Floo: Automatic, Lightweight Memoization for Faster Mobile Apps

Murali Ramanujam
Princeton University
muralisr@cs.princeton.edu

Helen Chen
Princeton University
hc20@cs.princeton.edu

Shaghayegh Mardani
UCLA
shaghayegh@cs.ucla.edu

Ravi Netravali
Princeton University
rnetravali@cs.princeton.edu

ABSTRACT

Owing to growing feature sets and sluggish improvements to smartphone CPUs (relative to mobile networks), mobile app response times have increasingly become bottlenecked on client-side computations. In designing a solution to this emerging issue, our primary insight is that app computations exhibit substantial stability over time in that they are entirely performed in rarely-updated codebases within app binaries and the OS. Building on this, we present Floo, a system that aims to automatically reuse (or memoize) computation results during app operation in an effort to reduce the amount of compute needed to handle user interactions. To ensure practicality – the struggle with any memoization effort – in the face of limited mobile device resources and the short-lived nature of each app computation, Floo embeds several new techniques that collectively enable it to mask cache lookup overheads and ensure high cache hit rates, all the while guaranteeing correctness for any reused computations. Across a wide range of apps, live networks, phones, and interaction traces, Floo reduces median and 95th percentile interaction response times by 32.7% and 72.3%.

CCS CONCEPTS

• **Human-centered computing** → *Ubiquitous and mobile computing systems and tools*; **Mobile phones**; *Smartphones*.

KEYWORDS

Smartphones, mobile apps, performance, caching, memoization

ACM Reference Format:
Murali Ramanujam, Helen Chen, Shaghayegh Mardani, and Ravi Netravali. 2022. Floo: Automatic, Lightweight Memoization for Faster Mobile Apps. In *The 20th Annual International Conference on Mobile Systems, Applications and Services (MobiSys '22), June 25–July 1, 2022, Portland, OR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3498361.3538929

1 INTRODUCTION

Mobile apps continue to surge in popularity, accounting for upwards of 70% of user interactions with digital media services and 80% of user attention on smartphones [48]. A governing factor to the success of apps is their ability to respond to user interactions quickly. Indeed, recent reports highlight that users will abandon apps whose response times regularly exceed 2-3 seconds, and grow frustrated with even 100 ms of added delays [6, 8, 28, 49]. Yet, despite their global importance [66, 68, 93], apps continue to trail user expectations in the wild, with nearly half of interaction response times exceeding 3 seconds even on state-of-the-art phones and mobile networks (§2.2).

A recent wave of studies and optimizations have highlighted and addressed network bottlenecks in the process that apps follow to handle user interactions [13, 23, 41, 46, 55, 62, 111]. Due (in part) to these efforts, we now find that client-side computations are a major contributor to suboptimal app responsiveness, accounting for 63.4% and 43.8% of median response times on WiFi and LTE networks, respectively. Worse, this compute bottleneck is slated to persist and worsen (§2.2) as mobile network improvements [22] continue to outpace those of energy-constrained smartphone CPUs [34, 82] (especially with the advent of 5G [69, 106]), and apps continue to become more feature-rich (and thus, computationally intensive).

To tackle this emerging bottleneck, our key insight is that app computations are entirely performed in the source code of (rarely updated [17, 101]) app binaries and operating systems – not files and programs downloaded during each interaction, as in the traditional web [63, 71, 103, 104] – and thus exhibit substantial stability across user interactions and over time. Building on this, we pursue a conceptually simple, yet historically difficult to realize approach: reduce the computations required to handle user interactions by reusing results from past computations, i.e., *computation memoization*. As with prior memoization efforts across other domains [36, 78, 95, 96], the overarching challenges are in (1) ensuring correctness when reusing any computations, and (2) minimizing the overheads associated with reusing those results. However, unlike prior efforts, we additionally seek automation (i.e., no developer effort) and compatibility with the existing app ecosystem. Further, we require especially high computation cache hit rates because app interactions typically involve millions of short function invocations, rather than a few lengthy ones (§3.2).

Our solution, **Floo**, begins by automatically analyzing and instrumenting app and platform bytecode prior to installation to extract insights about function behavior, i.e., state accesses. During operation, apps respond to interactions as normal. However, prior to each function invocation, Floo queries its cache of past computation results in search of an entry that identically matches the context (i.e., input values) of the upcoming invocation. Hits result in foregoing function execution and instead applying the external effects listed in the cache entry, while misses result in normal function executions and the additions of new cache entries. Atop this basic workflow, three main techniques guide the design of Floo, bringing collective adherence to the goals above.

First, Floo embeds a novel caching strategy that programmatically guarantees correctness while preserving most potential cache hits. To ensure that any introduced memoization leaves app behavior unaltered, cache queries list all possible state that a function *might* access during its upcoming invocation based on Floo's offline static

MobiSys '22, June 25–July 1, 2022, Portland, OR, USA

ACM ISBN 978-1-4503-9185-6/22/06.
https://doi.org/10.1145/3498361.3538929

analysis. However, such conservatism can unnecessarily forego cache hits by mandating matches across all of the function's potential control flows, rather than only the ones that will actually be traversed. To safely regain those precious hits, cache entries with Floo only list information about the state accessed during each invocation, but do so in a manner that implicitly captures the precise set of control flows that were (*and were not*) traversed. Floo then uses a new cache hit criterion that remedies the discrepancies in state listed between queries and entries to aggressively memoize invocations in a control- and data-flow aware manner (ensuring correctness).

Second, to reduce the overheads of querying the cache for the many sub-millisecond invocations in apps, Floo judiciously performs cache queries *ahead of time* for upcoming invocations that are predicted using the app's (static) call graph. Key to Floo's lookahead queries is the careful handling of potential state dependencies between upcoming invocations, and the resultant uncertainty in specific values to embed in each query. Further, to speed up each query and keep prediction horizons short, cache entries are oriented using a custom data structure that enables aggressive pruning of the number of records to consider and the ability to fail fast on each one.

Third, to cope with resource restrictions on mobile devices in light of the millions of invocations (and thus, queries and entries) in app interactions, Floo introduces domain-specific cache admission and eviction policies; both are based on several empirical observations that highlight the consistency in each function's memoization behavior over time. For admission, Floo quickly identifies the sizeable set of functions that consistently yield low hit rates or net slowdowns with memoization, and deactivates them from the caching process, thereby freeing compute threads for more useful queries. For eviction, Floo eschews generic policies such as LRU in favor of a new utility metric per entry that estimates hit counts and predicted speedups from memoization using only (lightweight) passive observations on the cache.

Floo does not require app source code (or developer support), and instead works by modifying app binaries and Android bytecode; it only makes client-side changes, enabling direct and secure deployment in the wild. As a result, we evaluated Floo using a wide range of 50 popular Android apps, real phones, live mobile networks and origin servers, and realistic user interaction traces. Overall, we find that Floo reduces median and 95th percentile computation overheads by 43.8% and 77.6% (0.55 and 0.98 seconds), which translates to speedups in app responsiveness of 32.7% and 72.3%. Further, Floo outperforms (1) computation offloading systems [42] by 6-22% while sidestepping the security and management issues of proxy servers, and (2) recent (complementary) network optimizations [86] by 1.1-1.7×. The source code for Floo is available at https://github.com/muralisr/floo.

2 BACKGROUND AND MOTIVATION

We first detail the procedure that apps follow to respond to user interactions (§2.1), and then present measurements highlighting the significant negative effects that on-device computations have on app responsiveness (§2.2). §6.1 explains the experimental setup and methodology used to collect all results in this section.

2.1 App Interaction Handling

Post installation, mobile apps operate in an event-driven manner, whereby users interact with the app via device sensors (e.g., the screen, microphone), and those interactions trigger computations and network fetches that recursively resolve until the interaction is fully handled. The overarching performance goal for apps is to minimize interaction response times (IRTs) which characterize the end-to-end

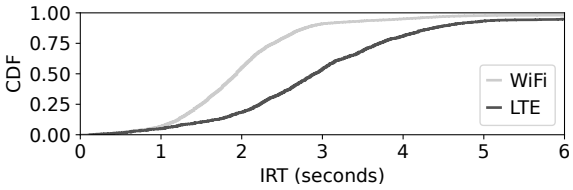


Figure 1: App interaction response times on WiFi and LTE.

delay between an interaction being initiated and the final screen for that interaction being rendered to the user [20, 42, 86].

From a network perspective, apps operate much like the traditional web [16, 70, 75, 91]. Requests are typically issued using the HTTP protocol, and response times are governed by both network transmission and server-side delays, e.g., from dynamically generating or locating responses. To mitigate network overheads, apps routinely maintain local HTTP caches housing both up-to-date resources and recently expired files that enable data-saving conditional requests [39, 86].

App computations, on the other hand, deviate from the traditional web, and employ a multi-process and multi-threaded execution model. When an app is spawned, it generates one Linux foreground process that houses the *main (or UI) thread*. The main thread is responsible for ingesting user interactions, resolving and managing the corresponding computations, and displaying the effects back to the user. To remain responsive and able to absorb new user actions at any time, the main thread may (1) offload tasks to other threads in the same process [29], or (2) spawn new (background) processes for asynchronous tasks that do not influence immediate updates to show the user.

The computations performed in response to a user interaction are defined by a *call graph* that is specified entirely by code in the app binary (or APK) and the underlying computation stack, i.e., the Android codebase. More specifically, the call graph embeds event handler functions that are defined to fire in response to specific user interactions, e.g., `onTouch()`. The subgraphs beneath those event handlers list the complete series of nested function invocations and callbacks that might be traversed to ultimately resolve the interaction. Interactions involve resolving the entire subgraph rooted at the corresponding event handler unless explicitly preempted by code in an intermediate invocation or by a subsequent user action. Note that a function may appear at multiple places in the call graph depending on its utility to different interaction handling paths.

For Android apps (our focus), each function in the call graph can pertain to one of two classes: app-defined code or code that is part of the Android Open Source Project (AOSP). App-defined code includes the source code written by app developers, including that embedded in linked third-party libraries, e.g., for locking or network management. Unsurprisingly, app-defined code can issue API calls that interact with the underlying Java Virtual Machine or device hardware; such APIs invoke code that is part of AOSP. For example, creating a new `String` variable *tmp* and invoking *tmp.hashcode*() entails calling constructors and functions defined as part of the `String` class in OpenJDK. Similarly, displaying a list of items involves constructing a `ListView` object using elements of *android.view.ViewGroup* in the Android source.

2.2 The Problem: Client-Side Computation Delays

App speeds trail user expectations. As shown in Figure 1, app response times in the wild often exceed the 2-3 seconds that users are willing to tolerate [8, 28, 49]. For example, 45.9% and 81.6% have IRTs >2 seconds on WiFi and LTE; 95th percentile IRTs reach 4.0 and 6.1 seconds on the two networks.

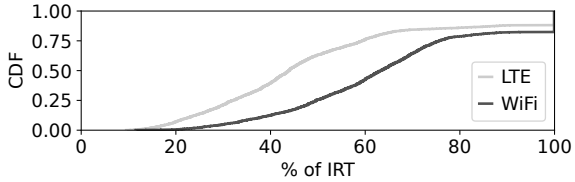


Figure 2: % of IRT accounted for by client-side computation. 100% symbolizes interactions with only local computations.

