PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Ahmed Mohamed Abdelhaffiez Hussein

Entitled
EFFECTIVE MEMORY MANAGEMENT FOR MOBILE ENVIRONMENTS

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Antony L. Hosking	
Mathias Payer	
Suresh Jagannathan	
Xiangyu Zhang	

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy of Integrity in Research" and the use of copyright material.

Approved by Major Professor(s): Antony L. Hosking and Mathias Payer

Approved by: William Gorman 11/10/2016
Head of the Departmental Graduate Program Date

EFFECTIVE MEMORY MANAGEMENT

FOR MOBILE ENVIRONMENTS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Ahmed Mohamed Abd-elhaffiez Hussein

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2016

Purdue University

West Lafayette, Indiana

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

ART	Android runtime
ASLR	address space layout randomization
DVFS	dynamic voltage and frequency scaling
CMS	concurrent mark-sweep
CPI	cycles per instruction
GC	garbage collection
HAL	hardware abstraction layer
JIT	just in time
LMK	low memory killer
OOM	out of memory
SoC	system on chip
VM	virtual machine
WCPT	worst case pause time

ABSTRACT

Hussein, Ahmed Mohamed Abd-elhaffiez PhD, Purdue University, December 2016. Effective Memory Managementfor Mobile Environments. Major Professors: Antony L. Hosking and Mathias Payer.

Smartphones, tablets, and other mobile devices exhibit vastly different constraints compared to *regular* or *classic* computing environments like desktops, laptops, or servers. Mobile devices run dozens of so-called "*apps*" hosted by independent virtual machines (VM). All these VMs run concurrently and each VM deploys *purely local* heuristics to organize resources like memory, performance, and power. Such a design causes conflicts across all layers of the software stack, calling for the evaluation of VMs and the optimization techniques specific for mobile frameworks.

In this dissertation, we study the design of managed runtime systems for mobile platforms. More specifically, we deepen the understanding of interactions between *garbage collection* (GC) and system layers. We develop tools to monitor the memory behavior of Android-based apps and to characterize GC performance, leading to the development of new techniques for memory management that address energy constraints, time performance, and responsiveness.

We implement a GC-aware frequency scaling governor for Android devices. We also explore the tradeoffs of power and performance *in vivo* for a range of realistic GC variants, with established benchmarks and real applications running on Android virtual machines. We control for variation due to dynamic voltage and frequency scaling (DVFS), Just-in-time (JIT) compilation, and across established dimensions of heap memory size and concurrency. Finally, we provision GC as a global service that collects statistics from all running VMs and then makes an informed decision that optimizes across all them (and not just locally), and across all layers of the stack.

Our evaluation illustrates the power of such a central coordination service and garbage collection mechanism in improving memory utilization, throughput, and adaptability to user activities. In fact, our techniques aim at a sweet spot, where total on-chip energy is reduced (20-30%) with minimal impact on throughput and responsiveness (5-10%). The simplicity and efficacy of our approach reaches well beyond the usual optimization techniques.

1 INTRODUCTION

The mobile telecommunications industry has grown rapidly over recent decades, fuelled by immense shift to mobile broadband technologies (i.e., 2G, 4G, LTE, etc.) [54]. Expanding broadband coverage lead to an economic shift, providing unique subscribers all over the world with affordable devices and services centralized around the digital ecosystem. Modern mobile devices are shipped as systems on a chip (SoC), featuring heterogeneous multi-core hardware with on-die hardware peripherals such as WiFi, GPS, and cameras.

To enrich the set of software applications on mobile platforms, each vendor provides a software distribution platform for software applications ("*mobile apps*") written to deliver specific functions for a specific operating system.

Such a rapid spike of innovation in the mobile industry owes its existence in large part to the design interfaces between different sub-modules. In particular, the abstract interfaces serving as abstract layers play a major role in enabling contributors to develop and experiment with the design of a single component without interfering with other layers.

1.1 The Double Edged Sword in Fast Mobile Advances

Modern mobile platforms run complete operating systems (OS). A Mobile device combines features of general purpose computing platforms with those of other popular consumer devices, such as digital camera, GPS navigation, WiFi, and a vast set of sensors. To support faster, more powerful, and richer apps, hardware vendors compete in providing heterogeneous multicore devices shipping with hardware level optimizations and computation offloading to assure power and time efficiency. These devices often exploit vendor-specific libraries and customized drivers that increase the diversity and heterogeneity of the mobile eco-system.

On the software side, mobile platforms typically run apps using a managed run-time system, Virtual Machine (VM), that includes services such as garbage collection and dynamic “just-in-time” (JIT) compilation.

Conserving power on mobile platforms requires optimizations across all levels of the stack, from hardware, operating system, and VM. However, modularity and abstraction create a fragmentation of work between different communities. This ties the scope of the analysis to a single component, without a thorough understanding of how each influences the global system performance.

In the course of delivering product releases, several software methodologies in mobile platforms were adopted from server platforms. The incentive is driven by the fact that porting well-studied and refined components over decades has shown to cut software development time. However, several components were not redefined to better fit the mobile platform. For example, a computation-heavy server application is evaluated by minimizing the environment overhead (i.e., running in single user mode) and by building statistical methods to generate consistent results across different runs [71]. Mobile platforms, on the other hand, introduce additional dimensions of environment that are not principal components of the design—i.e., user interaction, application response to events, and restrictions on available resources.

1.2 Thesis Statement

> Modern mobile platforms run complete operating systems to manage features that combine those of general purpose computing and those of other popular consumer devices, such as digital camera, GPS navigation, WiFi, and a vast set of sensors. These platforms run apps using a managed run-time system, Virtual Machine (VM), to provide abstraction over hardware specifications. Mobile devices must conserve power while still providing the desired responsiveness and throughput. This offers new dimensions for the evaluation of the VM for devices that rely on a managed run-time. These dimensions

> offer not just the usual tradeoffs, but also a sweet spot where the energy cost of a single run-time service (i.e., garbage collection, or just-in-time compiler) can be lowered with minimal impact on throughput and responsiveness. In addition, coordination between heuristics across several non-adjacent software layers and dozens of concurrent VMs empowers informed decisions and optimization across all of the VMs.

We claim that different VM implementations have different energy requirements that do not always correlate with their app's throughput. Varying policies, such as memory allocation or concurrency, can significantly reduce the energy consumed and the worst-case pause time. Moreover, app throughput does not necessarily correlate with power consumption. In addition, integrating VM components with power mechanisms on mobile devices leads to significant and efficient performance and power savings.

While our study explores a wide variety of software techniques, we identify the missing coordination between several software components as an opportunity for optimization of mobile systems along the different evaluation criterion: *(i)* throughput, *(ii)* energy consumption, and *(iii)* responsiveness. We illustrate the power of combining vertical cross-layered heuristics to achieve efficient adaptive decisions.

The dissertation focuses on analyzing the effect of *garbage collection* (GC) behavior on the Android system performance including power, execution time, and response time to user actions.

1.3 Choice of Platform

Perhaps the most important milestone is the selection of the platform and the VM service. Of equal importance is the demonstration of a need for refining the software evaluation and design of the service of choice.

We targeted the Android run-time [50] and its automatic memory manager as infrastructure for our study. Since Android is available as open source, it has allowed us to easily extend and modify the virtual machine. We also considered the wide availability of hardware platforms that runs Android, including both commercial devices and development kits. This has allowed us to apply new techniques *in vivo*.

We have carefully considered the possibility of adopting other VM components (i.e., compiler or class loader), but we concluded that there were several limitations tying those components. For example, Android up to JellyBean run a VM that uses a JIT to produce dynamically optimized code for frequently executed methods. The substantial energy and performance overhead of the optimization reaches its peak in the early phases of launching the app. The cost degrades over time once all possible optimizations are performed.

We explor several extensions to the default GC configuration of Android, including a generational collector, spreading the GC load to different cores, and adjusting the speed of different cores during GC collection. Beyond extending stock Android platorm, we implement a new GC-aware frequency scaling governor for Android devices. In addition, we explore the tradeoffs of power and performance *in vivo* for a range of realistic GC variants, with established benchmarks and real applications running on Android virtual machines. We control for variation due to dynamic voltage and frequency scaling (DVFS), JIT compilation, and across established dimensions of heap memory size and concurrency.

Moreover, we introduce a VM design that allows a central service to observe performance critical parameters of concurrent yet independent VMs and carry out decisions optimized across the whole system instead of just locally. Our prototype then addresses a major resource bottleneck (i.e., memory) by presenting a central GC service that executes GC decisions across all running VMs and optimizes for a global set of heuristics. Here, we present a first set of sensible heuristics although further research and tuning is necessary.

1.4 Work Plan

In order to validate the dissertation claims, we aim to answer the following research questions:

Q1: What are the challenges offered by the new mobile devices?

Q2: What is the methodology to evaluate VM services on mobile devices?

Q3: Why is the GC important?

Q4: What is the impact of disjoint heuristics across dozens of concurrent VMs?

Q1: Challenges

Mobile platforms are sensitive to thermal issues (using only passive heat sinks) and are more aggressive in the power management of sub-systems to save power compared to laptop, desktop, or server systems. Answering these questions is challenging: *(i)* we have migrated the VM profiling spanning four major Android versions from HoneyComb, Ice Cream Sandwich, JellyBean to KitKat; *(ii)* we have carefully examined several parameters and configurations in order to deduce correct conclusions; and *(iii)* we have built a background knowledge of the heuristics and interactions throughout the whole platform stack starting from hardware and up to the user level.

The endeavor proved to be much deeper and challenging than an engineering task. Reevaluating the software components design from the overall system perspective raised research questions, leading to the production of superior virtual machine performance across all running applications.

Q2: Software evaluation

We develop a benchmark suite *Etalon* that combines both Android and stock Java applications. The Java ports aid in differentiating the apps running on Android VM from standard desktop programs. Our goal is to experiment with common Android apps available on Google Play, in addition to a set of ported Java applications from SPECjvm98 [105] and DaCapo [12].

We develop and apply a memory profiling framework for measuring the platform-independent memory behavior of applications running on Android. Validation of the resulting profiling framework is achieved through the Java ports. The profiling framework thus allows for the evaluation of industry-standard Android benchmarks to be done with confidence.

We run a VM profiler as a daemon thread inside the Android VM. Profiling is only enabled to gather execution statistics, but not to capture measurements that are sensitive to timing or scheduling, such as total execution time or OS context switching. The profiling periodically gathers per-mutator statistics, without synchronization to avoid perturbing them. Large-volume traces (such as lifetime statistics) are buffered to avoid imposing I/O costs at every sample point, and are periodically dumped to Flash RAM.

We measure a complete Android development platform in-vivo, avoiding emulation. We use the APQ8074 DragonBoard development kit, based on Qualcomm's Snapdragon S4 SoC using the quad-core 2.3 GHz Krait CPU. We measure current flow at the circuit level between the Krait application microprocessors and the voltage regulator.

Q3: GC significance

The garbage collection, which is common task between all apps, is characterized as memory-bound workload that runs in phases. The garbage collector traverses all the references starting from the "root" to reclaim memory occupied by non-reachable objects. As a result, There is a lot of inter-thread communication, there is a high amount of memory operations, and a lot of processing power needed for automated memory management, making it a perfect opportunity to study any effects.

We discuss alternative GC designs that extend Dalvik's default *mostly concurrent, mark-sweep* collector with *generations*, and *on-the-fly* scanning of thread roots. We perform an extensive evaluation of the different GC configurations using the Java benchmarks and other Android apps. We correlate energy consumption with GC, showing tradeoffs with other performance metrics to understand how GC overhead affects different system layers.

We analyze the GC within the system scope to serve as a guide of how to evaluate the coordination between design decisions across all the layers of the system stacks (software and hardware). Our correlation between GC and system stacks reveal that GC has a significant impact on energy consumption due to implicit scheduling decisions by the OS with respect to CPU cores. We present a modified frequency governor that exploits phase changes of the app running in the VM and adapts the frequency of individual cores during garbage collection.

Q4: Centralized system service for memory management

We identify the lack of coordination between dozens of parallel virtual machines as an opportunity for optimization for mobile systems over: *(i)* memory usage, *(ii)* runtime performance, and *(iii)* power consumption.

We implement a global service that can collect statistics from all running virtual machines and can then make an informed decision that optimizes across all the virtual machines (and not just locally). The new system is 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, and context-aware task-killing mechanisms.

1.5 Results

We show that GC has significant impact on energy consumption, not only from its explicit overhead in CPU and memory cycles, but also because of implicit scheduling decisions by the OS with respect to CPU cores [63]. Our results show that existing DVFS policies should be informed of GC events by the VM to make more informed hotplugging and frequency scaling decisions. Similarly, app developers need a range of GC strategies to choose from, so they can tune for responsiveness, utilization, and power consumption.

We present a full Android system, evaluated in vivo. Our techniques reduce total on-chip energy (up to 30%) for throughput tradeoff of 10%. [62; 64] We also present a new benchmark suite that builds the foundation for a robust evaluation methodology for mobile platforms. Our work quantifies the direct impact of GC on mobile systems, enumerates the main factors and layers of this impact, and offers a guide for analysis of memory behavior in understanding and tuning system performance.

1.6 Thesis Organization

Chapter 2 describes the architecture and the internals of the Android platform along with the related work. Chapter 3 lists in details the challenges imposed by the mobile frameworks and our recommendations in building a robust methodology to evaluate the mobile systems followed by a brief description of the *Etalon* benchmark.

Chapters 4 and 5 describe the implementation of our GC extensions, the integration with frequency governor, and the analysis performed to reach our conclusions. Chapter 6 presents a new Android virtual machine design that allows the garbage collection to be performed as a centralized service for all the parallel virtual machines. Finally, Chapters 7 and 8 present future directions and conclusions.

2 BACKGROUND AND RELATED WORK

Modern mobile devices combine features of general purpose computing platforms with those of other popular consumer devices, such as digital camera, GPS navigation, wifi, and a vast set of sensors. Users install third party applications (apps) on a managed run-time system through digital distribution platforms such as the Apple store, the windows phone store, or Google play. In addition, mobile devices feature a wide range of self-adaptive systems that continuously adapt run-time parameters according to environmental inputs and to achieve local goals (e.g., reducing power by enabling/disabling cores).

In this chapter we will explain the Android architecture, and give a high level feedback on the main Android components.

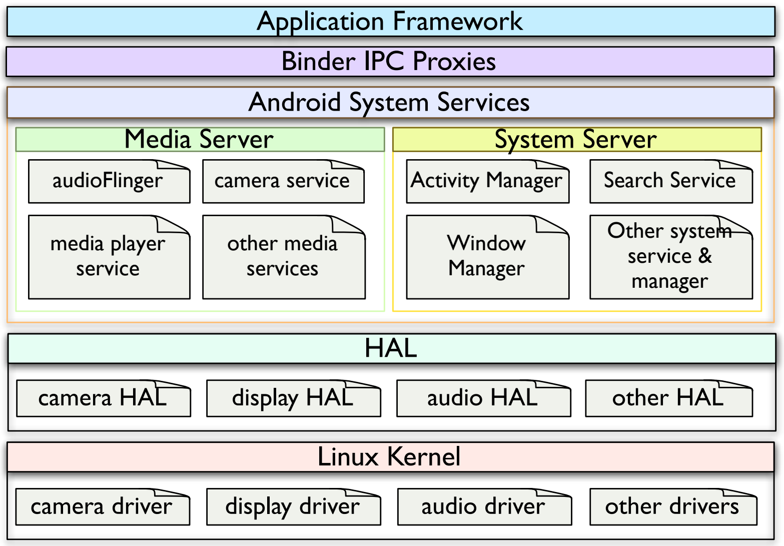


Figure 2.1.: Android system architecture

Table 2.1.: Android versions

Code name	Date	API	Release	Kernel
Honeycomb	02/2011	11-13	3.0, 3.1 and 3.2-3.2.6	2.6.36
Ice Cream Sandwich	10/2011	14-15	4.0-4.0.2, and 4.0.3-4.04	3.0.1 - 3.0.31
JellyBean	07/2012	16-18	4.1-4.1.1, 4.2-4.2.2 and 4.3	3.0.31 and 3.4
KitKat	10/2013	19-20	4.4-4.4.4	3.4
Lollipop	11/2014	21-22	5.0-5.0.2, and 5.1-5.1.1	3.4(armv7); 3.10(arm64)

2.1 The Android Software Stack

The Android platform is designed in the form of a software stack comprising various layers running on top of each other in a way that the lower-level layers are providing services to upper-level components. Figure 2.1 shows the applications, services, and the operating system in android architecture. Both native and third party applications are located at the top of the Android software stack. Application framework provides services to apps. The most important parts of the framework are as follows: *(i)* activity manager: manages the app life cycle, *(ii)* content provider: provides encapsulation of data that needs to be shared between applications, and *(iii)* notification manager: handles events such as arriving messages and alerts.

