One Process to Reap Them All

Garbage Collection as-a-Service

Ahmed Hussein[†‡] Mathias Payer[†] Antony L. Hosking[†*] Chris Vick[§]

[‡]Huawei, USA [†]Purdue U., USA [*]ANU / Data61, Australia [§]Qualcomm, USA

ahmed.hussein2@huawei.com mpayer@purdue.edu antony.hosking@anu.edu.au cvick@quicinc.com

Abstract

Ubiquitous mobile platforms such as Android rely on managed language run-time environments, also known as *language virtual machines* (VMs), to run a diverse range of user applications (apps). Each app runs in its own private VM instance, and each VM makes its own private *local* decisions in managing its use of processor and memory resources. Moreover, the operating system and the hardware do not communicate their low-level decisions regarding power management with the high-level app environment. This lack of coordination across layers and across apps restricts more effective *global* use of resources on the device.

We address this problem by devising and implementing a *global memory manager service* for Android that optimizes memory usage, run-time performance, and power consumption globally across all apps running on the device. The service focuses on the impact of *garbage collection* (GC) along these dimensions, since GC poses a significant overhead within managed run-time environments. Our prototype collects system-wide statistics from all running VMs, makes centralized decisions about memory management across apps and across software layers, and also collects garbage centrally. Furthermore, the global memory manager coordinates with the power manager to tune collector scheduling. In our evaluation, we illustrate the impact of such a central memory management service in reducing total energy consumption (up to 18%) and increasing throughput (up to 12%), and improving memory utilization and adaptability to user activities.

VEE' 17, April 08-09, 2017, Xi'an, China
© 2017 ACM. ISBN 978-1-4503-4948-2/17/04...$15.00
DOI: http://dx.doi.org/10.1145/3050748.3050754

Categories and Subject Descriptors C.1.4 [*Parallel Architectures*]: Mobile processors; D.3.4 [*Programming Languages*]: Processors—Memory management (garbage collection), Run-time environments; D.4.8 [*Performance*]: Measurements

Keywords mobile, power, energy, Android, smartphones

1. Introduction

Mobile devices must balance performance and responsiveness while being constrained by energy consumption and thermal dissipation. With performance, heat, and power consumption strongly tied together, mobile devices come bundled with software components such as kernel governors [11] and proprietary thermal engines that control power and thermal properties (*dynamic frequency*). The crude decisions made by these engines are orthogonal to heuristics for resource management embedded within software components at the user space level.

With the number of Android [28, 31] devices exceeding a billion,[1] the dominance of the Android run-time environment introduces an interesting challenge: we are faced with devices that continuously run dozens of VMs in parallel as *apps* and as services. All these VMs share a set of constrained and over-committed resources. Without global coordination, each VM optimizes independently across competing goals: performance, responsiveness, and power consumption. For example, each VM decides independently what to compile (JIT), when to garbage collect, and what to collect (e.g., using a minor or a major collection). This situation is vastly different from classic desktop or application server systems where VMs use dedicated resources, and where only one or a handful of VM processes run concurrently.

1.1 Motivation

VMs substantially increase productivity by abstracting the hardware and offering useful services such as dynamic optimization, class resolution and, *garbage collection* (GC). However, VM services impose performance and energy costs, ranging from 5% to over 70% [14, 18, 34, 40, 53].

[1]http://expandedramblings.com/index.php/android-statistics/

Tuning of GC. VM services typically come with a number of optimization and scheduling heuristics designed to meet the performance needs of supported applications and users. The tuning of GC performance is achieved by designing a *GC policy* that uses a set of predefined heuristics and the state of app execution to decide *when* and *what* to collect [36]. Configuring a garbage collector is a tedious task because a VM often uses tens of parameters when tuning the garbage collector, specific to the needs of a particular application: e.g., initial heap size, heap resizing, and the mode of collection to perform [10, 42]. Even for a single VM, it is extremely difficult to identify the best collector and heuristics for all service configurations [36, 39, 55].

GC Impact on Energy. Recent interest in fine-grained power measurement shows that GC has a significant impact on energy consumed by the apps, ranging from 7% to 30% [14, 53]. This happens not only because of its explicit overhead on CPU and memory cycles, but also because of implicit scheduling decisions by the OS and hardware with respect to CPU cores. Therefore, a potential approach to optimize GC cost per single VM is to take advantage of GC idleness and control the frequency of the core on which the concurrent collector thread is running [23, 34].

Distributed Controls. Mobile platforms have a software stack comprising various layers with lower-level layers providing services to upper-level layers. These layers of abstraction typically are not tuned in harmony with VM implementation. For example: (*i*) *Device Configurations*: The mobile system has globally fixed VM configurations such as the initial and maximum heap sizes; (*ii*) *OS*: Some heuristics and configurations may be applied on their own, without coordinating with the VM [37, 41]—e.g., the low memory killer that handles out-of-memory events.

Interference Across Running VMs. With dozens of VMs running concurrently on constrained devices, tuning memory configurations for mobile platforms is even more challenging due to interference between VMs across the layers of the hardware and software stack. Local per-VM tuning is a *sub-structured* approach that *mostly fails* to find the globally optimal policy for the entire device. For example, throttling the CPU during GC of the highest priority app [34] cannot handle concurrent GC tasks across the running apps.

1.2 Contributions

Here we consider the impact of GC on the device's overall performance. We identify the missing coordination between concurrent VMs as an opportunity for optimization on mobile systems along the dimensions of (*i*) memory usage, (*ii*) runtime performance, and (*iii*) power consumption. A global service that collects statistics from all running VMs can optimize across these dimensions, and it allows for coordination with power managers to achieve global energy optimization. The service can prioritize GC operations based on estimates of bytes freed, reducing the total work required by individual VMs. The benefits of a global service include efficient resource management, feasible methodology to analyze system behavior, fine control over tuning parameters, and excluded redundancy across the parallel VMs.

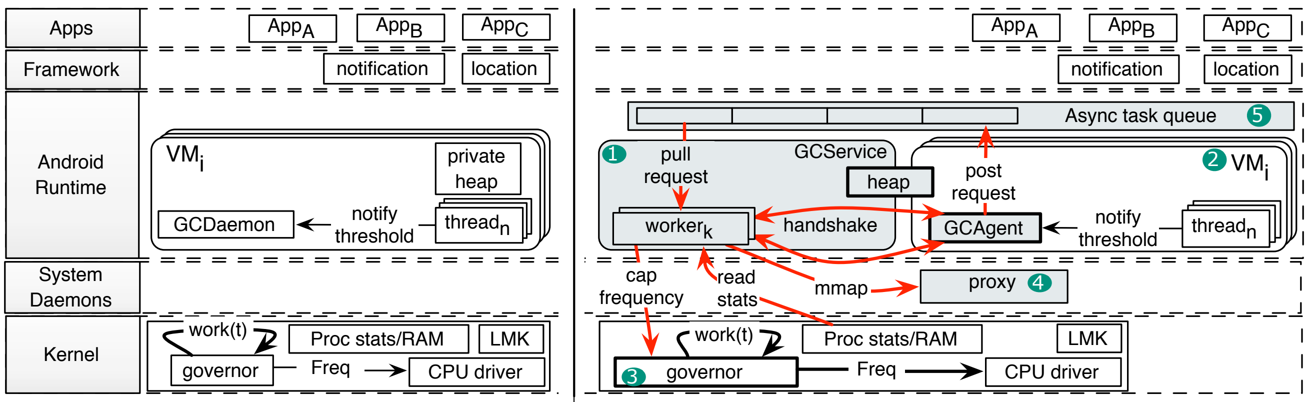
In this paper, we show that a global memory management service provides better control over GC costs and memory utilization. Unlike the existing execution mode, where each collector runs within its own VM, the new platform has a single *GC service* process that serves all running VMs. The GC service unifies interactions between nonadjacent system layers (i.e., the low-level OS power manager) and GC tasks. The service has OS-like access, capable of scanning and collecting per-process VM heaps remotely and gathering statistics about all the running VMs in the system, including process priority, allocation rate, and heap demographics. This allows for fine-grained control over the GC tasks being executed, and their scheduling, compared to just coarsely signaling individual VMs to start GC collections.

We illustrate the power of combining vertical cross-layered heuristics to achieve efficient heap decisions such as compaction, collection, and trimming. GC service efficiency is not limited to local heuristics, resulting in better utilization of system resources based on the workload. We make the following contributions:

- We identify a unique opportunity for optimization on mobile systems by coordinating and orchestrating all the concurrently running VMs.
- We design a global service that collects statistics from all VMs, and we implement a prototype that centralizes GC, including global GC heuristics that optimize memory usage *across* VMs and the actual collection tasks.
- We develop, implement, and evaluate, *in vivo*, a complete running mobile platform based on Android that distributes GC sub-tasks between applications and an OS-like control unit. These heuristics include: heap growth management, compaction, trimming, context-aware task-killing mechanisms, and energy optimization.