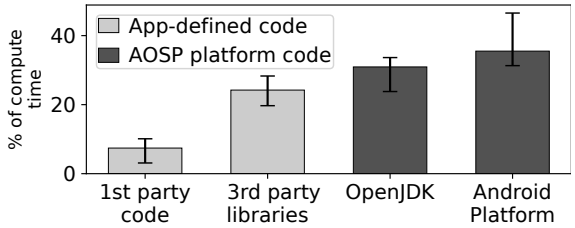


Figure 3: Per-interaction compute times for different types of app code. Bars list medians; error bars show 25-75th percentiles.

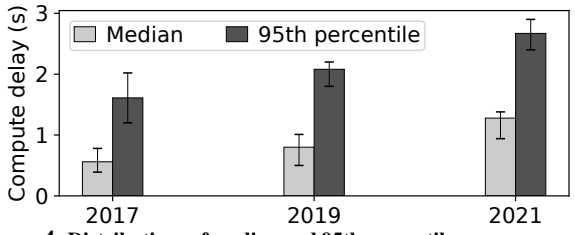


Figure 4: Distributions of median and 95th percentile per-app compute delays for app versions across three different years. Bars list medians, with error bars for 25-75th percentiles.

Compute delays are a primary bottleneck. To understand (and quantify) the role that on-device computations play in the IRTs reported above, we evaluated the same set of interactions in an environment where network delays were set to ≈0 ms. To create such an environment, we performed each interaction back-to-back and recorded the IRTs from both runs. Note that the second run leveraged a warm network cache to minimize fetch delays; any cache misses that arose were preferentially addressed using safe URL rewriting to point to a local file [76], and if still missing, were serviced over fast wired networks.

Figure 2 compares the IRTs per interaction with compute and compute+network delays. As shown, compute delays account for 63.4% (100%) and 43.8% (100%) of IRT for the median (95th percentile) interaction on WiFi and LTE. Moreover, compute delays for 13% of interactions exceed user tolerance levels on their own. Analysis of the Android in-built profiler revealed that app-defined and AOSP functions each contribute substantially to compute delays (Figure 3), and collectively account for 100% of the compute delays for each interaction.

The problem will persist (and worsen). To understand the long-term outlook of the computation bottleneck in apps, we performed two experiments that longitudinally analyzed the evolution of apps and their execution environments, respectively. First, for each app in our corpus, we collected the binaries that were available on the app store 2 and 4 years ago. For the 14 apps whose binaries were still functional on today's phones, and could communicate with live origin servers, we generated and applied interaction traces in the same way as described in §6.1. As Figure 4 shows, per-interaction compute delays have steadily increased across app versions, e.g., median compute time for the median per-app interaction has more than doubled over the past 4 years, growing from 0.6 to 1.3 seconds.

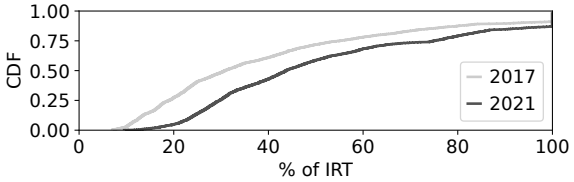


Figure 5: % of IRTs contributed by compute delays when using representative phones and LTE networks from 2017 and 2021.

Second, we characterized the fraction of IRTs accounted for by compute delays when existing apps are running in two different execution environments: those representative of today's app usage and that of 4 years ago. Execution environments are influenced by both network speeds and device compute resources. For network speeds, we leveraged Mahimahi LTE network traces captured in 2017 and 2021 [76]. For compute speeds, we considered two devices from those same time periods: Google Pixels v2 and v5.As shown in Figure 5, the impact of compute delays is steadily growing, with median contributions to IRT increasing from 30.9% to 43.8% over the past 4 years. The result is in line with two recent trends: mobile networks have rapidly improved [22], while speedups to phone CPUs continue to be hindered by energy restrictions [34, 82].

3 APPROACH: AUTOMATIC MEMOIZATION

A natural solution to tackling the compute overheads presented in §2.2 is to reduce the amount of computation needed to support each interaction. Along these lines, we advocate for the local reuse of client-side computations during and across user interaction sessions with apps, i.e., computation memoization [36, 64]. A key driver for our proposal is that app computations remain largely stable over time because they entirely involve traversing call graphs and executing source code defined by app binaries and the platform (§2.2); both are rarely (on the order of months) updated in practice [17, 101]. This is in contrast to the web where computations predominantly pertain to executing programs downloaded during each interaction, e.g., JavaScript files [63, 71, 103]. Further, user interaction patterns with apps exhibit high degrees of repetition over time [50].

Of course, not all workloads are amenable to memoization. In particular, computations must inherently repeat over time for results to be reused. Further, the effects of reusing a computation should be indistinguishable from running the computation inline in that the user-perceived app operation and generated app state should be identical – we refer to this property as *correctness*, and elaborate on it in §4.5. We next demonstrate the promise of memoization in accelerating app interactions while preserving correctness (§3.1), and the challenges associated with realizing those potential benefits in practice (§3.2).

3.1 Potential Benefits

To estimate potential memoization speedups on compute delays, we analyzed the computations that the apps in our corpus performed to support realistic user interaction traces in an environment with no network delays (§6.1). In line with the structure of app call graphs, we focus our analysis on computation reuse for function invocations, i.e., making memoization decisions independently for each node traversed in the graph during interaction handling. For each function invocation, we follow the methodology detailed in §4.1 to record a unique ID for the corresponding function, the set of values that the function's code read during execution (from the global heap and arguments), and the runtime of the invocation.

Using these logs, we estimated an *upper bound* on memoization benefits by joining the resultant list of invocations with the subgraph

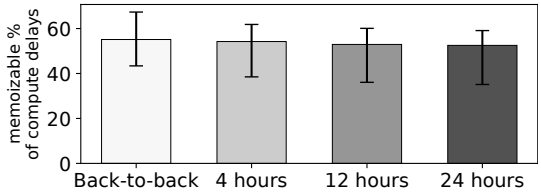


Figure 6: Potential computation speedups from memoization across interactions carried out with different time gaps between them. Bars list medians, with error bars for 25-75th percentiles.

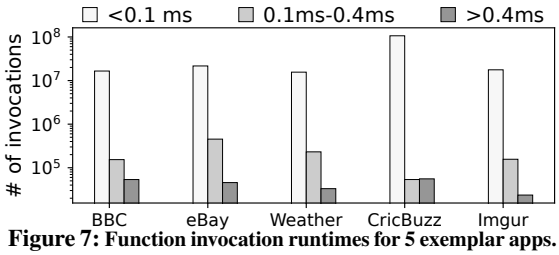


Figure 7: Function invocation runtimes for 5 exemplar apps.

of the call graph that was traversed for each interaction. More specifically, we applied maximal reuse of computations while ensuring correctness such that if a given invocation *inv* had the same function ID and the *exact* same read state as a prior invocation, then *inv*'s runtime was set to 0 ms. Computation speedups were then determined by comparing the runtime on the critical paths of the original and post-memoization call graphs for each interaction.

As shown in Figure 6, existing apps are largely amenable to reaping memoization benefits over time. For instance, potential computation reductions were 55.1% and 67.3% for the median and 75th percentile interactions in the second of back-to-back interaction sessions. These benefits remain largely unchanged (within 6.7%) for interactions carried out 24 hours later, highlighting the stability in app computations alluded to earlier.

3.2 Goals and Challenges

In aiming to realize the potential benefits shown above, we target practicality by focusing on ease of adoption and compatibility in existing app ecosystems. Consequently, unlike prior efforts that require developers to manually (1) add annotations to guide memoization [89], or (2) rewrite their apps (or Java programs) to leverage memoization opportunities [25], we seek a fully automated solution that *operates with any legacy app and without any developer effort*. Moreover, our solution must *guarantee correctness* – i.e., being indistinguishable from unmodified apps in terms of user-perceived functionality and generated state – at all times. Finally, we aim to match the potential benefits as closely as possible by *pursuing all (correctness-preserving) memoization opportunities* that arise during user app sessions. This is in contrast to prior memoization systems that either target testing environments with pre-determined execution patterns [35], or operate only on pure functions [90].

Achieving these goals for real apps and phones in the wild involves numerous challenges. The heart of the issue is that the high computation overheads that apps present manifest in the form of many short invocations, rather than a few long invocations. For instance, across the representative apps in Figure 7, 98-99% of invocations last for fewer than 0.1 ms, with the longest per-app invocations consuming only 5.1-7.4 ms. Further, user interaction sessions for these apps each involve 15-100 million invocations, with 0.4-0.8 million for the median per-app interactions alone. Taken together, these patterns lead to the following complexities for automatic memoization.

Challenge 1: high lookup overheads. Determining whether the results of a prior invocation can be safely reused involves collecting the input values for the current invocation and comparing them fully (e.g., potentially using deep object comparisons) with those from the prior invocation. Worse, owing to the massive number of invocations that apps carry out over time, this analysis may need to be performed on a large number of cache entries. The net is that lookup overheads can quickly dwarf the low runtimes for certain invocations, eliminating benefits (and in fact introducing slowdowns).

Challenge 2: limited device resources. Mobile devices present restricted computation resources along multiple axes: CPU threads, memory, and energy [31, 54, 102]. Unfortunately, the large number of invocations in apps can quickly stress these resources by generating a commensurate number of queries in search of memoization opportunities (each of which consumes a thread and energy), and a significant number of cache entries that consume device memory, i.e., one entry per cache miss.

Challenge 3: correctness-speedup tension. As noted above, our first-order goal is to ensure correctness for any memoization that is applied. The main difficulty is that each function can exhibit diverse behavior across its invocations by following different control and data flows based on interaction orderings, nondeterministic APIs, or downloaded resources. And yet, memoization decisions must be made *prior* to invocations to yield benefits. The natural solution is to bake in a degree of conservatism into the decision-making process, e.g., by reusing a result only if all possible inputs identically match the values for the current invocation. However, doing so inherently brings the potential for missed cache hit opportunities – something we cannot afford with app memoization since each invocation is short, and thus the benefit of each hit is small. Instead, practical (and fruitful) memoization mandates a method to guarantee correctness while foregoing as few cache hits as possible.

4 DESIGN OF FLOO

Figure 8 illustrates the high-level components and operation of Floo, from its offline static analysis on app binaries and platform code (§4.1) to its online attempt to memoize each function invocation involved in responding to a user interaction. To bolster practicality, Floo incorporates several optimizations that enable it to realize the achievable high cache hit rates while ensuring correctness, adhering to resource constraints, and masking lookup overheads (Table 2). In the rest of this section, we first describe the general approach to memoization, and then detail these optimizations in turn.

4.1 Offline Instrumentation for Memoization

Employing memoization for a given invocation (i.e., populating or querying the cache) requires knowledge of the reads and writes that the function performs on state that will be *externally visible* after the invocation is complete, e.g., class fields on the Java heap, values passed through nested invocations or return statements, and updates to the screen or file system. To extract this information, Floo statically analyzes the Java bytecode in app binaries (APKs) and the Android platform prior to installation on a user's phone. The same analysis could be performed on source code, but targeting APKs obviates the need for developer support and enables operation on any app in the app store.

