End the Senseless Killing: Improving Memory Management for Mobile Operating Systems

Technical Report UW-CSE-2019-04-01

Niel Lebeck
University of Washington

Arvind Krishnamurthy
University of Washington

Henry M. Levy
University of Washington

Irene Zhang
Microsoft Research

Abstract

To ensure low-latency memory allocation, mobile operating systems needing memory kill applications instead of swapping memory to disk. This design choice shifts the burden of managing over-utilized memory to application programmers, requiring them to constantly checkpoint their application state to disk. This paper presents Marvin, a new memory manager for mobile platforms that efficiently supports swapping while meeting the strict performance requirements of mobile apps. Marvin's swap-enabled language runtime is co-designed with OS-level memory management to avoid common pitfalls of traditional swap mechanisms. Its three key features include: (1) a new swap mechanism, called *ahead-of-time (AOT) swap*, which pre-writes memory to disk, then harvests it quickly when needed, (2) a modified bookmarking garbage collector that avoids swapping in unused memory, and (3) an object-granularity working set estimator. Our experiments show that Marvin can run more than 2x as many concurrent apps as Android, and that Marvin can reclaim memory over 60x faster than Android with a Linux swap file can allocate memory under memory pressure.

1 Introduction

Over the past decade, mobile apps have grown appreciably in complexity and size [21], though mobile device memory has not [4]. This trend has increased memory pressure on mobile operating systems as apps compete for limited space. Going forward, mobile OSes must more efficiently share memory across demanding apps, or user experience will suffer.

Unfortunately, while mobile apps have become more sophisticated, mobile memory management remains in its infancy. Today's popular mobile operating systems set a fixed upper bound on memory for each running application (e.g., 1.4GB for iOS running on an iPhone X [30] and 512MB for Android running on a Google Pixel XL). They never overcommit memory; instead, they kill running applications when memory runs out and restart them later.

This simplistic approach worked well when mobile apps were small and largely stateless. However, it is unsustainable as mobile apps move closer towards replacing desktop applications (e.g., Google Docs and Word Online replacing Microsoft Word). Today's apps already do not fit into their memory allocation, so they manually swap objects between memory and local storage or use libraries to meet their needs [10]. Because apps are increasingly likely to be killed due to memory pressure, they must also continuously save execution state to disk and strive to minimize their start-up times to cope with frequent restarts. Despite significant engineering effort [17, 20, 26], it still takes several seconds to kill and restart popular apps.

Improving mobile memory management is difficult. Mobile apps run in high-level language runtimes (e.g., Swift, ART), which limit OS insight (e.g., working set estimation is impossible) and are notoriously difficult for OS-level memory managers to work with [16]. Further, mobile apps often allocate large amounts of memory quickly (e.g., when starting, or for cloud downloads); unless the OS keeps a large pool of free memory, this is easier to accommodate by killing entire applications. Finally, touch-based interfaces impose strict latency requirements, which swapping to disk cannot meet.

In this paper, we improve mobile memory management with a key observation: unlike other operating systems, all mobile OSes run their apps in a *common* language runtime. For example, all apps running on Android must run in the Android Runtime (ART). We can therefore co-design the language runtime to assist the mobile OS in optimizing memory management instead of hindering it. Due to its knowledge of memory usage, the language runtime becomes an ideal place for mechanisms that can better manage memory.

This paper demonstrates the value of leveraging the runtime. We present *Marvin*, a new memory manager for Android that efficiently supports memory overcommit while meeting the strict performance requirements of mobile apps. Marvin implements most memory management in the language runtime, which has more insight into an application's memory usage. Marvin relies on the operating system only for cross-application resource allocation.

By integrating with the language runtime, Marvin can offer three new features that enhance memory management:

- A new swap mechanism, which performs *ahead-of-time* swapping to disk to meet app latency requirements

- A new object-level working set estimator, which distinguishes between garbage collector (GC) accesses and app accesses and avoids false sharing by tracking accesses at object granularity
- A new bookmarking garbage collector [16], which preserves exact liveness information without accessing swapped-out objects

We implemented a prototype of Marvin by modifying the interpreter and compiler of the Android Runtime (ART). Experiments show that our Marvin prototype is able to run more than 2x as many concurrent apps as Android, and that Marvin can reclaim memory over 60x faster than Android with a Linux swap file can allocate memory under memory pressure.

2 Limitations of Modern Mobile OS Memory Resource Management

Although mobile OSes may be based on traditional OSes (e.g., Android and Linux), they diverge in two important ways: (1) for each app, they bound memory usage to a fraction of physical memory (e.g., 512MB on a 4GB device), rather than letting apps allocate as much memory as they need, and (2) they kill applications when physical memory runs out rather than overcommitting memory through paging or other mechanisms. To motivate our work, we ran experiments with popular apps that show the reasoning and cost for these design decisions. We ran all experiments on a Pixel XL phone with 4GB RAM and a quad-core Qualcomm Snapdragon 821 CPU.

2.1 Fixed Memory Allocation

Mobile OSes have poor insight into app memory usage. The runtime garbage collector regularly touches all objects and moves objects for heap compaction, and the OS cannot distinguish this activity from app accesses. Mobile apps also access language-level objects, which vary in size, while the OS can only track memory accesses at page granularity. To understand the impact of object-level accesses on page-sized access tracking, we measured the size of objects in popular apps. Figure 1 shows a CDF of the size distribution. Depending on the app, up to 40% of objects are less than 4KB in size, which means that the OS cannot accurately track their usage.

Without good insight into app memory usage, today's mobile OSes allocate all apps a fixed memory budget. On Android, this memory limit is the same whether an app is in the foreground or background. Android attempts to minimize the memory footprint of apps using techniques such as forking all apps from a single "zygote" process with copy-on-write pages. These techniques help avoid duplication of framework data structures and shared libraries, but they cannot do anything about objects allocated by the app itself in its Java heap. Using Marvin's object-level working set estimator, we measured the working set of popular apps. Figure 2 shows that although the heap footprint of these apps is large, their working sets actually account for a small fraction of their total heap size.

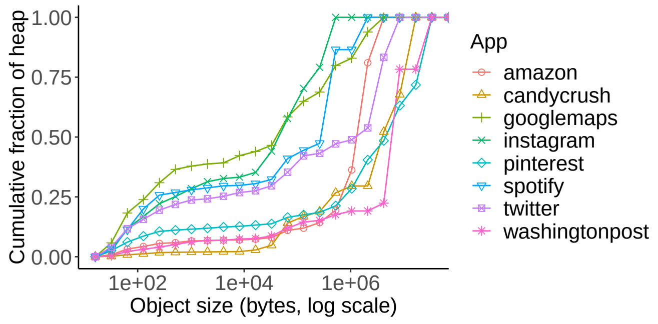


Figure 1. CDF of object size and heap percentage occupied by objects that size or smaller. Popular Android apps have a bimodal distribution where most objects are either significantly smaller or larger than a 4KB page.

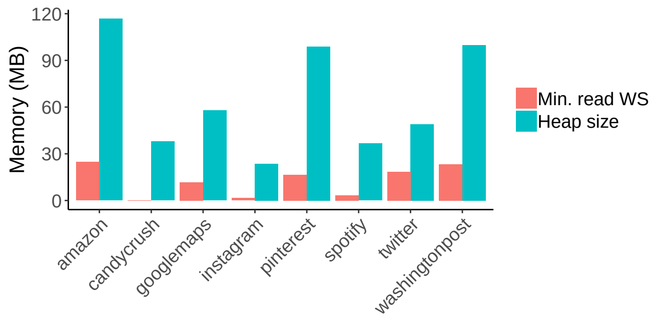


Figure 2. The cost of fixed allocation. Each bar shows the total Java heap size of a popular app alongside its minimum Java working set during active use. While apps have large memory footprints, they do not use most of that memory, which could be better utilized for running another app.