Over the course of our study Android has already undergone several versions. Table 2.1 shows the Android history since we started our research while mapping the official release version to that of the API and the kernel.

Honeycomb introduced multi-core support into Android[1].

Ice Cream Sandwich introduced `ontrimMemory` callback to reduce the memory pressure.

JellyBean introduced `SELinux` to Android.

KitKat introduced several APIs to read the available RAM, and the low RAM components. It also added `procstat` and memory swapping. Kitkat was the first release that allows optional switch between two different Android runtimes (Dalvik and Android Runtime).

[1]Android Honeycomb was not made available.

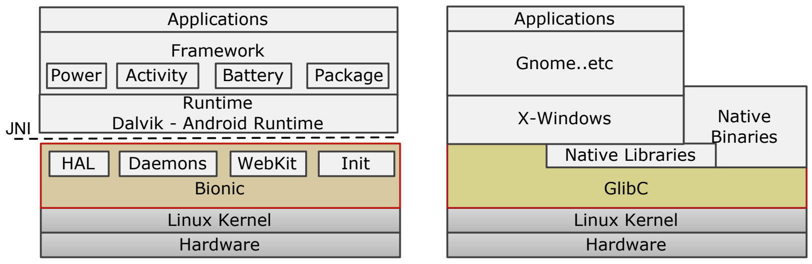


Figure 2.2.: Android architecture (left) and mainstream Linux (right)

Lollipop replaced the just intime compiling virtual machine Dalvik with Ahead of Time runtime (Android Runtime). Beside supporting 64 bits, the new version introduced several enhancements to the garbage collector implementation.

2.2 Kernel Layer

Android is built on Linux, providing a software stack of kernel, drivers, GUI platform and a set of frameworks. Android introduces a Java runtime to support the framework allowing the developers to write their mobile applications in Java. Android modifications to mainstream Linux are referred to *Androidisms*. Among those modifications: ARM platform, drivers, interprocess communication, and memory. At the user level, Android introduces runtime componen and hardware abstraction. Figure 2.2 shows the main differences between Linux and Android [78; 116].

2.2.1 Power management

Mobile platforms employ aggressive power management of sub-systems to improve battery life and to control thermal conditions (mobile platforms only have passive heat sinks and components like the CPU cannot be run at full power continuously). The OS relies on *dynamic voltage and frequency scaling* (DVFS) in software controlled by a global governor [18]. The governor collects runtime statistics (e.g., system load and core temperature) and then applies complex heuristics to meet optimization criteria [23; 65; 85]. This feedback relation between the OS modules and the workload imposes challenges to the system design due to the dependency between the device performance and the non-deterministic response of these software modules.

Most policies adopt an approach known as *race-to-idle*: reacting to a growing workload by enabling more cores and/or increasing their speed; when the workload decreases, the cores are disabled and/or the frequencies are lowered.

Some mobile multi-cores provide *asynchronous multiprocessing* that allows to vary operating frequencies per core as a function of the workload. Thus, coarsely measuring high-level throughput and execution time yields different results, especially in multithreaded apps, as the results depend on the core frequency on which the evaluated task was scheduled.

The OS kernel reacts to the app workload by adjusting the core frequency and the voltage (DVFS) to meet the performance requirements at ideal power and thermal levels. The policy by which the scaling decisions are taken is defined by the governor configuration. In addition to the governor module, vendors usually ship their own proprietary software to increase the efficiency of core *hotplugging* and to manage other peripherals such as the GPU. For example, the `mpdecision` is a proprietary binary component shipped with the Snapdragon processors.

The power mechanisms can be represented as a discrete model. For simplicity, we consider that the time stamps of all events are multiples of a predefined time interval, `sampling_rate` (parameter of the `ondemand` governor). The values between the sampling events are found by transforming the discrete model to the continuous model.

Figure 2.3 illustrates the discrete model in which the kernel tasks are run each sample period of time (`sampling_rate`) to inquire the per-core workload w to decide on the operation frequency f, followed by a zero-order hold (ZOH) to convert the discrete time values into continuous time values. The `mpdecision` reads the data generated by the thermal sensors h to integrate the thermal conditions with the workload. The impact of the `mpdecision` on the governor policy is captured by signal p. We use the energy signal e as an input to the governor superblock G for the following two reasons: *(i)* reduce the non-determinism

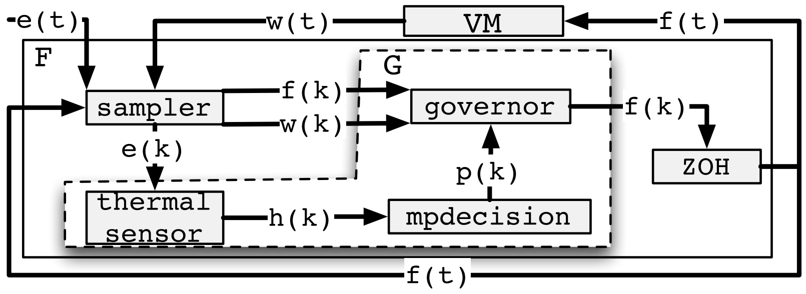


Figure 2.3.: Interaction between VM and power mechanisms on Android

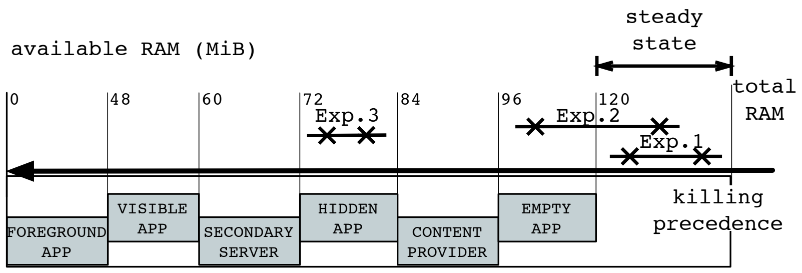


Figure 2.4.: Low memory killer configurations

by omitting the thermal environment (room temperature is a factor influencing the device thermal efficiency), and *(ii)* reduce the domain input, because the device temperature is an implicit function of the amount of energy consumed by the app.

2.2.2 Memory management

Modern mobile platforms run dozens of concurrent VMs. Those concurrent VMs share a set of constrained and over-committed resources such as memory. Mobile system rely mostly on caching the processes in order to reduce the latency of loading the applications [47]; hence mobile systems increase their own responsiveness.

2.2.2.1 Low memory killer

When an Android app is closed, the process stays in 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 low-memory-killer (LMK) which kills processes when the available free RAM reaches a low threshold. The LMK allows specifying a set of "*out of memory thresholds (OOM)*" to decide on which and how many processes to kill.

Figure 2.4 shows an example of low-memory regions in MiB. When free memory (RAM) in a system is less than 120 MiB, the LMK starts killing "empty apps" in the background. The app in `exp.1` runs in a stable steady state while the app in `exp.2` causes the free memory to drop below 120 MiB, triggering the LMK which will start killing background tasks. Finally, the app in `exp.3` executes in a stressed environment with low memory, causing side effects from potentially both the LMK and the OOM.

2.2.2.2 Memory usage characterization

There are main four categories to classify physical memory usages: process VMs, OS file caches, device I/O buffer, and kernel memory [76].

Process VM pages. VM pages constitute the largest portion of memory usage [76]. The VM pages specifies the maximum amount of memory available to allocate the heap and the stacks of a single VM. The memory driver *lazily* brings the page into physical memory in a discontiguous address range.

File pages. The OS maintains caches of data buffers and data to reduce the overhead of accessing the main memory. Page caches are managed at the page granularity [76].

Kernel memory. The kernel uses *slabs* to allocate kernel objects—i.e., page tables and stack. Slabs are contiguous physical memory pages. The granularity of management varies based on the allocated object. For example, small objects such as semaphores are allocated as a small virtual memory [76].

2.3 Android Virtual Machine

Android is designed for rich user interaction via a touch screen on mobile devices. The Android software layers include runtime component (Dalvik VM or Android Runtime) and core java libraries. The *application framework* provides the required interface to build Android apps, which belong to the top *applications* layer. The main components of Dalvik include the garbage collector, the just-in-time (JIT) compiler, the debugger, and the main thread.

2.3.1 App isolation

Android applications run in their own VM instances within separate os processes. The access of each instance to the platform API and other system resources is defined by a manifest configuration. Since each VM is a process, switching between apps requires swapping the memory addresses and page tables, and flushing processor caches. To guarantee application responsiveness, the scheduler assigns higher priorities to user interface threads, while background threads are given lower priorities. In addition, android moves all low-priority background threads to *cgroups*, where they are limited to a small percentage of CPU time. This ensures that the foreground application does not starve, regardless of the number of running background tasks.

When an app is started, it skips the initialization steps by inheriting the preloaded classes and resources created by the "*Zygote*" template process. Zygote reduces the global memory usage by keeping copy-on-write space of common resources. Once the process is forked, the VM heap is managed by a per-app garbage collector.

Up to Android 4.4, Android ran a register based Java VM "*Dalvik VM*", optimized for memory constrained devices. Dalvik runs a register-based byte code format called *Dalvik executable (DEX)* [50; 51] through a JIT. The *Android runtime (ART)* replaced the Dalvik JIT compiler through ahead-of-time compilation. During installation, ART translates dex files into native code.

Android binder is a simple subset of an *open binder*, [2]which allows binding functions or data from one execution process to another. In Android, the binder is used for every flow across processes.

2.3.2 Garbage collection in Dalvik

The Dalvik garbage collector runs in its own background daemon, with the application level Java threads. The default Dalvik collector is *mostly concurrent* in that it periodically stops all the Java (*mutator*) threads, but otherwise runs concurrently in the background and synchronizes only occasionally. It operates as a *marksweep* collector, tracing references from roots, which include both thread stacks and other global variables, marking objects reachable via those references, and recursively via the references stored in reachable objects. when all the reachable objects have been reached it sweeps the heap to free up unmarked objects.

The default Dalvik collector suspends all the mutator threads as a batch at the beginning of each collection cycle, scans their stacks for heap roots, and then restarts them all before continuing to mark reachable objects concurrently. Concurrent marking is supported by a *write barrier* that records *dirty* objects that have been modified by any mutator thread during the mark phase. When concurrent marking is finished, the collector once more suspends all the mutator threads as a batch, marks any remaining unmarked objects that are reachable from the dirty objects, and then restarts the mutator threads. It can then safely (and concurrently) sweep up and free the remaining unmarked objects as garbage.

The Dalvik collector uses simple heuristics to balance the tension between frequency of garbage collection and heap size Brecht et al. [16]. The primary parameter controlling heap size and garbage collection is the *target heap utilization* (`targetutil`) ratio, used to resize the heap after each GC cycle. The threshold *softLimit* is set such that the ratio of the volume of live data (*live*) to the *softLimit* is equal to the target utilization. Thus, the bigger the target utilization, the tighter the heap allocated to the app. The available space (*room*) is

[2]openbinder is a system for IPC for BEOS, windows, and Palmos cobalt: `http://www.angryredplanet.com/~hackbod/openbinder/`

constrained to the range 2-8 MiB. The threshold *concurrentstartbytes* (*CSB*) is set at some delta ($\delta = 128kib$) below the *softLimit*. The relationship among these parameters, at time t, is given by the following equation:

$$\begin{aligned}\text{room}(t) &= (1 - \text{targetutil}) \times \text{live}(t) \\ \text{softlimit}(t) &= \text{live}(t) + \min(\max(\text{room}(t),\ 2mib),\ 8mib) \\ \text{csb} &= \text{softlimit}(t) - \delta\end{aligned} \tag{2.1}$$

Memory events in Dalvik GC can be grouped into *(i)* GC triggers, and *(ii)* heap resizing decisions. GC triggers include:

GC-CONC after any successful allocation, if the allocation exceeds the *CSB* threshold, then the mutator signals the GC daemon to start a new *background* GC cycle, if it is not already active.

GC-ALLOC when allocation would exceed the *softLimit* threshold, or if allocation otherwise fails, then the mutator directly performs a *foreground* GC cycle, so long as the gc daemon is not already active in which case it waits for it to finish. the mutator boosts its priority while performing the foreground GC cycle. in the case of a failing allocation the mutator retries the allocation after the GC cycle ends.

GC-EXPLICIT the mutator directly performs a foreground collection cycle in response to an explicit call to `System.gc()`, so long as the GC daemon is not already active. the mutator does not boost its priority to perform the explicit GC.

In the absence of mutator signals, the GC daemon does not remain idle forever. the time it waits for a mutator signal is limited to five seconds, after which it performs a spontaneous concurrent collection cycle; this also *trims* excess virtual pages from the heap, returning them to the operating system.

2.3.3 Garbage collection in Android Runtime

The *Android runtime (ART)* replaced the Dalvik JIT compiler through ahead-of-time compilation introducing several enhancements to the Android GC [51]: *(i)* parallel processing during marking of mutator roots, *(ii)* a separate heap for large objects, and *(iii)* a pseudo-generational sticky collector to deal with short-lived objects.

In recent releases, ART introduced compacting GC to reduce memory usage and fragmentation.

2.3.4 Android compiler

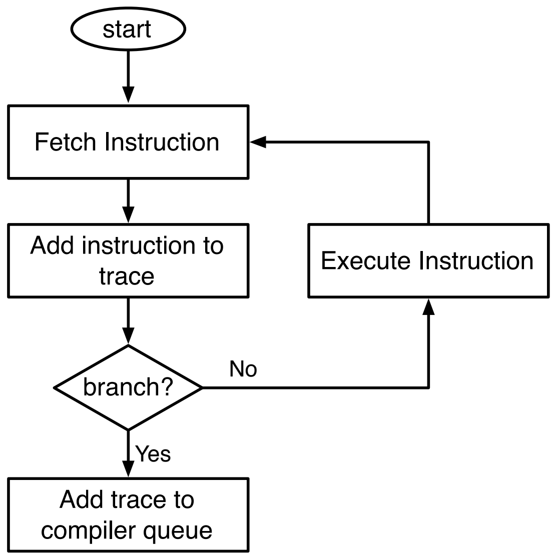
Starting from Android 2.2, Android Dalvik introduced Just-in-time compiler to increase the performance of the Android platform. Dalvik JIT is *trace* based that records a linear sequence of frequently executed operations and translated them to faster form of native machine code.

Dalvik defines an Low Level IR (LIR) that closely resembles the target machine dubbed "ArmLIR" (for ARM). The Dalvik instructions are pulled from the trace to perform the first stage of code generation that is to convert Machine Level IR (MIR) to LIR. The Machine Level IR (MIR) stream is build by wrapping an MIR structure around each Dalvik instruction. Basic Blocks are created, linked together, and added to the *Compilation Unit*. At this level, the generated code is highly portable amongst different ARM and x86 architectures.

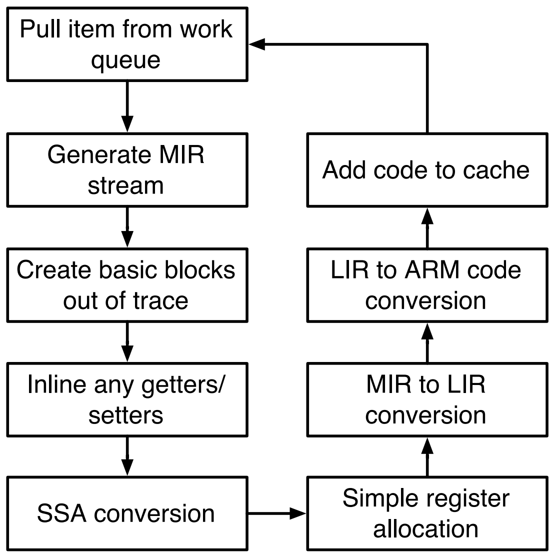
The Assembler generates a block of ARM instructions for each LIR instruction. This second phase of code generation outputs the ARM instructions from LIR. Figure 2.5 shows the flow of the interpreter and the compiler JIT in the Dalvik VM.