In line with prior static data flow trackers for Android and Java [10, 40, 105], Floo iterates over each line of code to extract lists of read and write operations. For instance, for assignment statements, reads and writes are extracted by processing RHS and LHS components, respectively. Similarly, branches are parsed to extract reads and writes in

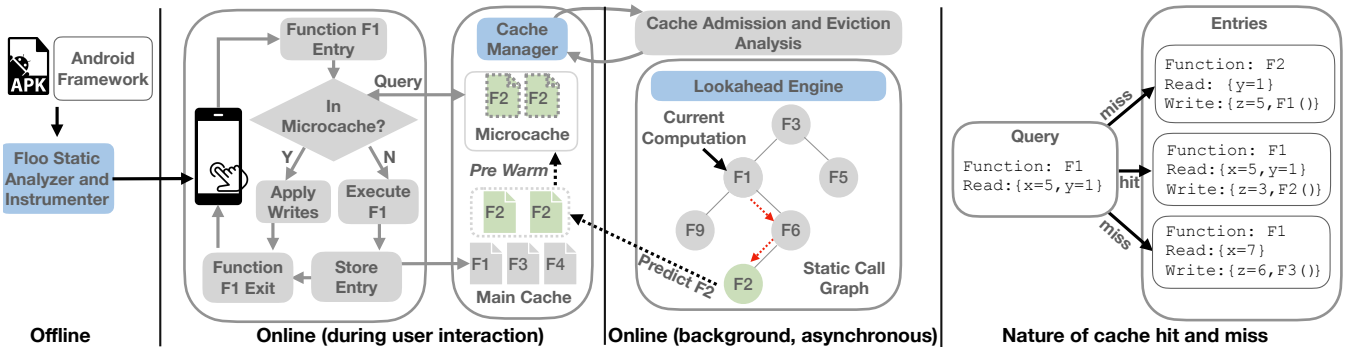


Figure 8: Overview of Floo. App binaries and platform code are instrumented prior to installation on a user device. During operation, each function invocation opportunistically queries a small, per-app microcache (via the cache manager) with the goal of skipping execution. In the background, two optimizations asynchronously churn: (1) the lookahead engine predicts upcoming invocations to pre-warm the microcache, and (2) the cache manager periodically evicts low-utility entries and halts memoization for functions with low hit rates.

both predicates and bodies. This static analysis also records function arguments, return values, and nested invocations. Tracking is done at the finest granularity possible, e.g., *a.b.c* rather than the entire object *a*.

Aliasing. Within each function, Floo tracks aliasing across reads and writes to different variables (i.e., where variables point to the same underlying object) by simply following assignment statements. However, to handle nested invocations and ahead-of-time cache queries (§4.3), Floo also supports cross-function alias tracking. To do so statically, Floo performs (conservative) points-to analysis [40, 94] by propagating reads and writes backwards in potential call graphs to conservatively determine if they might point to the same memory object.

External visibility. The above analysis operates at a variable level, collecting read and write information on each variable accessed within the scope of a function. However, recall that from the perspective of memoization, Floo must record accesses only to externally-visible state; variable-level tracking does not directly elucidate this subset of state accesses. Digging deeper, within the scope of a function, values (primitives or objects) can be accessed in one of three ways, each of which warrants a different process to determine external visibility.

- *Class fields.* Fields of different classes represent the only 'global' variables in Java in that they persist beyond the scope of a given invocation without explicitly needing to be passed out of the function. Thus, setting aside arguments, all accesses to class fields (both static and instance) are externally visible and must be tracked. Note that although reinstantiations for objects in class fields (i.e., via `new`) generate objects at new memory locations, they also update externally-visible pointers for the original object; accesses to class fields thus remain externally visible beyond reinstantiations.
- *Arguments.* Primitive arguments enter functions as pass-by-value, so accesses to them are only externally visible if the underlying values ultimately exit the function via a return, nested invocation, or class field. Object arguments are also passed in by value, with the argument being a pointer to the object's underlying memory location. Writes to that object are generally externally visible, with one exception. If the object is reinstantiated, then the argument variable's pointer will be updated to the memory location for the newly created object; further accesses will not be externally visible unless the new object exits the function via the above mechanisms. Note that updates to an object prior to reinstantiation must still be recorded because the original object is not overwritten in memory and its pointer may be accessible elsewhere.

API Category	Examples
Randomness	`Random, SecureRandom`
Time	`Date, LocalTime`
File System	`File, Paths`
Network	`URLConnection, ContentHandler`
Device Sensors	`SensorManager, GnssAntennaInfo`

Table 1: Categories of nondeterminism that Floo detects. Functions using these APIs are excluded from the memoization process.

- *Local values.* Updates to variables that are assigned to new primitive or object values (created in the function) only become externally visible when those updates exit the function via return statements, invocations, or addition to class fields.

Once only reads and writes for externally visible state remain, Floo iterates over the per-function records and consolidates nested functions that pertain to the same class as the parent functions that invoke them. Consolidation involves adding the nested function's reads and writes to the corresponding sets for the parent function, and removing the invocation from the parent's write set. Floo does not consolidate nested functions belonging to different classes as those functions may access private class variables that the parent could not access. Lastly, Floo instruments each function with lightweight statements to collect read and write state, and issue the appropriate query or population messages to the computation cache (§4.2).

App-level nondeterminism. Certain functions may embed calls to nondeterministic APIs, e.g., for logging the current date/time, generating random numbers, etc. [47]. To safely preserve this behavior, Floo detects such API calls and excludes the housing functions from the memoization process; Table 1 lists the specific API categories that our current implementation detects. A more aggressive approach would attempt to memoize finer-grained compute blocks on either side of the nondeterministic call. However, Floo opts against this for two reasons: (1) as per §3.2, function runtimes are low and can tolerate minimal memoization overheads, and (2) across our apps, only 4.3% of functions use nondeterministic APIs, accounting for only 1.8% of runtime. §4.5 describes Floo's relation to nondeterminism lower in the stack.

4.2 Online Memoization

Basic computation caching. Floo's cache manager runs on a dedicated thread, and operates in a multi-threaded manner to handle incoming messages. At the start of each invocation, functions issue a blocking query to the cache manager that includes (1) the function's Floo-assigned unique identifier, and (2) a list of the {name,

Goal	Techniques	Section
Ensure correctness without degrading hit rates	Control flow-aware cache entries and cache hit criteria.	§4.2
Reduce+mask query overheads to avoid slowdowns	(1) Trie-like cache structure to prune entries to consider and fail fast on each, and (2) Cache lookaheads that safely perform ahead-of-time cache queries for upcoming invocations.	§4.2 §4.3
Maximize benefits with limited device resources	Domain-specific cache admission and eviction policies to eliminate low-probability queries and low-utility entries from the perspective of memoization.	§4.4

Table 2: Overview of the main techniques that Floo uses to address the challenges outlined in §3.

current value, type} three-tuple for each variable that is part of the function's read state based on Floo's offline analysis. For objects, queries list references and the cache manager generates a new copy for query execution. From there, a hit requires that a cache entry and the incoming query identically match on the function id and the three-tuple for all listed variables. Note that hash codes from developer-defined serialization functions are used when present to compare object values; else, (expensive) deep comparisons are used.

Upon a cache miss, the function executes as normal and prior to terminating its context, Floo aggregates the above three-tuples for each variable and nested invocation in the function's predetermined write state. This information (along with the function id) is sent to the cache to form a new entry that reflects this invocation. In contrast, upon a hit, the cache manager returns the three-tuples for all variables in the corresponding entry's write set. The function eschews normal execution in favor of directly applying each write, i.e., by executing assignments or the listed invocations. Importantly, when pooling read or write state to issue cache messages, Floo applies readers-writer locks to objects until the cache manager has generated copies in its context, thereby precluding races with other threads.

Fast query execution. To execute a given query efficiently, the computation cache is organized using a custom data structure. At the highest level, cache entries are partitioned based on whether their functions reside in AOSP or app-defined code. Within each, there exists a single map that is keyed by function ids, with each id pointing to a group of cache entries. Each function's entries are represented as a trie-like structure whereby links are maintained across entries when their values for a specific variable are identical. Using those links, Floo's cache manager prunes the set of entries to consider per query and the set of variables+values to compare per entry. More specifically, once a value for a variable is compared with that in the query, (1) if it does not match, we can quickly eliminate all entries that contain the mismatched value without further analysis, and (2) if it matches, we do not need to compare it again and can immediately remove all other entries with a different value. The cache manager also navigates the trie-like structure by prioritizing lightweight comparisons to fail quickly, i.e., primitives, then hash codes, and then deep comparisons.

To ensure consistency in cache accesses, the cache manager maintains reader-writers locks for all entries pertaining to each function id. The reason is that, although an entry cannot be overwritten by app execution (since we consider only deterministic functions), the corresponding trie could be modified via cache eviction (§4.4) or a cache miss (new entry). The overhead of such locks is low in part because Floo steadily deactivates memoization for functions with low hit rates (§4.4).

Control flow-aware caching. Although the online memoization approach described thus far is functionally correct, it can forego precious cache hits due to its conservative nature. The issue is that, as described, cache entries list all possible reads and writes that a function *could* make based on offline static analysis, rather than the subset of those accesses that were actually carried out during the populating invocation. Consequently, if a function could read a, b, and c, but an invocation only reads a and b, then the value of c should not influence the usability of a given cache entry for that invocation.

To tackle this, cache entries with Floo are control flow-aware in that they only list the reads and writes actually made during each invocation. However, this presents a challenge for queries: at the start of a function, Floo does not know the precise control flows that will be taken, and discovering this would entail similar overheads to executing the function. Thus, cache queries must still include the conservative set of all possible read state for a function, posing a mismatch with control flow-aware entries that list invocation-specific read state. To resolve this, Floo defines a new cache hit criterion: if all variables in the intersection between the query and a given entry have identical values, then the entry can safely be considered a hit and query execution can halt without considering any other entries.

The underlying idea is that any invocation can only pertain to (and traverse) one set of control flows in a function. Further, the initial values for any variables actually read during a function will uniquely map to the specific set of control flows (i.e., branches) that are taken *and* not taken. The reason is that branches are defined by predicates involving reads (or constants) and are explored hierarchically based on pre-defined source code (recall that we exclude nondeterministic functions). Thus, matching on all variables in a cache entry ensures that the set of control flows match, and that no other entry could match. For example, consider the following code snippet:

```
1 if (a == 5) { // branch 1: {a: 5}
2     ...
3 else if(b == 10) { // branch 2: {a: !5, b: 10}
4     a = 5;
5     x = a;
6 } else { // branch 3: {a: !5, b: !10}
7     ...
8 }
```

There are three possible control flows in this branching sequence, each of which is annotated with the read state in its cache entry. As shown, each entry not only specifies which branch was taken, but it also elucidates which branches were not taken, e.g., branch 2's entry precludes branch 1 from being taken due to a's value (of something other than 5). The example also highlights the importance of considering the initial read values for each accessed variable, e.g., if branch 2 was taken, there would also be a read on a with a value of 5 (line 5); considering this read would incorrectly imply that branch 1 was taken.

Floo's approach is conceptually similar to symbolic execution [11], whereby branch traversals (and control flows) are characterized using symbolic expressions on read state. However, with Floo, branch traversal is represented using *concrete* values for an invocation's read state. This distinction is paramount for memoization. On the one hand, using concrete values fails to account for the fact that a set of control flows may pertain to multiple read states (and thus, multiple cache entries), e.g., invocations with read state $a == 7$ and $a == 8$ will both traverse a branch with predicate $a > 5$; symbolic expressions would capture this similarity. On the other hand, cache

hits involve not only finding entries with matching control flows, but also applying the corresponding writes. Consequently, matching control flows is insufficient for correctness. Data flows (and thus, the precise read state values) must also be matched to ensure that the write values match those from normal execution.

4.3 Cache Lookaheads

Though optimized, blocking cache queries often exceed the sub-millisecond runtime of most app invocations (§3.2). To overcome this, Floo incorporates a lookahead engine that opportunistically performs queries *ahead of time* for upcoming function invocations. The key challenge is in coping with uncertainty in downstream computations and state accesses.

