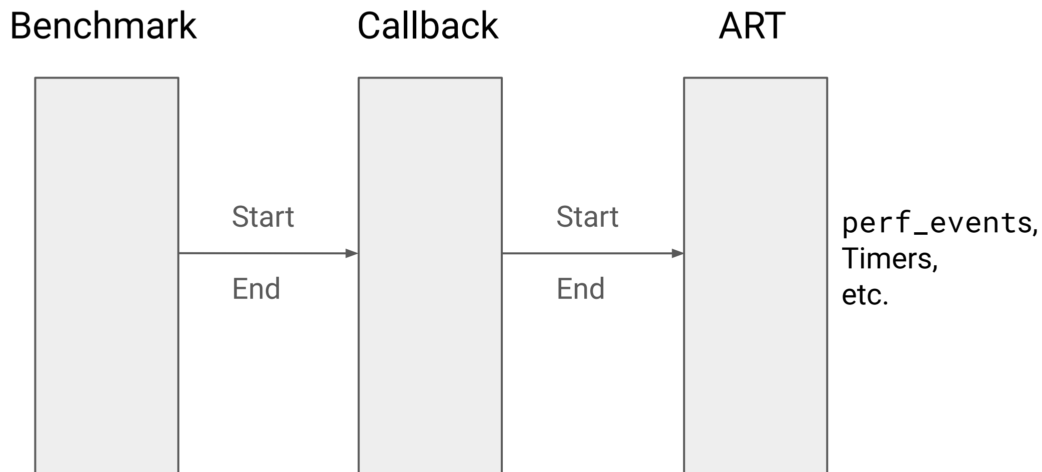
Image from this [presentation](#) by Zixian Cai (https://www.zcai.org).

Experimental Methodology

Attribution of costs

GC cost

Benchmark iteration

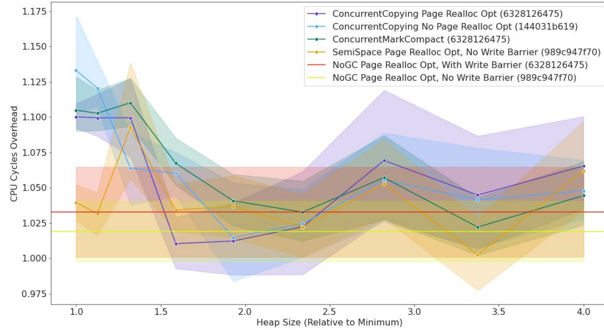


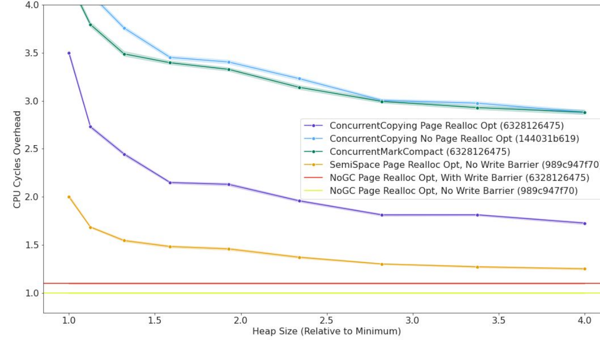
Experimental Methodology

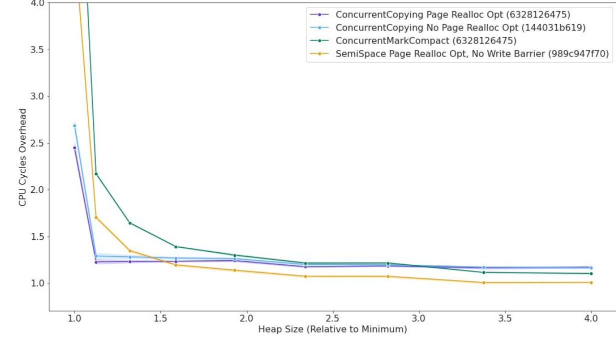
Use best practices from GC and performance analysis literature