This rarely accessed memory would be better utilized keeping other apps alive, rather than wasting space not being used.

Popular apps often have large memory footprints but small working set sizes because they cache as much as possible from the cloud. This caching improves performance at no cost to the app, but it leads to poor memory utilization, and choosing the correct cache parameters is difficult [23]. Worse, modern applications frequently exceed their memory budgets. Coping with this problem requires applications to implement manual swap-to-storage, which adds significant programming complexity to them, as demonstrated by the amount of documentation and tutorials on this topic [9, 10]. While caching libraries like Glide [5] and Fresco [28] are helpful, they do not apply to all memory objects. Therefore, today's apps use a complex combination of libraries and manually shuffling data between memory and disk.

2.2 No Memory Overcommit

Today's mobile OSes kill applications rather than swapping to disk when physical memory runs out. They take this approach because mobile apps need to respond to user input

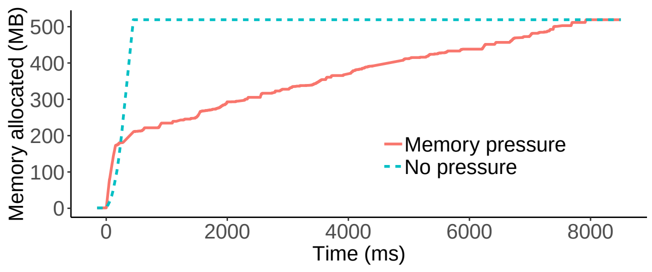


Figure 3. Android memory allocation speed with and without swap. Android allocates 512MB of memory in 450 ms when memory is free; however, when required to swap, Android takes almost 8 seconds to allocate the same amount of memory.

quickly, within hundreds of milliseconds or a second, and traditional swap mechanisms impose too much latency for apps allocating memory in response to user input. To measure the effect of swap on memory allocation, we enabled a Linux swap file on our Android test device [29] and measured the amount of time required to allocate 512MB when the Android OS had free memory and when it did not. Figure 3 shows the memory allocated by the OS over time. With memory available, the OS allocated all 512MB in 450ms; however, under memory pressure, it took almost 8 seconds for the OS to allocate the same amount. Such high allocation latency would be unacceptable if the app were allocating memory in response to user input.

Unfortunately, killing and restarting apps comes at a cost. As shown in Figure 2, modern apps have large memory footprints, and a restarted app must fetch all of its cached data from the network or disk. We measured the amount of time needed to restart popular apps and compared it to that needed to fetch their entire checkpointed memory image from disk. As shown in Figure 4, restarting apps takes 4-27x longer than fetching the app's memory from disk.

The ability to kill and restart apps at any time also imposes a programming burden on app developers. Modern OSes give apps a limited time budget to perform cleanup before being killed. This limit leads apps to constantly write state to storage; in fact, Android encourages it:

> You will probably want to commit your data even more aggressively at key times during your activity's lifecycle: for example before starting a new activity, before finishing your own activity, when the user switches between input fields, etc. [8]

Such constant checkpointing in response to app lifecycle events adds programming complexity, a challenge described in prior work [13]. Not only do app developers have to manage the checkpointing process, they have to correctly use a variety of mechanisms to do so with good performance. According to Android documentation:

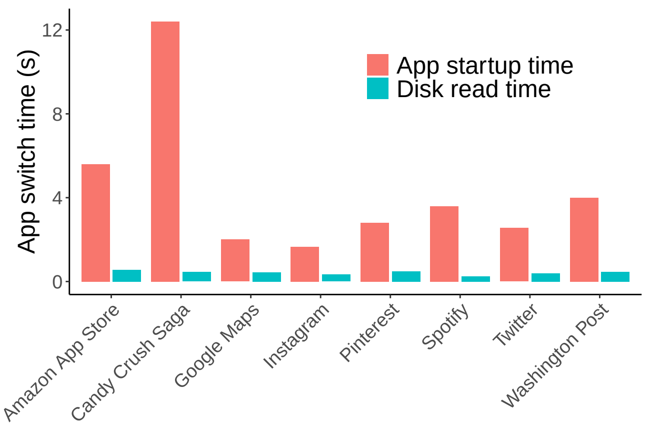


Figure 4. The cost of re-starting apps. Modern mobile OSes kill apps when memory runs out rather than swapping to disk. This wastes significant time for popular apps, which take anywhere from 4x to 27x longer to restart than to read *the entire app memory image from disk.*

> Deciding how to combine these options depends on the complexity of your UI data, use cases for your app, and consideration of speed of retrieval versus memory usage. [11]

The programming effort required to prepare for unexpected app deaths is an additional cost that app developers must pay.

3 Our Approach

The primary barrier to improving memory resource management in mobile operating systems is the OS's lack of insight into the language runtime. To overcome this barrier, we *co-designed* the language runtime and the mobile OS. Mobile operating systems are uniquely suited to such co-design because, unlike their desktop counterparts (e.g., Linux), they force all applications to use the same language runtime.

Marvin's design focuses on Android and the Android Runtime (ART) due to Android's popularity [18]. Using the language runtime, Marvin manages memory entirely at object granularity, tracking, swapping, reclaiming and faulting in entire objects. This section describes in more detail the barriers to better memory resource management in a mobile OS and how Marvin addresses those challenges.

3.1 Object-Level Working Set Estimation

The first barrier to better memory resource management is a better understanding of each application's working set. As a first step, Marvin implements language-aware working set estimation in the language runtime, which tracks app accesses at object granularity. This helps Marvin identify candidates for ahead-of-time swap (Section 3.2) and separate garbage collector accesses from app accesses (Section 3.3). Lacking hardware access bits to help with this tracking, Marvin must implement software access tracking in both the ART interpreter and compiler, as modern mobile language runtimes run both interpreted and compiled code.

3.2 Ahead-of-time Swap

As noted, swapping to disk when the OS needs memory is not feasible for mobile OSes and their touch-based apps. Marvin takes a different approach. While traditional swapping mechanisms write to disk when memory is needed, Marvin uses a new *ahead-of-time* swap technique. This technique saves memory to disk *before* it is needed and then reclaims those pages under memory pressure. Ahead-of-time swap separates swapping to disk from reclaiming memory; thus, we distinguish between *saved* objects, which have been copied to disk but still reside in memory, and *reclaimed* objects, which no longer reside in memory but are only on disk.

Swapping objects before they are needed leaves a large pool of clean memory that the OS can quickly reclaim and reallocate. While this technique lets apps continue using memory until the OS reclaims it, whenever the app dirties a page, the swap mechanism must update the on-disk copy before the OS can reclaim it. Due to this trade-off, Marvin prioritizes swapping objects that are infrequently or never written.

3.3 Bookmarking Garbage Collector

Like traditional swapping, ahead-of-time swapping is affected by friction with the language-level garbage collector. As noted by Hertz et al. [16], the garbage collector can inadvertently page in memory when walking the object heap to look for unused objects. With ahead-of-time swapping, the garbage collector can also inadvertently dirty pages when updating references, causing unnecessary writes to disk. Marvin solves this problem by integrating a modified bookmarking garbage collector [16] into the Android Runtime.

Marvin's swap mechanism leaves *stubs* – analogous to bookmarks – for each reclaimed object that detail the objects' references to other objects. Using these stubs, its garbage collector can process a reclaimed object during a mark-and-sweep run without faulting in the entire swapped object. Marvin's swap mechanism can further optimize swapping from disk by dropping dead objects without faulting them in.

4 Marvin Overview

Marvin is a new mobile memory manager that supports flexible memory allocation between apps and memory overcommit through swapping. Marvin includes components in the language runtime and OS, which are co-designed to provide better memory management. This section gives a design overview of both components.