Determining upcoming invocations. During offline analysis (§4.1), Floo extracts the call graph for the app and stores it in the lookahead engine's memory; recall from §2.1 that call graphs specify inter-function relationships (including invocation ordering). Call graphs are extracted using Gator [108, 109], and Floo adds additional annotations to each function listing its conservative read/write sets, and whether or not each child is dependent on specific branches being traversed or not.

To determine the set of upcoming invocations at any point in time, the lookahead engine must understand where in the call graph execution currently is. Unfortunately, simply observing function ids for current cache queries does not suffice because a given function can appear in multiple locations in the call graph (§2.1). Instead, during offline analysis, Floo identifies the first function fired in response to each user interaction, e.g., the `onTouch()` handler. These functions each appear only once in the app's call graph, and Floo modifies them such that, when fired, they embed in their cache queries information indicating that a new interaction is being handled, and they are the root of the corresponding subgraph that is about to be traversed. This information is passed from the cache manager to the lookahead engine, which maintains (and updates) a pointer in its in-memory call graph indicating where computation currently is. Localized within that subgraph, the pointer is subsequently updated based on the function ids embedded in cache queries.

Even with an up-to-date pointer in the call graph, the lookahead engine can only identify the set of *potential* upcoming invocations. The uncertainty is rooted in the fact that functions will traverse specific control flows based on their specific input values and nondeterminism, and each set of control flows may invoke only a subset of the function's children in the call graph. To bound the risk of performing queries for functions that will not be invoked, the lookahead engine limits the number of uncertain functions (and their children) that it queries for using the annotations described above about branching.

Performing lookahead queries. The main challenge with lookahead queries is that other functions will be executed before the predicted one, and those functions may modify state that the predicted one accesses. Thus, values for certain read state may not be known at the time the lookahead query is performed. To handle this, upon deciding to perform a query for a downstream invocation of function $func$, the lookahead engine conservatively traverses the call graph from the current pointer up until $func$ and aggregates the write state across all passed functions. The conservative nature of the traversal manifests in two ways: (1) any uncertainty in the call graph resolution is resolved by considering all possible traversals, and (2) the execution of any registered but unexecuted asynchronous handlers are considered possible at any time, e.g., network handlers.

The aggregate write set represents the state that *may* be modified by the time $func$ is invoked, and the lookahead engine compares it with the (conservative) read set for $func$. Any variables in $func$'s read set that do not appear in the aggregate write set will use values that are known and can be collected immediately; the lookahead engine uses Floo accessor methods to extract values from the appropriate contexts. All other variables that intersect across the two lists are assigned a wildcard value of $*$.The lookahead engine then packages the query in its normal form and issues it to the cache manager.

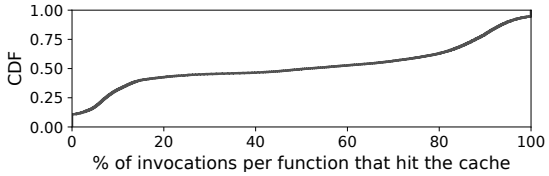


Figure 9: Per-function (memoization) cache hit rates.

Query execution operates as normal, but with one change to handle wildcards: variables with $*$ can match an entry on any value. Thus, queries without wildcards can return at most one result, while queries with wildcards can return multiple matches. Return entries are stored in a small section of Floo's cache called the microcache; in our current implementation, the microcache size is limited to $10^{-4}\times$ the size of the entire compute cache. The cache manager maintains a list of function id's whose lookahead results currently reside in the microcache. Upon receiving a standard query, the cache manager first consults the id list to determine if the function has been part of a lookahead. If so, the query is executed on the few microcache entries, resulting in a near-instantaneous lookup; wildcards are resolved based on the actual values in the query. If not, the normal cache is queried. Upon a microcache hit, the entries for that function are evicted and the id list is updated.

4.4 Cache Management

Floo embeds custom cache admission and eviction policies that aim to maximize speedups within a resource budget by leveraging several empirical observations about app computations (and memoization benefits). We describe these policies in turn.

Intelligent admission. To reduce the stress on cache memory and thread resources, Floo leverages our observation (Figure 9) that certain functions almost never hit in the cache across their invocations. To detect such functions, Floo's cache manager keeps track of the hit rate for each function after its initial entry is added to the cache. If that hit rate ever drops below a pre-defined threshold (5% in our implementation), Floo deactives that function from the memoization process, i.e., the function stops querying or populating the cache, and its existing entries are removed. Floo periodically reactivates such functions and restarts tracking to cope with app or user changes.

Intelligent eviction. To handle scarcity in cache space, Floo eschews generic policies such as LRU, and instead opts for a custom one that directly ties entry utility to memoization speedups. More specifically, the expected speedup that a given entry yields is governed by the (1) runtime of the original function, (2) query execution delay, (3) time to apply the write set, and (4) number of hits. Given the periodicity in user interaction patterns with apps [50] and the stability in app codebases, Floo uses previously delivered speedups for an entry as an approximate indicator of expected future benefits. The key focus is then on how to collect past speedup information with low overhead.

Floo's lightweight eviction strategy is rooted in the observations shown in Figure 10. Despite variations in incurred control and data flows, function runtimes are stable both with and without memoization (i.e., across cache entries). For instance, standard deviation and CV across runtimes for the median function in our corpus were only

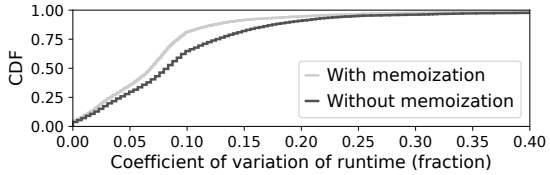


Figure 10: Coefficient of variation (CV) in runtime across invocations of each function with and without memoization.

7 µs and .08 without memoization. Building on these findings, rather than recording all invocation runtimes, Floo's cache manager records the runtimes for each function with and without memoization by analyzing only the first cache query and hit for that function; speedup per hit for the function is defined as the difference in these values.[1] In addition, Floo records the number of hits and the time of the last hit per entry by *passively* observing cache messages and updating only on hits. Per-entry speedups are then computed as the product of the hit count for the entry and speedup per hit for the function, and per-entry utilities normalize those values by space consumed in the cache.

Using the above information, Floo employs a tiered eviction strategy. As cache space is required, Floo first determines if any entry has not been part of a cache hit over a predefined time window (900 seconds in our implementation). If so, those entries are first removed in the order of increasing utility. This ensures that entries which once delivered high speedups but no longer do (e.g., due to app updates) are eventually cycled out of the cache. If more cache space is required, additional entries are then removed in order of increasing utilities.

4.5 Verifying Correctness

Recall the correctness definition from §3, i.e., indistinguishability in user-perceived app operation and generated app state. Preserving correctness is central to Floo's design, manifesting via the use of offline (conservative) static analysis, control/data-flow aware caching, locking semantics on the cache, and the exclusion of functions using nondeterministic APIs. However, we note that Floo's efforts to preserve correctness come at the app level, not lower parts of the computation stack. For example, race conditions across runs of an app could arise due to variations in thread scheduling decisions by the OS, resulting in a violation of the correctness definition above [73]. We deem such discrepancies as acceptable from a correctness perspective as they could arise across runs of an app without Floo. Floo does replay the most common app-level locking mechanisms that developers include to protect against races (via standard reads and writes to Android Locks and Java synchronization monitors), but it does not provide protection to applications that use lower-level synchronization techniques such as Java atomic classes (Java.Util.Concurrent.Atomic). While this is a fundamental limitation of Floo, future improvements in program analysis techniques to detect arbitrary synchronization mechanisms may be leveraged to improve Floo.

To study Floo's ability to preserve this definition of correctness, we performed multiple experiments in which we recorded content downloads while loading all of the apps and traces in our corpus (§6.1), and then replayed that content (and those interactions) with and without Floo [47]. The experiments (described in turn, below) force the same return values for the nondeterministic APIs in Table 1 across both runs during replay. Additionally, to enable more fine-grained correctness analysis, we record all reads and writes to global variables in each function invocation during replay; note that performance is not considered in this experiment, enabling the use of such costly dynamic instrumentation. Correctness was measured by comparing runs with and without Floo on two metrics: (1) pixel-wise screen comparisons and identical Android views [7] after each interaction, and (2) full heap equivalence after each trace.

In our first experiment, we forced computations during replay to operate on a single thread. This avoids scheduling races from lower in the computation stack, and thus implies that any observed heap variations could be attributed to errors with Floo. For all apps and traces, we verified that Floo always met both parts of the correctness definition described above. To understand why, we used the read/write logs collected during each trace to investigate the efficacy of Floo's correctness techniques described above. First, recall that Floo uses static analysis to determine the potential read state for each function which, although not precise to a particular run, aims to be comprehensive. As a stress test of our implementation, we confirmed that the set of variable reads made during each invocation was *always* a subset of the static analysis output, i.e., the function's signature. Second, to ensure that Floo's cache entires were complete and that its cache lookup logic (i.e., deep comparisons and object `hashCodes` when available) was properly enforced, we verified that the writes applied after each cache hit matched those made during the run without Floo.

The setup for our second experiment matched that from the first, except that we relaxed the single-threading requirement, and instead allowed apps to operate across as many threads as they normally would. Out of the 50 apps in our corpus, Floo achieved both screen and heap equivalence on 33 apps. The remaining 17 apps all exhibited some form of heap divergence; 11 of the 17 also involved screen differences. Using our fine-grained logs, we dug deeper into the causes of these divergences, and have categorized them below and provided representative examples.

Variations in cross-thread compute schedules (14 apps): The first category of divergence occurs as the thread schedule across multiple runs of an app may be different. For instance, PicsArt (500M+ downloads) is a video editing app for Android. In this app, the camera is activated briefly during one of our interaction traces. The app receives a sequence of byte arrays (`Byte[]`) corresponding to the sequence of frames captured when the camera is active. Each `Byte[]` is processed by a background worker into a Bitmap and stored in memory. The next interaction in the trace deactivates the camera, upon which the app cancels any lingering frame processing operations (by sending `kill` signals to the corresponding threads). With Floo, due to compute speedups, more frame processing operations have been completed successfully by the time the next interaction is issued (although the same number of frames were sent to both versions deterministically), leading to a Bitmap vector with more entries. A similar heap divergence could also occur as threads may complete in a different order across runs. For instance, WebToon (100M+ downloads) is a comic book reader for Android. When displaying the list of books available, since the item-ordering is unimportant, the app uses fork-join parallelism to load the cached comic books, causing the order of entries on the screen, as well as in the data structure on the heap, to differ.

We note that divergences of this type arise even without Floo. Indeed, when applying every trace 20 times on the unmodified versions of these 14 apps (while also forcing determinism for the APIs in Table 1), we found at least one such heap divergence per app.

Modified spin lock behavior leading to app alterations (3 apps): This category of divergence pertains to scenarios in which app logic is altered by Floo's use of memoization. For example, U-Dictionary (100M+ downloads) is a translation and dictionary app for Android. The app busy-waits in a loop using an `AtomicBoolean` until a wordList variable is initialized and

[1] Functions with negative speedups are deactivated as with admission.

loaded by another thread. With Floo, the busy-wait is eliminated, causing a `NullPointerException` crash when the wordList is accessed before being initialized. Such behavior need not always cause a crash — for instance, in the Hopper Travel app (10M+ downloads) a similar `Exception` has been handled with a retry by the developer, preventing a crash. Note that such cases are not observed in a single-threaded setting as event-ordering is maintained. Further, Floo correctly handles other forms of locking that make use of Android or Java primitives.