2. Background

Mobile platforms employ aggressive power management of sub-systems to improve battery life and to control thermal conditions since these platforms only have passive heat sinks, and for example, their CPUs cannot run continuously at full speed. The governor [11] collects run-time statistics at time t (e.g., work load `work(t)` and core temperature) and then applies complex heuristics—*dynamic voltage and frequency scaling* (DVFS)—to meet optimization criteria [15, 35, 46]. Smartphone energy profiling is a tedious task because it requires (*i*) hardware access to device components, and (*ii*) an understanding of the distribution of power among the device components. A common methodology is to use external instruments to measure the total energy consumed by the device [12, 47, 54], and then extract the



(a) Default Android stack

(b) GC service system overview: new components are shaded

Figure 1: Comparing the default Android and GC service stacks

contributions of subcomponents based on a defined power model [49, 61].

Android. Figure 1a lists the different layers of the Android software stack: (*i*) A modified Linux kernel that provides drivers, shared memory, and interprocess communication; (*ii*) Native libraries and system daemons (i.e., bionic, and thermal engines); (*iii*) Android run-time environment, which is the core VM library that hosts an app; (*iv*) The framework layer that provides services to apps; and (*v*) The application layer that compromises the third party applications installed by the user and the native applications (i.e., browser).

App Isolation. Each app runs in its own VM instance within a single OS process. Isolated processes communicate through RPC using the *Android binder* driver which is implemented in kernel space and is optimized to avoid copying. To guarantee application responsiveness, the VM assigns higher priorities to user interface threads, while background threads are given lower priorities. In addition, Android moves all low-priority background threads to *Linux control groups* (*cgroups*), where they are limited to a small percentage of CPU time. This ensures that the foreground app does not starve, regardless of the number of running background tasks. When an app starts, it skips some initialization steps by inheriting resources created by the *zygote* template process. The zygote reduces app startup time and global memory usage by keeping common resources in copy-on-write memory. Once the process is forked, a per-app garbage collector manages the VM private heap.

Until Android 4.4, Android relied upon the *Dalvik* VM, optimized for memory-constrained devices. Dalvik uses a register-based byte-code format called *Dalvik Executable* (DEX) [28, 31] through a just-in-time (JIT) compiler. The *Android Run-Time* (ART) replaced Dalvik's JIT compiler with ahead-of-time compilation. At app installation time, ART compiles DEX files into native code.

Memory Management. There are four main categories of Android physical memory usage: OS file caches, device I/O buffer, kernel memory, and process VMs. The VM pages constitute the largest portion of memory usage [41]. Each Android per-app GC runs in its own background daemon thread (*GC daemon*) concurrently with the application-level threads (*mutators*). The default Android GC operates as a *concurrent mark-sweep* collector, tracing references from roots that include both thread stacks and other global variables, marking reachable objects via those references, and then marking reachable objects recursively via references stored in those objects. When all of the reachable objects have been marked, the collector sweeps the heap to free the unmarked objects.

ART introduced several enhancements to Dalvik GC, including a pseudo-*generational sticky* garbage collector to deal with short-lived objects [44, 56]. This *sticky mark-sweep* collector is a non-moving generational GC that focuses its effort on separately collecting young objects by (logically) segregating them from older objects. This offers improved locality and shorter collection cycles for *minor* collections of the young space. Both of the Android VMs (ART & Dalvik) use several heuristics to decide how to balance heap growth with collector work, the mode of each collection cycle (minor vs. major), and whether to trim the heap (returning unused pages to the system). The *target heap utilization* ratio controls heap size. If live data exceeds (or falls below) this factor of the heap size (plus or minus some threshold), then the collector reacts by increasing (or decreasing) the size of the available heap. After any successful allocation, if the allocation exceeds the heap threshold, the mutator signals the GC daemon to start a new background GC cycle if it is not already active.

Low-memory Killer (LMK). Android apps are designed to tolerate random restarts. We refer to the app currently on the screen and running as *foreground*. Otherwise, the app is labeled *background* [30]. When a user navigates away from an Android app, the process stays in the memory in order to reduce the start-up overhead when the application is used again. To clean up the physical memory, Android uses a controlled LMK, which kills processes when the available free memory is close to being exhausted. LMK allows specifying a set of out-of-memory (OOM) thresholds to decide on which and how many processes to kill [25].

3. Design and Architecture

We introduce a service that runs as a separate VM process and collects information (i.e., RAM consumed and workload) from running apps and run-time services. Building on this central service that runs as its own separate Linux process, we design a *GC service* component that maps the VM heaps of all running apps into the central service and carries out all GC decisions using *global* (not local) heuristics. Figure 1b shows the high-level interaction between the components of our system:

1. The *server* process maps the client heap into its private virtual address space and manages all coordination and synchronization with other system layers—e.g., power managers. The server has a pool of *worker* threads to support parallel collection tasks. A single worker handles a remote GC task on the client's heap. The worker coordinates with the client and performs *concurrent GC phases*—i.e., recursive marking and sweeping.
2. The *client VM* that hosts the app execution is an isolated Linux process. Instead of making independent GC decisions, a client yields GC policy decisions to the server, including (*i*) heap growth management, (*ii*) GC triggers, and (*iii*) the type of GC operation to perform (e.g., trimming or compaction). Each client coordinates with the server using a dedicated *GC agent* daemon.
3. A GC-aware governor that works in tandem with the server to improve power consumption and performance according to the current workload, taking special measures during GC activity.
4. A platform-specific *proxy* that abstracts the mechanism of sharing the client's heap with the server.
5. An asynchronous *task queue* that allows the GC agent to post requests to the GC service.

When the allocation exceeds the heap limit (*hLimit*), the mutator signals the GC agent, which posts a new GC request, if it does not already have one pending. A worker thread signals the agent to start a collection cycle. The agent performs the initial mark and the remark pauses. The worker thread performs concurrent tracing of the reachable object graph, and then performs a concurrent sweeping phase in which the objects identified as unreachable are collected. During concurrent phases, all the mutators are resumed except the agent that waits for the completion of the phase.

3.1 Global Collector and Energy Optimization

In theory, an optimal GC policy leads to *optimal GC scheduling*. The latter is the trace of GC events throughout the program execution that produces the lowest GC cost [36]. However, with the introduction of other system components into the cost equation (i.e., scheduling and CPU throttling), the GC scheduling can be tuned by hiding expensive GC operations inside of small, otherwise unused idle portions of application execution, which results in *better* overall execution [23]. For mobile devices, tuning the GC implementation to meet performance and power goals is exceptionally difficult, because per-app GC cost is defined as a function of several controls [23, 34] such as: (*i*) the power manager reacting to CPU idle-time during memory-bound GC phases; (*ii*) VM configurations, GC algorithm and the heuristics controlling the heap size; and (*iii*) the workload and memory stress at run time.

In a tracing GC, the collector task is inherently memory bound. Not only the collector is subject to total memory bandwidth, but the mutators are more likely to stall due to stop-the-world phases, and when they are waiting for the collector to finish so they can allocate. Therefore, the mutators are unlikely to make full use of fast cores. Thus, our goal is to tune the GC using the following mechanisms:

1. The prioritization of GC tasks across dozens of simultaneously running VMs needs fine-grained control over scheduling. Put another way, given a set of parallel VMs and a global state of execution, define the selection criteria to pick a VM and the GC task to apply next.
2. The reduction of GC energy cost while allowing for better responsiveness and throughput. This can be achieved either (*i*) by coordinating between the system scheduler and the VM [23], or (*ii*) by capping the frequency of the core during GC phases [34].

Tuning the GC across all the VMs cannot be achieved with each VM independently sending information to the power manager. Therefore, the clients need to offload GC management to a single process. The GC service unifies the interface between the system components and the concurrent VMs. In this way, coordination between the power manager and a single process is feasible and practical.

Having a single process handle the launch of GC tasks allows for more fine-grained control over estimating the memory management overhead and coordinating with other system components. At a high level, the GC service aims to make the most effective decision in a specific situation. The GC service does not (necessarily) collect the heap of the app that is currently running (and is likely requesting memory), but the heap that contains the most garbage. Executing GC tasks by a single process also reduces the code footprint and code cache pressure from individual threads that are running per-app GC and from negative interactions with the scheduler.

3.2 Global GC Service vs. Global GC Policy

Some studies investigate auto-tuning of the GC policy *locally* per single VM [55, 57]. Yet, to date there is no published work on a *global* tuning methodology that combines both the GC configuration policy and the global scheduling decisions on the system. Our service allows holistic and central control of (*i*) detecting conflicts and overlaps between heuristics of components scattered across non-adjacent layers of the system stack, (*ii*) removing redundant functionality that exists between non-adjacent software layers, and (*iii*) identifying unhandled scenarios that result from distributing the memory tuning task across several libraries.

We augment the GC service with the following extensions: (*i*) global device stats—i.e., available RAM and workload, (*ii*) per-process system stats, (*iii*) per-heap stats such as heap variables, fragment distribution, and allocation rate, and (*iv*) the ability to perform GC phases remotely on behalf of other processes. With global system information, the centralized GC service makes more efficient decisions such as trimming the heap that contains the highest fragmentation first, delaying collections when unused memory is available, and even adjusting the heap thresholds based on allocation rates rather than static thresholds.