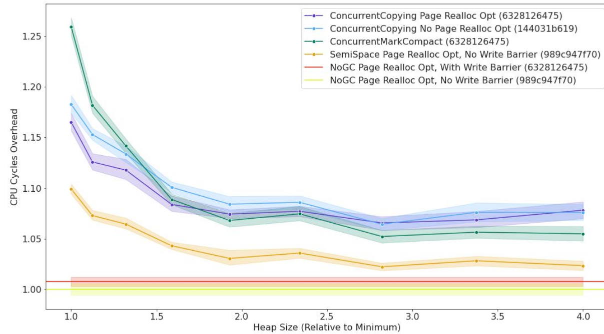
- Control heap sizes
- Interleaved execution
- Reducing sources of experimental noise

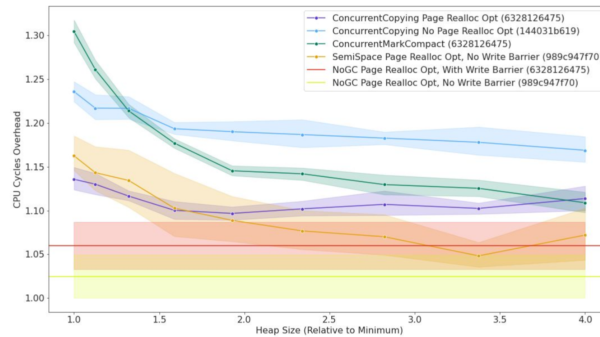
Results

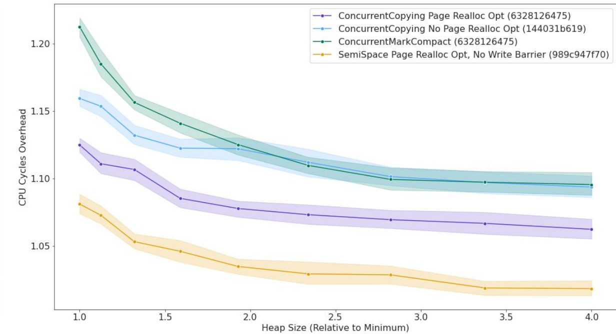




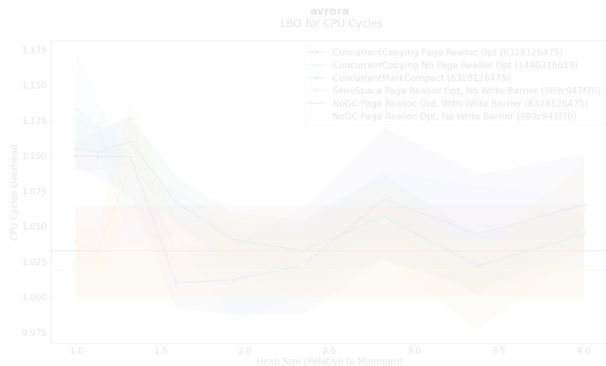