4.1 Design Goals

Our goals with respect to Marvin's design follow:

1. **Fast memory allocation.** Marvin must allocate memory quickly on-demand, avoiding disk accesses on the critical path for memory allocation.
2. **High memory utilization.** Marvin must provide the illusion of unlimited memory, provided working sets do not exceed the size of physical memory.
3. **Minimal overhead.** Marvin must impose low runtime overhead and require no app code changes.

While the last two goals are common to all memory management systems, existing mobile platforms sacrifice high memory utilization for fast memory allocation. Marvin aims to achieve all of these goals.

4.2 Marvin System Model

Marvin assumes a systems environment that meets three requirements: (1) all apps are written in a single language (e.g., Java), (2) all apps run in a single managed language runtime (e.g., ART), and (3) the runtime performs garbage collection or some form of automatic memory management. Marvin's design targets Android, which meets all of these requirements. Note, however, that much of its design could apply to other operating systems (e.g., iOS). For example, Swift uses automatic reference counting as an alternative to garbage collection, so it would require a bookmarking reference counter that can track references without faulting in the entire object.

Marvin runs unmodified Android apps on ARM64-based devices. Android distributes apps in a bytecode format called DEX. ART runs DEX bytecode directly in an interpreter and also compiles DEX to native ARM64 instructions both at install time (ahead-of-time, or AOT, compilation) and at runtime (just-in-time, or JIT, compilation). Marvin modifies both the interpreter and compiler.

4.3 Marvin Architecture

Marvin has two key components: (1) the *Marvin Kernel* (MK), a modified Android/Linux kernel, and (2) the *Marvin Runtime* (MRT), a modified ART. Most memory management occurs in MRT; it performs working set estimation, ahead-of-time swapping, and bookmarking garbage collection. MK's sole responsibility is to balance memory allocation among apps by deciding when and from which app to reclaim memory.

Marvin performs working set estimation and swapping at object granularity; however, there is no CPU support for object-level access bits and memory faults. As a result, Marvin implements software object access tracking and faulting in the MRT interpreter and compiler. The interpreter marks access bits and checks for swapped-out objects as it runs DEX bytecode; the compiler inserts that functionality as additional ARM64 instructions. Marvin reserves four bytes in each object header and uses those bytes to store swapping metadata and access bits. These software features impose an app overhead, which we quantify in Section 8.

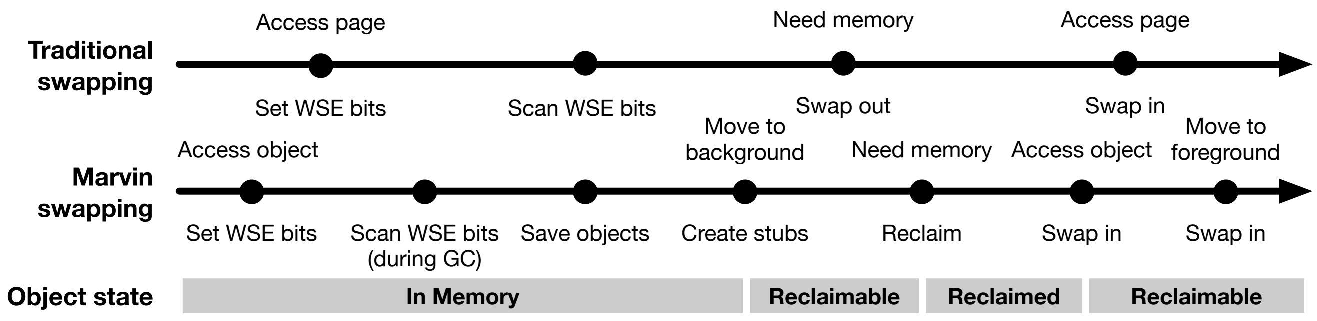


Figure 5. A timeline of actions performed by Marvin's swap mechanism as compared to traditional (e.g., Linux) swap mechanisms. Events are listed above the timeline while Marvin's actions in response are listed below.

4.4 Marvin Memory Management Timeline

Objects managed by Marvin move through several states over time, driven by app behavior and app lifecycle events. Figure 5 illustrates these events and states and compares Marvin's swapping to a traditional swap mechanism. When an app first starts, MRT begins estimating its working set. It identifies objects that are suited for swapping by examining whether they are cold (have not been accessed recently by the app). MRT begins saving checkpoints of those objects to disk in the background. We refer to an object with a saved checkpoint as a *saved* object.

When the app moves from foreground to background, MRT pauses app threads and creates stubs, small proxy objects that add a layer of indirection over swap candidate objects. Stubs ensure that Marvin can catch accesses to objects and fault them back in, if necessary. Once MRT creates a stub for an object, that object becomes *reclaimable*; the object is still memory-resident, but MK can reclaim its memory at any time. When MK reclaims an object, it enters the *reclaimed* state; only the object's stub remains in memory, and the object's checkpoint will need to be faulted back into memory before the object can be accessed again. The garbage collector uses only the stub and need not fault the object back into memory.

5 Marvin Core Mechanisms

As noted in Section 3, Marvin's key features are ahead-of-time swap, language-aware working set estimation, and bookmarking garbage collection. Designing these features required addressing three challenges: adding a layer of indirection for object references, coordinating between the OS and runtime, and interposing on object accesses. This section describes Marvin's mechanisms for addressing these challenges.

5.1 Stubs for Object Reference Indirection

According to the Java language specification, object references are opaque. However, in practice, object references in the Android Runtime are direct pointers to the heap memory holding the referenced object. Several of Marvin's features—such as its software object faulting mechanism and its bookmarking garbage collector—require a layer of indirection between references and their referent objects. Marvin creates this layer of indirection using special objects, called *stubs*. Each stub contains a pointer to its underlying object along with a copy of each reference held by the object. All references to an object point instead to its stub, and only the stub holds a pointer to its underlying object. Accessing an object through a stub adds overhead, so Marvin creates stubs only for objects that are cold and at least 2KB in size.

When creating a stub for an object, Marvin must redirect all references to the object to point to its stub instead. This task requires that all app threads be paused. Therefore, it can be performed more efficiently if Marvin can create stubs for many objects at once, using a single scan of the heap to redirect all affected references. As a result, Marvin periodically executes a heap task that pauses all app threads and creates stubs, and it executes this heap task only when the app is in the background and its threads can be safely paused. A stub needs to be created only once for a given object. Once the stub is in place, Marvin can manipulate the underlying object without pausing app threads.

5.2 Reclamation Table for OS-Runtime Coordination

Modern mobile platforms have multicore processors that let system services run concurrently with apps. In this environment, the OS should be able to reclaim memory quickly from a running app without scheduling the app's threads for execution. However, the OS cannot simply seize memory from an app whose threads are not scheduled—a pointer to the memory in question may be present in an app thread's stack or registers, waiting to be used as soon as the thread is scheduled once again. As a result, the OS and runtime need a way to coordinate concurrent accesses to objects so the OS does not try to reclaim one that the runtime is accessing.

Marvin uses a shared-memory *reclamation table* to provide this coordination. MRT populates the table with reclaimable objects, and MK uses it to identify memory to reclaim. Each reclamation table entry is a small, fixed-size data structure that holds the address of an object, a set of flags indicating whether the object is memory-resident and the entry is valid, and a set of bits used for locking by the runtime and OS. To reclaim an object, MK first acquires an exclusive lock on the object's reclamation table entry. Similarly, whenever an app thread prepares to access an object, MRT acquires a shared lock on the reclamation table entry.

5.3 Object Access Interposition

All of Marvin's features require the runtime to interpose and perform specific tasks whenever an app accesses an object. On every object access, MRT must set read and write bits for working set estimation; check for the presence of a stub and redirect the object access through the stub if necessary; and fault in the object if it has been reclaimed. It must also set a dirty bit whenever an object is modified so the ahead-of-time swap mechanism knows which objects need to be saved, and it must update stubs whenever reference member variables in their corresponding objects change to support the bookmarking garbage collector.