3.3 Performance

Our system aims at reducing the latency of app responses while assuring better performance and longer (battery) lifetimes. However, with a centralized GC service, a slow messaging mechanism can introduce a new bottleneck. In order to concurrently process multiple GC requests each *worker* can independently handle a GC task in parallel with other workers. A worker pulls the next pending request from the queue once it finalizes a GC task.

Pause Times. GC implementation affects responsiveness as observed by users. GC pauses can prevent threads that service user interface tasks from giving timely responses to user input events. Humans typically perceive interaction pauses greater than 50 ms [24], so any greater pause is likely to be noticed by users. A mutator is interrupted by the GC (*i*) during the initial marking and remarking phases, and (*ii*) when the mutator executes an explicit GC (`System.gc()`). Explicit GC accounts for the worst case pause. We tune the mutator utilization not just by shortening the length of pause phases, but also by preventing the client's mutators from performing GC tasks. All mutators offload *all* GC work to the agent which eliminates the worst case pause scenario. When an allocation request exceeds the hLimit, or fails, the mutator forces allocation (by resizing the heap) and signals the agent before continuing. When a mutator executes an explicit GC call, it simply signals the agent.

Communication Mechanism. The coordination between the agent and the worker implies an inter-process communication (IPC) between the client and the server. One possible shortcoming of poor IPC design is frequent *process context switching*, which is known to have high overhead compared to thread switching [22, 43]. Thus, the performance benefit of the GC service has to be greater than the IPC overhead. Our goal is to mitigate the overhead of interprocess communication as much as possible by: (*i*) scheduling the GC triggers to balance the tension between the frequency of GC, heap size, and IPC overhead [7, 10, 33]; and (*ii*) designing a robust asynchronous messaging mechanism to allow for a fast *handshake* between the worker and the agent threads.

The GC service processes the requests based on priority queues, which provides flexibility to support various heuristics—i.e., processing the foreground application at higher priority. The longest duration of time a request stays pending can be represented as a function

$$\text{latency}(\text{IPC}) = f(\upsilon, \eta, \rho) \qquad (1)$$

where η is the number of requests with higher priority, υ is the total time overhead in context switching, and ρ is the duration spent processing one request.

During GC, the agent and worker threads execute GC phases synchronously. Thus, it is conceivable that both threads cannot be scheduled concurrently on different cores. Because the OS scheduler does not consider shared resources, some scheduling decision may not be optimal for multithreaded execution. We feed this information to the scheduler by *pinning* both threads to a single core until the end of the GC cycle. Forcing the two threads to run on the same core corrects scheduler decisions and prevents thread preemption. This results in better cache performance and locality [9, 59].

3.4 Design Considerations

There are many challenges that need to be addressed when implementing a centralized GC service.

Portability. Placing the GC service in the VM layer enhances portability while keeping the OS unmodified. The communication model between the GC service and the clients must not be platform-specific.

Reliability. It is essential that the GC service provides a uniform mechanism for managing app life cycles in isolation from each other [38]. Therefore, it must be feasible to change the status of a VM without affecting the remaining VMs. This requires separating the internal structure of a client's VM (i.e., static objects). In addition, it is essential that failures in the GC service do not bring the system down.

The stock Android run-time environment restarts apps when they become unresponsive. Like all other Android services, the GC service is designed to restart after crashing. When the GC service is offline, clients perform GC locally until the server is back online. However, the service may halt in the middle of a critical section while locking the client's heap. In this scenario, a client will be restarted for being unresponsive.

Security. Since Android allows execution of native code, serious security issues arise with previous approaches like *Multi-tasking VMs* (MVMs) that allow a single VM to run multiple applications [21, 60]. Such an approach consists of sharing one heap across all running apps, making the system highly vulnerable to security exploits that have low-level access to memory. Our design, on the other hand, offers a secure approach where the code of the central GC service is trusted, and each VM has access to only its own local heap. In the GC service the heap layout must not be identical across all the VMs (including the zygote) to support the shuffling introduced by techniques such as *address space layout randomization* (ASLR). This reduces the risk of memory attacks. Also, we avoid reusing GC internal structures across different VMs—i.e., mark-bitmap and mark-stack are exclusive to a single client.

4. Implementation

Our prototype GC service implementation is based on Android 4.4 (KitKat) ART. KitKat is the latest release that runs on our development hardware platform and uses Linux kernel 3.4. We start our modification based on the open-source SDK and the default configuration. We extend the Android VM with 4K LoC to implement the GC service and the GC agent. We also extend the kernel layer with a few hundred lines of code to allow direct access to run-time statistics.

4.1 System Startup and IPC Support

The system boot process follows the standard steps of loading hardware-dependent components and initializing subsystems and drivers. After starting the kernel, the *proxy* starts as a native service. Following the creation of native daemons the zygote starts by preloading common Java classes and resources. The last step of zygote's initialization is to fork the GC service, which initializes the shared memory regions and the pool of worker threads. The server has a singleton *listener* daemon that fetches tasks from the task queues and inserts them into local queues to be handled by the workers. Occasionally, the server updates the global statistics to adjust its decisions. There are three types of client requests: (*i*) *Registration:* A new client VM sends a request, including the VM process id (PID), the continuous space addresses, and the file descriptor to memory region; (*ii*) *Shutdown:* A hook that is executed during the client VM shutdown sequence; and (*iii*) *Collection:* The agent requests a GC when the allocation exceeds the hLimit threshold, or when a mutator executes an explicit GC call.

When the zygote receives a request to start an app, it forks itself and launches the new app. The new client VM creates the *agent daemon* that sends a registration request to the server. A successful registration shares the client's allocation space with the GC service. A heap collection is triggered when a mutator signals the agent, which in turn forwards a request to the GC service. The agent waits for a message from the server that defines actions to be executed (i.e., GC or trimming).

The shared regions are created using *Android shared memory (ashmem)* that uses *reference-counted virtual memory*, assuring that a shared region is automatically reclaimed when all file descriptors referring to it are closed. Thus, it prevents memory leaks caused by a process crash or by software bugs. Also, ashmem assures that a failure on the server side will not remove the client's heap.

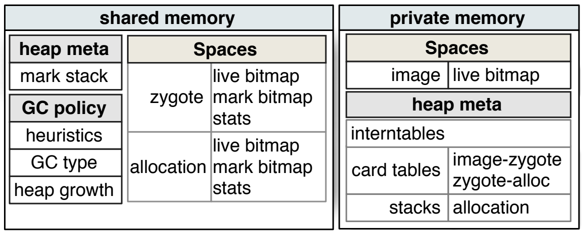


Figure 2: The shared heap layout in the GC service

The IPC messages and signals are implemented using *futexes* [26] to synchronize in user space. The server utilizes a pool of work-stealing threads to reduce the overhead of scheduling. Although Android provides the binder as an IPC mechanism, we implement our communication model on top of shared memory for the following reasons: (*i*) Binder provides synchronous calling, which increases the possibility of context switching between the sender and the receiver, leading to performance degradation [22, 43]; (*ii*) Binder restricts the maximum number of calls that can be handled concurrently (currently 16); and (*iii*) Shared memory makes the system portable and independent of platform-specific features.

4.2 Client Memory Management

The client VM starts by creating a GC agent daemon, which registers the VM with the service. Thus, a subset of heap metadata, zygote, and allocation spaces become accessible to the service. Figure 2 shows the shared memory layout in the new system. The heap layout of a client VM comprises the following main blocks: (*i*) image-space: an immutable contiguous memory region created from an image file, (*ii*) zygote-space: a contiguous memory space inherited from the zygote process (the zygote space is occasionally collected during full GC events), (*iii*) allocation-space: the active contiguous space used by the app. For each individual space, the mark bits are stored in two separate bitmap tables (live and mark) to allow for concurrent progress of the mutator while the collector is tracing the objects. Concurrent marking is supported by a *write barrier* that records dirty objects that have been modified by mutators in a global *card-table*. Inter-space references are stored in internal tables to reduce the overhead of tracing the reachable objects in the allocation space (active). The collector stores the marked objects that need to be scanned during the concurrent tracing in a *mark stack*.