Android Runtime Compiler

Android Lollipop 5.0 replaced the JIT Dalvik compiler with an Ahead-of-Time compiler that compiles the Dalvik code into native code during the installation. Even though Dalvik JIT is replaced, the applications are still packaged with Dalvik bytecode. Android



(a) Interpreter flow


Pull item from work queue
Generate MIR stream
Create basic blocks out of trace
Inline any getters/ setters
SSA conversion
Simple register allocation
MIR to LIR conversion
LIR to ARM code conversion
Add code to cache

(b) Compiler thread flow

Figure 2.5.: Dalvik tracing Just-in-time compiler

Runtime (ART) compiles the `classes.dex` to native code leaving the optimization phase to `dexopt`.

2.4 GC Evaluations

Several studies have addressed GC requirements when deployed in restricted environments. Chen et al. [25, 26] tune the collector to enable shutting down memory banks that hold only garbage objects. Griffin et al. [53] implement a hybrid mark-sweep/reference-counting collector to reduce power consumption. The three major sources of energy leakage of the GC are identified as: *(i)* instructions executed by the CPU core, *(ii)* cache access (data and instruction), and *(iii)* memory access due to misses.

Sartor and Eeckhout [96] explored tradeoffs with separating JVM threads (e.g., garbage collector) and its effect on performance for a multi-socket server environment (8-core Intel Nehalem). For managed run-time systems on general purpose platforms, there is much recent interest in fine-grained power and to understand the energy needs of VM components [20; 110]. Occasionally, GC has been evaluated as an asymmetric activity that can be isolated on a separate core [20; 96]. However, the presented methodology relies on dedicated hardware which is not practical for modern mobile devices.

For mobile devices, several power studies involve software and hardware layers leading to fine-grained tools to profile the system level to detect power bugs and to determine the application blocks that leak large amounts of energy [89; 90]. Hao et al. [58] presented an approach for power estimation based on off-line program analysis. The responsiveness of embedded systems was throughly 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 [113]. However, the relation between the WCET analysis and power consumption is less understood, because of the challenge in assuming a direct correlation between execution bounds that involve different components such as compiler, scheduler, and hardware specifications [2; 7; 113].

In this dissertation, we demonstrate that it is necessary to define GC requirements as a function of system mechanisms such as the governor and scheduling policies. While Schwarzer et al. [97] suggest methods to estimate performance requirements of software tasks using simulation, our approach is based on the observation made by Sherwood et al. [99] that a program's execution changes over time in phases. Our work characterizes the GC workload, which is common between all apps, as having a lower CPI compared to the average mutator workload.

Taking advantage of DVFS [65], we cap the speed of the collector thread in order to reduce the power consumption within each collection cycle. Our study is characterized by its unique contribution in evaluating the GC design and configurations as an integrated system component on mobile devices, in the spirit of Kambadur and Kim [73]. We show that the energy bill for GC can be reduced by simple integration across system layers (i.e., managed run-time system and governor).

Our results differ from the work of Hao et al. [58] in fitting the run-time performance within the whole system stack (i.e., hardware, kernel, and power management). The results generated in this dissertation reflect real executions involving synchronization overhead, induced by spin-locks and context-switching as noted by others [46; 88].

The efforts to efficiently manage the independent Java VM heaps turned to be fruitful to update the shared memory resources between the running virtual machines [19; 84]. These studies differ from our study in the following: *(i)* our system focuses on restricted mobile platform hosting dozens of parallel virtual machines, and *(ii)* the GCService performs GC phases on all virtual machines (not just locally).

Our work is the first to present a runtime service on mobile platform that can manage the heap of all the running VMs while keeping each instance in its own process which is different from other approaches like Multi-tasking VM [31; 117].

2.5 Mobile Application Behavior

We considered adopting existing evaluation methodologies in our platform evaluation. First, we surveyed existing set of characteristics that label any application available on the app stores. Then, we surveyed existing benchmark suites used in industry and academia to evaluate the system. We concluded that none of the existing techniques serves our needs as we aimed at evaluating the whole system by analyzing the interaction between the system components. In this section we summarize the three major categories of studies interested in analyzing the application behavior.

2.5.1 Security

One of the most active research areas in mobile platforms taming all possible vulnerabilities. Security oriented studies focus on combining static and dynamic analyses in order to detect control and data flows in mobile applications. This is achieved by evaluating the app call graph and I/O operations behavior from the API perspective [15; 21; 29; 32; 52; 59; 69; 118; 119]. Unauthorized access, or suspicious functionality is reported as a possible threat to the device security.

This category of studies do not consider performance or implicit impact and interaction between non-adjacent layers. Therefore, it did not fit our needs in studying the managed runtime design choices and their impact on the system performance.

2.5.2 Usage patterns and context-aware approaches

With millions of applications on the software stores, it becomes challenging for users to select the right set of applications to install. This urged the interest in analyzing the usage pattern of applications among a group of users in order to provide better choice for installations [6; 37; 43; 60; 74; 75; 98; 101; 104]. The analysis does not just include the daily usage of each individual to his device, but it may extend to monitor inter-communication and dependency between different applications [80; 107].

More recent studies extended the usage pattern methodology to optimize the mobile framework [115; 120]. This is achieved by offering app usage prediction model constructed through logging the following events: *(i)* observed app preferences; *(ii)* user triggered events and readings as observed through environmental sensor-based contextual signals; and *(iii)* the common patterns of app behavior among different group of users

Apparently, this inspired other researchers to take one step further applying the prediction module in analyzing and optimizing the energy profile [82; 86; 100; 108]. The methodology is an event based model that captures the relevant power consumption using monitoring tools.

While we found many of studies belonging to that criteria are interesting, they are more suitable for package and device management. For example, app prediction can be used in better device management to reduce launch overhead. However, they do not provide metrics, or answers to our research questions.

2.5.3 Managed runtime and microarchitecture evaluation

The third category in our survey is composed of studies analyzing the performance of a single app. The vast majority of evaluation are based on hardware counters as evaluation metrics [39; 55; 56; 57; 61; 106]. Given the complexity of the system stacks on mobile devices, we found that it is nearly impossible to attribute a certain profile behavior to individual hardware counters. Eventually, we concluded that hardware metrics are strongly bound to the VM implementation.

The most relevant study that focuses on mobile device responsiveness consists of analyzing critical path of the application code [94]. Blocking tasks within this path are labeled to degrade the system responsiveness. Our approach in analyzing the impact of managed runtime service on responsiveness focuses on measuring the time spent in by the runtime system within the critical path.

3 RIGOROUS EXPERIMENTATION ON MOBILE PLATFORMS

Evaluating performance of modern mobile platforms, comprising complex hardware and software stacks, is notoriously difficult. These complex layers include feedback mechanisms that adapt to the power and performance profile of the application, as well as the environment. Thus, obtaining deterministic and repeatable experimental results requires care. It is not sufficient to repeat measurements to eliminate noise, because the platform itself exhibits widely variable behavior. Nor is it sufficient to control and measure the application layer alone, without also controlling lower layers of the software and hardware stack. These cause non-deterministic behaviors dependent on the environment, configuration of hardware (e.g., power status, temperature, peripherals), and software (e.g., services, scheduler, power-governor, networking).

In this chapter, we consider metrics for evaluating Android apps running on a real device (the Snapdragon multi-core platform), and steps needed to obtain controlled results for those metrics, like limiting interference from non-salient layers, and controlling variability due to adaptive components that perturb the target metric. Understanding results requires correlating metrics with underlying platform (e.g., hardware, OS, run-time, and application) events. The metrics we consider include power, performance (throughput and utilization), and responsiveness, within a study of the memory management behavior of the Android virtual machine. For each metric we describe the techniques and controls used to obtain reliable and meaningful results. We also characterize the variability that ensues when controls are not carefully applied.

3.1 Motivation

To support faster, more powerful, and richer apps, hardware vendors compete in providing heterogeneous multi-core devices shipping with hardware level optimizations and computation offloading to assure power and time efficiency. These devices often exploit vendor-specific libraries and customized drivers that increase the diversity and heterogeneity of the mobile eco-system.

Benchmarking on general purpose computing devices is a well-studied problem and has been refined over decades. For example, a computation-heavy server application is evaluated by minimizing the environment overhead (i.e., running in single user mode) and building statistical methods to generate consistent results across different runs [71]. Compared to this well-studied problem, mobile platforms introduce additional dimensions of environment such as user interaction, application response to events, and restrictions on available resources. In addition, mobile devices feature a wide range of self-adaptive systems that continuously adapt run-time parameters according to environmental inputs and to achieve local goals (e.g., reducing power by enabling/disabling cores).

Metrics used to experiment on mobile platforms must prioritize factors that affect user interaction. For example, server benchmarks have thoroughly studied throughput, execution time, and response time. Recently, with the emergence of highly parallelized hardware, more attention has been paid to other metrics such as power efficiency and scalability. For a mobile device, scalability may be of secondary concern so long as the user is satisfied with the response time and the device reliability (e.g., battery lifetime). Thus, it is necessary to qualify the overall system along three dimensions: *(i)* responsiveness to user actions; *(ii)* power efficiency throughout the system stack; and *(iii)* performance over time, space, and thermal profile.

We address the challenges one faces in performing consistent and meaningful experimental evaluation of mobile devices across these dimensions. We discuss how to reduce non-determinism across these dimensions and how they correlate with each other.
Our contributions are as follows: *(i)* we explain the challenges of controlled experimental evaluation on mobile platforms; *(ii)* we survey metrics affecting user experience; *(iii)* we develop a sound methodology for reliable measurement of the suggested metrics; *(iv)* we characterize the resulting improvements in accuracy compared to naïve experimentation; and *(v)* we present a benchmark suite that captures the mobile applications behaviors.

3.2 Experimental Challenges on Mobile Platforms

Mobile devices are event-based systems. Users interact with mobile systems through a set of touch events and gestures. These events propagate across the complete hardware and software stack. The device's response is nondeterministic, considering side effects of the mechanisms, shaping the hardware and software behavior. For example, the system assures that the highest priority task, the foreground process, has enough resources to proceed within reasonable response times. This may include killing background processes when the device runs out of physical memory.

The unpredictable nature of such asynchronous actions taken by the adaptive layers of the system makes teasing out the salient performance impact of candidate implementation alternatives particularly difficult. Such conditions demand new techniques for controlled experiments to reduce and filter noise efficiently, and to devise better policies to adapt to usage profiles.

We develop techniques to tame variability and non-determinisim across the system layers to demonstrate the relationship between non-adjacent layers in the evaluation of garbage collection (GC) for the Android VM [50].

3.2.1 System complexity

3.2.1.1 Architecture

Shipped as systems on a chip (SoC), mobile platforms feature heterogeneous multicore hardware with on-die hardware peripherals such as WiFi and GPS, increasing the level of complexity in functional and architectural aspects. Evaluating performance on such devices must consider non-deterministic behavior caused by architecture functionality such as cache tuning and branch prediction which yield to different results across multiple runs. The complexity of the functionality in dealing with a SoC comes from the fact that many SoC features are managed by proprietary software components so as to increase productivity and code reusability of systems. Software and hardware are in this way closely intertwined.

Mobile multi-cores can run asymmetrically at varying frequencies per core depending on their workload. As a result, coarsely measuring high-level throughput and execution time will yield different results depending on the core frequency on which the evaluated task was scheduled. Obtaining valid results is even more complicated when considering that varying core frequencies also have an impact on concurrent apps. Dynamic voltage and frequency scaling (DVFS) adjusts voltage and core frequencies to meet optimization criteria that are not always self-evident. These policies are often managed through an OS kernel module or *governor* [18].

The OS can selectively disable separate components to reduce leakage power. Most policies adopt an approach known as *race-to-idle*: reacting to a growing workload by enabling more cores and/or increasing their speed; when the workload decreases they disable cores and/or lower frequencies. Yet, even only adapting a single parameter may result in nondeterminism and disturb measurements.

Moreover, some DVFS policies are implemented as proprietary, vendor-specific binaries. Such inscrutable adaptive components make the challenge of understanding performance behavior even harder.

In addition, device services — such as for location, phone, and networking — run as background services that compete for resources with apps. These device services cannot easily be disabled, making it impossible to use “single-user” mode for experiments on mobile platforms. Interference from these background services can also lead to experimental variability.

3.2.1.2 VM configuration

Mobile platforms typically run apps using a managed run-time system that includes services such as garbage collection and dynamic “just-in-time” (JIT) compilation. These services also have side effects on power efficiency by inserting idle periods and changing the way memory is accessed by the app.

The policy by which the VM manages the heap has an impact on the overall app performance. Defining the heap size and the frequency of the GC introduces a tension between memory utilizations and the responsiveness of the app. Frequent GC pauses increase pause times of mutators, reducing the app efficiency. Another dimension is introduced by performing concurrent GCs which cannot be analyzed by static techniques. The efficiency of the GC is determined by the interaction of several layers: allocation rate, trade-off between scheduling overhead and the live set size, memory bandwidth and the core speed.

3.2.1.3 Application level

Mobile apps are characterized by their event based behavior, adapting to user actions. Developers use available cores through the API concurrent libraries provided by the managed runtime system. The efficiency of the execution relies on the scalability of the byte code and the VM's success to execute the code efficiently on the underlying chip. There are many sources of non-determinism at the application level, e.g., interference through shared data for concurrent computational tasks, tasks racing to access peripheral devices, or interference from scheduled background tasks.

3.2.1.4 Tools

The OS kernel allows users to access hardware performance counters on the CPU. Gathering hardware counters may help to build analytical models and to correlate between the efficiency and the hardware events. However, hardware counters are limited on mobile devices [111]. For example, L2 memory counters are not available on some ARM processors and commercial devices often disable access to the hardware registers. This limitation prevents importing existing analytical models relying entirely on hardware performance counters.

3.2.2 Characterizing suitable benchmarks and workloads

Characterizing programs is key to system development. Java programs can be characterized by set of continuous metrics including: memory use, polymorphism, and the level of concurrency [12; 40]. With the arrival of highly parallel hardware architectures concurrency and scalability became one of the most important metrics to provide insight on the concurrency pattern of the individual programs and the scalability of the VM [72].

While many of the standard metrics are still relevant for mobile development, the weight of the metrics may be different. For example, while energy consumption and security are crucial for mobile platforms, scalability on mobile devices is not as critical as for server applications. Mobile devices are still a young platform and standard mobile benchmarks have yet to emerge. Hence, there is a lack of accurate analysis on the interfaces and system calls widely used by the developers. Such questions are important for the system designers to prioritize their optimizations and their evaluations.

It is important to consider to consider that shared libraries are main factors in app behaviors. Dong et al. [39] found that 72% of the instruction fetches are from native-code shared libraries. Moreover, 62% of Android apps use *Android Support Library* [5]. Hence, it is necessary to provide a variety set of applications that use several shared native libraries.

Previous studies relied on micro-architecture metrics (i.e., hardware performance counters) to characterize a suitable mobile benchmark [56; 106]. However, we show that architectural metrics is challenging given the depth of the software stack. For example, each app exhibits different characteristics under different Android releases. The most common Performance metrics are the following: *(i)* Performance: Cycles, instructions, cycles per instruction (CPI), and stalled cycle per component; *(ii)* Branch misprediction rate: The branch predictor plays an important role in ensuring efficient out-of-order execution and exploiting instruction level parallelism; *(iii)* Cache: L1, L2 and TLB instruction/data cache; and *(iv)* Core Utilization.

3.2.2.1 Using existing apps as benchmarks

Mobile apps differ from server applications by an adaptive behavior. Measuring their performance requires a clear definition of executed functionality that is missing in the majority of commercial apps. Instead, they function as background services that adapt to user requests. Hence, comparing time execution is not applicable on the vast majority of mobile applications.

Given that applications are provided by third parties, they represent a black-box to the system designer and deciding on their suitability for evaluating the new system requires a detailed analysis of their programming pattern. Some apps are designed for benchmarking but we note that they provide a scoring formula that may not be relevant or representative for mobile systems (or the current aspect under test). A majority of these apps runs with phase behavior with each phase focusing on a specific system feature, limiting generality. Furthermore, during the execution of well-known commercial benchmarks, the vendor-specific daemon enables all cores and locks them to their maximum speed, shadowing the effect of the system changes.