5 IMPLEMENTATION

For each app, we extract Java bytecode by disassembling the app's APK into *.smali* files using apktool [98]. Bytecode files for AOSP code (including for the Android platform and OpenJDK) are extracted from the corresponding *dex* files. The bytecode from both sources is then passed through Soot [99] and converted into an intermediate representation called Jimple [100], on which we perform static analysis and instrumentation. The decision to run on Jimple was purely a pragmatic one: it is a typed, 3-address, statement-based IR that involves only tens of statement types compared to the >200 possible instructions in Java bytecode. Across these steps, our offline analysis components comprise ≈1100 lines of Java code.

Updated app-defined code is combined into a new APK with bytecode versions of Floo's cache manager and lookahead engine that are generated using AndroidStudio. Post re-assembly, the APK for the median app in our corpus grew from 29 to 34 MB with Floo. In contrast, updated AOSP code is directly flashed onto the target smartphone. AOSP code grew by 71 MB; however, we note that this would be incurred only once per device (rather than once per app). Importantly, AOSP code is modified to include a registration function that points cache accesses to the appropriate caching classes inside each APK.

During operation, the cache manager and lookahead engine each run on a dedicated background thread, and they were collectively implemented in ≈4600 lines of Java code. Cache entries and callgraphs are stored in memory (siloed across apps), and Java ExecutorServices are used as background workers to service cache operations. Between app sessions, cache entries are persisted to the disk using Kyro v5 [4].

6 EVALUATION

We evaluated Floo across a wide range of popular apps, live mobile networks (and origin servers), real phones, and realistic user interaction traces. Our key findings are:

- Floo reduces median and 95th percentile compute delays by 43.8% and 77.6% (0.55 and 0.98 seconds) for the median per-app interaction in each case, which translates to overall interaction response time improvements of 32.7% and 72.3%.
- Floo delivers larger speedups than recent app optimizations that (1) offload computations to cloud servers (by 6-22%), and (2) reduce network delays via caching and prefetching (by 1.1-1.7×); Floo is complementary to the latter, and running them together lowers response times by 40.3-43.9%.
- Floo does not inflate energy usage, and its speedups are within 3.6-15.2% of the unachievable optimal that performs zero-overhead and perfect memoization; both findings are due to Floo's ability to efficiently mask query overheads and preserve (and enforce) most potential cache hits.
- Floo is amenable to partial deployment: computation improvements are 24.1% and 11.2% when only performing memoization on platform code (Android modifications) or app-defined code (app APK modifications).

6.1 Methodology

Apps. We crawled the Google Play Store [38] lists for popular Android apps in a wide range of categories such as entertainment, sports, lifestyle, weather, and shopping. Our crawl took place in October 2021 and returned 75 apps. From this set, we focus our experiments on the 50 apps that the Soot bytecode analyzer (which Floo builds atop, §5) could run on without error, as the other 25 apps crash during static analysis or launch due to known limitations of Soot [1, 2].

Smartphones and networks. Our experiments consider two smartphones: a Google Pixel 5 (Android 11) [32] and a less powerful Samsung Galaxy Note 9 (Android 10) [33]. We ran all experiments on both phones; trends for any results shown for only one phone (due to space constraints) hold for the other phone. Experiments were run over live (campus) WiFi and Verizon LTE mobile networks with strong signal strength.

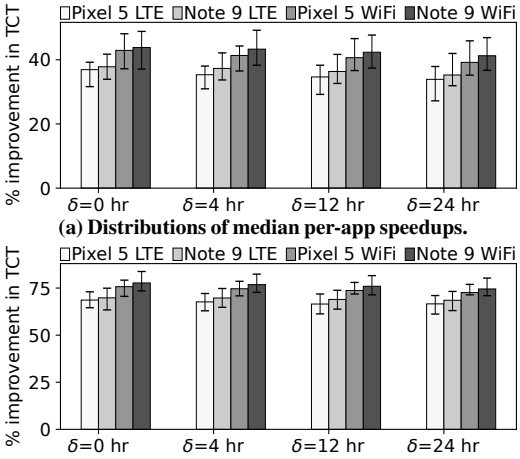
Interaction traces. As in recent studies [86], to generate realistic user interaction traces for our apps, we use the Humanoid app testing framework [56]. In particular, we generated 20 traces per app using the default Humanoid deep neural network that was trained on real user traces over the Rico dataset [24]. Each trace lasts for 3 minutes (matching prior reports about user sessions with apps [30, 111]), and specifies an ordered list of timestamped actions to perform (e.g., taps, scrolls) that accounts for user think time. In our traces, taps (53%) and swipes (39%) comprise the majority of interactions, with long presses accounting for the rest. These traces trigger meaningful parts of the app; for instance, an interaction trace on the BBC News app opens two articles from the app's main screen in succession, and swipes through the embedded gallery of images in the second article. Following this, the trace navigates to the "Popular" tab, then taps and scrolls on an article.

Performance metrics. We evaluated Floo on two metrics: (1) *total computation time* (TCT), or the critical path of time spent only executing app binary or platform code (i.e., no network operation) during an interaction, and (2) *interaction response time* (IRT) measured as the time an interaction is performed to the time when the final response screen is fully rendered. IRTs are measured in a manner analogous with the Speed Index web performance metric [37, 74], i.e., by recording the phone's screen using ffmpeg [3] and tracking when visual changes for non-dynamic pixels (i.e., excluding videos) have halted.

Experimental workflow. Matching recent testbeds [20, 86] and studies about user-app interaction patterns [50], in each experiment, we randomly select and apply an interaction trace, wait δ minutes, and repeat this for a total of 5 traces per app. We consider δ values ranging from 0 mins (i.e., back to back) to 1 day. In each experiment, to ensure fair comparisons despite using live origin servers, networks, and app content, we run each trace back-to-back using each system under test. Unless otherwise noted, Floo was granted access to 32 device threads and 512 MB of memory.

6.2 Interaction Speedups

Figure 11 shows Floo's ability to reduce app compute delays. Median TCT speedups were 36.9-43.8% (0.46-0.54 sec) and 68.6-77.6% (0.86-0.98 sec) for the median and 95th percentile per-app interactions across all considered conditions. There are two main trends to note. First, for a given network, Floo's speedups are slightly larger on the Note 9 than the Pixel 5 as the former possesses a slower CPU, thereby increasing the effective time savings for each cache hit that Floo brings. For example, on WiFi, TCT improvements for the median per-app interactions are 39.1-42.9% to 41.2-43.8% on the two devices. Second, for a given phone, TCT speedups are



(a) Distributions of median per-app speedups.

(b) Distributions of 95th percentile per-app speedups.

Figure 11: TCT improvements with Floo compared to default apps. Bars list median or 95th percentile speedups for the median app, and error bars span the 25-75th percentiles.

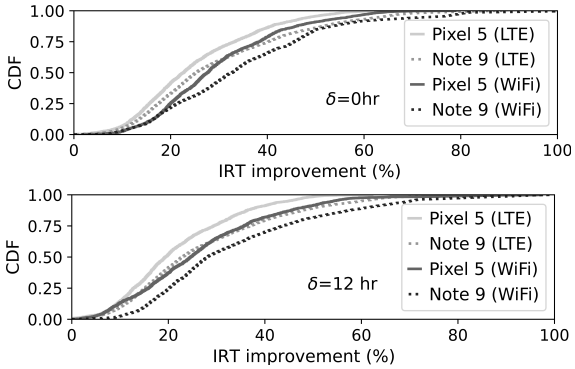


Figure 12: IRT improvements with Floo vs. default apps.

larger on WiFi than LTE, e.g., median speedups grow from 36.8% to 42.9% with the Pixel 5. The reason is the lower round trip times on WiFi which result in additional time blocked on client-side compute.

Figure 12 shows how these compute speedups translate into faster interaction response times. Overall, median IRT improvements were 19.3-32.7%, while 95th percentile speedups were 44.8-72.3%. The same cross-device and cross-network trends as discussed above for TCT hold for IRT.

Figures 11-12 also highlight that Floo's improvements marginally decrease as the time between interaction sessions (i.e., δ) increases. For instance, with LTE and a Note 9, TCT speedup for the median per-app interaction drops from 37.8% to 35.2% as δ grows from 0 to 12 hours. The reason is that, although app computations are entirely in stable app binary code, downloaded content can influence the specific set of functions invoked during interaction handling (i.e., the path through the call graph), and the control/data flows traversed in each invocation. For example, in the BBC News app, a launch of a category screen operates differently depending on whether a downloaded JSON file embeds a 'breaking news' heading. Content changes grow with larger δ values, resulting in previously unseen traversals, and thus fewer cache hits (Table 3).

6.3 Comparison with State-of-the-Art

Network optimizations. We first compared Floo with the recent Marauder app accelerator [86] that alleviates network overheads in interaction handling by using intelligent and lightweight caching and prefetching strategies. Our comparison focused on the 27 apps in our corpus that used the OkHttp caching library (and are thus compatible with Marauder).

δ	Median (95th percentile) cache hit rate
Back-to-back	83.1% (89.4%)
4 hours	80.9% (85.3%)
12 hours	79.9% (83.7%)
24 hours	77.5% (82.6%)

Table 3: Overall cache hit rates with varying δ values.

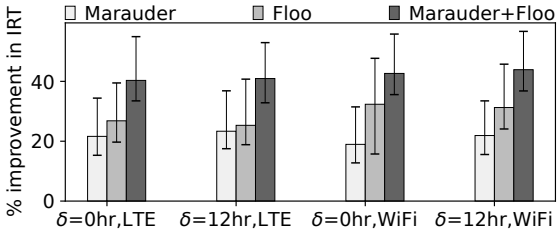


Figure 13: Comparing Floo and Marauder [86] on a Pixel 5. Bars list medians, with error bars for 25-75th percentiles.

As shown in Figure 13, Floo delivers larger IRT speedups than Marauder. For instance, on the LTE network with a δ of 0, median and 75th percentile IRT improvements (compared to default apps) are 1.2× and 1.14× larger with Floo than Marauder. Floo's relative benefits compared to Marauder grow as experiments (1) shift to WiFi, e.g., median speedups jump to 1.7×, or (2) use smaller δ values, e.g., median speedups on LTE drop to 1.08× when δ grows to 12 hours. The former trend occurs due to WiFi's lower round trip times, which lead to more interactions being bottlenecked by compute. The second trend occurs because larger δ values entail more (unnecessary) cache misses (e.g., due to suboptimal TTLs), and thus more opportunities for Marauder to accelerate requests. In contrast, as noted above, Floo exhibits slightly lower cache hit rates as δ increases (Table 3). Importantly, Figure 13 also confirms that Floo and Marauder are largely complementary to one another, with the combined systems outperforming each in isolation.

Compute optimizations. We compared Floo with Tango [42], an app accelerator that offloads certain compute tasks to powerful cloud servers. Unlike prior offloading approaches [23, 52], Tango opts to replicate (not partition) all computations across the phone and cloud. During app operation, Tango shares user interactions with the cloud, and then flip flops between the replicas on a per-invocation basis, aiming to display each screen update to the user based on whichever (leader) replica generates it first. Deterministic replay techniques are used to keep the two replicas in sync, and the functions that issue network requests are pinned to the cloud to leverage its fast wired links. Note that only screen updates are shared (in a pipelined fashion) from the cloud to the phone; the phone still executes all interaction handling as it may become the leader at any time.