Android apps execute both as DEX bytecode running in an interpreter and as compiled native code running directly on the hardware, and Marvin must interpose on all object accesses in both kinds of code. As a result, Marvin features a set of paired interpreter and compiler modifications that add the required object access interposition. For each additional task performed by the interpreter when it accesses an object, there is a corresponding change to the compiler, adding assembly instructions performing the same task to compiled code.

6 Marvin Memory Management

This section describes how we use Marvin's mechanisms (stubs, the reclamation table, and object access interposition) to design the features that make up Marvin's memory management system.

6.1 Working Set Estimation

MRT performs object-granularity working set estimation by maintaining two access bits in each object header, a read bit and a write bit, and scanning those access bits.

Setting access bits. MRT uses object access interposition to set an object's read and write bits whenever that object is read or written from either interpreted or compiled code. It avoids including garbage collector reads in its working set estimation by setting a flag in the object header when the garbage collector is visiting an object and leaving the read bit untouched if that flag is set.

Access tracking in MRT is performed on a best-effort basis to minimize its overhead: MRT uses non-atomic operations with relaxed memory ordering semantics when setting read and write bits. As a result, concurrent reads and writes to the same object could result in a lost update to one of the access bits. An update to the read bit could also be lost if an app thread reads an object that the garbage collector is processing. These optimizations may decrease swapping performance if the estimated and actual working sets differ significantly, but they do not affect correctness.

Scanning access bits. MRT periodically walks the heap and uses the Clock algorithm [7] to track each object's long-term usage. Each object header holds two four-bit *shift registers*, one for reads and the other for writes. The time between heap walks constitutes an access-tracking "round," and each shift register tracks whether the object was read or written in the last four rounds. During a heap walk, MRT updates an object's shift registers and then clears the object's access bits.

Producing the working set. As MRT walks the heap and scans access bits, it tabulates the app's working set. Our current MRT implementation considers an object part of the working set if it has been read or written within the last four access-tracking rounds. The precise policy is an implementation detail that can be easily changed.

6.2 Ahead-of-Time Swapping

In Marvin's ahead-of-time swap mechanism, MK reclaims objects and decides which apps to target for reclamation. MRT performs all other functions, including saving object checkpoints to disk, restoring reclaimed objects, and preventing the operating system from reclaiming objects being used by app code.

Saving objects to disk. MRT identifies suitable objects for swapping (i.e., cold objects) using its working set estimation feature, and it proactively saves checkpoints of those objects to a swap file on disk so they can be reclaimed quickly under memory pressure. MRT saves objects to disk in a periodic heap task that runs on a background thread concurrently with app code.

After app code modifies an object, MRT must save an updated copy of that object to disk. It does not need to save the updated copy immediately as long as it prevents the kernel from reclaiming the object while it is "dirty." To do so, MRT maintains a *dirty bit* in the object header. It uses object access interposition to set this dirty bit whenever app code writes to an object, and its object-saving heap task clears this dirty bit when saving the object to disk. MK checks dirty bits when looking for objects to reclaim and avoids reclaiming dirty objects. MRT and MK use strong memory-ordering semantics when reading and writing the dirty bit to ensure that no modifications to objects are lost.

MRT begins saving swap candidate objects to disk even before those objects have had stubs created for them. Once MRT creates a batch of stubs, those objects become immediately reclaimable without requiring further disk I/O.

Reclaiming objects. MK selects apps to target for reclamation and reclaims objects from the MRT instances corresponding to those apps. After selecting an MRT instance to target, MK scans the MRT instance's reclamation table until it finds an entry for an object that is neither dirty nor locked by the runtime. MK then locks that entry and reclaims the object's pages. It continues scanning the reclamation table and reclaiming objects until it has harvested the desired amount of memory from the MRT instance.

MRT ensures that MK does not reclaim an object currently being accessed by its app code. To do so, it uses object access interposition to detect whenever a reclaimable object is being read or written, and it locks the object's reclamation table entry before the access and unlocks its entry after the access.

Restoring objects. MRT restores reclaimed objects either eagerly or on-demand. Either way, whenever MRT restores an object, it locks the object's reclamation table entry, copies the saved checkpoint data of the object into memory, and copies any modified references from the object's stub into the object itself. This last step is necessary because references in the stub may have been modified by the garbage collector while the object was not memory-resident.

Marvin's eager object restoration uses app lifecycle information to restore objects before app code needs them. We implemented a simple eager restoration policy, where an MRT instance restores all reclaimed objects when it transitions to the foreground. This policy ensures that no user-perceptible delays or stuttering result from swapping activity. Our design is flexible and could support more advanced policies; for instance, the runtime could predict which objects are likely to be touched immediately after a foreground transition and restore those objects first, trading off a shorter pause time in exchange for the risk of user-perceptible stuttering.

If an object has not been eagerly restored, MRT restores it on-demand when app code accesses it, a process that we call *software object faulting*. Whenever app code accesses a reclaimable object, MRT uses object access interposition to check if the object is memory-resident by inspecting a bit in its reclamation table entry. If not, MRT executes an object fault handler that performs a procedure call into its C++ object restoration function.

6.3 Bookmarking Garbage Collector

A tracing garbage collector touches every object in the heap (or a subset of the heap), causing live objects to be swapped back into memory even if app code is not using them. Marvin's garbage collector avoids touching reclaimed objects by storing an object's references inside its stub and using the stub during the mark phase of garbage collection. Stubs play a similar role as bookmarks in the bookmarking garbage collector [16].

During the mark phase, the garbage collector maintains a *mark stack* and repeatedly pops an object from the mark stack, marks all its references, and pushes those references onto the mark stack. Marvin's garbage collector checks whether an object is a stub when it pops the object from the mark stack; if so, it reads the references off the stub instead of accessing the underlying object.

For the garbage collector to use stubs in place of their objects, MRT must ensure that the stub of a memory-resident object has up-to-date copies of the object's references. It uses object access interposition to update the stub of a reclaimable object whenever Java code modifies one of the object's reference member variables.

MRT must also properly clean up after any saved objects that are freed by the garbage collector. MRT records when saved objects have been freed by the garbage collector, and when the fraction of the swap file consisting of freed objects passes a set threshold (25% in our implementation), it compacts the swap file in a heap task. MRT also cleans up after reclaimable and reclaimed objects by checking whether an object being freed is a stub; if so, it deletes the reclamation table entry corresponding to the stub. If an object is reclaimable, MRT deletes the memory-resident copy of the underlying object; if the object is reclaimed, MRT simply marks its saved copy in the swap file for deletion without needing to fault in the object.

6.4 Design Tradeoffs and Alternatives

By tracking working sets and faulting in objects in software at the runtime level, Marvin achieves a clean design, albeit with some drawbacks. First, Marvin cannot reclaim objects accessed by native libraries: native libraries have no way to detect stubs and no recourse for faulting in reclaimed objects. Second, software working set estimation and object faulting add overhead, particularly to compiled code. We evaluate this overhead in Section 8.

Marvin moves almost all memory management into the runtime because we believe that the runtime's better access to information about app behavior makes it better suited for managing memory. The functionality remaining in the kernel is the minimum required by existing Linux kernel design; if Marvin was built on top of an exokernel, it could move even more functionality into the runtime. A variety of other designs are possible that split functionality between the runtime and kernel in different ways.

Kernel-level working set estimation would reduce the overhead of accessing objects, but it would suffer from false sharing if an app's working set is mixed with unused objects across 4KB pages. Faulting in memory at the kernel level would similarly reduce object access overhead but would require more extensive re-design of the runtime garbage collector to avoid unnecessary swapping activity. By tying the granularity of memory management to the size of pages, kernel-level memory management will also become inflexible as large pages become more common and the disparity between object sizes and page sizes widens. In any case,

`array-length`	`iget-*`
`instance-of`	`iput-*`
`check-cast`	`aget-*`
`invoke-interface`	`aput-*`
`invoke-virtual`	

Table 1. DEX bytecode instructions for which Marvin performs object access interposition.

kernel-level memory management would require some sort of ahead-of-time swap mechanism to satisfy the latency requirements of modern mobile platforms (Figure 3), and even adding ahead-of-time swap to the existing Linux kernel would require significant implementation effort.