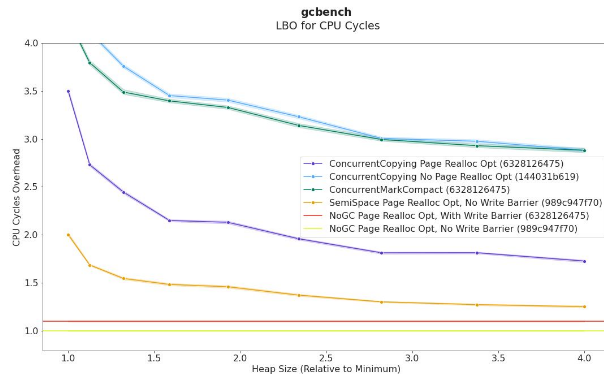


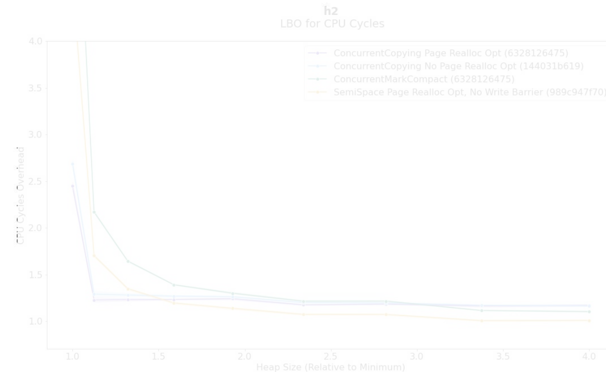


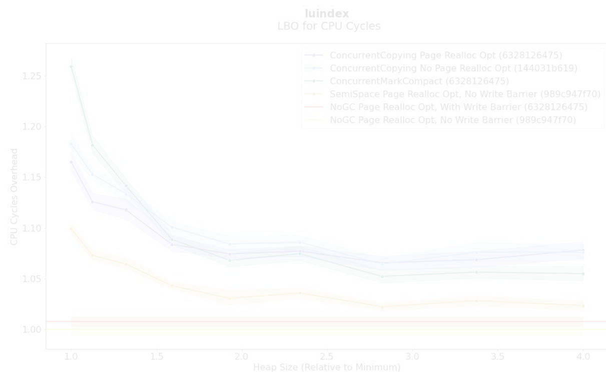


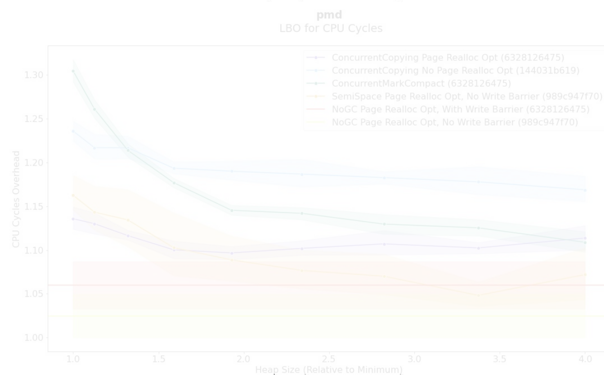
Results

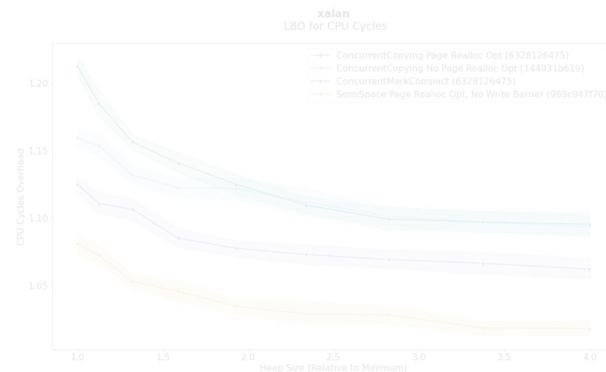