Tango is not publicly available, so we instead use an in-house version. Prior to the experiment, our Tango variant executes interaction traces and records all nondeterministic values from Android APIs and network fetches. These values are served at both the phone and cloud server (a 32 core, 128 GB RAM server) during replay; network fetches are handled by a server that is co-located with the Tango cloud server, which sits on the other side of the phone's mobile access link. This represents a favorable scenario for Tango for two reasons. First, network fetches incur near-zero latency from the cloud server. Second, all (blocking) coordination for exchanging nondeterministic values from the leader to follower is eliminated.

As shown in Figure 14, speedups with Floo are 1.06-1.14× and 1.19-1.22× larger than those with Tango on WiFi and LTE; for context, Floo's speedups come without the management and security

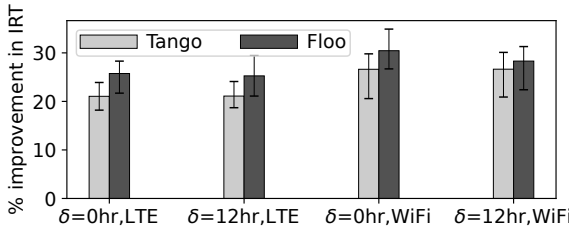


Figure 14: Floo vs. Tango [42] on Note 9. Bars list medians, with error bars for 25-75th percentiles.

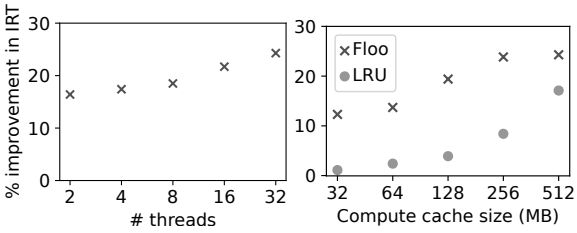


Figure 15: Floo's speedups with varying mobile device resources. Caching results compare with a generic LRU policy. Results are for the Pixel 5 and LTE, and list median IRT wins across all apps.

overheads of running a proxy server (§7). The reason is that, in scenarios where the cloud is deemed the leader (the only ones where Tango can provide a speedup), Tango must share updates from each invocation that alters the screen. 0.3% (1800) of invocations on the median interaction update the screen, adding substantial blocking delays to those already incurred from sharing interactions with the cloud. Floo's larger wins on LTE versus WiFi are because each blocking delay with Tango involves a larger round trip time on LTE.

We also considered recent memoization efforts for Android [83, 90] by comparing Floo with a version of itself that only enables reuse of computations for pure functions, i.e., those whose only external effects are return values that are influenced solely by function arguments. This is a favorable comparison since prior efforts do not consider platform code that accounts for a large fraction of compute delays (§2). Overall, we find that Floo delivers 15-21.3× larger speedups than these efforts across all phones and δ values. The underlying reason is straightforward: only 1.4% of invocations and 2.1% of runtime in our apps come from pure functions.

6.4 Analyzing Floo

Varying device resources. We stress-tested Floo's ability to maximize the utility of limited smartphone resources by (separately) varying the compute threads it had access to, and the memory available for its compute cache. Figure 15 illustrates the trends in Floo's performance, highlighting two takeaways. First, fewer compute threads result in lower speedups as fewer lookaheads can be performed, which in turn leads to more blocking cache queries and a larger number of functions being deactivated from memoization. Second, Floo's performance (unsurprisingly) degrades as cache sizes decrease and fewer results can be stored for reuse. However, owing to Floo's memoization-specific eviction policy, the degradation is minimal and far slower than when generic caching policies are used, e.g., cutting the cache from 512 to 32 MB drops speedups by 2.0× and 15.5× with Floo's eviction strategy and LRU.

Importance of each technique. Table 4 shows the performance impact of selectively disabling Floo's optimizations: (1) control flow-aware caching, (2) lookahead queries, and (3) memoization-centric cache admission (Figure 15 studies eviction). As shown, all three techniques are core to Floo's speedups, with raw degradations in

Disabled feature	Median (95 %ile) IRT improvement
None (complete Floo)	29.0% (65.4%)
Control flow-aware caching	11.2% (21.4%)
Lookahead queries	2.3% (12.1%)
Intelligent cache admission	20.1% (39.1%)

Table 4: Importance of Floo's optimizations. Results are for the Pixel 5 and LTE.

Computation type	Raw compute time (ms)	Floo's TCT speedup (%)	Potential TCT speedup (%)
AOSP (platform)	894	24.1	39.3
App-defined	371	11.2	14.8
Total	1265	36.9	55.9

Table 5: Speedups from Floo and (unachievable) potential memoization when run on different types of app computations. Results are for the median interactions across apps, LTE, and the Pixel 5.

median IRT wins of 17.8%, 26.7%, and 8.9% when they are disabled, respectively. Cache lookaheads and control flow-aware caching are most crucial due to the many short invocations in apps; blocking lookups would exceed runtimes for most invocations, while reduced hit rates would yield minimal compute savings. When lookahead is enabled, the microcache is populated 83µs prior to the median lookup. Floo's cache admission policy importantly deactivates functions that do not benefit from memoization; removing this feature increases the invocations slowed down by Floo from 3% to 31%.

Dissecting Floo's speedups. Table 5 breaks down Floo's TCT wins when running on different types of app computations. As shown, both AOSP platform code and app-defined code are amenable to Floo's memoization optimization, with median TCT speedups of 24.1% and 11.2%, respectively. Table 5 also highlights that Floo's benefits are within 3.6-15.2% of the potential memoization savings from §3. Recall that those savings represent an unachievable upper bound in that they assume perfect hit rates, and zero delay for querying the cache and applying the writes for each hit. Thus, these results illustrate Floo's ability to effectively mask cache management delays and deliver most potential hits in an entirely online fashion.

Case studies. After studying the nature of apps and interactions in our corpus, we identify two broad categories of apps based on their client-side computations. The characteristics of these computations influence the impact of Floo's optimizations. The first category of apps embed a significant amount of client-side computations, while the second category of apps act as a "thin-client" with minimal client-side computations, or load their computations dynamically at runtime. Results in §3.1 indicate that apps in the first category are expected to be amenable to Floo's optimizations — examples include the PicsArt video editor (500M+ downloads), and shopping apps such as Nike (10M+ downloads) and Bath&Body Works (1M+ downloads). Specifically, in the Nike app, interactions that open product details were largely accelerated by Floo. In these interactions, we found that the app used an image processing library to apply transformations (decode, crop, blur or rotate) on a series of product images. These transformations repeatedly impose significant computational overheads every time a product page is reopened, allowing such interactions to be largely accelerated by Floo. In the second category of apps, due to the offline analysis adopted by Floo (§4.1), apps that load their computations dynamically are not amenable to Floo's optimizations. Examples for this category of apps include NewsBreak (50M+ downloads) and RSS Reader (100K+ downloads). Specifically, we found that Floo was unable to accelerate a large portion of interactions in the NewsBreak app. In this app, we found that articles were loaded in

a WebView, with JavaScript computations being downloaded and executed at runtime. Such computations are beyond Floo's purview, and remain entirely unoptimized, leading to unaccelerated interactions.

Impact on device resources. We executed all interaction traces per app with and without Floo, and recorded the CPU usage with Android Profiler [26] and total energy usage with Android BatteryStats [27]. Overall, while Floo increases the peak CPU usage of the median app in our corpus by 1.07× , it is 'net neutral' on energy usage, with mild savings of 2% and 5% for the median and 95th percentile apps. In other words, the additional compute used for cache management and lookaheads is negated by computation reductions from memoization, i.e., Floo can boost IRT and TCT for a given energy usage. Key to this is the judiciousness with which Floo performs lookaheads; only 4.1% of lookahead queries go unused by downstream invocations.

Cross-app benefits. Our implementation currently siloes apps and their caches to preserve existing privacy and data sharing semantics. However, apps involve significant computations that are part of the (shared) platform AOSP code, and also often build on third-party libraries (§2). Further analysis reveals promise for reusing computations across apps: 6.3% and 19.2% of invocations and runtime for the median app overlapping with the computations in at least one other app in our corpus.

7 RELATED WORK

App network optimizations. Numerous systems address network bottlenecks in app operation using prefetching and caching. For prefetching, one line of systems uses static analysis on app source code to identify resources to fetch early, either one callback early via a local proxy [112] or in batches via a remote proxy [20]. Other systems generate prefetching strategies by passively monitoring issued requests to identify inter-resource dependencies [45], or by tuning app-specified prefetching policies according to network conditions or past hit rates [13, 46]. For caching, many studies have documented the inefficiencies of HTTP caches [59, 61, 72, 84, 85, 110, 112], and provided solutions in the form of finer-grained caching [65, 81, 104] and prefetching to refresh TTLs [86]. As shown in §6.3, Floo delivers larger (but complementary) speedups than network-focused accelerators since apps are increasingly compute-bottlenecked (§2.2).

App compute optimizations. Multiple systems tackle smartphone resource limitations by offloading app tasks to well-provisioned cloud servers [21, 23, 43, 58]. For example, MAUI [23] offloads functions based on an optimization engine that considers energy usage, compute delays, and state sharing costs. More recently, Tango [42] eschews task partitioning in favor of replication. Floo provides larger speedups than offloading by avoiding costly phone-server coordination overheads (§6.3). More importantly, by focusing purely on client-side memoization, Floo sidesteps the scalability, cost, and security issues of using a proxy server [75, 92].

SmartIO[77] accelerates app computations by carefully reordering disk accesses to cater to the large number of (concurrent) reads made during app startup and interaction handling. These I/O optimizations are enforced via a modified operating system. Floo and SmartIO share the same goal – improved app responsiveness – and can run alongside one another as both operate in a purely reactive manner relative to the computations that are actually triggered at runtime.

A slew of systems improve the performance of specific classes of mobile apps using domain-specific optimizations, e.g., lowering inference overheads for vision recognition [14, 19, 44, 57]. In contrast, Floo aims to reduce app compute delays in any Android app by maximally reusing Java computations.

App energy optimizations. Certain frameworks profile app operation offline, and leverage the garnered insights to lower energy usage by tuning knobs in the computation stack (e.g., CPU frequency, memory bandwidth) [87] or refactoring/reorganizing API calls [12]. Other efforts automatically detect and bundle HTTP requests to lower energy usage via longer radio idle times [55]. Floo is complementary to these efforts, and instead focuses on maximizing user-perceived performance for a fixed amount of energy usage (§6.4). Floo can work on the energy-optimized code output by these frameworks.

Predicting user behaviors in apps. Falcon [107] and PREPP [80] predict (and preload) app launches based on observations about user location, access patterns, and device sensors. Floo eschews (error-prone [79, 88]) prediction and instead applies memoization to already-issued computations. That said, Floo could reduce the overheads of preloading apps, enabling more preloads in a fixed resource budget. Relatedly, other efforts predict user interactions for improved analytics [60, 97]; these works are orthogonal to Floo, but could enable longer-term predictions for lookahead queries.

Java program optimization. Floo is inspired by a larger body of prior work that optimize Java programs, but instead targets the Android framework (not general Java), and integrates new techniques to tackle the computation patterns and resource restrictions in mobile apps and phones. Beyond memoization [5, 25], other efforts automatically parallelize Java programs to take advantage of multi-core platforms [9, 15, 18, 51, 53, 67]. Though conceptually complementary to Floo, we note that our results highlight substantial state sharing across invocations in app interaction handling, thereby limiting the potential for safe parallelism (§3).