In the absence of the GC service, the client activates sticky collections to reduce the overhead of the GC. The agent uses an *allocation stack* to keep track of the objects

Algorithm 1: Server-side concurrent mark sweep

```
procedure GARBAGECOLLECT                              /* worker thread */
    preparePhase()
    Ⓐ sync gcPhase ←markRoot
        end sync (gcPhase = recursiveMark) Ⓓ
    function RECURSIVEMARK(mark-bitmap, mark-stack)
        delayedList ← empty subset of mark-stack
        while mark-stack ≠ delayedList do
            obj ← mark-stack.pop()
            if ¬ isMarked(mark-bitmap, obj) then
                setMarked(mark-bitmap, obj)
                for fld ← obj.fields do
                    ref ← *fld
                    if ref ∈ allocationSpace then
                        mark-stack.push(ref)
                    else                         /* ref ∈ private memory */
                        delayedList.add(ref)
    Ⓔ sync gcPhase ←remark
        end sync (gcPhase = reclaim) Ⓗ
    function SWEEPALLOCSPACE(mark-bitmap, live-bitmap)
        for i ← 1, mark-bitmap.length() do
            if mark-bitmap[i] &¬(live-bitmap[i]) then
                free(getAddress(mark-bitmap, i)
    adjustHeapLimit()
    Ⓘ Async gcPhase ← finalize
    finishPhase()
```

Algorithm 2: Client-side concurrent mark sweep

```
procedure GARBAGECOLLECT                                  /* GC agent */
    sync gcPhase
        end sync (gcPhase = markRoot) Ⓑ
    function INITIALMARK(mark-bitmap, live-bitmap)
        StopTheWorld();
            mark-ThreadStack()
            mark-InternalTables()
        ResumeTheWorld()
        scanRoots()
        mark-InterSpaceReferences()
    Ⓒ sync gcPhase ←recursiveMark
        end sync (gcPhase = remark) Ⓕ
    StopTheWorld();
        function HANDLEDIRTY
            while ¬(mark-stack.isEmpty()) do
                obj ← mark-stack.pop()
                if ¬(isMarked(mark-bitmap, obj)) then
                    setMarked(mark-bitmap, obj)
                    for fld ← obj.fields do
                        mark-stack.push(*fld)
        swap(mark-bitmap, live-bitmap)
    ResumeTheWorld()
    Ⓖ sync gcPhase ←reclaim
        end sync (gcPhase = finalize) Ⓘ
    finalize();
```

that were allocated since the last GC. The sticky mark-sweep does not adjust the heap size after it completes a collection.

4.2.1 Concurrent Mark Sweep

Algorithms 1 and 2 show how the collection is handled in the GC service. (*A*) The server notifies the GC agent to start a new collection cycle. (*B*) The app-local agent first marks the heap roots including those from thread stacks and globals. (*C*) The agent extends the root set by adding the references from immuned spaces (i.e., zygote and image) to the allocation space). (*D*) The shared *mark-bitmap* is then used by the server to mark the reachable objects recursively. (*E*) The worker yields control to the agent after all objects in allocation stack are scanned. (*F*) The agent starts the remark phase. It *stops the world* and revisits any remaining dirty objects pushed on the *mark-stack* due to concurrent updates by the client application threads (mutators). (*G*) The agent swaps the bitmap tables, resumes the mutators, and signals the worker. (*H*) The worker sweeps the space to create the list of free objects, computes fragment distributions, and calculates the new size of the heap. (*I*) Finally, the worker finalizes the GC cycle concurrently while the agent enforces the new heap threshold.

Since handling the dirty objects requires pausing all threads, we avoid IPC between the agent and the server to assure that the app threads are resumed in a short period of time. On the server side the collector recalculates reference fields based on the base address of the mapped heap. The server scans the reachable objects except for a small set located in the client's private memory range. If the adjusted address does not belong to the mapped range (shared space), the server adds the object to a "*delayed*" list. The GC agent processes the delayed list as a subset of the dirty objects.

4.2.2 Heap Size Management

Mobile apps often exhibit an execution pattern that makes static GC policies ineffective—e.g., music players and games tend to allocate large chunks of data at the beginning of each phase (track/level), causing a spike in the allocation rate and the heap size, followed by minutes with little allocation. Our profiling data shows clear traces of this behavior where the heap oscillates indefinitely.

We address this challenge by avoiding a static threshold. Algorithm 3 illustrates the steps followed by the GC service to set the heap limit for the next GC cycle. (*i*) an app in the start-up phase (*nursery*) grows the heap more aggressively; (*ii*) app priority is a factor for allowed heap growth, (*iii*) for each VM, the GC service stores allocation and resizing information of the last 20 events to adjust to memory usage dynamically (line 24). The service uses the memory allocation rates to auto-adjust the heap growth policy dynamically. This allows for identifying the steady state of the heap volume in smooth steps that eliminate inefficient heap bounds. The GC service updates global statistics when a new app is created, or when an important app changes status, possibly being replaced by another app.

Algorithm 3: Heap growth procedure following a GC task

```
procedure ADJUSTHEAPLIMIT
    $Pri \leftarrow App\ priority$
    $updateFlag \leftarrow$ False
    $resizeFactor \leftarrow$ sizeFactors[$Pri$]
    if App.isNew() then
        App.setLabel(nursery)
        $resizeFactor \leftarrow$ sizeFactors[nursery]
        $updateFlag \leftarrow$ True
    else
        if App.label = nursey then
            if $\neg$(promote(App)) then
                $resizeFactor \leftarrow$ sizeFactors[nursey]
            else
                $updateFlag \leftarrow$ True
                App.setLabel(tenure)
        else
            $Pri_{\text{prev}} \leftarrow App\ priority\ from\ previous\ GC$
            if $Pri \neq Pri_{\text{prev}}$ then
                $updateFlag \leftarrow$ True
    $ipcDelay \leftarrow$ latency(IPC)                /* Equation (1) */
    $allocRate \leftarrow$ (hLimit $-$ heapSize$_{\text{prev}}$)/($\Delta$ time)
    $w \leftarrow$ heapSize $*$ ($resizeFactor$ + 1)
            + $allocRate * ipcDelay$
    hLimit $\leftarrow$ sample(heaplimit, $w$)
    if $updateFlag$ then
        updateGlobalState()
```

4.2.3 Extension: Compaction vs. Trimming

Due to the scarcity of available memory, following a GC cycle the Android VM occasionally scans the heap spaces, releasing empty pages to the system. This *trimming* event is executed on lower priority VMs where the live set occupation falls below a given threshold. This periodic trimming comes at a high price with long-running VMs oscillating indefinitely around the triggering threshold. If the system needs more memory, Android simply kills inactive apps to release their memory pages. The efficiency of trimming depends on the distribution of heap fragments. Note that ART (Android 4.4) does not compact the heap, so any remaining object on a page reduces trimming effectiveness. Knowing that the space leakage in a tracing collector grows much faster than linearly with a heap size [52], it is intuitive to see that a live object occupying just few bytes can prevent the release of a full memory page.

In order to tackle this challenge, we implement a variant of the GC service with *compaction* capability called *GCS-Compact*. The GCS-Compact keeps statistics about empty slots following each full GC. When memory becomes scarce, the GC service *lazily* picks the VM with the highest fragmentation score in the list of low priority VMs. Once picked, the server signals the GC agent to perform a heap compaction. For idle processes, taking advantage of the fact that the VM is already inactive, the GCS-Compact performs the compaction in an *offline* mode.

It is important to distinguish between the *remote compaction* mechanism in the GC service and having a centralized GC manager that signals a specific VM to release the unused pages. In the latter case, each VM needs to perform the compaction task, implying that the process changes its state from *inactive* to *running*. Heap compaction also requires significant per-VM overhead to store the forwarding references (space overhead) and to synchronize attempts to access moved objects [6, 19]. With offline compaction, the server process, which is already *running*, (*i*) avoids signaling an inactive process and (*ii*) omits the need for forwarding references.

4.3 Energy Optimization

The *ondemand* governor controls the energy consumption of the multicore processor based on the observed workload. The governor collects run-time statistics and applies heuristics in an attempt to meet optimization criteria. We integrate the GC service with the CPU power driver, making the governor aware of GC activities (a user-space activity). This allows the governor policy to account for distinct phases of GC behavior in the application workload. By monitoring the workload the GC service makes informed decisions to schedule background tasks with lower GC costs [23].

When the workload across the online cores exceeds a threshold, then the cores ramp up to *optimal_freq* that is set to less than the maximum frequency. If the *optimal_freq* is sufficient to handle the workload, the cores will go back to idle. Otherwise, the cores ramp up to the maximum frequency. At the beginning of a GC cycle, the modified ondemand governor *caps* the maximum frequency of the core on which the collector daemon is scheduled. We calculate the *capped frequency* as the median between the current core frequency and the governor *optimal_freq*. Following the collection cycle, the governor is free to adjust the frequency according to the observed workload and the default settings (see Table 1). GC service coordination with power managers differs from local power optimizations that may inherit conflicting GC scheduling decisions across concurrent VMs [34].

5. Experimental Results

Our centralized framework cuts across multiple layers of the Android 4.4.2 "KitKat" software stack and touches both hardware and operating system aspects. For each experiment, we consider the following main runtime contexts to assess the efficiency of GC decisions on apps that compete on scarce resources:

ART The app runs in *foreground* mode on the default Android ART.

ART-Bgd The app runs in *background* mode on Android ART. By default, ART assigns lower priority to the app, causing the GC to increase the memory constraints.