Marvin's garbage collector is different from the original bookmarking collector [16] in that it maintains exact reachability information with stubs rather than conservatively storing approximate reachability information. The latter approach requires the garbage collector to perform less work when evicting pages and scanning the heap, but it can result in the heap being needlessly occupied with dead objects.

7 Marvin Prototype

We implemented a prototype of MRT by modifying ART on Android 7.1.1 (which includes Linux 3.18.31). Our implementation includes a modified version of ART's ARM64 compiler, allowing our prototype to support Android devices with 64-bit ARM processors. Our changes to the ART codebase resulted in approximately 3500 additional lines of code, as measured by SLOCCount.

In addition to modifying ART, we made a small modification to the version of OpenJDK included with Android 7.1.1, namely, we added fields to the Object class definition to mirror the bytes added to the object header in ART. We also changed a source file in the Android framework (`ProcessList.java`) to increase a hard-coded limit on the number of concurrently running apps.

In our experiments, we manually triggered reclamation, so we did not prototype the kernel modifications required for MK. However, our MRT implementation includes the reclamation table and performs all operations required to support kernel memory reclamation.

7.1 Object Access Interposition

We implemented MRT's object access interposition by adding specialized functionality to the ART interpreter and compiler. This lets MRT interpose on object accesses from both DEX bytecode running in the interpreter and compiled OAT code running natively. The following section describes in detail how we modified each component.

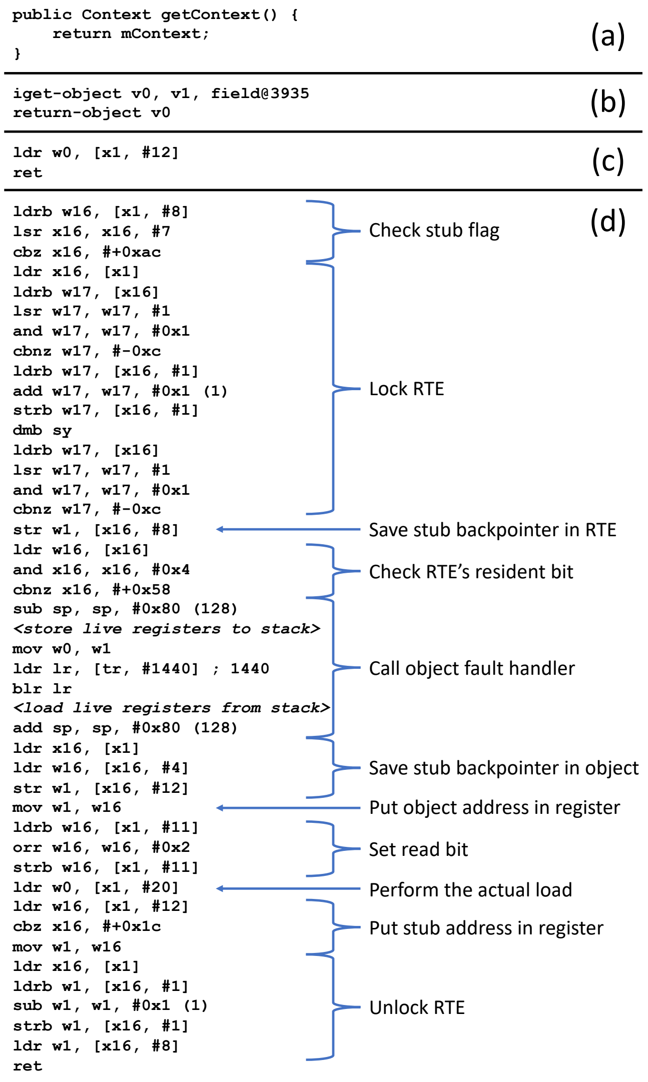
MRT interpreter. The ART interpreter internally represents each Java object as a C++ *mirror object*, which it manipulates when executing DEX bytecode instructions that read or write that object. The mirror object's type definition includes methods to read or write the data at a given offset within the object's memory footprint, and the interpreter code calls these methods when executing DEX instructions. To add object access interposition to the interpreter, we modified the interpreter's mirror object methods to implement Marvin's features.

For example, to redirect object accesses through stubs and perform on-demand object faulting, we added a preamble macro to each mirror object method. The preamble first checks if the object is actually a stub. If so, it casts the `this` pointer to a stub, calls a method that locks the stub's reclamation table entry (RTE), checks the RTE's resident bit, and if the resident bit is cleared, calls a method to fault in the object from disk. The preamble then gets the address of the underlying object from the RTE and invokes the mirror object method on the underlying object. Finally, the preamble unlocks the RTE and returns the result of the mirror object method, if any.

ART contains multiple interpreter implementations, and the default is the "mterp interpreter," an interpreter written in assembly. When the mterp interpreter executes DEX instructions that read or write an array, it directly accesses the array's memory, bypassing the mirror object methods. To allow Marvin to interpose on array accesses, we instead use the "switch interpreter," an interpreter written in C++ that calls the mirror object methods when executing array accesses.

MRT compiler. Java code in Android framework libraries and portions of app Java code execute as native code, which is compiled by the ART compiler either statically after the app is installed or dynamically with just-in-time (JIT) compilation. We added object access interposition to this compiled code by modifying the MRT compiler's assembler to generate additional assembly instructions that implement Marvin's features when it performs code generation for object accesses. Since we used ARM64 devices for our testing and evaluation, we added support for object access interposition to the ARM64 assembler.

Figure 6 illustrates the added ARM64 instructions in Marvin's version of compiled code compared to the same compiled code in Android. Each operation described above for the interpreter's implementation of stub redirection and object faulting has a corresponding block of ARM64 instructions in the compiled code. The main difference is that when a stub is detected, the compiled code must explicitly overwrite the register holding the stub's address with the address of the underlying "real object;" it then loads the stub's address back into that register when it is done with the object. In the common case, when an object is not a stub (or when it is, but



```
public Context getContext() {
    return mContext;
}
```

(a)

```
iget-object v0, v1, field@3935
return-object v0
```

(b)

```
ldr w0, [x1, #12]
ret
```

(c)

```
ldrb w16, [x1, #8]
lsr x16, x16, #7
cbz x16, #+0xac
ldr x16, [x1]
ldrb w17, [x16]
lsr w17, w17, #1
and w17, w17, #0x1
cbnz w17, #-0xc
ldrb w17, [x16, #1]
add w17, w17, #0x1 (1)
strb w17, [x16, #1]
dmb sy
ldrb w17, [x16]
lsr w17, w17, #1
and w17, w17, #0x1
cbnz w17, #-0xc
str w1, [x16, #8]
ldr w16, [x16]
and x16, x16, #0x4
cbnz x16, #+0x58
sub sp, sp, #0x80 (128)
<store live registers to stack>
mov w0, w1
ldr lr, [tr, #1440] ; 1440
blr lr
<load live registers from stack>
add sp, sp, #0x80 (128)
ldr x16, [x1]
ldr w16, [x16, #4]
str w1, [x16, #12]
mov w1, w16
ldrb w16, [x1, #11]
orr w16, w16, #0x2
strb w16, [x1, #11]
ldr w0, [x1, #20]
ldr w16, [x1, #12]
cbz x16, #+0x1c
mov w1, w16
ldr x16, [x1]
ldrb w1, [x16, #1]
sub w1, w1, #0x1 (1)
strb w1, [x16, #1]
ldr w1, [x16, #8]
ret
```

(d)

Check stub flag
Lock RTE
Save stub backpointer in RTE
Check RTE's resident bit
Call object fault handler
Save stub backpointer in object
Put object address in register
Set read bit
Perform the actual load
Put stub address in register
Unlock RTE

Figure 6. (a) Java code, (b) DEX bytecode, (c) compiled ARM64 code on Android, and (d) compiled ARM64 code on Marvin corresponding to the Android framework's `LayoutInflater.getContext()` method.

its underlying "real object" is memory-resident), execution branches past many of the added object-faulting instructions.