8 CONCLUSION

Computation delays from executing source code in mobile app binaries and OSes increasingly govern the responsiveness that apps deliver to end users. To address this worsening bottleneck, we presented Floo, a system that automatically integrates memoization into app binaries to reduce the amount of computation required to respond to latency-sensitive user interactions. Key to Floo are its techniques to simultaneously ensure correctness for all reused computations while reaping most potential speedups via masked cache lookup overheads and high cache hit rates. Overall, Floo reduces median and 95th percentile interaction response times by 32.7% and 72.3%. More broadly, we hope that the adaptation mechanisms that Floo employs help promote next-generation optimizations that explicitly (and more directly) cater to the increasingly heterogeneous device and network profiles that global mobile app users provide.

Acknowledgements: We thank Harsha Madhyastha, Amit Levy, Anirudh Sivaraman, and Aishwarya Sivaraman for their valuable feedback on earlier drafts of the paper. We also thank our shepherd, Alec Wolman, and the anonymous MobiSys reviewers for their constructive comments. This work was supported in part by NSF grants CNS-2140552, CNS-2151630, and CNS-2152313, as well as a Sloan Research Fellowship.

REFERENCES

[1] 2018. Soot Issue: APKs are crashing after a simple unpack/repack with Soot. https://github.com/soot-oss/soot/issues/999.
[2] 2019. Soot Issue: Android App crashes after Soot Instrumentation even with empty internalTransform() body. https://github.com/soot-oss/soot/issues/1152.
[3] 2020. FFmpeg. https://ffmpeg.org/.
[4] 2021. Kyro. https://github.com/EsotericSoftware/kryo.
[5] Giovanni Agosta, Marco Bessi, Eugenio Capra, and Chiara Francalanci. 2011. Dynamic memoization for energy efficiency in financial applications. In *2011 International Green Computing Conference and Workshops*. IEEE, 1–8.

[6] Android Developers. 2020. Keeping your app responsive. https://developer.android.com/training/articles/perf-anr#Reinforcing.

[7] Android Developers. 2021. Android API Reference. https://developer.android.com/reference/android/view/View.

[8] AppDynamics. 2014. AppDynamics Releases App Attention Span Study Which Shows Nearly 90 Percent Surveyed Stopped. https://bit.ly/39pMAXM.

[9] Pedro V Artigas, Manish Gupta, Samuel P Midkiff, and José E Moreira. 2000. Automatic loop transformations and parallelization for Java. In *Proceedings of the 14th international conference on Supercomputing*. 1–10.

[10] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

[11] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39.

[12] Abhijeet Banerjee and Abhik Roychoudhury. 2016. Automated re-factoring of android apps to enhance energy-efficiency. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*. 139–150.

[13] Paul Baumann and Silvia Santini. 2017. Every byte counts: Selective prefetching for mobile applications. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 1, 2 (2017), 1–29.

[14] Kevin Boos, David Chu, and Eduardo Cuervo. 2016. Flashback: Immersive virtual reality on mobile devices via rendering memoization. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*. 291–304.

[15] Borys J Bradel and Tarek S Abdelrahman. 2007. Automatic trace-based parallelization of java programs. In *2007 International Conference on Parallel Processing (ICPP 2007)*. IEEE, 26–26.

[16] Michael Butkiewicz, Daimeng Wang, Zhe Wu, Harsha Madhyastha, and Vyas Sekar. 2015. Klotski: Reprioritizing Web Content to Improve User Experience on Mobile Devices. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Oakland, CA) *(NSDI)*. USENIX Association.

[17] Bogdan Carbunar and Rahul Potharaju. 2015. A longitudinal study of the google app market. In *Proceedings of the 2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2015*. 242–249.

[18] Michael K Chen and Kunle Olukotun. 2003. The Jrpm system for dynamically parallelizing Java programs. In *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.* IEEE, 434–445.

[19] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. 2015. Glimpse: Continuous, Real-Time Object Recognition on Mobile Devices. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*. 155–168.

[20] Byungkwon Choi, Jeongmin Kim, Daeyang Cho, Seongmin Kim, and Dongsu Han. 2018. Appx: an automated app acceleration framework for low latency mobile app. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*. 27–40.

[21] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. 2011. CloneCloud: Elastic Execution between Mobile Device and Cloud. In *Proceedings of the Sixth Conference on Computer Systems* (Salzburg, Austria) *(EuroSys '11)*. Association for Computing Machinery, 301–314.

[22] Cisco. 2021. Cisco Annual Internet Report - Cisco Annual Internet Report Highlights Tool. https://www.cisco.com/c/en/us/solutions/executive-perspectives/annual-internet-report/air-highlights.html.

[23] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. 2010. MAUI: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*. 49–62.

[24] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 2017. Rico: A Mobile App Dataset for Building Data-Driven Design Applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology* (Québec City, QC, Canada) *(UIST '17)*. Association for Computing Machinery, 845–854.

[25] Luca Della Toffola, Michael Pradel, and Thomas R Gross. 2015. Performance problems you can fix: A dynamic analysis of memoization opportunities. *ACM SIGPLAN Notices* 50, 10 (2015), 607–622.

[26] Android Developers. 2019. Inspect CPU activity with CPU Profiler.

[27] Android Developers. 2019. Profile battery usage with Batterystats and Battery Historian.

[28] Dimensional Research. 2015. Failing to Meet Mobile App User Expectations: A Mobile App User Survey. https://techbeacon.com/sites/default/files/gated_asset/mobile-app-user-survey-failing-meet-user-expectations.pdf.

[29] Zhen Dong, Abhishek Tiwari, Xiao Liang Yu, and Abhik Roychoudhury. 2020. Concurrency-related flaky test detection in android apps. *arXiv preprint arXiv:2005.10762* (2020).

[30] Hossein Falaki, Ratul Mahajan, Srikanth Kandula, Dimitrios Lymberopoulos, Ramesh Govindan, and Deborah Estrin. 2010. Diversity in Smartphone Usage. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services* (San Francisco, California, USA) *(MobiSys '10)*. Association for Computing Machinery, 179–194.

[31] Jiaojiao Fu, Yaohui Wang, Yangfan Zhou, and Xin Wang. 2022. How resource utilization influences UI responsiveness of Android software. *Information and Software Technology* 141 (2022), 106728.

[32] GeekBench. 2021. Google Pixel 5. https://browser.geekbench.com/v5/cpu/11564175.

[33] GeekBench. 2021. Samsung Galaxy Note 9. https://browser.geekbench.com/v5/cpu/11562629.

[34] Y. Geng, Y. Yang, and G. Cao. 2018. Energy-Efficient Computation Offloading for Multicore-Based Mobile Devices. In *IEEE Conference on Computer Communications (INFOCOM)*. 46–54.

[35] Ali Ghanbari and Andrian Marcus. 2021. Toward Speeding up Mutation Analysis by Memoizing Expensive Methods. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. 71–75. https://doi.org/10.1109/ICSE-NIER52604.2021.00023

[36] Ayush Goel, Vaspol Ruamviboonsuk, Ravi Netravali, and Harsha V Madhyastha. 2021. Rethinking Client-Side Caching for the Mobile Web. In *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*. 112–118.

[37] Google. 2012. Speed Index - WebPagetest Documentation. https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index.

[38] Google. 2020. Google Play Store. https://play.google.com/store.

[39] Google. 2020. Volley. https://developer.android.com/training/volley/index.html.

[40] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. 2015. Information flow analysis of android applications in droidsafe.. In *NDSS*, Vol. 15. 110.

[41] Mark S Gordon, David Ke Hong, Peter M Chen, Jason Flinn, Scott Mahlke, and Zhuoqing Morley Mao. 2015. Accelerating mobile applications through flip-flop replication. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. 137–150.

[42] Mark S Gordon, David Ke Hong, Peter M Chen, Jason Flinn, Scott Mahlke, and Zhuoqing Morley Mao. 2015. Tango: accelerating mobile applications through flip-flop replication. *GetMobile: Mobile Computing and Communications* 19, 3 (2015), 10–13.

[43] Mark S. Gordon, D. Anoushe Jamshidi, Scott Mahlke, Z. Morley Mao, and Xu Chen. 2012. COMET: Code Offload by Migrating Execution Transparently. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, 93–106.

[44] Peizhen Guo and Wenjun Hu. 2018. Potluck: Cross-application approximate deduplication for computation-intensive mobile applications. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 271–284.

[45] Yao Guo, Mengxin Liu, and Xiangqun Chen. 2017. Looxy: Web Access Optimization for Mobile Applications with a Local Proxy. In *2017 IEEE 85th Vehicular Technology Conference (VTC Spring)*. IEEE, 1–5.

[46] Brett D Higgins, Jason Flinn, Thomas J Giuli, Brian Noble, Christopher Peplin, and David Watson. 2012. Informed mobile prefetching. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*. 155–168.

[47] Yongjian Hu, Tanzirul Azim, and Iulian Neamtiu. 2015. Versatile yet lightweight record-and-replay for android. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 349–366.

[48] Immobi. 2020. Mobile App Vs Website Statistics: How App Usage Compares To Mobile Web Visits In The United States. https://bit.ly/3sgObI9.

[49] Tushar Jain. 2017. 7 pre-launch mobile app performance metrics to measure. https://kaysharbor.com/blog/mobile/7-pre-launch-mobile-app-performance-metrics.

[50] Simon L. Jones, Denzil Ferreira, Simo Hosio, Jorge Goncalves, and Vassilis Kostakos. 2015. Revisitation Analysis of Smartphone App Use. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing* (Osaka, Japan) *(UbiComp '15)*. Association for Computing Machinery, 1197–1208.

[51] Iffat H Kazi, Howard H Chen, Berdenia Stanley, and David J Lilja. 2000. Techniques for obtaining high performance in Java programs. *ACM Computing Surveys (CSUR)* 32, 3 (2000), 213–240.

[52] Roelof Kemp, Nicholas Palmer, Thilo Kielmann, and Henri Bal. 2010. Cuckoo: a computation offloading framework for smartphones. In *International Conference on Mobile Computing, Applications, and Services*. Springer, 59–79.

[53] Raffi Khatchadourian, Yiming Tang, and Mehdi Bagherzadeh. 2020. Safe automated refactoring for intelligent parallelization of Java 8 streams. *Science of Computer Programming* 195 (2020), 102476.

[54] Seyeon Kim, Kyungmin Bin, Sangtae Ha, Kyunghan Lee, and Song Chong. 2021. zTT: learning-based DVFS with zero thermal throttling for mobile devices. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*. 41–53.

[55] Ding Li, Yingjun Lyu, Jiaping Gui, and William GJ Halfond. 2016. Automated energy optimization of http requests for mobile applications. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 249–260.

[56] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. Humanoid: A Deep Learning-Based Approach to Automated Black-Box Android App Testing. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 1070–1073.

[57] Robert LiKamWa and Lin Zhong. 2015. Starfish: Efficient concurrency support for computer vision applications. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. 213–226.

[58] Chit-Kwan Lin and H. T. Kung. 2014. Mobile App Acceleration via Fine-Grain Offloading to the Cloud. In *Proceedings of the 6th USENIX Conference on Hot Topics in Cloud Computing* (Philadelphia, PA) *(HotCloud'14)*. USENIX Association, 8.