Mobile apps have inconsistent workloads due to the variety of downloaded data through the network which makes their memory usages vary across different runs. Interestingly, the impact of the workload is not limited to memory profiling. Pathak et al. [90] show that free mobile apps using third-party services to display advertising consume considerably more battery. For example, an app spends 75% of its total power consumption on advertisements. The latter behavior suggests that many apps are not actually engineered with power efficiency considerations. In other contexts, app developers may explicitly force components to stay awake introducing more drains to battery since the individual app is not aware of the global system utilization [89].

3.3 The Etalon Benchmark Suite

In this section, we introduce *Etalon*, a benchmark suite designed to make it easier to evaluate mobile applications on real devices (the Snapdragon multi-core platform) considering metrics that are relevant to user interaction and simplify correlation between underlying platform (e.g., hardware, OS, run-time, and application) events. Etalon contains popular Android applications. We characterize the features compared to previous methodologies. Our results show that Etalon exhibits various behaviors, and deterministic replays. The platform is useful for both system and application developers.

3.3.1 Specifications

Popular mobile applications are commercial, which complicates instrumentation because their source codes are unavailable. Also, customizing a benchmark to stress a specific system module is not feasible. While, we include commercial applications in the benchmark suite, we considered applications available as open source. In that way, the benchmark suite provides flexibility to fit any evaluation target.

3.3.1.1 Memory behavior

Objects allocation rate on apps determines the *garbage collection* overhead on the device. The latter impacts the energy consumption, app throughput, and system responsiveness to user events [62; 63]. Allocation rate, object demographics and reference distances are strongly bound to the app workload which make them a good fit to characterize the app. Etalon allows configuring the memory workload in order to cover more benchmarking needs.

3.3.1.2 Responsiveness

On mobile devices responsiveness is a primary virtue in providing usable user interfaces. A simple user request triggers multiple asynchronous calls, with complex synchronization between threads. Identifying performance bottlenecks in such code requires correctly tracking causality across asynchronous boundaries.

Execution time of the tasks performed across the UI stacks has to meet a target which is less than the user perception. To evaluate a software module within the platform layers, it is important to study the distributions of pause times to statistically estimate the efficiency and utilization. Hence, a benchmark used to evaluate responsiveness needs to exploit sensitivity to *hot path*. A hot path is the bottleneck path in a user transaction, such that changing the length of any part of the critical path will change the user-perceived latency.

Therefore, it is important to have a set of applications that exhibits different patterns to evaluate the responsiveness under stress. Etalon allows different configurations in which the user can force the app to execute extended tasks blocking the system from responding to user events.

The suite benchmark provides consistent steps to capture user events to allow precise record and replay. The most important events include:

Press-and-Release represents a simple press;

Press-and-Hold used to open menus;

Swipe switching between screens;

Zoom-and-pinch multitouch input commonly used in maps and photos applications.

In all applications, we used *Monkeyrunner* tool to automate user inputs [3]. On the other hand, we avoid events that are provided through systeem services (e.g., camera, GPS and WiFi). Instead, all applications (including browsers) must access local files. Our intuition is that delays caused by the environment does not provide helpful analysis for system designers.

3.3.1.3 Execution time

Mobile apps differ from server applications by an adaptive behavior. Measuring their performance requires a clear definition of executed functionality that is missing in the majority of commercial apps. Instead, they function as background services that adapt to user requests. Hence, comparing time execution is not applicable on the vast majority of mobile applications. Etalon offers a subset of applications or subcomponents that can be evaluated as end-to-end points.

3.3.1.4 Microarchitecture characterization

Our aim is to offer a set of applications that exhibit different behaviors on michroarchitectural level. Therefore, we include a large set of apps with different functionality making the benchmark suite a good fit for high level evaluation.

3.3.2 Origins of the source code

3.3.2.1 Porting Java benchmark to mobile platform

We established a subset of applications from Java benchmarks that are already well understood, at least in the desktop and server space. These apps are helpful for system developers interested in evaluating VM componenets (e.g., garbage collection or compiler optimizations). They also may exhibit behaviors (e.g., scalability and concurrency) that existing Android apps do not (yet) display.

Android supports many Java packages while some libraries are unavailable. To port Java benchmarks unsupported libraries must be replaced by equivalent or comparable mobile API calls. Clearly, the new port will have different behavior since the supporting libraries are different. Although, ported Java benchmarks may offer a clearer indication of performance compared to using commercial apps, we still note that more standardized benchmarks are needed on mobile platforms.

We have faithfully ported all eight SPECjvm98 [105] applications. Due to API incompatibilities between Android and Java (Android apps are written in Java but use different standard libraries) we have restricted the port of DaCapo 9.12 [12; 13] benchmarks to lusearch, xalan and pmd benchmarks[1].

[1]we maintained luindex up until the relase of Android KitKat.

While there are complete studies on Java benchmarks suites [12; 35; 72], porting them to the mobile platform requires extending the characteristics and prioritizing them to consider user interaction, responsiveness and power efficiency. Our experiments show that our Android ports exhibit similar behaviors to standard Android applications. For example, Xalan-Java is an `XSLT` processor for transforming XML documents into HTML, text, or other XML document types which is not supported by default on Android platforms. This makes xalan a good fit for the benchmark suite since its workload is identical to typical Android applications.

3.3.2.2 Considering real world mobile apps

We call the set of Android apps in our benchmarks *smart-benchmarks*. Using Android apps in profiling is challenging and requires refinements. Our methodology is to invoke the smart-benchmarks from the Android-Runtime, avoiding the standalone invocation (performed through command line). Hence, the profiler runs the app from the same context experienced by the user. The first run is excluded because it has a bigger workload due to initializations and user profiling. The Smart-Benchmarks used in our study are:

Quadrant provides an overall of 21 tests covering the processor, memory, input, output, 2D graphics and 3D graphics performance. Our results are generated from running version Professional 2.1.

AnTuTu evaluates the device based on various tests: user experience, CPU, RAM, GPU Tests and I/O.

Pandora automatically recommends music based on the Music Genome Project. It is ad supported, which makes the amount of memory allocated by the app non-deterministic. We show results running version of 5.4 in the experiments section.

Spotify music streaming app.

3.3.3 Android apps

In addition to app store, we developed a set of Android apps that provide common mobile functionalities while exhibiting a deterministic behavior.

3.3.3.1 SQliteEtalon

Android comes with `SQLite` for data persistence. The *SQliteEtalon* is a multithreaded app that executes in-memory a number of transactions against a model of a banking application to measure how fast a device processes SQL queries. Each thread in the app represents a client performing multiple transaction.

3.3.3.2 JSONEtalon

JavaScript Object Notation, `JSON` is wdiely used nowadays on mobile applications to transmit data objects. At least 12% of Android applications use JSON libraries to convert Java Objects into their JSON representation [5]. These libraries can also be used to convert a JSON string to an equivalent Java object.

Based on the benchmark source code provided from `LoganSquare` [14], the *JSONEtalon* parses and serializes a set of input streams using four different JSON libraries: *(i) Jackson*: a suite of data-processing tools for Java (and JVM platform) [44]; *(ii) Gson*: most popular library to process JSON data [49]; and *(iii) Moshi*: a modern JSON library for Android and Java [103]. *(iv) LoganSquare*: based on Jackson's streaming API and *Butter-Knife* annotation library.

3.3.3.3 SVGEtalon

Based on `AndroidSVG` [4], it is a SVG parser and renderer for Android. It has almost complete support for the static visual elements of the `SVG-1.1` and `SVG-1.2` tiny specifications (except for filters).

The application loads a list of SVG files into an `imageviewer`. The workload varies based on the size of the files loaded from memory. Best way to demonstrate the app is to load a list of SVG saved maps downloaded from *Wikimedia*.

3.3.3.4 Vellamo

We consider Qualcomm's Vellamo open-source [93] for offline benchmarking of the browser tasks. The tests are entirely HTML and JavaScript and run inside Android `WebView` views which use the Android WebKit browser. *Vellamo* runs the mongoose webserver [24] to access the pages from localhost. modified to be compatible with Android KK. The benchmark is executed offline using mongoose webserver [24].

The application allows the user to select which test to be performed: *(i) Image Scroller*: to measure image decoding and rendering; *(ii) SurfWax Binder*: long series of nested calls to Javascript functions to evaluate the VM; *(iii) Inline Video*: tests the core video support; *(iv) Ocean Scroller*: tests the smoothness of scrolling; and *(v) Ocean Zoomer*: tests the browser's zooming capabilities.

Table 3.1 summarizes the execution characteristics of these benchmarks. We obtain the GC events and overhead columns when running the default Dalvik CMS collector. The allocation statistics (Heap, Objects, and Threads) are obtained by running the default CMS collector in a mode where it performs GC at very frequent fine-grained intervals (every 64KiB of allocation) to obtain tight estimates of their value. Similarly, the lifetime column reports the percentage of objects collected within the corresponding nursery size. Thus, it is a rough estimate of the extent to which the benchmark follows the generational hypothesis. The degree of concurrent allocation occurring within the benchmarks is represented by the heap contentions column.

The maximum pause time is measured as the worst-case pause time experienced by any of the mutator threads when responding to GC-safepoint suspension requests or when performing a foreground GC. The CPU overhead of GC records the percentage of CPU cycles over the execution of the benchmark that are spent performing GC, measured using the hardware CPU performance counters. Finally, the last columns show the following statistics about the code and the compiler: loaded classes, declared, methods, fields, count of compiled unit in the code cache (*count*), and the size of the compiled code (in KiB).

We invoke the benchmarks directly from the Android Runtime, which spawns each Dalvik VM instance from the pre-initialized *zygote* VM (as opposed to spawning a new Dalvik VM process from the command line). This ensures that our results mirror actual Dalvik app behavior. The Heap and Objects results for the ported Java benchmarks are similar to those reported by others using different VMs [12; 35].

Table 3.1.: Benchmark characteristics for Dalvik CMS (ignoring *zygote* process)

	Heap (MiB)		**Objects (M)**		**Lifetime (%)**			**Threads**		**GC events**				**GC overhead**		**Code**				**Compilations**	
Benchmark	Alloc	Live	Alloc	Live	128KiB	256KiB	512KiB	Total	Heap Contentions	GC-CONC	GC-ALLOC	GC-EXPLICIT	trims	Max. Pause (ms)	GC CPU (%)	Classes	Methods	Static Fields	Inst. Fields	Count	Size (KiB)
Android																					
Quadrant	28.71	8.23	0.46	0.22	9.50	20.49	40.71	16	448	6	4	42	3	30.5	2.8	1721	11,891	895	2,582	3,961	41.80
Pandora	48.91	18.76	0.28	0.06	13.01	24.89	46.03	77	829	7	18	0	4	33.1	6.2	1596	14,419	4,402	3,701	6,302	97.10
SPECjvm98																					
javac	217.47	10.19	6.15	0.27	7.60	15.78	32.68	7	276	55	42	6	1	99.0	19.7	227	1,464	674	320	5,308	68.90
jack	180.22	0.87	5.52	0.02	11.74	23.83	48.16	8	4133	105	0	2	1	24.0	8.0	131	717	275	199	2,018	41.80
DaCapo																					
lusearch	686.75	1.22	11.65	0.01	10.27	22.45	47.17	26	2.63e6	356	0	5	1	35.0	5.4	326	3,016	615	781	3,473	56.20
xalan	395.06	2.26	4.14	0.02	9.46	19.28	39.01	26	4.38e5	199	1	5	1	37.2	3.5	489	5,287	915	1,029	5,449	67.60

3.4 Experimental Environment

We measure a complete Android development platform in-vivo, avoiding emulation. We use the APQ8074 DragonBoard development kit, based on Qualcomm's Snapdragon S4 SoC using the quad-core 2.3 GHz Krait CPU, which has 4 KiB + 4 KiB direct mapped L0 cache, 16 KiB + 16 KiB 4-way set associative L1 cache, and 2 MiB 8-way set associative L2 cache. Importantly, Krait allows cores to run *asymmetrically* at different frequencies, or different voltages. Software calls can change both frequency and voltage for each core.

Our board runs on Android version 4.3-4.4 (JellyBean and KitKat) with Linux kernel version 3.4. We modified the kernel and Android VM *(i)* to allow direct access to hardware performance counters from the VM, *(ii)* to control *enabling/disabling* of the cores, and *(iii)* to expose the VM profiler to other kernel-level events.

Table 3.2.: Build properties in our experimental environment

VM parameter	value	Governor parameter	value
`heapstartsize`	8 MiB	`optimal_freq`	0.96 GHz
`heapgrowthlimit`	96 MiB	`sampling_rate`	50 ms
`heapsize`	256 MiB	`scaling_max_freq`	2.1 GHz
`heapmaxfree`	8 MiB	`scaling_min_freq`	0.3 GHz
`heapminfree`	2 MiB	`sync_freq`	0.96 GHz
`heaptargetutil`	75 %	lowmem_minfree	(page)
`large obj threshold`	12 KiB	{ 12288 15360 18432	
`trim_threshold`	75 %	21504 24576 30720 }	

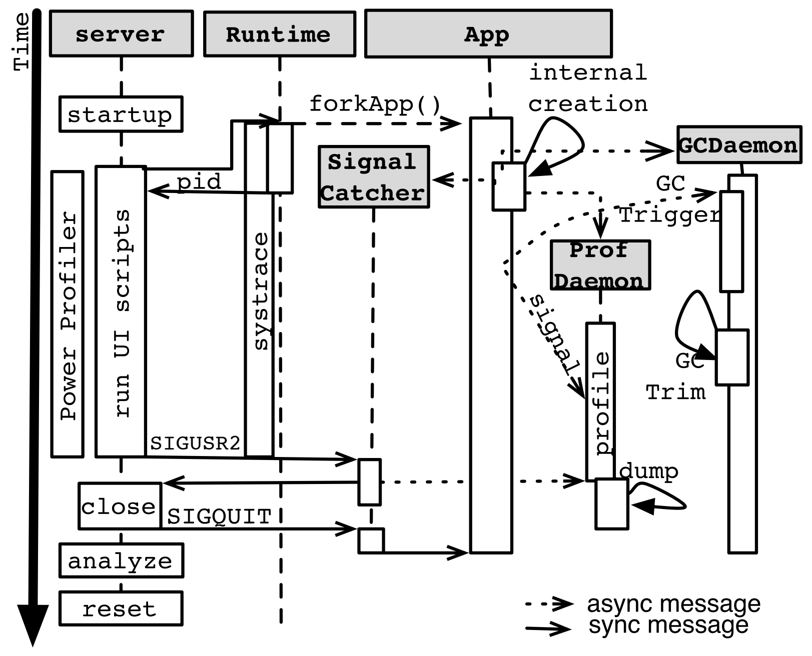


Figure 3.1.: Profiler sequence diagram

We capture both events that enable/disable cores and frequency change events due to DVFS using a modified Android `systrace`. The default device configurations are shown in Table 3.2, as shipped in the APQ8074 Android distribution.

3.5 VM Profiling

3.5.1 Gathering memory events

Figure 3.1 shows the setup of our measurement platform. The separate server drives the experiment and measures power of the device cores. The server configures the set of enabled cores and their frequencies and sets the VM parameters for the experiment (including thread affinities for individual services). Peripherals like WiFi or Bluetooth are disabled, as they are not needed to measure GC performance. The Android runtime then initializes systrace to record core frequency changes and events that enable or disable cores. The server then orchestrates the execution of the benchmark app inside the VM.

Our VM profiler runs as a separated thread inside Dalvik (the "`profDaemon`"). This daemon is disabled when measuring metrics sensitive to timing such as execution time. Otherwise, the daemon is responsible for gathering per thread statistics such as heap demographics or performance counters and correlating these values with GC events.

During the app execution, `profDaemon` waits to be signaled by the mutator to iterate on all the threads to gather the profiling data (e.g., pause times, and hardware counters) storing the results in a cyclic buffer. In our experiments, profDaemon is signalled after every 64 KiB of allocation, whereupon it gathers per-mutator statistics, without synchronization to avoid perturbing them. Gathered data is written to cyclic buffer to reduce I/O costs during execution. Finally, when a `SIGUSR2` is received by the signal-catcher, profDaemon dumps the buffer to Flash RAM.