Table 1: Experimental environment specifications

Hardware	Architecture:	Qualcomm's Snapdragon S4 SoC				
	CPU:	quad-core 2.3GHz Krait				
	Memory:	2GiB				
	Cache:	4KiB + 4KiB direct mapped L0				
		16KiB + 16KiB 4-way set associative L1				
		2MiB 8-way set associative L2				
Software	**VM parameter**		**ondemand parameter**			
	start size:	8 MiB	optimal freq:	0.96 GHz		
	size:	256 MiB	sampling rate:	50 ms		
	targetUtil:	8 MiB	scaling max freq:	2.1GHz		
	LOS threshold:	12 KiB	scaling min freq:	0.3 GHz		
	trim threshold:	75 %	sync freq:	0.96 GHz		
	LMK minfree:	12288; 15360; 18432; 2150; 24576; 30720 (pages)				
Microarchitecture		Level	Size	Miss Lat.	Line	Replace
	TLB	1	32	4.27 ns	–	–
		2	128	33.39ns	–	–
	Cache	1	16 KiB	3.21ns	64B	3.28ns
		2	2 MiB	10.03ns	128B	10.63ns
	CPU/L1	1.85ns	Proc Ctx.	$43.41\mu s$	Thread Ctx.	$9.56\mu s$

GCService The app runs in foreground mode on a system that deploys GC as a service.

GCService-Bgd The app runs in *background* mode on GCService. The heap growth of the app is lowered only if the global memory is scarce.

GCS-Compact The app runs on GCService with compaction enabled.

For all applications, we use the *Monkeyrunner* tool to automate user inputs [29]. We use the APQ8074 DragonBoard™ Development Kit with specifications described in Table 1.

5.1 Methodology and Benchmarking

For each metric we describe the techniques and controls used to obtain the results. We also characterize a set of applications used in the experiments.

Consistent Lightweight Profiling. We have instrumented the Dalvik/ART VMs and the kernel to record statistics on demand. To avoid perturbing mutators the profiling daemon does not synchronize with them. To avoid environmental perturbation, we run experiments that are sensitive to time and scheduling with the thermal engine disabled. We note that the thermal engine controls the CPU frequency, increasing non-determinism of the experiments—i.e., execution time and power consumption will change depending on the temperature. The VM profiler runs as a C-coded daemon thread inside ART and Dalvik. This daemon only runs when we are collecting execution statistics such as performance counters or GC events and not for measurements that are sensitive to timing or scheduling such as total execution time and OS context switching. The data from this daemon are *not* used for our heuristics, but to evaluate the system *in-vivo*. The profiler daemon does not synchronize with app threads to avoid perturbing app execution.

Table 2: Workload description

xalan:	A multithreaded `XSLT` processor for transforming XML documents into HTML, or text which is not supported by default on Android platforms [8].
lusearch:	A multithreaded text search over a set of input files [8].
sqlite:	A multithreaded app that executes, in memory, a number of transactions against SQL queries using Android `SQLite` [8].
JSON:	A multithreaded app that serializes a set of JSON files to Java objects. At least 12% of Android apps use JSON libraries [3].
SVG:	A single threaded parser based on `AndroidSVG` [2].
App store:	Quadrant, Pandora, AngryBirds, and Spotify

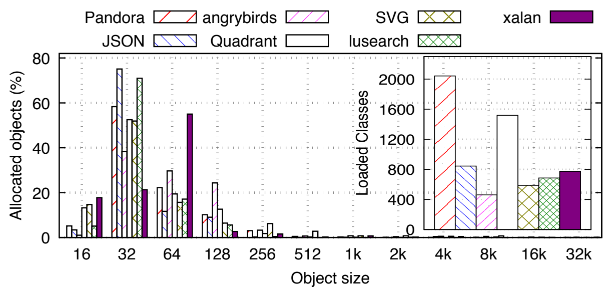


Figure 3: Object size histograms and loaded classes

Workload. Mobile apps are characterized by their event based behavior. There are many sources of non-determinism at the application level, including interference caused by shared data for concurrent tasks, tasks racing to access peripheral devices, and interference from scheduled background tasks. The goal is to define a benchmark suite that is used to optimize mobile environments, considering metrics that are relevant to user interactions and metrics that simplify correlation of underlying platform events across the layers (e.g., hardware, OS, runtime, and application). Table 2 lists a set of applications that wrap popular Android libraries, which allow various workload sizes and number of iterations.

We profile the object size in a perfectly compacted heap (64KiB). Figure 3 plots the percentage of objects (y-axis) in each object size (x-axis) that the app allocates. The number of loaded classes reflect the variance in object types.

Energy Profiling. GC energy consumption exhibits nonlinear relationship with resource utilization—i.e., CPU and memory cycles. Therefore, a *utilization-based* power model is not suitable for our experiments [61]. Instead, we use an *event-based* model [5, 49] that captures the relationship between GC events and total power consumption of the device. We measure the total physical energy consumed during the app execution using a hall-effect linear current sensor [1], and we read the output voltage using a National Instruments NI-6009 data acquisition device [48]. We correlate the re-

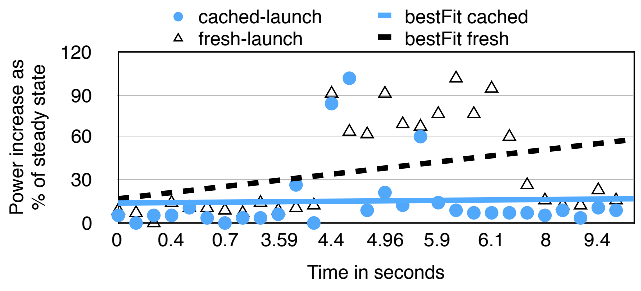


Figure 4: Power trends when launching apps for first time (fresh) vs. invoking background apps (cached)

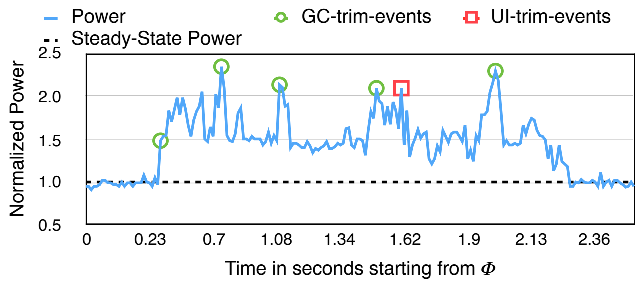


Figure 5: The overall power measured when running Spotify in the background normalized to the steady state, which shows the impact of trimming on power consumption

sults and the configurations of several layers considering different controls. Once the app starts execution, the profiler reads the voltage drop across the device at a sample rate of 2 kS/s.

Pause Times. The responsiveness of embedded systems was thoroughly studied and evaluated by estimating the *Worst-Case Execution Time* (WCET) of individual tasks leading to the existence of several commercial tools and research prototypes [58]. However, worst case and average mutator pause times do not adequately characterize the impact of GC on responsiveness because of the complexity of the system stacks. Thus, we use *minimum mutator utilization* (MMU) [20, 39, 51] over a range of time intervals. For each individual mutator we gather the pauses during the following events: (*i*) safepoint pauses, when a mutator stops in response to a suspension request (e.g., for marking mutator roots), (*ii*) foreground pauses, when a mutator performs a foreground GC cycle, and (*iii*) concurrent pauses, when a mutator waits for a concurrent GC cycle to finish.

We compute the MMU for a multithreaded app having a total execution time T and M mutators $m_1, \ldots, m_M$, each experiencing p_i GC pauses $\delta_1, \ldots, \delta_{p_i}$, we define MMU for a window of length w as the MMU (for all mutators) over all time slices of length w in the execution.

5.2 Energy and Performance Evaluation

Table 1 shows the micro-architectural characteristics of our platform. The metrics include: TLB, Cache, and process context switch performance. Although the process context switching overhead is known to be high compared to thread switching, our profiling of the scheduling statistics on Android ART and the GC service are indifferent.

The default Android memory system is tuned for single monolithic applications. First, following a collection, the default collector iterates through all allocated heap memory and trims free pages if the app is in the background. This scenario is inefficient as (*i*) trimming is executed for every collection (as long as the heap utilization is less than the trimming threshold), leading to diminishing returns for trimming sparse heaps, (*ii*) low priority applications with sparse heaps do not trigger GC and therefore hold on to empty pages, and (*iii*) the trimming decision does not consider global state, leading to unnecessary GC overhead in unstressed environments. Second, the default Android LMK is aggressive, killing apps even when memory is not exhausted [27]. Process killing is especially problematic for apps that are designed to run in the background like music players.

Restarting Android Apps. The killing of VM processes has an implicit penalty overhead when the user reopens the apps. We measure the average power consumed when we launch a set of apps for the first time, and we compare the same power traces when the apps are cached in background. Figure 4 demonstrates that the re-launch of the apps that were killed by the LMK has a large impact on energy consumption. In addition, our experiments reveal that local GC trimming operations increase the power leaks for apps running in the background.

GC Impact on Low Priority Apps. To analyze the impact of background GC tasks on energy, we calculate the *steady state* power consumption (device is idle) as a baseline, and we correlate between power measurements and GC events. *Spotify*—one of the most popular apps on Android store—provides a streaming functionality that lets a user listen to music. Our script launches Spotify, enters the login credentials, and then listens to the default music channel for a specified amount of time. Once Spotify is launched, the VM profiler collects the memory behavior and heap characteristics as a function of time in two different settings: (*i*) Spotify is the foreground app, and (*ii*) Spotify is sent to the background after four minutes. Figure 5 shows a time window (starting at time ϕ) obtained when Spotify is pushed to the background, demonstrating the high cost of heap trimming.