7.2 Potential Optimizations

Our implementation of object access interposition in the MRT compiler is unoptimized, and the per-object-access overhead of compiled code could be reduced with deeper compiler integration. We implemented object access interposition in the compiler by modifying its ARM64 assembler, which translates intermediate representation (IR) instructions to ARM64 binary code. As a result, our prototype compiler generates ARM64 instructions to check for the presence of a stub, perform object faulting, and lock/unlock the object's RTE on every IR instruction that reads or writes an object, even if further IR instructions read or write the same object while it is allocated to the same ARM64 register. A more optimized version of the MRT compiler could generate code to check for the presence of a stub, perform object faulting, and lock the object's RTE only once when the object is initially allocated to a register, and it could generate code to unlock the object's RTE only once after the object's register allocation expires. This optimization would require compiler modifications at the IR level, and it would decrease both the execution overhead of compiled code and the code size overhead.

The MRT compiler has other areas for optimization. For instance, we noticed situations where ARM64 parameter registers (`x0`–`x7` and `d0`–`d7`) appear to be live but are not reported as such by the ART compiler's `LocationSummary` class; therefore, we conservatively save all parameter registers to the stack when performing a procedure call. Saving only live parameter registers would further reduce code size overhead. In addition, the ARM64 assembler makes a maximum of two scratch registers available at any time, which required us to save some backpointers and add more instructions to juggle required state among the limited available registers.

8 Evaluation

Our evaluation demonstrated that Marvin successfully met its design goals. It could:

1. Quickly reclaim memory on-demand
2. Maintain high memory utilization by sharing memory efficiently among apps rather than killing them
3. Achieve the previous two goals with low overhead and no app changes

We now describe our evaluation experiments in detail.

8.1 Evaluation Setup

We ran our experiments on a Google Pixel XL smartphone with 4GB RAM and a quad-core Qualcomm Snapdragon 821 CPU. The smartphone ran either the open-source release of Android 7.1.1 (AOSP tag android-7.1.1_r57) or our Marvin implementation based on that release. Both our Marvin implementation and our baseline Android build included a change to the Android framework to increase a hard-coded cap on the number of concurrently running apps.

For some experiments, we built synthetic apps that simulate various memory footprints and working set sizes. These apps allocate the entire memory footprint at startup and then repeatedly walk the working set. In this way, we could measure the effect of various working set sizes on Marvin compared to Android. We also used PCMark for Android, a commercial benchmark app based on real-world apps, to measure Marvin's overhead on real apps [3]. PCMark's Work 2.0 benchmark suite contains five benchmarks; we selected two to use for evaluation (Writing 2.0 and Data Manipulation) since the remaining three test the performance of native libraries.

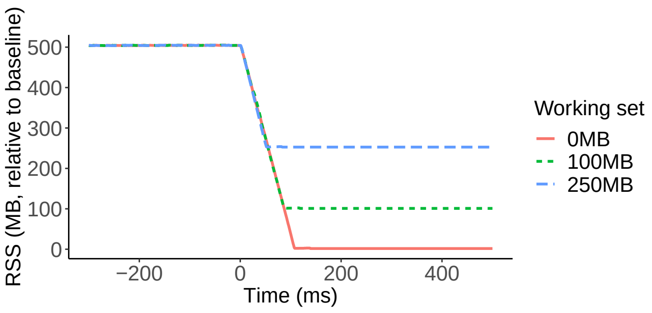


Figure 7. Memory usage as Marvin reclaims memory from a benchmark app with a 500MB heap and different working set sizes. Marvin took 108ms to reclaim 500MB, much faster than the nearly 8 seconds required by Android with Linux swap to allocate the same amount of memory.

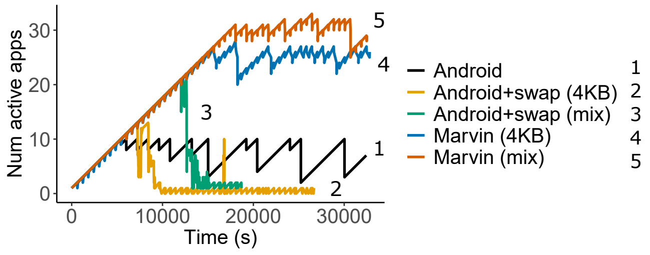


Figure 8. Count of active apps over time when starting instances of our benchmark app. Marvin runs more than twice as many apps as regular Android before needing to kill any apps; on Android with a swap file, most apps are alive but inactive due to constant swapping activity.

8.2 Memory Reclamation

Marvin must be able to quickly reclaim memory from running apps when a new or existing app needs to allocate large amounts of memory. Its ahead-of-time swap mechanism ensures that each MRT instance has a pool of clean memory that can be quickly reclaimed without swapping to disk. In this section, we measure the latency of reclaiming memory for apps with different working set sizes. Because Marvin limits itself to reclaiming only non-working-set memory, the bigger the app's working set, the less memory that Marvin can reclaim.

For our prototype, we use `madvise` to return memory from MRT to the kernel. This design minimizes our changes to Android's Linux kernel and lets us trigger reclamation rather than waiting for memory pressure.

Figure 7 shows memory usage over time for apps with a 500MB heap and differing working set sizes. RSS values shown are relative to the RSS reported for a minimal Android app with a single empty Activity (approx. 80MB). At time 0ms, MRT begins to return memory from the app to the OS. Marvin returned 250MB of memory in 52ms and 500MB of memory in 108ms. In comparison, as shown in Figure 3, Android with a Linux swap file took nearly 8 seconds to free and allocate 500MB of memory under memory pressure. Using ahead-of-time swap let Marvin reclaim memory over 60x faster than Android with Linux swap could allocate the same amount of memory, allowing Marvin to meet the strict latency requirements of mobile apps.

8.3 Memory Utilization

To demonstrate Marvin's efficient memory utilization compared to Android, we ran multiple instances of an app with a large memory footprint and a limited working set, and we counted the number of *active* apps that were alive and making progress on their workloads. Each app had a 220MB heap filled with arrays, and it deleted and reallocated 20MB of those arrays every 5 seconds. We used two different heap compositions: one where the apps had heaps filled with 4KB arrays, and one where they had an even mix of 4KB and 1MB arrays (similar to the bimodal distribution of real apps in Figure 1). We consider an app "inactive" if it fails to perform a round of its workload for 20 seconds after the previous round; we consider it "active" once again if it succeeds in performing a round of its workload within 7 seconds of the previous round. We started a new app instance every 10 minutes to give Marvin time to perform background work. For unmodified Android, only the data for the apps with 4KB arrays is shown, because its behavior was nearly identical for the apps with a 4KB/1MB mix.

As shown in Figure 8, Marvin ran over 2x as many apps concurrently as unmodified Android and over 1.5x-2x as Android with a Linux swap file enabled; however, swapping left almost all apps unusable. While baseline Android begins killing apps when physical memory runs out, Android with swap keeps more apps alive. However, without a bookmarking garbage collector, the system experienced constant swapping activity, which prevented most apps from making progress on their workloads. Our experimental runs of Android with a swap file consistently ended early due to the device crashing and rebooting.

Marvin made better use of device memory because it reclaimed unused memory from apps and used its bookmarking garbage collector to avoid touching that unused memory when running garbage collection. While Android only ran 10 apps concurrently, and Android with a swap file only briefly reached a maximum of 13 concurrent apps (4KB arrays) or 20 apps (4KB/1MB mix), Marvin ran 27 apps (4KB arrays) or 30 apps (4KB/1MB mix) concurrently. Marvin's memory reclamation and bookmarking garbage collector let it execute 1.5-2x as many apps concurrently while neither killing apps nor suffering performance degradation.