Overall we record data that allows us to correlate *(i)* systrace data, *(ii)* performance counters, and *(iii)* internal GC events, resulting in a fine-grained and detailed picture of internal VM behavior, including app and GC characteristics.

First, the server script configures the system governor to control core frequencies and enabling/disabling of cores. The VM with the instrumented code is then pushed to the device and its time is set to match the server clock. The device is rebooted with VM parameters to control the heap parameters and thread affinities. Peripherals such as WiFi and Bluetooth are disabled. After all service initialization has completed, the server scripts start the UI automated events and wait until the end of execution. Finally, measurement data is pulled from the device and the default governor and VM are restored.

To monitor scheduling events, the profiler starts systrace to account for the core frequencies changes and events that enable or disable cores. The frequency updates are correlated with the GC events and the allocation behavior. Synchronized with the UI scripts, the power measurements run on the separate server to read the values of the voltage regulators.

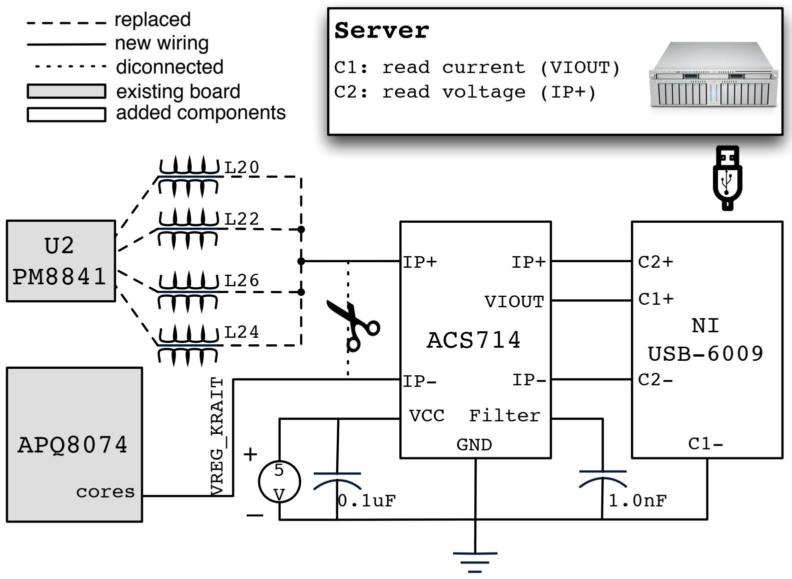
VM level allows us to analyze an app along different dimensions, e.g., compiler, class loading, or memory allocation. We found that controlling general VM parmeters (i.e., heap utilization) is not enough to run controlled experiments since the parameters are reused among all VM instances on the device, leading to different initial state conditions in each independent run. For example, changing the heap utilization changes the total RAM consumed by all the apps that start before launching the benchmark. Thus, it is necessary to account for the side effects of VM initialization on the remaining services and apps.

One possible approach is to enable the parameterization only for the benchmarks being evaluated. In addition, tuning the profiler is key for accurate results. For example, the profiler daemon should change its state before I/O operations so that the VM does not stall when the profiler writes measurements to disk. Otherwise, the responsiveness evaluation is governed by the pause times caused by the profiler I/O.

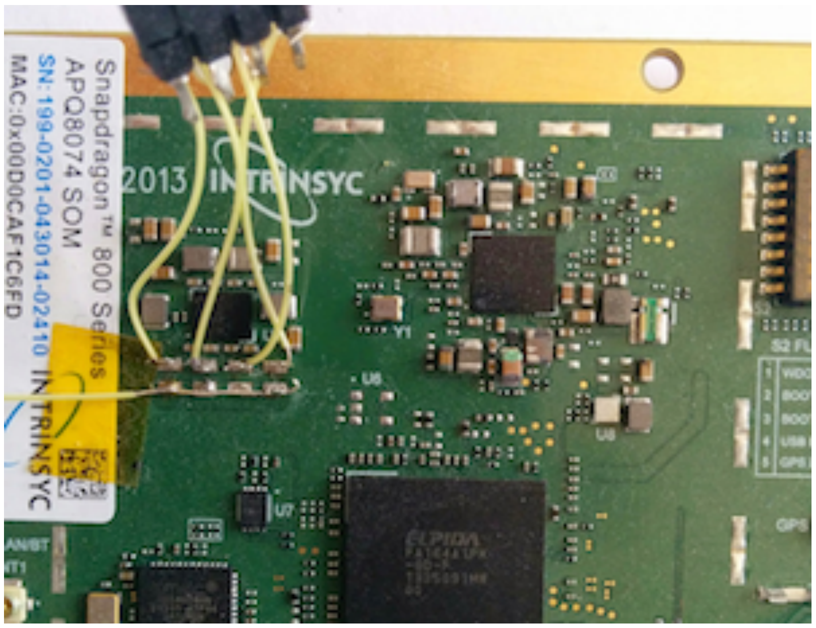
3.5.2 Power measurements

For battery operated devices, the amount of time the device stays on is key for user satisfaction. The battery life time is relevant only within a specific usage pattern, and it can be measured by dividing the battery capacity by the total power consumed by the device subsystems (e.g., CPU, display, GPU, or GPS). Estimating the total energy consumption on a mobile platform can be performed by dividing the measurements into subsystems [89; 90] which allows isolating irrelevant components.

Measuring total AC current to the device with a clamp ammeter is not precise enough to measure the effect of the VM components on the CPU power. Nevertheless, measuring the



(a) Circuit-level power measurement



(b) APQ8074 System-On-Module (SOM) modifications to measure power for the quad-Krait application processors

Figure 3.2.: CPU power measurement

power on the SoC level shadows the GC contribution on the CPU power since it accounts for the total power consumed by individual components (e.g., modem, GPU, or sensors).

Here, we measure the total physical on-chip energy consumed for each core during the app execution and we correlate the results and the configurations of several layers considering different controls. Once the app starts execution, the profiler reads the voltage drop across microprocessor applications. By controlling system configurations, we reach a qualitative description of the power behavior.

The power at the circuit level is calculated as the product of the two vectors, I and V, where V is the voltage drop across the microprocessors:

$$\mathrm{P}(t) \;=\; V(t) \,\cdot\, I(t) \tag{3.1}$$

The energy consumed by n cores is defined [22; 23] as:

$$P_{\text{CPU}} \;=\; P_{\text{uncore}} \;+\; n(P_{\text{dynamic}} \;+\; P_{\text{static}}) \tag{3.2a}$$

$$E_{\text{CPU}} \;\propto\; (P_{\text{dynamic}}) \tag{3.2b}$$

P_{static} is the static power consumed by an online core while P_{dynamic} is the power consumed by an active core and is dependent on the workload (instructions executed), i, and the core frequency ($P_{\text{dynamic}} \propto \frac{i}{\text{cycles}}$). The P_{dynamic} is calculated as a function of the effective capacitance, the frequency and the voltage as ($C_{\text{eff}} f V^2$). Although, the P_{uncore} as it is independent of the number of online cores and the workload. The energy is the integral of Equation (3.2a) over total execution time T:

$$E_{\text{CPU}} \;=\; (P_{\text{uncore}} \;+\; nP_{\text{static}})\,T + \sum_{i=1}^{n} \int_0^T P_{\text{dynamic}}\,\mathrm{d}t \tag{3.3}$$

Since the power mechanisms (e.g., governor) react to the workload by adjusting the core speed, the dynamic energy is affected by these decisions.

We measure overall current flow at the circuit level as shown in Figure 3.2(a), using a `Pololu-ACS714` Hall-effect linear current sensor [1], positioned between the CPU and the voltage regulator. We read the output voltage using a National Instruments NI-6009 data acquisition device [87]. From these we calculate instantaneous power and thence energy over time. On the board we replaced the four inductors `L20, L22, L24` and `L26` `PIFE20161T` power-choke inductors (0.24 μH, 20%, `DCR` = 19 mΩ, `ISAT` = 4.7 A ROHS) by four power-choke `WE-TPC-8012` shielded inductors (0.24 μH, `DCR` = 19 mΩ, `ISAT` = 5.8 A) in series with the hall-effect sensor [30; 114].

Figure 3.2(b) shows the modifications on a real board. The output sensor is connected to the `VREG_KRAIT_OP9` generated from the power manager (`PM8841`).

The `Pololu-ACS714` has total output error of $\pm 1.5\%$ at room temperature with factory calibration. NI-6009 allows 48 kS/s sampling rate with typical absolute accuracy 1.5 mV (error 0.9%). We eliminate noise for analog signals using two bias resistors 50 kΩ to satisfy the bias current path requirement of the instrumentation to the ground. At sample rate 2 kS/s, we read the voltage across the voltage regulator and the sensor output using the differential method and we take simple moving average for each 20 points.

3.6 Summary

Measuring performance on mobile systems is challenging due to the complex hardware and software stacks. Different feedback mechanisms continuously adapt system parameters, resulting in changed response time, power consumption, and time performance. These metrics are deeply entangled and must be evaluated in unison in a controlled environment.

Addressing these challenges, we discuss a systematic approach that tames individual feedback systems, reducing variations across experiments by disabling thermal throttling, adaptive governors, and unneeded system services. In addition, we ensure stable conditions by controlling the system image and parameters the experiment runs in.

We capture the discussed metrics by collecting *(i)* fine grained microprocessors level power, *(ii)* detailed performance counters, *(iii)* system events, and *(iv)* VM events, correlating all of them across the experiment.

Individual collected data is selected based on the target metric, ensuring that measuring does not influence our experiment. In an in-vivo case study on the Dalvik VM we use both existing applications and ported benchmarks to measure GC behavior and report precise results that can be used for future optimizations in response time, power consumption, and time performance.

4 IMPACT OF GC DESIGN ON POWER AND PERFORMANCE

Here we study the impact of GC implementation on the device including its impact on application throughput, responsiveness, and energy consumption. We explore a range of different collector implementations (including both the default Dalvik collector and extensions designed to improve both concurrency and locality), across several dimensions, including heap sizing, concurrency, multi-core scheduling, and frequency scaling.

We propose several extensions to the default GC configuration of Android, including a generational collector, spreading the GC load to different cores, and adjusting the speed of different cores during GC collection. Our evaluation shows that Dalvik's asynchronous GC thread consumes a significant amount of energy. Therefore, varying the GC strategy can reduce total on-chip energy (by 20-30%) whilst slightly impacting application throughput (by 10-40%) and increasing worst-case pause times (by 20-30%). This leads to the identification of a sweet spot between reducing energy consumptions with minimal performance tradeoff.
The contributions are:

- Discussion of alternative GC designs that extend Dalvik's default *mostly-concurrent, mark-sweep* collector with *generations*, and *on-the-fly* scanning of thread roots.

- An extensive evaluation of our measurement methodology for the different GC configurations using a set of ported standard Java benchmarks and other Android apps.

- Correlating energy consumption with GC, showing tradeoffs with other performance metrics to understand how GC overhead affects different system layers.

We refer to the default Dalvik collector as the *concurrent mark-sweep* (CMS) collector. It suspends all the mutator threads as a batch ("stop-the-world") at the beginning of each collection cycle, scans their stacks for heap roots, and then restarts them all before continuing concurrently to mark reachable objects.

4.1 GC Extensions

We consider both generational and on-the-fly variants of the default Dalvik CMS collector. These allow us to compare tradeoffs among different GC variants for mobile devices.

4.1.1 Generational CMS

We implemented a *generational* variant of the CMS collector (GenCMS) to study its effect on app performance, considering metrics that include pause times, throughput, and energy consumption. Generational collectors [81; 109] assume that recently allocated objects have a lower probability of surviving collections, splitting the heap into a young and a mature space. *Minor* collections only propagate surviving young objects to the mature space, *major* collections collect both spaces.

Our extension reuses the dirty object information already maintained for the CMS collector to find references from survivor objects (those that are live after a GC cycle) to new objects allocated since the previous cycle. This approach treats all surviving objects as *old* and newly-allocated objects as *young*.

The mark phase of a *minor* generational GC ignores old objects, marking only the reachable young objects. At the end of marking, the mark bits record the objects that survived the current GC cycle, which we merge into a *survivor* bitmap to record old objects. The survivor bitmap is cleared before each *major* (whole-heap) GC, but otherwise accumulates the survivors through each successive minor GC.

GenCMS uses complementary heap sizing heuristics to those of CMS, performing minor collections so long as the accumulated survivors do not exceed the *softLimit* computed at the most recent *major* collection. The size of the young generation is set to the *room* in the heap at the last major collection (i.e., the difference between the volume of the last major collection's survivors and the *softLimit*). As a result, GenCMS will use more space than CMS (up to the *softLimit* plus the *room*). Concurrent GCs are triggered with a *CSB* threshold set slightly below that of the *softLimit* plus *room*. Trimming collections always perform a major GC.

Trigger policies for the generational collector aim to reduce mutator pauses (by having mutators never directly perform major GCs), while also respecting the heap heuristics employed by the CMS collector:

GC-CONC as for CMS, except that the background GC daemon may perform a minor or major GC depending on the heuristics described above;

GC-ALLOC as for CMS, but the mutator performs a minor GC, noting that the next `GC-CONC` should be major;

GC-EXPLICIT as for CMS, but the mutator ignores the explicit GC call, noting that the next `GC-CONC` should be major.

4.1.2 On-the-fly

The default CMS collector has brief stop-the-world phases in which all Java threads are brought together to a halt: *(i)* while marking the thread stack roots, and *(ii)* while rescanning dirty objects to terminate marking. Each thread is notified to execute until it reaches a *GC-safepoint*, whereupon it notifies the collector that it has stopped.

Ideally, stop-the-world phases should be shortened or eliminated to improve application scalability and minimize mutator pauses. *On-the-fly* collectors [36; 38] avoid the stop-the-world phase during the marking phase.

We have extended the default Dalvik CMS collector to address the first of these pauses. The second remains future work. Once a mutator thread has had its stack roots marked we immediately signal it to resume execution. Moreover, we process threads in the order in which they arrive at their GC-safepoint, so early responders receive service before later responders. We refer to our on-the-fly collector implementation as `CMSFly`.

4.1.3 Concurrency policies

We consider variations regarding the presence, requests to, and core placement of the background GC daemon threads, as follows:

background (bg) Mutators yield *all* GC triggers to the GC daemon, without foreground GC. When an allocation request exceeds the *softLimit* or fails then the mutator instead forces allocation, and signals the GC daemon to start a new background GC cycle, before continuing. `GC-EXPLICIT` triggers simply signal the GC daemon.

foreground (fg) There is no `GC-CONC` trigger (the GC daemon is disabled). Mutators perform all `GC-ALLOC` work in foreground, concurrently to other mutators at boosted priority. `GC-EXPLICIT` remains the same.

pinned (pin) The GC daemon runs exclusively on one of the cores, with its maximum frequency limited, while other threads run on the remaining available cores. Pinning allows direct exploration of the relationship between frequency scaling and GC performance, as well as the impact of OS scheduling (which can otherwise migrate the GC daemon to a different core).

4.2 Results

Our results show that varying GC policies, such as heap growth or concurrency can reduce the energy consumed by 20-30% or can reduce the worst-case pause time by 30-50%. Moreover, app throughput is not necessarily correlated with power. GC work is inherently memory-bound but current governor heuristics focus on system load and do not incorporate the execution profile into their decisions.