Sending Top App to Background. This experiment assesses the efficiency of GC decisions on low priority apps. We evaluate GC behavior when the front app is pushed to the background during a non-stressed state of execution (i.e., the device has plenty of free memory). Figure 6 shows the execution time and power results, with a confidence interval (5%), of running each benchmark for eight iterations after a warmup under the two different Android systems. For Android ART, apps running in background exhibit considerable

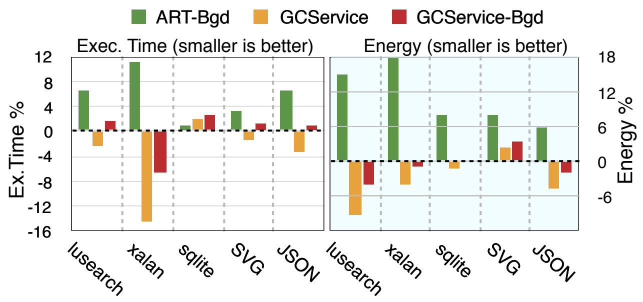


Figure 6: App execution time & energy in foreground and background modes compared to the default execution

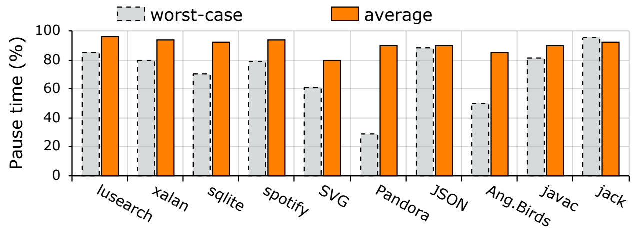


Figure 7: Worst case and average pauses in GC service as % of respective ART pauses

trimming, slowing down app execution and leaking more energy. Our experiments show that trimming phase may span up to 0.6s. For GCService, the GC does not perform any trimming, because the memory is not stressed. Note that sqlite, having bigger live set, exhibits a slow down due to the heap growth thresholds (see Algorithm 3) used in our experiments.

Responsiveness. We instrumented the pause segments during execution of each thread. For each mutator, the MMU generates the maximum pause time (the longest window for which mutator CPU utilization is zero). Figure 7 shows the average and the maximum value (worst-case) recorded across all the mutators in the execution. Pandora and Angrybirds execute several explicit GC calls during execution. This implies that the app mutator executes the GC cycles, increasing the maximum pause times of that mutator. For the GC service, delegating GC to the service process avoids GC delays and context switches between threads of the app. The second reason for reduced GC pause times is the existence of an upper bound for the number of objects to be allocated between two collection cycles. Finally, special handling for apps in the start-up phase reduces the average time needed to launch an app.

5.3 Space Analysis

Here we demonstrate how the GC service meets user requirements and executes seamlessly on real devices. With an increase in memory used by the foreground app, physical memory may become insufficient. According to the values of `minfree` in Table 1, the LMK starts killing processes from the lowest priority group. Profiling global device resources by running experiments that simulate real world scenarios is difficult due to the non-deterministic execution of mobile platforms—e.g., some random services may fail during system start-up, resulting in a variable amount of available memory for each run.

The GC service Space Overhead. As we describe in Section 4.1, the shared regions are created as reference-counted virtual memory. The metadata overhead is small and bound by the number of running applications.

Android ART puts new objects in the *large object space* (LOS) when its absolute size exceeds a predefined threshold, “*large object threshold*”. The current GC service implementation limits the heap to contiguous memory regions, causing slightly more overhead when scanning the heap. Figure 3 shows that few object allocations would be affected by disabling the LOS.

Sequential App Execution and Compaction. Here, we compare between the compactor variant of the GC service, GCS-Compact, and the default Android system. This benchmark launches several apps, switching between them by pressing the *home* button. When an app A_i is brought back to the foreground it (*i*) may have been killed, triggering a fresh start, or (*ii*) it is still running, refreshing existing pages. In both cases, switching to A_i increases memory pressure. Occasionally, Android responds to this increase by killing processes from the lowest priority group to release their memory. Throughout execution, the system running Android ART kills 27 processes including the Browser and BBCNews-Service. The GCS-Compact, on the other hand, reduces the number of killed processes to only 14–19 (depending on the individual run) without coordinating with the *Android runtime manager*.

Figure 8 shows the variation of memory through the sequence of events. The stacked bars indicate the total memory used for each app process at a given point of time. Since we do not have *precise* control on the number of processes running at the beginning of the experiment, we present the different memory curves of the GCS-Compact and Android ART. The service (the right column) reserves more memory for the foreground app as a result of the heuristics that allow the high priority apps to consume more memory. However, the service is more effective in releasing memory from apps running in the background by executing compaction followed by trimming.

Figure 9 shows the number of trims performed by the apps (excluding `System` processes). Android ART performs fewer trimming operations on a limited set of apps compared to the GCS-Compact. Therefore, the default Android ART tends to kill more apps as a result of not reclaiming memory from inactive apps.

The Impact of Growth Policies on App. Users flag apps as battery and memory drainers when they cause issues on

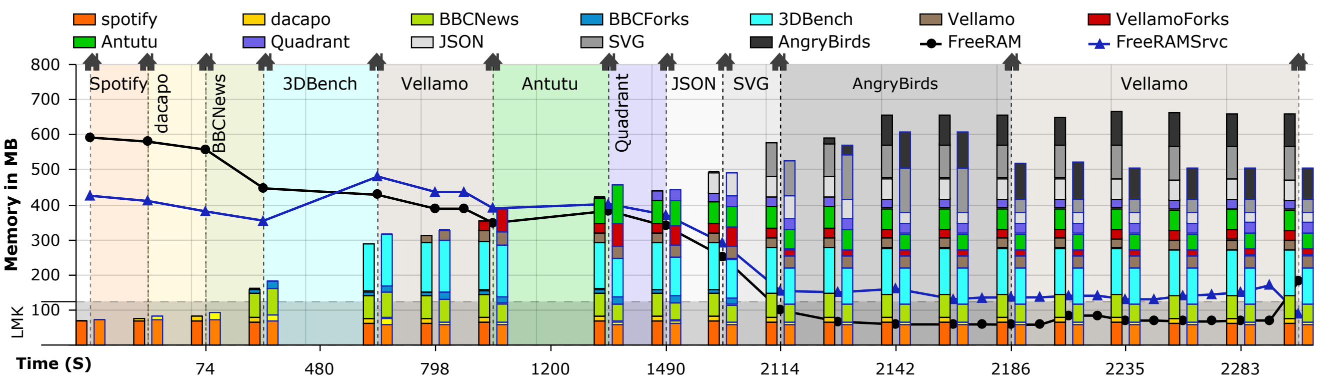


Figure 8: Memory stats vs. time in seconds: Stacked bars are the total apps' memory in ART (left) and the GCS-Compact (right); available RAM (FreeRAM/Srvc); the LMK range; and the current foreground app (vertical guides)

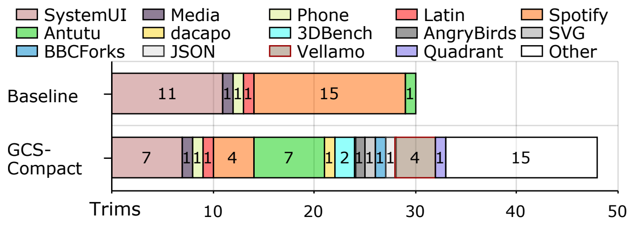


Figure 9: Trim counts per app throughout the execution time

the device. This benchmark analyzes the memory behavior of Spotify—frequently flagged by users as a drain on the battery [4]—that leads to the power leaks in Figure 5. As a foreground app, Spotify executes 58 concurrent garbage collections. These GC events consume up to 7% of the total app CPU cycles excluding idle cycles (measured using hardware performance counters). In the background experiment, Spotify executes 60 concurrent collections. However, listening for 5 minutes of music triggers 8 trimming operations in the background. This increases the GC overhead from 7% to 10% of the total CPU cycles. Note that Spotify gets less CPU slots when it falls to the background based on Android scheduling policies.

To explain the high frequency of trimming operations, we profile heap variables and the distribution of free slots following each concurrent cycle. The results reveal that trim operations are not effective, because the gaps after collecting small size objects do not form contiguous memory chunks that can be released to the system (see Figure 3). Figure 10 shows that the heap characteristics of both settings are very close to each other, despite the extra work done to restrict the heap size in the background mode.

Compared to ART, the GC service reduces the total garbage collections to 24 (50% fewer). Not only the collection overhead is reduced, but the total heap space is also reduced by 10%. The main reasons leading to these improvements are: (*i*) The heap growth manager improves the resizing decisions by removing steps that reach a local maximum; and (*ii*) Executing major collections (young and old objects) during the start-up phase reduces the fragments; hence, the heap utilization is high and the total space occupied is small. For Android ART, low heap utilization caused by fragmentation occasionally falls below the trimming threshold.