[59] Xuanzhe Liu, Yun Ma, Yunxin Liu, Tao Xie, and Gang Huang. 2015. Demystifying the imperfect client-side cache performance of mobile web browsing. *IEEE Transactions on Mobile Computing* 15, 9 (2015), 2206–2220.

[60] Yozen Liu, Xiaolin Shi, Lucas Pierce, and Xiang Ren. 2019. Characterizing and forecasting user engagement with in-app action graph: A case study of snapchat. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2023–2031.

[61] Yun Ma, Xuanzhe Liu, Shuhui Zhang, Ruirui Xiang, Yunxin Liu, and Tao Xie. 2015. Measurement and analysis of mobile web cache performance. In *Proceedings of the 24th International Conference on World Wide Web*. 691–701.

[62] Ivano Malavolta, Francesco Nocera, Patricia Lago, and Marina Mongiello. 2019. Navigation-aware and personalized prefetching of network requests in Android apps. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 17–20.

[63] Shaghayegh Mardani, Ayush Goel, Ronny Ko, Harsha V. Madhyastha, and Ravi Netravali. 2021. Horcrux: Automatic JavaScript Parallelism for Resource-Efficient Web Computation. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 461–477.

[64] Donald Michie. 1968. "Memo" functions and machine learning. *Nature* 218, 5136 (1968), 19–22.

[65] James Mickens. 2010. Silo: Exploiting JavaScript and DOM Storage for Faster Page Loads.. In *WebApps*.

[66] MindSea. 2020. 28 Mobile App Statistics To Know In 2020. https://mindsea.com/app-stats/.

[67] Anders Møller and Oskar Haarklou Veileborg. 2020. Eliminating abstraction overhead of Java stream pipelines using ahead-of-time program optimization. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–29.

[68] San Murugesan. 2013. Mobile Apps in Africa. *IT Professional* 15, 5 (2013), 8–11. https://doi.org/10.1109/MITP.2013.83

[69] Arvind Narayanan, Eman Ramadan, Jason Carpenter, Qingxu Liu, Yu Liu, Feng Qian, and Zhi-Li Zhang. 2020. A First Look at Commercial 5G Performance on Smartphones. In *Proceedings of The Web Conference 2020* (Taipei, Taiwan) *(WWW '20)*. Association for Computing Machinery, New York, NY, USA, 894–905. https://doi.org/10.1145/3366423.3380169

[70] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. 2016. Polaris: Faster Page Loads Using Fine-grained Dependency Tracking. In *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation* (Santa Clara, CA) *(NSDI)*. USENIX Association, Berkeley, CA, USA.

[71] Ravi Netravali and James Mickens. 2018. Prophecy: Accelerating Mobile Page Loads Using Final-state Write Logs. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation* (Renton, WA) *(NSDI)*. USENIX Association, Berkeley, CA, USA.

[72] Ravi Netravali and James Mickens. 2018. Remote-control caching: Proxy-based url rewriting to decrease mobile browsing bandwidth. In *Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications*. 63–68.

[73] Ravi Netravali and James Mickens. 2019. Reverb: Speculative Debugging for Web Applications. In *Proceedings of the ACM Symposium on Cloud Computing*. 428–440.

[74] Ravi Netravali, Vikram Nathan, James Mickens, and Hari Balakrishnan. 2018. Vesper: Measuring Time-to-Interactivity for Web Pages. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation* (Renton, WA) *(NSDI)*. USENIX Association, Renton, WA, USA.

[75] Ravi Netravali, Anirudh Sivaraman, James Mickens, and Hari Balakrishnan. 2019. WatchTower: Fast, Secure Mobile Page Loads Using Remote Dependency Resolution. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services* (Seoul, Republic of Korea) *(MobiSys '19)*. ACM, 430–443.

[76] R. Netravali, A. Sivaraman, K. Winstein, S. Das, A. Goyal, J. Mickens, and H. Balakrishnan. 2015. Mahimahi: Accurate Record-and-Replay for HTTP *(Proceedings of ATC '15)*. USENIX.

[77] David T Nguyen, Gang Zhou, Guoliang Xing, Xin Qi, Zijiang Hao, Ge Peng, and Qing Yang. 2015. Reducing smartphone application delay through read/write isolation. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. 287–300.

[78] Peter Norvig. 1991. Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics* 17, 1 (1991), 91–98.

[79] Andreas Pamboris and Peter Pietzuch. 2015. Edge Reduce: Eliminating Mobile Network Traffic Using Application-Specific Edge Proxies. In *2015 2nd ACM International Conference on Mobile Software Engineering and Systems*. IEEE, 72–82.

[80] Abhinav Parate, Matthias Böhmer, David Chu, Deepak Ganesan, and Benjamin M Marlin. 2013. Practical prediction and prefetch for faster access to applications on mobile phones. In *Proceedings of the 2013 ACM international joint conference on Pervasive and ubiquitous computing*. 275–284.

[81] KyoungSoo Park, Sunghwan Ihm, Mic Bowman, and Vivek S Pai. 2007. Supporting Practical Content-Addressable Caching with CZIP Compression.. In *USENIX Annual Technical Conference*. 185–198.

[82] Gavin Phillips. 2020. Smartphones vs. Desktops: Why Is My Phone Slower Than My PC? https://www.makeuseof.com/tag/smartphone-desktop-processor-differences/.

[83] Adriano Pinto, Marco Couto, and Jácome Cunha. 2018. Memoization for Saving Energy in Android Applications. (2018).

[84] Feng Qian, Kee Shen Quah, Junxian Huang, Jeffrey Erman, Alexandre Gerber, Zhuoqing Mao, Subhabrata Sen, and Oliver Spatscheck. 2012. Web caching on smartphones: ideal vs. reality. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*. 127–140.

[85] Feng Qian, Subhabrata Sen, and Oliver Spatscheck. 2014. Characterizing resource usage for mobile web browsing. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. 218–231.

[86] Murali Ramanujam, Harsha V. Madhyastha, and Ravi Netravali. 2021. Marauder: Synergized Caching and Prefetching for Low-Risk Mobile App Acceleration. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services* (Virtual Event, Wisconsin) *(MobiSys '21)*. Association for Computing Machinery, New York, NY, USA, 350–362. https://doi.org/10.1145/3458864.3466866

[87] Karthik Rao, Jun Wang, Sudhakar Yalamanchili, Yorai Wardi, and Handong Ye. 2017. Application-specific performance-aware energy optimization on android mobile devices. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 169–180.

[88] Lenin Ravindranath, Sharad Agarwal, Jitendra Padhye, and Chris Riederer. 2014. Procrastinator: pacing mobile apps' usage of the network. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. 232–244.

[89] Hugo Rito and Joao Cachopo. 2010. Memoization of methods using software transactional memory to track internal state dependencies. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*. 89–98.

[90] Rui Rua, Marco Couto, Adriano Pinto, Jácome Cunha, and Joao Saraiva. 2019. Towards using memoization for saving energy in android. (2019).

[91] Vaspol Ruamviboonsuk, Ravi Netravali, Muhammed Uluyol, and Harsha V. Madhyastha. 2017. Vroom: Accelerating the Mobile Web with Server-Aided Dependency Resolution. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM)*. ACM.

[92] Ahiwan Sivakumar, Chuan Jiang, Seong Nam, P.N. Shankaranarayanan, Vijay Gopalakrishnan, Sanjay Rao, Subhabrata Sen, Mithuna Thottethodi, and T.N. Vijaykumar. 2017. Scalable Whittled Proxy Execution for Low-Latency Web over Cellular Networks. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking* (Snowbird, Utah) *(Mobicom)*. ACM.

[93] Daniel Staesser. 2018. The Emergence of Mobile Apps in Developing Countries. https://borgenproject.org/emergence-mobile-apps-in-developing-countries/.

[94] Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 32–41.

[95] Arjun Suresh, Erven Rohou, and André Seznec. 2017. Compile-time function memoization. In *Proceedings of the 26th International Conference on Compiler Construction*. 45–54.

[96] Arjun Suresh, Bharath Narasimha Swamy, Erven Rohou, and André Seznec. 2015. Intercepting functions for memoization: A case study using transcendental functions. *ACM Transactions on Architecture and Code Optimization (TACO)* 12, 2 (2015), 18–1.

[97] Yuan Tian, Ke Zhou, Mounia Lalmas, Yiqun Liu, and Dan Pelleg. 2020. Cohort Modeling Based App Category Usage Prediction. In *Proceedings of the 28th ACM Conference on User Modeling, Adaptation and Personalization*. 248–256.

[98] Connor Tumbleson and Ryszard Wisniewski. 2020. Apktool – A tool for reverse engineering Android apk files. https://ibotpeaches.github.io/Apktool/.

[99] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. 214–224.

[100] Raja Vallee-Rai and Laurie J Hendren. 1998. Jimple: Simplifying Java bytecode for analyses and transformations. (1998).

[101] Haoyu Wang, Hao Li, and Yao Guo. 2019. Understanding the evolution of mobile app ecosystems: A longitudinal measurement study of google play. In *The World Wide Web Conference*. 1988–1999.

[102] Jiadai Wang, Lei Zhao, Jiajia Liu, and Nei Kato. 2019. Smart resource allocation for mobile edge computing: A deep reinforcement learning approach. *IEEE Transactions on emerging topics in computing* (2019).

[103] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. 2013. Demystifying Page Load Performance with WProf. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Lombard, IL) *(NSDI)*. USENIX Association.

[104] Xiao Sophia Wang, Arvind Krishnamurthy, and David Wetherall. 2014. How much can we micro-cache web pages?. In *Proceedings of the 2014 Conference on Internet Measurement Conference*. 249–256.

[105] Fengguo Wei, Sankardas Roy, and Xinming Ou. 2014. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*. 1329–1341.

[106] Dongzhu Xu, Anfu Zhou, Xinyu Zhang, Guixian Wang, Xi Liu, Congkai An, Yiming Shi, Liang Liu, and Huadong Ma. 2020. Understanding Operational 5G: A First Measurement Study on Its Coverage, Performance and Energy Consumption. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM*

'*20)*. Association for Computing Machinery, New York, NY, USA, 479–494. https://doi.org/10.1145/3387514.3405882

[107] Tingxin Yan, David Chu, Deepak Ganesan, Aman Kansal, and Jie Liu. 2012. Fast app launching for mobile devices using predictive user context. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*. 113–126.

[108] Shengqian Yang, Haowei Wu, Hailong Zhang, Yan Wang, Chandrasekar Swaminathan, Dacong Yan, and Atanas Rountev. 2018. Static Window Transition Graphs for Android. *International Journal of Automated Software Engineering* 25, 4 (Dec. 2018), 833–873.

[109] Shengqian Yang, Hailong Zhang, Haowei Wu, Yan Wang, Dacong Yan, and Atanas Rountev. 2015. Static Window Transition Graphs for Android. In *IEEE/ACM International Conference on Automated Software Engineering*. 658–668.

[110] Yifan Zhang, Chiu Tan, and Li Qun. 2013. CacheKeeper: a system-wide web caching service for smartphones. In *Proceedings of the 2013 ACM international joint conference on Pervasive and ubiquitous computing*. 265–274.

[111] Yixue Zhao, Marcelo Schmitt Laser, Yingjun Lyu, and Nenad Medvidovic. 2018. Leveraging program analysis to reduce user-perceived latency in mobile applications. In *Proceedings of the 40th International Conference on Software Engineering*. 176–186.

[112] Yixue Zhao, Paul Wat, Marcelo Schmitt Laser, and Nenad Medvidović. 2018. Empirically assessing opportunities for prefetching and caching in mobile apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 554–564.