As described earlier, Dalvik uses dynamic heap sizing heuristics, which size the heap at some factor of the live set resulting from the most recent (full) heap GC. Thus, both the benchmark *and* the `targetutil` affect the GC workload, in the number of instructions executed, in the mix of those instructions, and in the scheduling of GC. More frequent GC iterations and a smaller heap typically result in more GC work as a fraction of

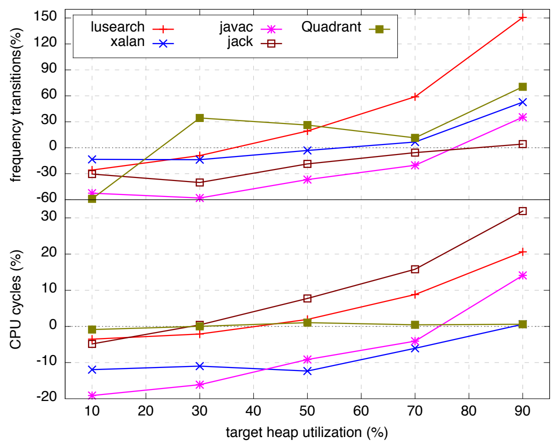


Figure 4.1.: Effect of `targetutil` on CPU cycles (bottom) & frequency transitions (top) normalized to default CMS

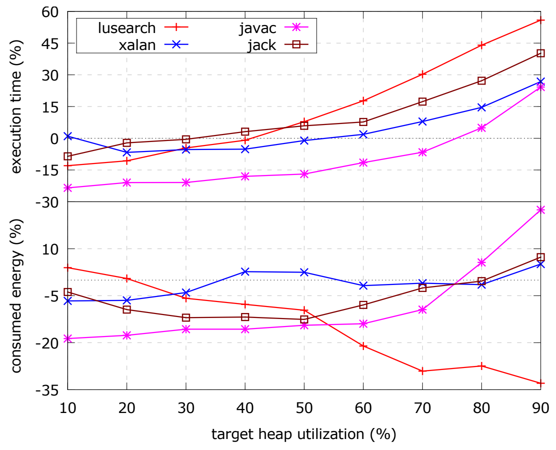


Figure 4.2.: Effect of `targetutil` on energy (bottom) & throughput (top) normalized to default CMS

total work, though the smaller heap can have second order effects on app locality. Figure 4.1 (bottom) shows the total CPU cycles executed by all app threads (normalized to CMS *per benchmark*) as `targetutil` varies. The trend is that the total app workload increases significantly with `targetutil`, except for Quadrant because of the large number of `GC-EXPLICIT` events.

Higher `targetutil` (smaller heaps but more frequent GC iterations) implies more frequency transitions, since GC workload characteristics are different from the app. Hotplugging and DVFS decisions respond to these differences. The app workload also affects the frequency of GC, so the number of transitions is quite different for each benchmark. Figure 4.1 (top) illustrates how `targetutil` affects the number of frequency transitions (normalized to CMS numbers *per benchmark*) imposed on the cores due to hotplugging and DVFS.

We now explore the tradeoff between power and throughput, versus heap size. Tighter heap imposes more frequent and higher total GC overhead. One expects app throughput to decrease (i.e., total execution time to increase) as GC overhead increases with `targetutil`. One might expect the same for energy consumption. Intriguingly, our results show that increased execution time does not always correspond to increased energy consumption.

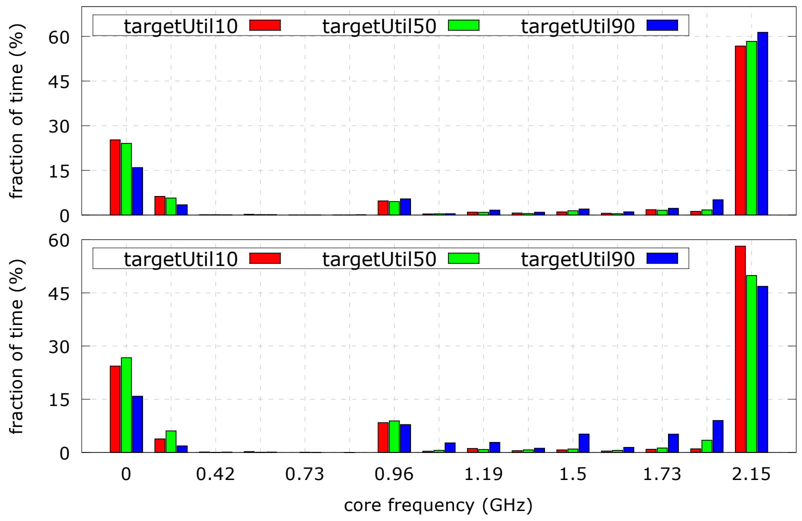


Figure 4.3.: xalan (top), lusearch (bottom): Core frequency distribution (as fraction of time)

Figure 4.2 shows both execution time and total energy consumed for each benchmark versus `targetutil` with the Dalvik CMS collector. As expected, all four of the benchmarks shown have longer execution times in tighter heaps. But lusearch, unlike the other benchmarks, consumes much less energy in tighter heaps.

The explanation for this seemingly anomalous behavior is that lusearch benefits from the system making more effective frequency transition decisions than the other benchmarks. Recall that GC work is memory-bound, resulting in increased memory cycles per instruction (MCPI). Thus, choosing a higher frequency to perform that work (commonly called "race to idle") does not necessarily improve throughput; a lower frequency can get the same work done in the same amount of time at lower energy.

Figure 4.3 shows the distribution of core frequency as a percentage of total execution time for lusearch and xalan, for `targetutil` values of 10; 50 and 90%. For lusearch, running with a tight heap (90%) causes the cores to spend a fraction of the execution time on a range of lower frequencies (more efficient), and offlined, more than for looser heaps. In contrast, xalan has more of its execution time spent at (less efficient) higher frequencies with tighter heaps.

Having all GC performed in background (bg) consumes more energy. The performance counters (i.e., L1 misses and CPU cycles) suggest that a background GC results in a higher workload across all the benchmarks due to heap synchronization (i.e., context switches) and trimming operations triggered when the heap is under utilized. The bg variant causes threads to wait longer than the other implementations, because they must wait at GC-safepoints for GC to mark the roots of all mutators. On-the-fly marking (CMSFly) instead allows the mutators to continue once their own roots have been marked.

4.3 Summary

Our results show that existing DVFS policies should be informed of GC events by the VM to make more informed hotplugging and frequency scaling decisions. Similarly, app developers need a range of GC strategies to choose from, so they can tune for responsiveness, utilization, and power consumption. This study is a first step to analyze the GC within the system scope to serve as a guide of how to evaluate the coordination between design decisions across all the layers of the system stacks (software and hardware).

5 CONTROLLING THE GC POWER CONSUMPTION

Dynamic voltage and frequency scaling (DVFS) is ubiquitous on mobile devices as a mechanism for saving energy. Reducing the clock frequency of a processor allows a corresponding reduction in power consumption, as does turning off idle cores. Garbage collection is a canonical example of the sort of memory-bound workload that best responds to such scaling. Here, we explore the impact of frequency scaling for garbage collection in a real mobile device running Android's Dalvik virtual machine, which uses a concurrent collector. By controlling the frequency of the core on which the concurrent collector thread runs we can reduce power significantly. Running established multi-threaded benchmarks shows that total processor energy can be reduced up to 30%, with end-to-end performance loss of at most 10%.

5.1 Objective

Shipped as systems-on-a-chip (SoC), mobile platforms feature heterogeneous multi-core hardware with on-die hardware peripherals such as WiFi and GPS. User experience is strongly based on device responsiveness and battery lifetime. To increase power efficiency, vendors often install binary-only, vendor-specific *thermal engines* that manage the throttling of core frequencies through *dynamic voltage and frequency scaling* (DVFS). DVFS heuristics aim for energy savings while maintaining reasonable performance [65; 85]. However, the complexity of modern mobile systems such as Android, with interactions across layers from hardware up through operating system and managed run-time system to application, makes managing this tradeoff difficult and complex.

Here, we focus on understanding and controlling the power-performance tradeoff of the garbage collector of Android's Dalvik virtual machine (DVM) running on a real mobile device. Prior work has thoroughly explored this tradeoff for general-purpose platforms

[20; 42; 91; 96], including surveying energy management across the stack [73], but the interactions of layers on mobile devices have not been directly addressed even as such devices are more sensitive to energy and thermal conditions. We focus on Dalvik as the most widely used mobile managed run-time system, treating it essentially as an opaque black box, though we observe and correlate significant memory management events with CPU power, performance, and responsiveness.

5.2 Approach

Tracing garbage collectors traverse heap references starting from the mutator roots to determine all the *reachable* objects [70]. The collector reclaims memory occupied by non-reachable objects. As a result, garbage collector instructions are dominated by memory operations to load and trace the references, making it primarily memory bound, and incur more memory cycles per instruction than compute-bound mutator workloads. Motivated by this specialized GC workload, several studies have explored offloading the GC workload to: (*i*) dedicated slow cores [20; 96], (*ii*) GPUs [83], and (*iii*) even specialized hardware [9; 10].

Here, we explore the direct power impact of Dalvik's concurrent garbage collector on the Android mobile platform. Mobile devices use sophisticated power management strategies in both hardware and software, with only simple communication among the layers. The DragonBoard APQ8074 development kit for Qualcomm's mobile platforms deploys a proprietary thermal engine to monitor temperature and workload which provides feedback to Android's Linux ondemand *governor* [18] to influence DVFS decisions. The DragonBoard's quad-core SnapDragon S4 processor supports *asymmetric SMP* with separate power domains for each core, so that each can be brought online, and its frequency controlled, independently of the other cores. Primary core 0 is always kept online (to service the OS as well as applications), though it may be throttled back to a very low idle frequency based on workload and demand.

Dalvik's concurrent garbage collector runs as a daemon thread in the Dalvik virtual machine. When triggered, it may be scheduled on any available core, at whatever frequency the governor sets for that core. The default governor has no special knowledge about the activity of the collector daemon. The default governor applies the same workload feedback mechanisms to the collector daemon as it does for all other threads. To isolate and control the impact of the collector daemon we make the following modifications to Android and Dalvik:

1. Pin the Dalvik collector daemon to primary core 0 so that we know which core it will run on. This core is always online, so we do not affect decisions for onlining cores for other threads. Moreover, we do not reserve a core and keep it online solely for the collector daemon, which runs only intermittently; this would otherwise result in unnecessary power consumption to keep a core online unnecessarily.

2. Modify the Dalvik virtual machine so that the ondemand governor knows when the concurrent collector daemon is active, by marking the beginning and end of each cycle of garbage collection.

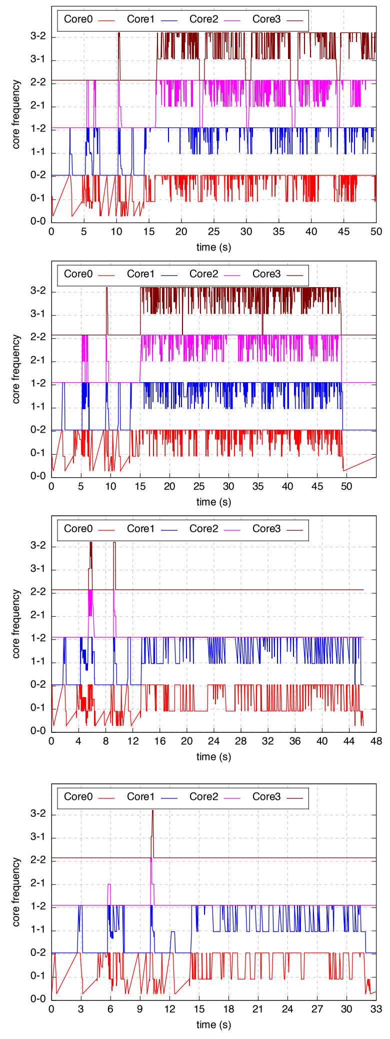
3. Modify the ondemand governor to *cap* the frequency of primary core 0, only for the duration of the concurrent GC cycle. The governor may choose to lower the frequency below this cap as it chooses. When the concurrent collector daemon is not active (i.e., outside the GC cycle), the governor is also free to adjust the frequency at will above the cap.

Given the memory-bound nature of garbage collection we expect lower frequencies to achieve the same work (instructions executed) without significantly degrading throughput, because at high frequencies many processor cycles (i.e., energy) will be wasted waiting for memory. Thus, one measure of garbage collector efficiency is cycles per instruction executed (CPI). The SnapDragon S4 allows the sampling of per-thread hardware counters, so we can directly measure CPI for the collector daemon. Our results demonstrate how CPI improves for the collector daemon when the frequency of its core is capped.

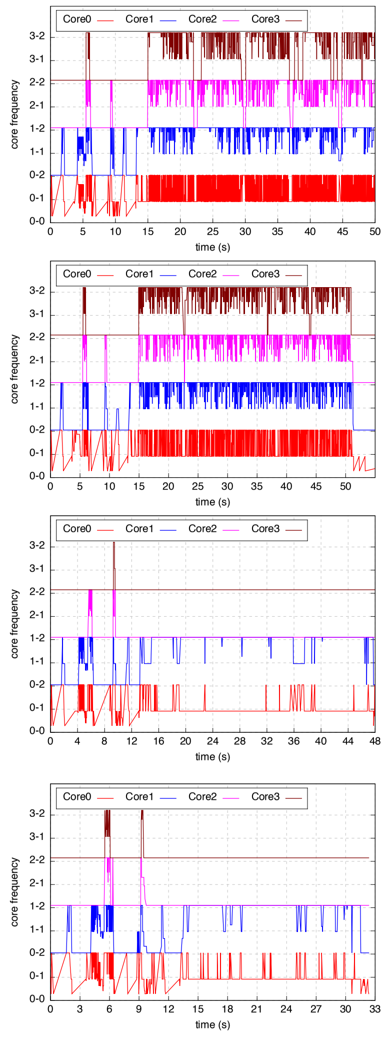
Of course, slowing the collector core also indirectly slows down the application because any attempt by the application to allocate will force it to wait for the collector daemon to finish the GC cycle. Thus, application throughput can be expected to decrease with a slower collector core. This tradeoff between application throughput and frequency of the collector core is the relationship we are interested in, because there turns out to be a sweet spot where slowing the collector core saves power without significantly reducing application throughput.

5.3 Results

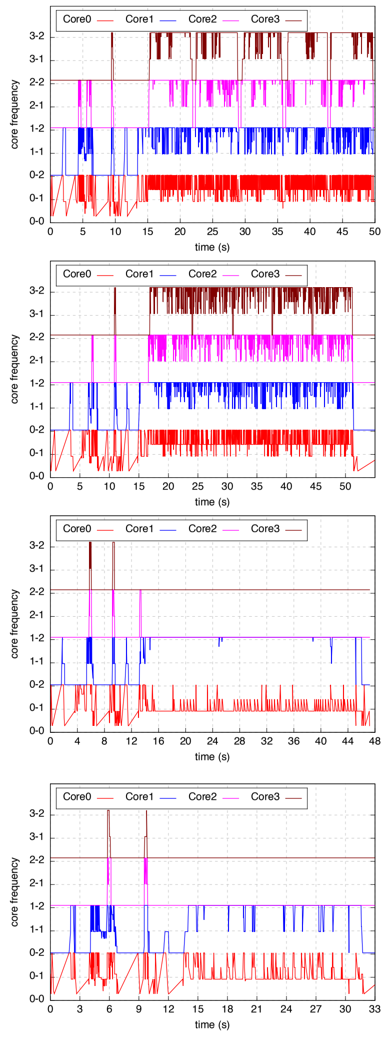
The power profile of an application is dictated by the core frequency transition and onlining/offlining DVFS events that occur during its execution. Moreover, when we cap the GC daemon's core frequency it will result in different feedback to the governor and different decisions about these events. To understand the impact of this for each of our benchmarks we compare the DVFS events and frequency values for the original Dalvik system with those of the GC-aware governor, for various values of GC core frequency caps. The profiles appear in Fig. 5.1. Figure 5.1(a) plots the frequency transitions of the apps running on the default Dalvik system. For DaCapo benchmarks, cores 2 and 3 are disabled between iterations, while the second core is disabled outside the main control loop for the iterations. The single threaded benchmarks (i.e., SPECjvm98) use only two cores. Figures 5.1(b) to 5.1(d) demonstrate the difference between the default and the GC-aware governors; each plots the frequency transitions at a different GC core frequency cap (0.96, 1.5, and 2.15GHz, respectively). Notice how capping affects not only the transitions for the GC core 0, but also the other cores servicing mutator threads. The reason for this is that changing the GC core 0 frequency affects the latency of stalls the mutator threads experience during stop-the-world phases or while waiting for the GC cycle to finish so they can allocate. This in turn changes their performance profiles that feed into the governor in its transition decisions for the other cores.