8.4 Runtime Overhead

Marvin has four main types of overhead:

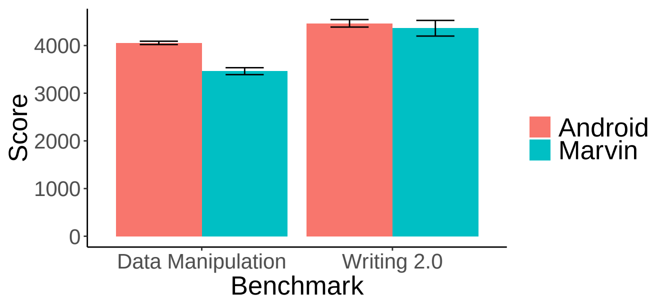


Figure 9. PCMark for Android benchmark results. Marvin's score is within 15% of Android's for both benchmarks, showing that Marvin adds low overhead for accessing non-reclaimable objects for real-world apps.

1. *Execution time overhead* caused by Marvin's object access interposition in compiled OAT code
2. *Increased size of compiled code* due to object access interposition
3. *CPU utilization overhead* caused by Marvin's heap walks for working set estimation
4. *Overhead of faulting* when an app accesses a reclaimed object

We explore each type of overhead in turn.

Execution time overhead of object access interposition. Native code produced by the MRT compiler has additional ARM64 instructions to support object access interposition. Some instructions (stub checks, dirty bit updates, and access-tracking bit updates) execute on every object access. Other instructions (indirecting object accesses through stubs and locking and unlocking RTEs) execute only on accesses to reclaimable objects. We measured the overhead of the added instructions that apply to all object accesses using PCMark for Android, and we used a synthetic benchmark to illustrate the dependence of that overhead on the makeup of application code.

Figure 9 compares the performance of Marvin and unmodified Android on the PCMark benchmarks in our test set. Each bar shows the mean and standard deviation of five runs. We turned off stub creation and swapping when running PCMark, using the benchmarks to measure the overhead of the instructions added to every object access for stub checks and working set estimation. Marvin's score on the Writing 2.0 benchmark was nearly identical to Android's, and its score on the Data Manipulation benchmark was only 15% lower. These scores show that Marvin's overhead for accessing regular (i.e., non-reclaimable) objects is low for real-world apps.

Figure 10 explores the dependence of Marvin's overhead on the DEX instruction mix of application code. The graph shows Marvin's overhead executing a synthetic workload that performed a tunable proportion of object accesses (array reads and writes) and integer operations (addition and multiplication). Each point represents the mean and standard deviation

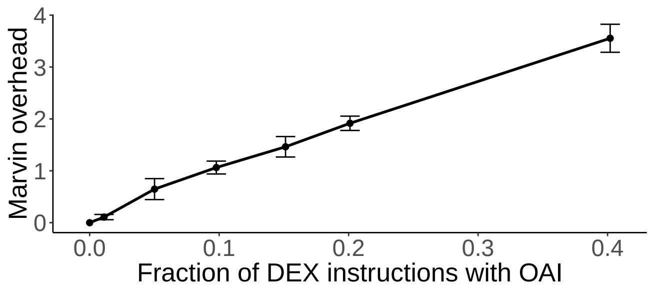


Figure 10. Overhead of Marvin for a synthetic workload with different proportions of DEX instructions with object access interposition (OAI). The point (0,0) represents the theoretical scenario of running no DEX instructions with OAI, while the other points show experimental results for the given workload mix. Marvin's overhead is lower when a smaller fraction of instructions have OAI.

of Marvin's execution time overhead relative to Android for 40 iterations of the workload. For large proportions of object accesses, Marvin had relatively high overhead (e.g., 350% overhead for 40% object accesses), while for low proportions of object accesses, Marvin's overhead was minimal (e.g., 10% overhead for 1% object accesses). PCMark's 15% overhead indicates that the real app workloads represented by PCMark have low proportions of object accesses.

Although these overheads are already reasonable, they could be further improved using additional optimizations. As noted in Section 7, deeper compiler integration would allow Marvin to reduce overhead by performing object access interposition less frequently.

Code size overhead of object access interposition. The ARM64 instructions added by Marvin's object access interposition also increase the size of compiled native code. To measure the increase in code size, we compared the compiled Android framework libraries generated by Marvin to the framework libraries on unmodified Android. Marvin increased the total size of the ARM64 framework libraries (in the `/system/framework/arm64` and `/system/framework/oat/arm64` directories on the Android filesystem) from 117 MB to 292 MB. This code size overhead, while relatively high, could be reduced significantly with deeper compiler integration (Section 7).

CPU utilization overhead of heap walks. Our Marvin prototype performs the heap walks required for working set estimation by invoking the concurrent garbage collector and piggybacking off its heap walk. Our prototype performs a heap walk every 5 seconds when an app is in the foreground and every 30 seconds for an app in the background. In theory, this periodic invocation of the garbage collector across multiple MRT instances could add CPU utilization overhead. In practice, when running multiple apps in the background, we found that the difference between Marvin's and unmodified

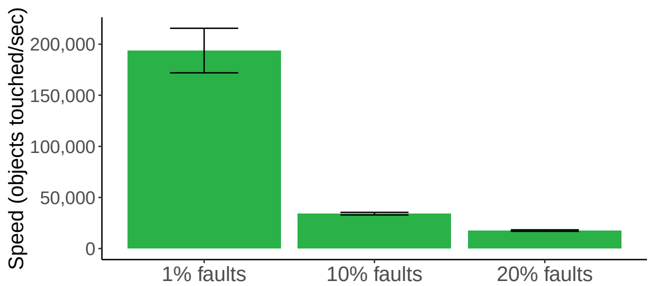


Figure 11. Speed of a benchmark app as it touches objects in its heap with different fractions of reclaimed objects. Object faulting never occurs in the user-visible foreground app due to Marvin's policies, but when Marvin does need to fault objects in from disk on-demand, its speed matches the expected result of trading off memory accesses for disk reads.

Android's CPU utilization was negligible, likely because the GC invocations were so infrequent for apps running in the background.

Overhead of faulting in objects. When an app first accesses a reclaimed object, Marvin must fault the object in from disk, adding significant latency to that initial access. Marvin's default policy eagerly restores all objects when an app moves to the foreground, so object-faulting latencies will never result in user-visible stuttering. Object faults may occur for background apps, but the Java working sets of apps in the background are generally quite small (less than 4MB for all commercial apps in our test set), so we expect object faulting to happen infrequently in practice.

We nonetheless studied the effect of object faulting on performance, in order to understand how Marvin might perform in situations where different policy choices or app workloads result in more object faulting activity. Figure 11 shows the effect of object faulting on a heap-walking benchmark app as it touches different fractions of reclaimed objects. The benchmark app looped over a set of 4KB arrays in its heap, reading five member variables of each array, and measured the speed at which it traversed the objects. Each bar shows the mean and standard deviation of five measurements, and the disk cache on the device was cleared before taking each measurement. As expected, there was an inverse relationship between heap-walking speed and fraction of faults; for instance, speed dropped by 49% as the fraction of faults increased from 10% to 20%. This speed/fraction-of-faults relationship is the natural result of exchanging low-latency memory accesses for high-latency disk accesses.

9 Related Work

A significant body of work examines the issue of providing persistent memory for object-oriented languages [1, 6, 19, 24, 27, 35]. These systems checkpoint objects to disk or non-volatile memory, but they do so to ensure safety in the face of failures rather than swapping out unused memory. As a result, they focus on supporting transactional programming models that provide strong guarantees under failure [6, 24] and on implementing crash-safe garbage collection [6, 35] rather than on maximizing the number of apps that can run concurrently.