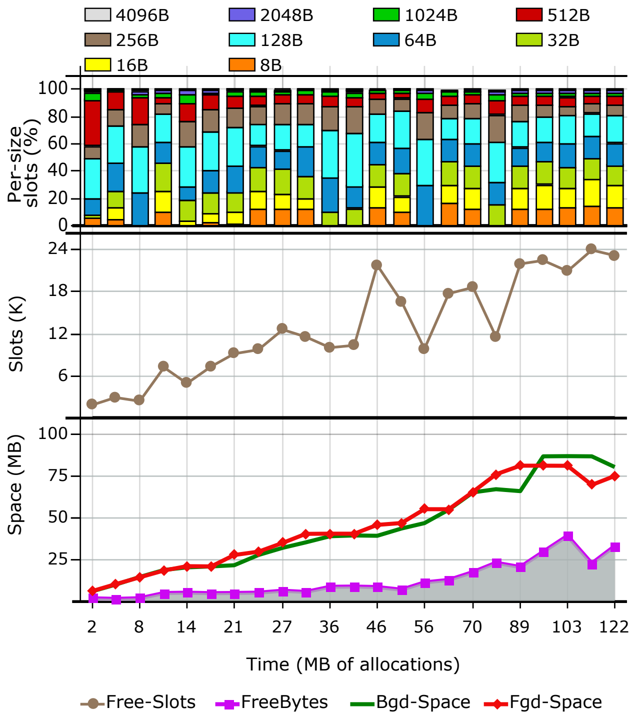


Figure 10: Heap characteristics of Spotify vs. time measured in bytes allocated: (1) the heap size in background and foreground modes (Bgd/Fgd-Space) and the free slots volume, (2) the count of free slots, and (3) the histogram of empty slots grouped by their size.

6. Discussion

One objective of this work is to reduce the GC's energy consumption through direct coordination with DVFS. In the future, we hope to extend the work to dynamically calculate the optimum core frequency based on the observed workload and memory allocation rates. This kind of optimization is infeasible if each individual VM communicates with the power manager directly to perform a collection cycle.

While we described the potential benefits of providing GC as a service in mobile systems, there are few items that bear further discussion: tuning of heuristics, limitations, and the benefit of deploying a centralized VM service.

Heuristic Manager. Existing GC tuning has focused on long running applications. For e.g., in server applications, a VM instance is initialized occasionally and executes continuously in a steady state after a sufficient warmup phase. On the other hand, mobile app VMs are designed to be killed and restarted frequently, increasing the per-app GC overhead.

We introduce a design that takes advantage of global statistics and improves the status quo. The GC service can be extended through a plugin-based system that allows for customization of its policies. Finally, the global service is capable of providing several important and useful information about the app memory usage and the user interactions with the app. Most importantly, the service can predict usage patterns (e.g., which apps may be used in a certain context).

Limitations. The GC service maps the heap into its address space. When the client exits, the GC service unmaps the heap. When the mapping fails, the service removes existing mapped heaps based on LRU. They will be re-mapped when they are used again. We consider another approach in which the client heap is directly added to the page table of the server process without calling map-unmap. This can increase the performance when a non existing VM needs to be added. The integration with the memory manager can improve the GC service performance. For example, replacing `mmap` with an efficient system call that directly inserts the client heap in the server page table will cut the overhead of sharing the heap with the server.

7. Related Work

Energy and thermal engines, such as DVFS, are invisible to user level developers. Several fine-grained profile tools [12, 47, 49, 54] show that apps still have room for significant improvements by reducing idle time and power leaks caused by inefficient software subcomponents [5, 18, 49, 50]. This knowledge fueled the interest in tuning the performance and energy by direct coordination of the application layer and the power managers [23, 34, 37, 40].

GC costs for restricted memory environments have been studied [16, 17, 32], but the precise relationship between the GC tradeoffs and energy consumption and illuminating the way GC is affected by the system layers were beyond these studies. Recently, the impact of memory management on modern mobile devices has been redefined on (*i*) the page-level [41], and (*ii*) the managed heap by the VM runtime [34].

Efforts to efficient management of independent Java VM heaps led to dynamic allocation of resources shared between the running virtual machines [13, 14, 45]. These studies differ from our study in that (*i*) our system focuses on a restricted mobile platform that hosts dozens of parallel virtual machines, (*ii*) we evaluate the GC as perceived globally on the system profiler, and (*iii*) the GC service performs GC phases on all virtual machines. Our work is the first to present a runtime service on mobile platforms that can manage the heap of all running VMs while keeping each instance in its own process, which is different from other approaches like MVM [21, 60].

8. Conclusion

Mobile devices pose novel challenges to system designers, because they juggle access to limited resources like battery usage against app performance and responsiveness to user actions. The Android system is running dozens of concurrent VMs, each running an app on a single device in a constrained environment. Unfortunately, the mobile system so far treats each VM as a monolithic instance.

We have introduced a VM architecture that addresses major resource bottlenecks (memory and energy) by presenting a central GC service that evaluates GC decisions across all running VMs and optimizes according to global heuristics. The GC service has the following benefits: (*i*) it reduces the cost of GC by tuning GC scheduling decisions and coordinating with the power manager, (*ii*) apps run in their own processes, ensuring separation between processes, (*iii*) it eliminates sparse heaps, releasing more pages back to the system, (*iv*) it performs opportunistic compaction and trimming on sparse heaps, reducing the total overhead needed to release memory from idle apps, (*v*) it reduces the number of processes killed by the system LMK by returning more pages, and (*vi*) it saves device resources during memory recycling.

We believe that centrally managed VM services on a mobile system will open up other research topics such as code optimization. Our central service has the power to remove redundancy and conflicting local heuristics, replacing them with a globally integrated alternative.

Acknowledgments

This work has been supported by Qualcomm and the National Science Foundation under grants number CNS-1161237 and CCF-1408896.

References

[1] *ACS714: Hall Effect-Based Linear Current Sensor*. Allegro MicroSystems, LLC. URL http://www.pololu.com/product/1185.

[2] AndroidSVG. *AndroidSVG — SVG rendering library for Android*, 2015. URL http://bigbadaboom.github.io/androidsvg.

[3] *AppBrain Android market*. AppTornado GmbH, 2016. URL http://www.appbrain.com/.

[4] *AVG Android App performance report Q3*. AVG. Now, 2015. URL http://now.avg.com/avg-android-app-performance-report-q3-2015/.

[5] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy consumption in mobile phones: A measurement study and implications for network applications. In *ACM SIGCOMM Conference on Internet Measurement Conference*, IMC'09, pages 280–293, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-771-4. doi: 10.1145/1644893.1644927.

[6] A. Bendersky and E. Petrank. Space overhead bounds for dynamic memory management with partial compaction. *ACM Transactions on Programming Languages and Systems*, 34(3):13:1–13:43, Nov. 2012. doi: 10.1145/2362389.2362392.

[7] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. In *The Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, pages 25–36, New York, NY, USA, 2004. ACM. ISBN 1-58113-873-3. doi: 10.1145/1005686.1005693.

[8] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 169–190, Portland, Oregon, Oct. 2006. doi: 10.1145/1167473.1167488.

[9] S. Blagodurov, S. Zhuravlev, and A. Fedorova. Contention-aware scheduling on multicore systems. *ACM Transactions on Computer Systems*, 28(4):8:1–8:45, Dec. 2010. ISSN 0734-2071. doi: 10.1145/1880018.1880019.

[10] T. Brecht, E. Arjomandi, C. Li, and H. Pham. Controlling garbage collection and heap growth to reduce the execution time of java applications. *ACM Transactions on Programming Languages and Systems*, 28(5):908–941, Sept. 2006. doi: 10.1145/1152649.1152652.

[11] D. Brodowski. *CPU frequency and voltage scaling code in the Linux kernel*. URL https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt.

[12] N. Brouwers, M. Zuniga, and K. Langendoen. Neat: A novel energy analysis toolkit for free-roaming smartphones. In *ACM Conference on Embedded Network Sensor Systems*, SenSys'14, pages 16–30, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3143-2. doi: 10.1145/2668332.2668337.

[13] C. Cameron, J. Singer, and D. Vengerov. The judgment of Forseti: Economic utility for dynamic heap sizing of multiple runtimes. In *ACM SIGPLAN International Symposium on Memory Management*, pages 143–156, Portland, Oregon, June 2015. doi: 10.1145/2754169.2754180.

[14] T. Cao, S. M. Blackburn, T. Gao, and K. S. McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *International Symposium on Computer Architecture*, pages 225–236, Portland, Oregon, June 2012. doi: 10.1109/ISCA.2012.6237020.

[15] A. Carroll and G. Heiser. Unifying DVFS and offlining in mobile multicores. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 287–296, Berlin, Germany, Apr. 2014. doi: 10.1109/RTAS.2014.6926010.

[16] G. Chen, M. T. Kandemir, N. Vijaykrishnan, M. J. Irwin, and M. Wolczko. Adaptive garbage collection for battery-operated environments. In *USENIX Java Virtual Machine Research and Technology Symposium*, pages 1–12, San Francisco, California, Aug. 2002. URL https://www.usenix.org/legacy/event/jvm02/chen_g.html.