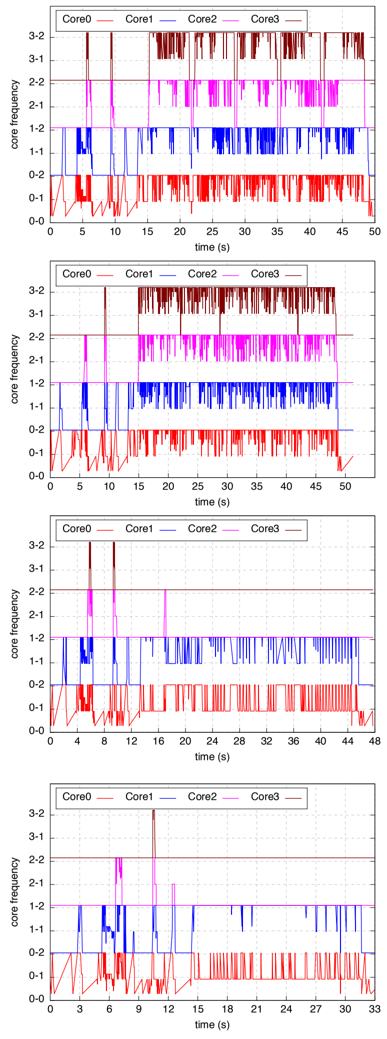
(a) default Dalvik



(b) 0.96GHz

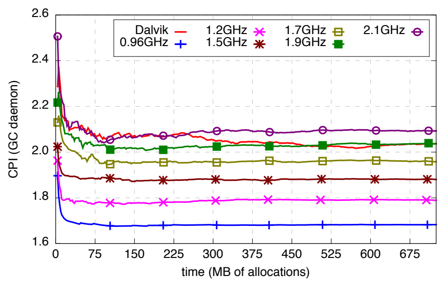


(c) 1.50GHz

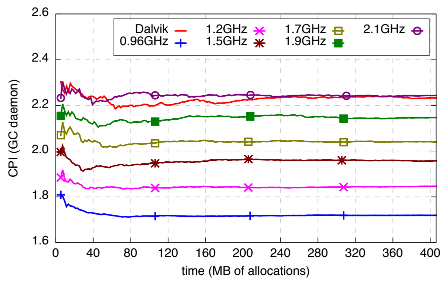


(d) 2.15GHz

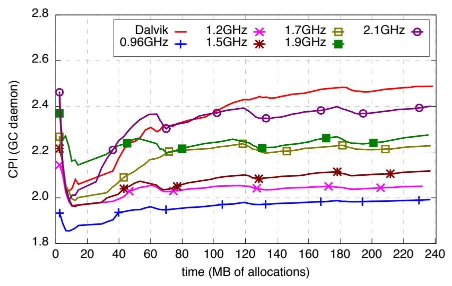
Figure 5.1.: Timeline of frequency transitions for lusearch (top), xalan, javac and jack (bottom) respectively



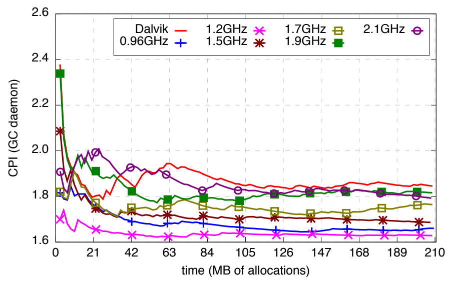
(a) lusearch



(b) xalan

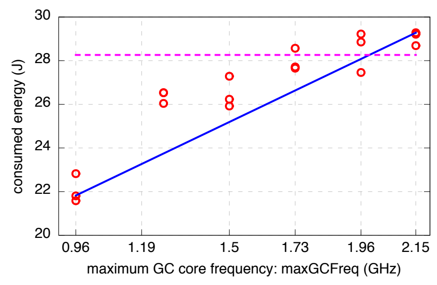


(c) javac

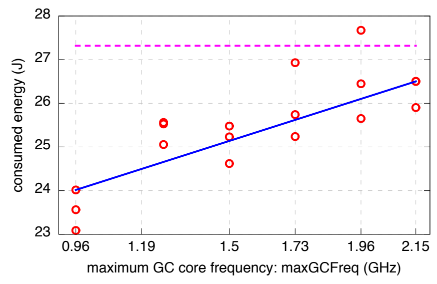


(d) jack

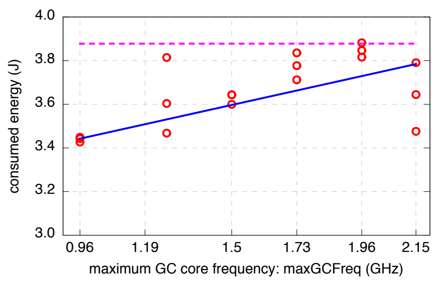
Figure 5.2.: Cumulative average GC daemon CPI



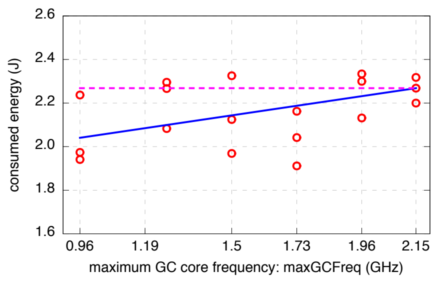
(a) lusearch



(b) xalan



(c) javac



(d) jack

Figure 5.3.: Total consumed energy

5.3.1 Energy and throughput

GC is a memory bound task so we measured the per-thread CPI of the concurrent GC daemon for each GC cycle. Figure 5.2 plots the cumulative average CPI (y-axis) for the GC daemon over time (measured in bytes allocated), for the default Dalvik and GC-aware capped governors. The clear trend is that the lower GC core cap, the lower the CPI. This is the primary reason why running the GC daemon at a slower speed can improve power efficiency without a proportional loss of performance.

In contrast, Fig. 5.4 shows the *overall* (rather than cumulative) average CPI for each benchmark while varying the GC core cap. This varies very little across GC core frequency caps, indicating that GC core CPI has little impact on overall CPI, which is dominated by the workload rather than GC. Thus, the GC daemon is a good candidate for targeted frequency capping to improve its efficiency.

The energy impact of capping the GC core frequency by the GC-aware governor is clear. Figure 5.3 plots the effect on energy consumed for a range of GC core frequency caps for each benchmark. Energy consumed with the default Dalvik governor for one execution is shown as a horizontal line. The trend lines are linear fits to the scatter plots (recall that energy consumed is proportional to frequency for a given fixed workload; computing more refined statistics such as confidence intervals is not feasible for so few data points). lusearch has a best energy consumption at 0.96GHz which is 30% lower than capping the GC core at highest frequency (2.15GHz).

Although both xalan and lusearch are multithreaded apps, xalan shows less energy savings (10%) than lusearch. The differences are due to the characteristics of the workloads exhibited by each benchmark. For example, xalan is known to perform more frequent memory operations [72], borne out by the higher overall CPI for xalan in Fig. 5.4.

Energy consumed for jack varies least. Referring back to the frequency transition diagrams for jack in Fig. 5.1 one notes that the profiles for jack are similar across frequencies indicating that the ondemand governor makes similar transition decisions regardless of the GC core frequency cap.

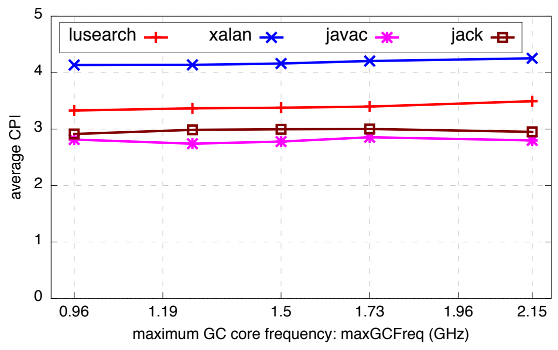
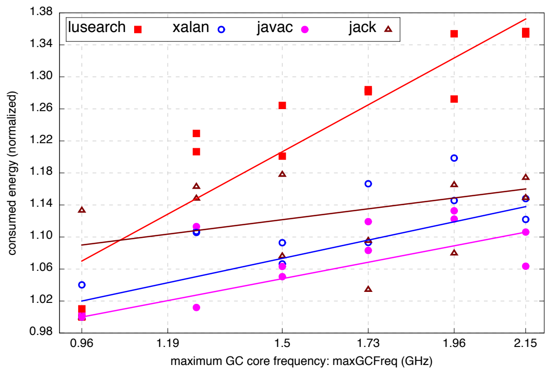
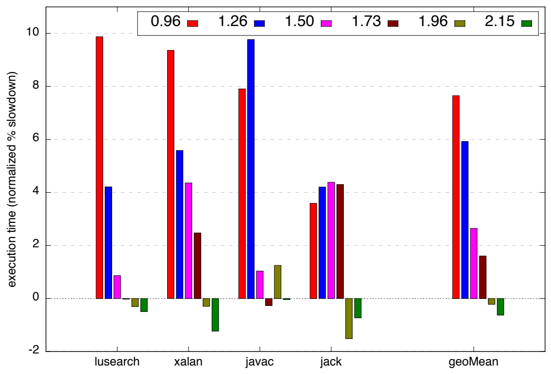


Figure 5.4.: Average overall CPI



(a) Energy consumed

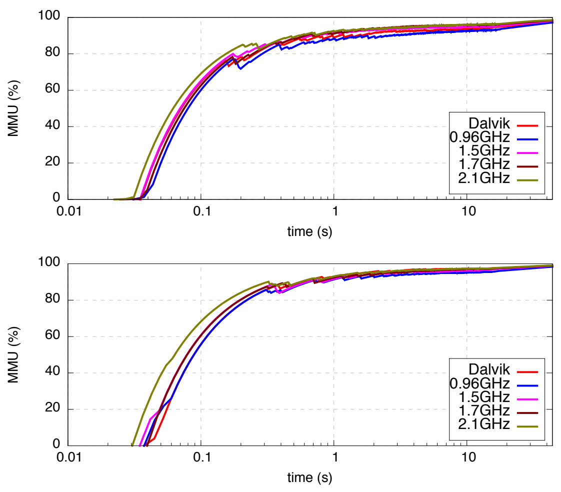


(b) Slowdown normalized to the default Dalvik

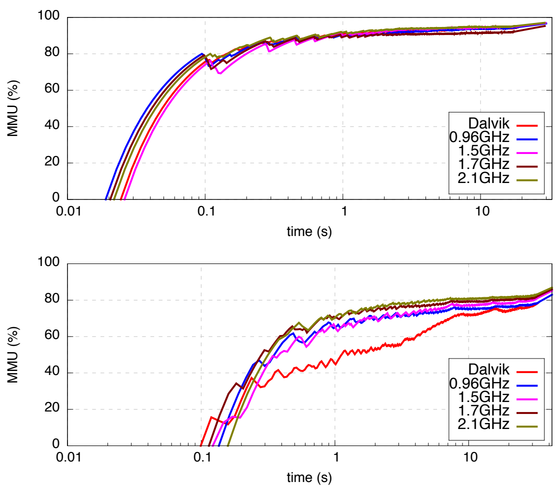
Figure 5.5.: Effect on energy and throughput

Figure 5.5(a) summarizes the effect of dynamic GC core frequency capping on the energy (normalized to the smallest value *per benchmark*). The clear trend is that higher frequency caps (faster collector thread and higher CPI) implies more energy consumption.

We now explore the *tradeoff* between power and throughput while varying the GC core frequency cap. One expects that capping core frequencies may affect mutators scheduled on the slower GC core interleaved with the GC daemon. Slowing the collector threads may also lead to longer collection windows during which mutators wait for the concurrent collection cycle to finish. Figure 5.5(b) shows the performance tradeoff with varying GC core frequency, normalized to the execution time of the default system.



(a) DaCapo: lusearch (top); xalan (bottom)



(b) SPECjvm98: jack (top); javac (bottom)

Figure 5.6.: Minimum mutator utilization

For three benchmarks (lusearch, xalan and javac), the throughput slowdown is at worst 10%. As noted earlier, jack is less sensitive to the value of the GC core frequency cap; its has throughput penalty at worst 4%.

Importantly, it is possible to obtain significant energy savings for modest reductions in throughput. For example, at the 1.5GHz cap the performance penalty is only around 4%, yet energy savings range up to 13%. And for a performance penalty of 10% energy savings are as high as 30%!

5.3.2 Responsiveness

Slowing down the GC daemon also affects mutator responsiveness by making allocators wait for the GC cycle to finish and to resume execution after the collector's relatively brief stop-the-world phases (to sample the roots and process weak references). On mobile devices responsiveness is a primary virtue in providing usable user interfaces. This is the primary reason for Dalvik to use a concurrent collector.

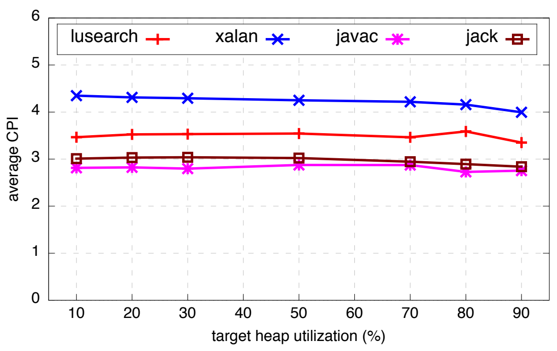
On their own, reporting worst case and average mutator pause times don't adequately characterize the impact of different collector implementations. Instead, minimum mutator utilization (MMU) over a range of time intervals yield a better understanding of the distribution and impact of pauses [28; 70; 92]. Our VM profiler records the pauses experienced by each mutator, classified into three categories: *(i)* GC-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.

Figure 5.6 shows the MMU results for each benchmark with varying GC core frequency caps. MMU graphs plot the fraction of CPU time spent in the mutator (as opposed to performing GC work) on the y-axis, for a given time window on the x-axis (from zero to total execution time for the application). The y-asymptote shows total garbage collection time as a fraction of total execution time (GC overhead), while the x-intercept shows the maximum pause time (the longest window for which mutator CPU utilization is zero). When comparing GC responsivenesses, those having curves that are higher (better utilization) and to the left (shorter pauses) can be considered to be better (with respect to mutator utilization).

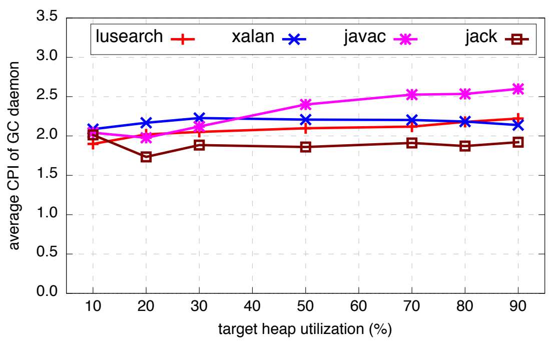
The GC-aware governor with 2.15GHz cap has the best MMU curve on the DaCapo benchmarks lusearch and xalan (Fig. 5.6(a)). The explanation for this behavior is that pinning the GC daemon reduces the number of task migrations on lusearch and xalan by a factor of 6 and 5%, respectively.

One might consider MMU for jack to be quite unintuitive as 0.96GHz has both smallest maximum pauses and best overall utilization. However, note that applying the GC-aware governor with a GC core cap of 0.96GHz, the ondemand governor responds by keeping core 1 on high frequency for a larger portion of execution time than the default governor, as illustrated in Fig. 5.1 (bottom). For javac (single threaded), the mutator spends more time waiting for collecting a relatively large heap (maximum heap size 14MiB). On the other hand, the GC-aware governor has a better overall utilization than the default Dalvik.

Overall, the GC-aware governor doesn't markedly degrade maximum pause times, and generally improves overall utilization.



(a) Average overall CPI varying `targetutil`



(b) Average GC daemon CPI varying `targetutil`

Figure 5.7.: Effect of `targetutil` on CPI

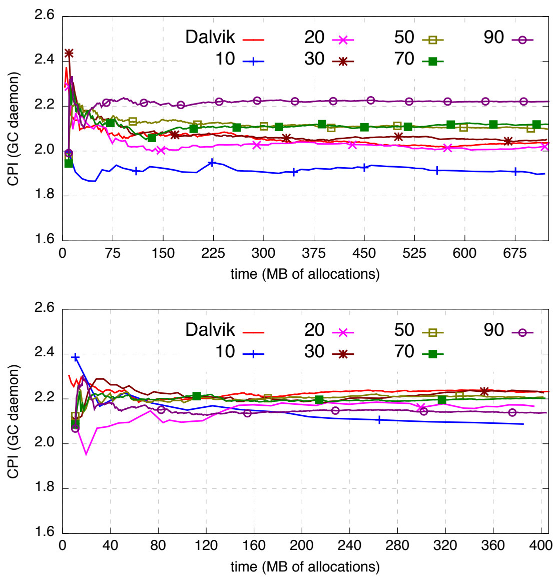
5.4 Discussion

Two items in our evaluation bear further discussion: use of CPI as an leading indicator for energy needs and the impact on CPI of heap size, plus the arrival of the new Android Run Time (ART).