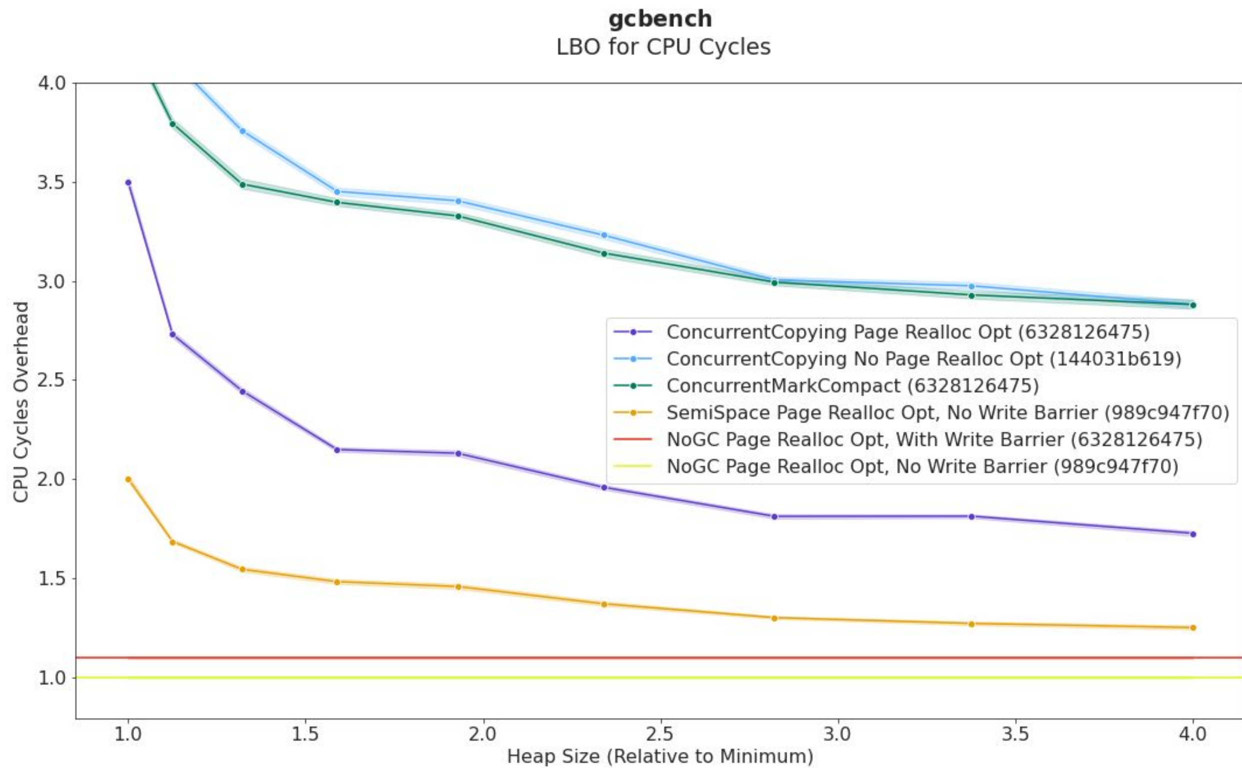




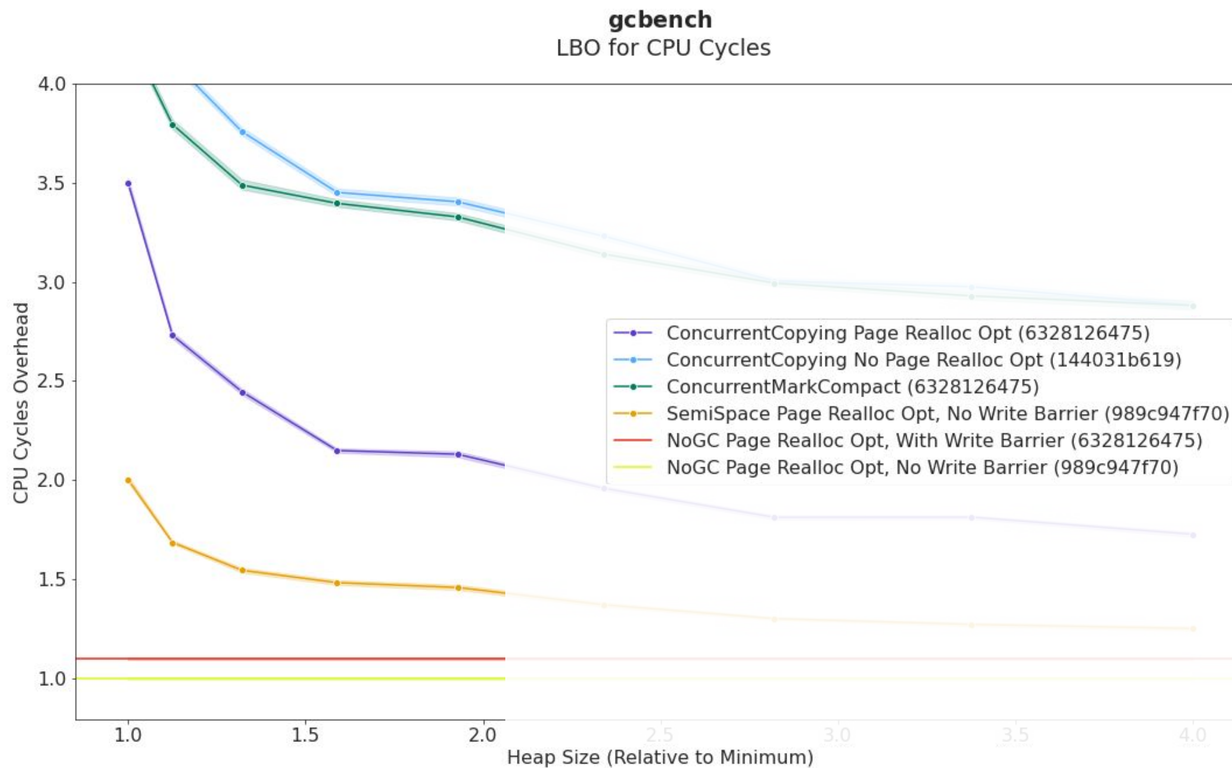




Results: GCBench



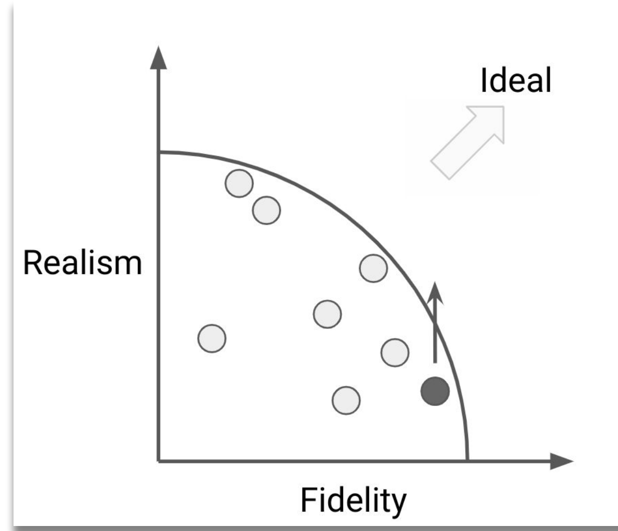
Results: GCBench



Workload Selection

Start with simple to drive but realistic application as proof-of-concept

Watching videos on YouTube



Experimental Methodology

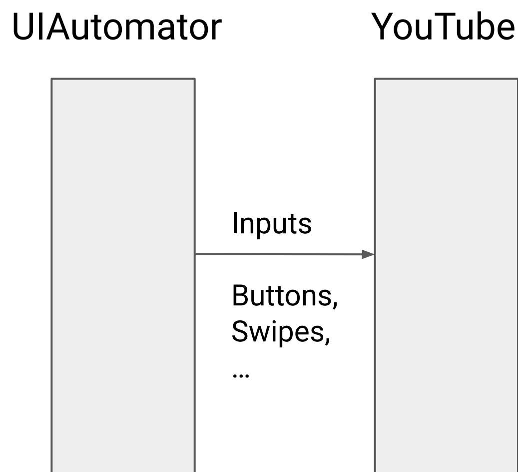
Running benchmarks

- Use UIAutomator tests
- However, tests use JUnit and other heavyweight frameworks
  - Don’t want to measure the test harness