[17] G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and M. Wolczko. Tuning garbage collection for reducing memory system energy in an embedded Java environment. *ACM Transactions on Embedded Computing Systems*, 1(1):27–55, Nov. 2002. doi: 10.1145/581888.581892.

[18] X. Chen, A. Jindal, N. Ding, Y. C. Hu, M. Gupta, and R. Vannithamby. Smartphone background activities in the wild: Origin, energy drain, and optimization. In *International Conference on Mobile Computing and Networking*, MobiCom'15, pages 40–52, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3619-2. doi: 10.1145/2789168.2790107.

[19] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, Nov. 1970. doi: 10.1145/362790.362798.

[20] P. Cheng and G. E. Blelloch. A parallel, real-time garbage collector. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 125–136, Snowbird, Utah, June 2001. doi: 10.1145/378795.378823.

[21] G. Czajkowski and L. Daynés. Multitasking without compromise: A virtual machine evolution. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 125–138, Tampa, Florida, 2001. doi: 10.1145/504282.504292.

[22] F. M. David, J. C. Carlyle, and R. H. Campbell. Context switch overheads for Linux on ARM platforms. In *Workshop on Experimental Computer Science*, ExpCS'07, San Diego, California, 2007. doi: 10.1145/1281700.1281703.

[23] U. Degenbaev, J. Eisinger, M. Ernst, R. McIlroy, and H. Payer. Idle time garbage collection scheduling. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 570–583, San

Jose, California, 2016. ISBN 978-1-4503-4261-2. doi: 10.1145/2908080.2908106.

[24] R. Efron. Conservation of temporal information by perceptual systems. *Perception & Psychophysics*, 14(3): 518–530, Oct. 1973. doi: 10.3758/BF03211193.

[25] *Taming the OOM killer*. Eklektix, Inc., 2009. URL http://lwn.net/Articles/317814/.

[26] H. Franke and R. Russell. Fuss, futexes and furwocks: Fast userlevel locking in Linux. In *Ottawa Linux Symposium*, pages 479–495, Ottawa, Canada, June 2002. URL http://www.kernel.org/doc/ols/2002/ols2002-pages-479-495.pdf.

[27] *AOSP issue tracker, 98332: Low memory killer is extremely aggressive*. Google Inc., 2015. URL https://code.google.com/p/android/issues/detail?id=98332.

[28] *Android ART and Dalvik*. Google Inc., 2015. URL https://source.android.com/devices/tech/dalvik/art.html.

[29] *monkeyrunner API*. Google Inc., 2015. URL http://developer.android.com/tools/help/monkeyrunner_concepts.html.

[30] *Android Processes and Threads*. Google Inc., 2016. URL https://developer.android.com/guide/components/processes-and-threads.html.

[31] *Android Open Source Project*. Google Inc., 2016. URL http://source.android.com.

[32] P. Griffin, W. Srisa-an, and J. M. Chang. An energy efficient garbage collector for Java embedded devices. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 230–238, Chicago, Illinois, June 2005. doi: 10.1145/1065910.1065943.

[33] X. Guan, W. Srisa-an, and C. Jia. Investigating the effects of using different nursery sizing policies on performance. In *ACM SIGPLAN International Symposium on Memory Management*, ISMM'09, pages 59–68, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-347-1. doi: 10.1145/1542431.1542441.

[34] A. Hussein, A. L. Hosking, M. Payer, and C. A. Vick. Don't race the memory bus: Taming the GC leadfoot. In *ACM SIGPLAN International Symposium on Memory Management*, pages 15–27, Portland, Oregon, 2015. doi: 10.1145/2754169.2754182.

[35] A. Iyer and D. Marculescu. Power efficiency of voltage scaling in multiple clock, multiple voltage cores. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 379–386, San Jose, California, Nov. 2002. doi: 10.1145/774572.774629.

[36] N. Jacek, M.-C. Chiu, B. Marlin, and E. Moss. Assessing the limits of program-specific garbage collection performance. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 584–598, Santa Barbara, California, June 2016. doi: 10.1145/2908080.2908120.

[37] M. R. Jantz, F. J. Robinson, P. A. Kulkarni, and K. A. Doshi. Cross-layer memory management for managed language applications. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 488–504, Pittsburgh, Pennsylvania, Oct. 2015. doi: 10.1145/2814270.2814322.

[38] *JSR 121: Application Isolation API Specification*. Java Community Process. URL https://jcp.org/en/jsr/detail?id=121.

[39] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC Press, 2011.

[40] M. Kambadur and M. A. Kim. An experimental survey of energy management across the stack. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 329–344, Portland, Oregon, Oct. 2014. doi: 10.1145/2660193.2660196.

[41] S.-H. Kim, S. Kwon, J.-S. Kim, and J. Jeong. Controlling physical memory fragmentation in mobile systems. In *ACM SIGPLAN International Symposium on Memory Management*, pages 1–14, Portland, Oregon, June 2015. doi: 10.1145/2754169.2754179.

[42] P. Lengauer and H. Mössenböck. The taming of the shrew: Increasing performance by automatic parameter tuning for Java garbage collectors. In *ACM/SPEC International Conference on Performance Engineering*, pages 111–122, Dublin, Ireland, 2014. ISBN 978-1-4503-2733-6. doi: 10.1145/2568088.2568091.

[43] C. Li, C. Ding, and K. Shen. Quantifying the cost of context switch. In *Workshop on Experimental Computer Science*, ExpCS'07, San Diego, California, 2007. doi: 10.1145/1281700.1281702.

[44] H. Lieberman and C. E. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6):419–429, June 1983. doi: 10.1145/358141.358147.

[45] M. Maas, K. Asanović, T. Harris, and J. Kubiatowicz. Taurus: A holistic language runtime system for coordinating distributed managed-language applications. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 457–471, Atlanta, Georgia, Apr. 2016. doi: 10.1145/2872362.2872386.

[46] A. Miyoshi, C. Lefurgy, E. Van Hensbergen, R. Rajamony, and R. Rajkumar. Critical power slope: Understanding the runtime effects of frequency scaling. In *International Conference on Supercomputing*, pages 35–44, New York, New York, June 2002. doi: 10.1145/514191.514200.

[47] *Monsoon Power Monitors*. Monsoon Solutions Inc., 2016. URL https://www.msoon.com/LabEquipment/PowerMonitor/.

[48] NI. *NI USB-6008/6009 user guide and specifications*, Feb. 2012. URL http://www.ni.com/pdf/manuals/371303m.pdf.

[49] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang. Fine-grained power modeling for smartphones using system call tracing. In *ACM European Conference on Computer Systems*, pages 153–168, Salzburg, Austria, Apr. 2011. doi: 10.1145/1966445.1966460.

[50] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In *ACM European Conference on*

Computer Systems, pages 29–42, Bern, Switzerland, Apr. 2012. doi: 10.1145/2168836.2168841.

[51] T. Printezis. On measuring garbage collection responsiveness. *Science of Computer Programming*, 62(2): 164–183, Oct. 2006. doi: 10.1016/j.scico.2006.02.004.

[52] J. M. Robson. Worst case fragmentation of first fit and best fit storage allocation strategies. *The Computer Journal*, 20 (3):242–244, Aug. 1977.

[53] J. B. Sartor and L. Eeckhout. Exploring multi-threaded Java application performance on multicore hardware. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 281–296, Tucson, Arizona, Oct. 2012. doi: 10.1145/2384616.2384638.

[54] A. Schulman, T. Schmid, P. Dutta, and N. Spring. *Demo: Phone Power Monitoring with Battor*, 2011. URL http://www.cs.umd.edu/~schulman/battor.html.

[55] J. Singer, G. Brown, I. Watson, and J. Cavazos. Intelligent selection of application-specific garbage collectors. In *ACM SIGPLAN International Symposium on Memory Management*, pages 91–102, Montréal, Canada, Oct. 2007. doi: 10.1145/1296907.1296920.

[56] D. M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, Pittsburgh, Pennsylvania, Apr. 1984. doi: 10.1145/800020.808261.

[57] D. R. White, J. Singer, J. M. Aitken, and R. E. Jones. Control theory for principled heap sizing. In *ACM SIGPLAN International Symposium on Memory Management*, pages 27–38, Seattle, Washington, June 2013. doi: 10.1145/2464157.2466481.

[58] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem — overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):36:1–36:53, May 2008. doi: 10.1145/1347375.1347389.

[59] F. Xian, W. Srisa-an, and H. Jiang. Contention-aware scheduler: Unlocking execution parallelism in multithreaded Java programs. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'08, pages 163–180, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-215-3. doi: 10.1145/1449764.1449778.

[60] Y. Yan, C. Chen, K. Dantu, S. Y. Ko, and L. Ziarek. Using a multi-tasking VM for mobile applications. In *International Workshop on Mobile Computing Systems and Applications*, HotMobile'16, pages 93–98, St. Augustine, Florida, Feb. 2016. doi: 10.1145/2873587.2873596.

[61] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *IEEE/ACM/IFIP international Conference on Hardware/Software Codesign and System Synthesis*, CODES/ISSS'10, pages 105–114, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-905-3. doi: 10.1145/1878961.1878982.