SSDAlloc [2] is a persistent memory system that, like Marvin, is motivated by the goal of helping apps with large memory footprints avoid memory pressure. Unlike Marvin's runtime-level object faulting and working set estimation, SSDAlloc allocates objects in separate virtual pages and uses the existing virtual memory system to estimate the working set and trigger its object fault handler.

Like Marvin, the bookmarking collector [16] aims to improve the performance of Java applications running in memory-constrained environments. It assumes that the OS uses a traditional page-level swapping mechanism and focuses on ensuring that the garbage collector can run without unnecessarily swapping in pages. The bookmarking collector conservatively stores approximate reachability information (bookmarks) that is used during garbage collection, whereas Marvin stores exact reachability information (stubs).

Other recent work on garbage collection focuses on co-designing the garbage collector and runtime to manage software caches more efficiently [23], co-designing the garbage collector and virtual memory manager to improve performance [36], measuring the effect of garbage collection on scalability [14], and designing garbage collectors or memory managers for specific application domains such as big data systems [15, 22].

With multiple runtimes performing their own memory management running on top of a single operating system, Android's architecture resembles the architecture of a virtual machine manager, where multiple guest operating systems run on top of a hypervisor. Marvin's runtime–OS cooperation is analogous to that between guest OS and hypervisor in the VMware ESX server [33], which uses a balloon driver to induce guest OSes to reclaim memory.

Wright et al. [34] present a system in which the architecture and Java runtime are co-designed for improved memory access performance. Their system features hardware modifications that allow an object-addressed CPU cache and an in-cache garbage collector.

Recent work has focused on modifying the Android platform for other purposes, such as improving security. TaintDroid [12] and TaintART [31] modify the Dalvik interpreter and the ART compiler, respectively, to add information-flow tracking; SandTrap [25] uses binary instrumentation to support information-flow tracking in native libraries across multiple threads. CleanOS [32] builds on TaintDroid to minimize the exposure of sensitive data by encrypting the data and evicting the on-device copy of the encryption key when the data is not in use.

10 Conclusion

Users of mobile devices expect to use apps and switch between apps with low latency. As mobile apps have become more memory-hungry, device RAM capacities have not kept pace, and traditional swapping mechanisms cannot meet user latency expectations. Marvin overcomes this challenge with a novel runtime-level swapping mechanism that accurately estimates working sets, moves disk I/O off the allocation critical path, and avoids unnecessary swapping during garbage collection. As our experiments demonstrate, Marvin lets more apps run simultaneously and reclaims memory faster than unmodified Android while adding reasonable overhead.

References

[1] Malcolm Atkinson and Ronald Morrison. Orthogonally persistent object systems. *The VLDB Journal*, 4(3):319–402, July 1995.

[2] Anirudh Badam and Vivek S. Pai. Ssdalloc: Hybrid ssd/ram memory management made easy. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11)*, 2011.

[3] UL Benchmarks. Pcmark for android. https://benchmarks.ul.com/pcmark-android. Accessed: 2019-1-9.

[4] Kofi Amankwah Boamah. iphone on-board RAM, July 2017. https://www.researchgate.net/figure/Phone-on-board-RAM-From-figure-8-it-is-clear-that-Apple-either-maintains-the-iPhone_fig1_319307164.

[5] Bumptech. Glide v4: Fast and efficient image loading for android. https://bumptech.github.io/glide/. Accessed: 2018-11-28.

[6] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 105–118, New York, NY, USA, 2011. ACM.

[7] Peter J Denning and Stuart C Schwartz. Properties of the working-set model. *Communications of the ACM*, 15(3):191–198, 1972.

[8] Android Documentation. Activity. https://developer.android.com/reference/android/app/Activity, 2018. Accessed: 2018-11-28.

[9] Android Documentation. Caching bitmaps. https://developer.android.com/topic/performance/graphics/cache-bitmap, 2018. Accessed: 2018-11-30.

[10] Android Documentation. Manage your app's memory. https://developer.android.com/topic/performance/memory, 2018. Accessed: 2018-11-28.

[11] Android Documentation. Saving ui states. https://developer.android.com/topic/libraries/architecture/saving-states, 2018. Accessed: 2018-11-28.

[12] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, 2010.

[13] Umar Farooq and Zhijia Zhao. Runtimedroid: Restarting-free runtime change handling for android apps. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '18, pages 110–122, 2018.

[14] Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. Assessing the scalability of garbage collectors on many cores. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems (PLOS '11)*, 2011.

[15] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, Derek G. Murray, Steven Hand, and Michael Isard. Broom: Sweeping out garbage collection from big data systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, 2015.

[16] Matthew Hertz, Yi Feng, and Emery D. Berger. Garbage collection without paging. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 143–153, 2005.

[17] Tyler Kieft. Building a better instagram app for android. https://instagram-engineering.com/building-a-better-instagram-app-for-android-c08f973662b, 2014. Accessed: 2018-11-9.

[18] Michelle Meyers. Android inches ahead of windows as most popular os. CNET, April 2017. https://www.cnet.com/news/android-most-popular-os-beats-windows-statcounter/.

[19] J. Eliot B. Moss. Design of the mneme persistent object store. *ACM Trans. Inf. Syst.*, 8(2):103–139, April 1990.

[20] Mike Nakhimovich. Improving startup time in the nytimes android app. https://open.blogs.nytimes.com/2016/02/11/improving-startup-time-in-the-nytimes-android-app/, 2016. Accessed: 2018-11-9.

[21] Randy Nelson. The size of iphone's top apps has increased by 1,000% in four years. Sensor Tower, Jun 2017. https://sensortower.com/blog/ios-app-size-growth.

[22] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, 2016.

[23] Diogenes Nunez, Samuel Z. Guyer, and Emery D. Berger. Prioritized garbage collection: Explicit gc support for software caches. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 695–710, New York, NY, USA, 2016. ACM.

[24] James O'Toole, Scott Nettles, and David Gifford. Concurrent compacting garbage collection of a persistent heap. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 161–174, New York, NY, USA, 1993. ACM.

[25] Ali Razeen, Alvin R. Lebeck, David H. Liu, Alexander Meijer, Valentin Pistol, and Landon Cox. Sandtrap: Tracking information flows on demand with parallel permissions. In *Proceedings of the 16th International Conference on Mobile Systems, Applications, and Services (MobiSys '18)*, 2018.

[26] Anshu Rustagi. How we improved our android app "cold start" time by 28%. https://redfin.engineering/how-we-improved-our-android-app-cold-start-time-by-28-a722e231314a, 2018. Accessed: 2018-11-9.

[27] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: An efficient, portable persistent store. In Antonio Albano and Ron Morrison, editors, *Persistent Object Systems*, pages 11–33, London, 1993. Springer London.

[28] Facebook Open Source. Fresco. https://frescolib.org/. Accessed: 2018-11-28.

[29] StackExchange. Creating and enabling an internal storage swap partition on rooted android kitkat. https://android.stackexchange.com/a/89030. Accessed: 2019-4-4.

[30] StackOverflow. ios app maximum memory budget. https://stackoverflow.com/a/15200855. Accessed: 2019-1-9.

[31] Mingshen Sun, Tao Wei, and John C.S. Lui. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*, 2016.

[32] Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, and Nikhil Sarda. Cleanos: Limiting mobile data exposure

with idle eviction. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, 2012.

[33] Carl A. Waldspurger. Memory resource management in vmware esx server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, 2002.

[34] Greg Wright, Matthew L. Seidl, and Mario Wolczko. An object-aware memory architecture. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 2005.

[35] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. Espresso: Brewing java for more non-volatility with non-volatile memory. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 70–83, New York, NY, USA, 2018. ACM.

[36] Ting Yang, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. Cramm: virtual memory support for garbage-collected applications. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, 2006.