Experimental Methodology

Simple, lightweight benchmark runner

Directly use UIAutomator to drive tests

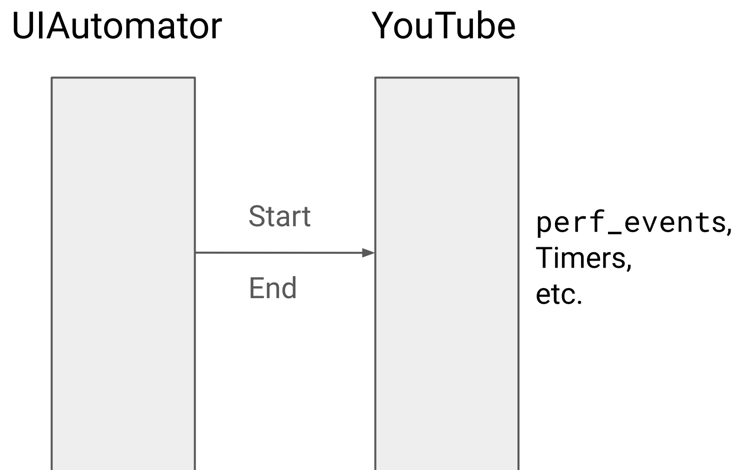


Experimental Methodology

Attribution of costs

GC cost

Benchmark iteration

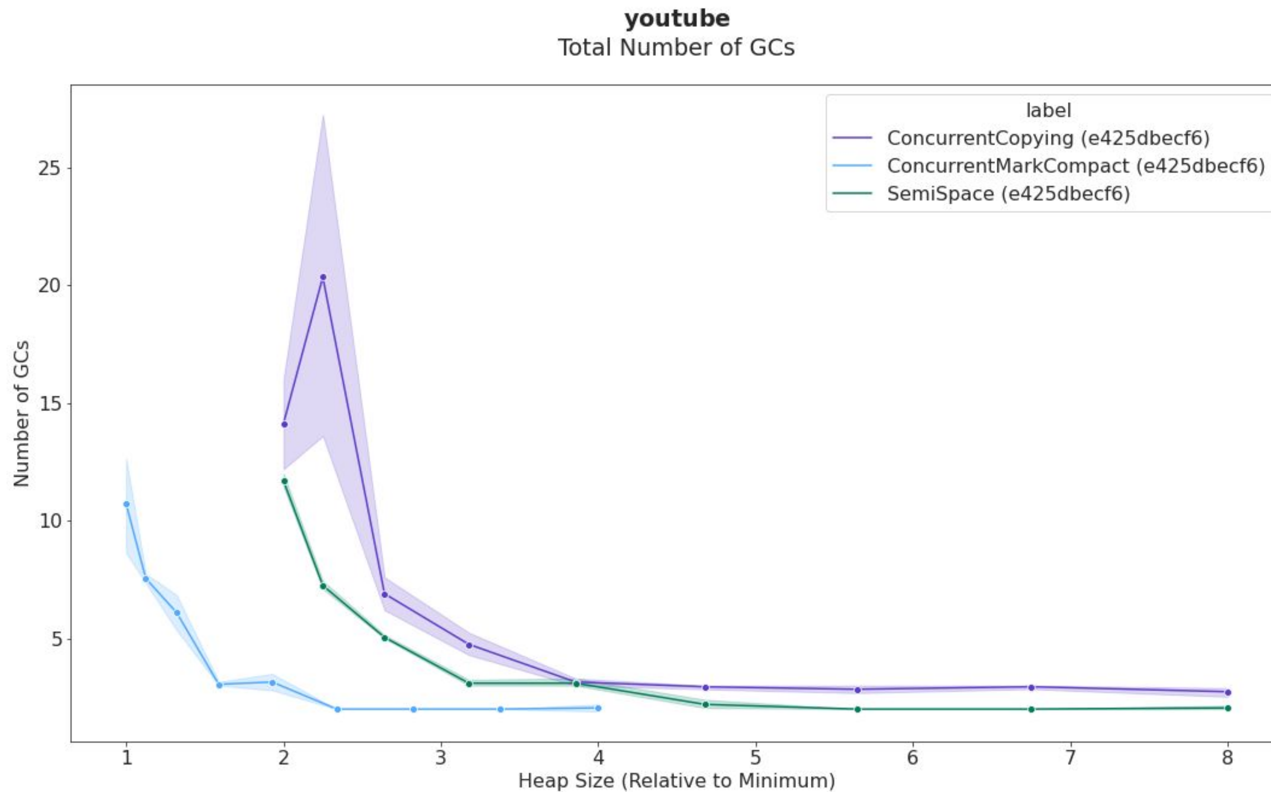


Experimental Methodology

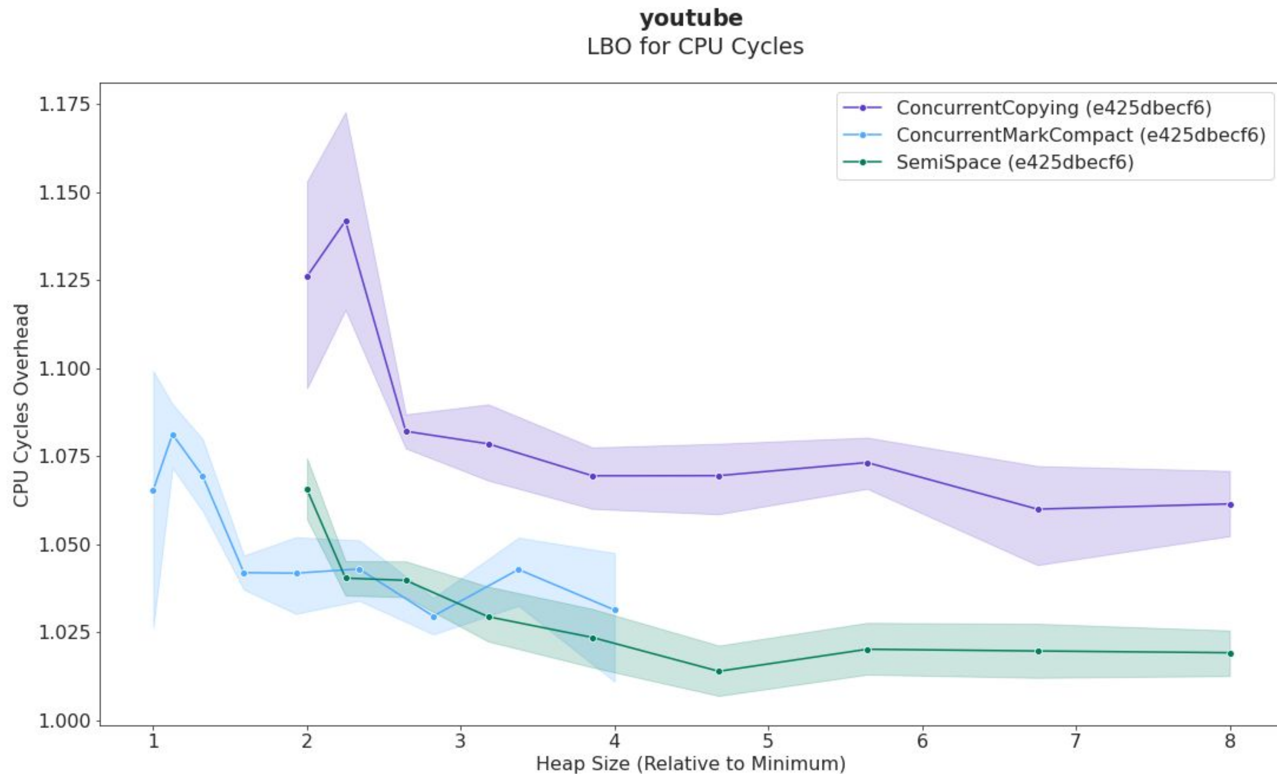
Controlling heap size

- Set only for target app (by default in Android heap settings affect all apps, globally)
- Configurable at execution time via file

Preliminary Results: YouTube



Preliminary Results: YouTube



Outcomes

Revealing GC costs for realistic, representative mobile workloads

Controlling heap sizes for real applications

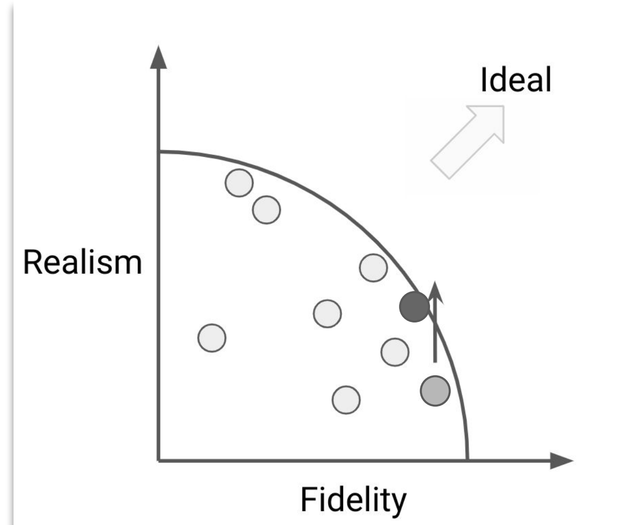
Gathering total costs of benchmark and attribute costs to GC

Running on the phone!

Future Work

Use more GC-sensitive benchmarks

Identify and remove sources of experimental noise



Reflections

Space-time trade-off is key to anchoring GC performance

Simple, well-understood baselines are invaluable

Running experiments in controlled environments can be insightful

Set of (constant-work) benchmarks are incredibly important