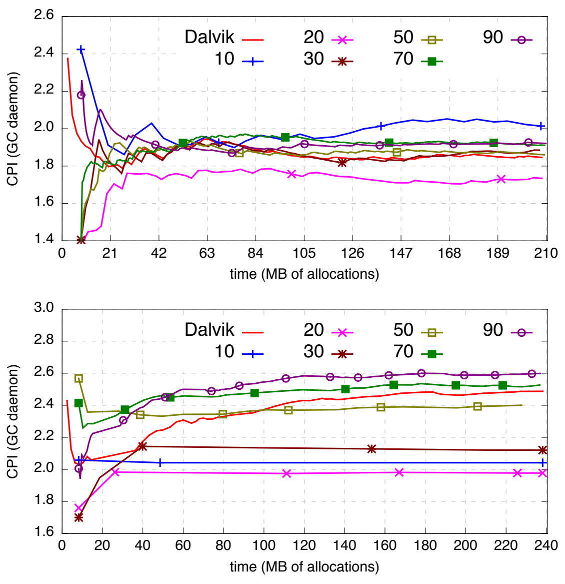
5.4.1 Choice criterion to characterize workload

Our study relies on CPI as an indicator for CPU energy requirements. The reported results do not explore CPI as a function of hardware architecture, which would be interesting for further study. Also, the number of instructions to run a fixed amount of work varies between different executions due to concurrency in the mutator threads. Taking into consideration that CPI does not measure I/O, OS interruptions, or GPU executions, our results show that app performance and energy consumption correlates well with the CPI.

Moreover, heap size can affect frequency scaling decisions and resulting energy effects and app throughput. Indeed, many GC studies treat heap size as the most important parameter to vary since it can have a significant impact on throughput and responsiveness. The parameter that controls the mix of collector work versus mutator work is the target heap utilization (`targetutil`), which affects heap sizing decisions. As described earlier, Dalvik uses dynamic heap sizing heuristics, which size the heap at some factor of the live set result-



(a) DaCapo: lusearch (top); xalan (bottom)



(b) SPECjvm98: jack (top); javac (bottom)

Figure 5.8.: Cumulative average GC daemon CPI varying `targetutil`

ing from the most recent (full) heap GC. Thus, both the benchmark and the `targetutil` affect the GC workload, in the number of instructions executed, in the mix of those instructions, and in the scheduling of GC. Figure 5.7(a) shows the average CPI for a range of `targetutil` values. The clear trend is that the total CPI does not vary significantly with heap utilization as long as the mutator workload is consistent. However, the average CPI of the GC daemon does vary somewhat since the amount of work done by the collector in each collection is different as illustrated in Figures 5.7(b) and 5.8. But the variation is not nearly as large as that obtained by capping the GC core frequency, which dominates the effect of different target heap utilization.

5.4.2 Android runtime extension

Google announced ART, a next-generation run-time system for Android 4.4 that relies on somewhat aggressive ahead-of-time compilation of apps [51], and which will replace Dalvik in the future. ART also implements several GC improvements that will impact our results: *(i)* the marking phase has one stop-the-world phase for marking the roots instead of two (no longer stop-the-world for weak references); *(ii)* introducing a pseudo-generational *sticky* collector to deal with short-lived objects; *(iii)* dedicating a separate heap for large objects; and *(iv)* enabling parallel processing during marking of mutator roots.

We do not expect the CPI of the GC daemon to change significantly as its work is dependent on the mutator heap data structures rather than the mutator code generated by the ahead-of-time compiler (and the daemon is implemented natively). Thus, the merit of controlling the frequency scaling decisions to reduce the GC daemon CPI still holds. Moreover, improved concurrency will reduce mutator pauses due to waiting for the collector. Thus, we are confident in advocating integration of governor decisions with GC activity as an effective mechanism to tune system performance for other systems including ART. As future work we will port our frequency governor to ART and study and improve settings for this platform.

5.5 Summary

On mobile devices, GC has significant impact on energy consumption, not only from its explicit overhead in CPU and memory cycles, but also because of implicit scheduling decisions by the OS with respect to CPU cores. Motivated by the fact that the kernel has the power to change core frequencies to adapt the system to changing workloads, we presented a new GC-aware governor that caps the frequency of the core while the concurrent collector thread is active. The new governor is evaluated *in vivo* showing that it reduces total on-chip energy (up to 30%) for comparably low throughput tradeoff (of at most 10%) on our workloads. The GC-aware governor has no negative impact on benchmarks experiencing optimum frequency scaling decisions by the default unmodified system. Our work is the first to analyze memory management on mobile devices across non-adjacent system layers (app, kernel and hardware).

6 GARBAGE COLLECTION AS A SERVICE

"The way to become rich is to put all your eggs in one basket and then watch that basket."
— Andrew Carnegie.

Mobile devices run dozens of so-called "apps" in protected *managed run-time environments*, also known as *virtual machines* (VMs). All these VMs run concurrently and each VM deploys **purely local** heuristics to organize resources like memory, performance, or power. Mobile frameworks deploy crude mechanisms to manage the scarcity of memory: (*i*) each VM decides locally when to garbage collect and what to collect (e.g., using a minor or major collection, collecting concurrently, or trimming the heap), (*ii*) kill background applications, or (*iii*) optimize/recompile running applications. We identify the lack of coordination among these VMs as an opportunity for optimization for mobile systems over: (*i*) memory usage, (*ii*) run-time performance, and (*iii*) power consumption. A **global memory manager service** can avoid conflicts across garbage collectors in separate VMs and make informed decisions based on global resource constraints. Our prototype implementation collects system-wide statistics from all running VMs, makes centralized decisions about memory management across all layers, and also collects garbage centrally. Furthermore, the global collector coordinates with the power manager to tune collector scheduling. In our evaluation we illustrate the power of such a central coordination service and garbage collection mechanism in reducing total energy consumption (up to 18%), throughput (up to 12%), improving memory utilization, and adaptability to user activities.

6.1 Motivation

Mobile devices are required to provide the desired performance and responsiveness while being constrained by energy consumption and thermal dissipation. With performance, heat, and power consumption strongly tied together it is common to use *dynamic* *frequency* at run-time to reduce power consumption and amount of generated heat (i.e., CPU throttling). Mobile platforms come bundled with software components such as kernel governors [18] and proprietary thermal engines that control power and thermal properties. The *crude* decisions made by these engines are orthogonal to resource management heuristics embedded within components in the user space level.

With the number of connected Android [50; 51] devices exceeding a billion[1], the dominance of Android's runtime introduces an interesting challenge: we are faced with devices that continuously run dozens of *managed language environments*—also known as *virtual machines* (VMs)—in parallel. These VMs run both as "apps" in the foreground and as services in the background. 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. For mobile devices, all concurrent VMs share a set of constrained and over-committed resources. Without global coordination, each VM optimizes independently across orthogonal goals: performance, responsiveness, and power consumption.

VM services such as *garbage collection* (GC) typically come with a number of optimization and scheduling heuristics designed to meet the performance needs of the 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 [66]. 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: i.e., initial heap size, heap resizing, and the mode of collection to perform [17; 77]. Even for a single VM it is extremely difficult to identify the best collector and heuristics for all service configurations [66; 70; 102]. Recent interest in fine-grained power measurement shows that GC has a significant impact on energy consumed by the apps [20; 96]. 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

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

VM is to take advantage of GC idleness and control the frequency of the core on which the concurrent collector thread is running [34; 62]. Although this approach increases the responsiveness of applications and reduces memory consumption as perceived from a single VM, it is not feasible to achieve a global optimization criterion with multiple 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. Among these layers, which are usually not tuned in harmony with the VM implementation, are:

1. **Device Configurations:** The mobile system has globally fixed VM configurations such as the initial and the maximum heap sizes. These configurations are device-specific and are based on several factors like RAM size and screen dimensions.

2. **OS:** Some heuristics and configurations may be applied on their own, without coordinating with the VM [67; 76]. Android employs the *low memory killer* (LMK) operating system module to monitor the available memory, and to kill arbitrary processes when memory runs short to reclaim memory for the system.

Here we consider the impact of just a single aspect of managed run-time environments, namely memory management (GC) on the device overall performance. We identify the missing coordination between concurrent VMs as an opportunity for optimization on mobile systems along (*i*) memory usage, (*ii*) run-time performance, and (*iii*) power consumption. A global service that collects statistics from all running VMs can optimize across them., and it allows for coordination with power managers to achieve global energy optimization. The service can prioritize GC operations based on the estimated freed bytes, 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.

We show that a global memory management service provides 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 *GCService* process that serves all running VMs.

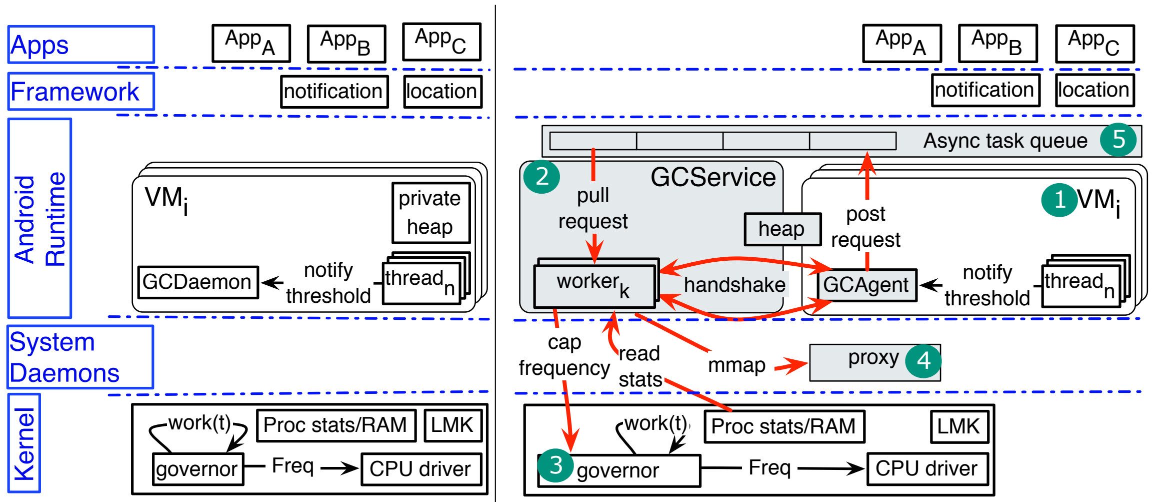
The GCService unifies interactions between nonadjacent system layers (i.e., the low-level OS power manager) and GC tasks. The service has OS-like access, capable of scanning and collecting per-process VM heaps remotely and gathering statistics about all the running VMs in the system, including: process priority, allocation rate, and heap demographics. This allows for a fine-grained control over the GC tasks being executed and their scheduling compared to just coarsely signaling individual VMs to start GC collections.

Contributions. We illustrate the power of combining vertical cross-layered heuristics to achieve efficient heap decisions such as compaction, collection, and trimming. GCService 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.

6.2 Design and Architecture

Android. The Android platform is designed in the form of a software stack that comprises various layers running on top of each other in a way that the lower-level layers are providing services to upper-level components. Figure 6.1(a) shows the following layers: (*i*) Android is built on a modified Linux (*Androidism*), providing a layer of drivers, shared memory, and interprocess communication (*binder*); (*ii*) Native libraries and system daemons (i.e., bionic, and thermal engines); (*iii*) Android runtime, which is the core VM library that hosts



(a) Default Android stack

(b) GCService system overview: new components are shaded

Figure 6.1.: Comparing default Android stack and GCService

an app; (*iv*) framework layer that provides services to apps—i.e., location manager; and (*v*) application layer that compromises the third party applications installed by the user and the native applications (i.e., browser).

We introduce a service that runs as a separate VM and collects information from running applications and runtime services—i.e., RAM and workload. Based on these statistics it then performs global decisions across all running applications. Building on this central service that runs as its own separate Linux process, we design a *GCService* 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, allowing for *more informed* decisions. Figure 6.1(b) shows the high-level interaction between the components of our system:

1. The client VM performs all necessary initialization including the heap initialization. The VM instance is augmented by a native daemon called *GC agent*. The agent daemon connects to the server, allowing it to synchronize on the client heap.

2. The *server* process maps the client heap into its private virtual address space. In addition, the server manages all coordination and synchronization with other system layers–i.e., power managers.

3. A GC-aware governor that communicates GC activities to CPU throttling decisions in order to control the power efficiency of the GC.

4. A platform-specific *proxy* that abstracts the mechanism of sharing the client's heap.

5. An asynchronous *task queue* that allows the GC agent to post requests to the GCService.

6.2.1 Challenges

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

Reliability. It is essential that the GCService provides a uniform mechanism for managing apps' life cycles in isolation from each other [68]. 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 GCService do not bring the system down.

Android apps are designed to tolerate random restarts. Stock Android-runtime restarts apps when they become unresponsive. Like all other Android services, the GCService is designed to restart after crashing. When the GCService is offline, clients perform GC locally until the server is back online. A client engaged in a GC cycle may hang depending on the lock status. The client will be restarted when it becomes unresponsive.

Security. Since Android allows execution of native code, serious security issues arise with previous approaches like *Multi-tasking VMs* (MVMs) that let a single VM to run multiple applications [31; 117]. Such approach consists of sharing one heap across all running apps, making the system highly vulnerable to security exploits. Our design, on the other hand, offers a secure approach where the code of the central GCService is trusted, and each VM has access to only its own local heap. In the GCService, the heap layout must not be identical across all the VMs (including the zygote), supporting the shuffling introduced by techniques such as *address space layout randomization* (ASLR). This reduces the risk of memory attacks.

Performance. Our system aims at reducing the latency of app responses while assuring better performance and longer (battery) lifetimes. However, with a centralized GCService, a slow process of GC requests can introduce a new bottleneck. Therefore, the new system has to provide a robust asynchronous mechanism to (*i*) allow for fast communication between the client and the server, and (*ii*) reduce the context switching overheads..

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

6.2.2 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 [66]. However, with the introduction of other system components into the cost equation (i.e., CPU scheduling and CPU throttling), the GC scheduling can be tuned by hiding expensive garbage collection operations inside of small, otherwise unused idle portions of application execution, which results in *better* overall execution [34].

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 [34; 62] 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.

During a GC cycle, the mutators are unlikely to make a full use of fast cores because (*i*) threads 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, and (*ii*) GC is inherently memory bound, which is subject to total memory bandwidth. Thus, GC scheduling can be tuned using the following mechanisms:

1. The prioritization of GC tasks across dozens of simultaneously running VMs, which need 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 a better responsiveness and throughput [34]. This works as a GC-aware governor that caps the frequency of the core.

A global GC policy cannot achieve energy goal on its own. Therefore, the GCService works as a global collector that handles several GC phases on behalf of other VMs. In this way, the coordination between the power manager and a single process is feasible and practical.

Having a single process to handle the launch of GC tasks allows for a more fine-grained control of 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. For e.g., the GCService 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 code footprint and code cache pressure from individual threads that are running per-app GC and from negative interactions with the scheduler. This results in better cache performance and locality, in addition to reducing the VM footprint and abstracting memory management from the rest of the VM implementation.

Therefore, the GCService coordinates with a power manager and a scheduler to hide the cost of the GC idleness [34] across all VMs–and not only for a single VM [62]. The GCService feeds the power managers with information about the GC tasks, capping the maximum frequency of the core on which the collector thread is running.

6.2.3 global GC service vs. global GC policy

Some studies investigate auto-tuning of the GC policy *locally* per single VM [102; 112]. 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.

Global Heuristic Manager. 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—i.e., priority, (*iii*) per-heap stats such as heap variable, fragments 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 then static thresholds.

6.3 Service 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 and the Linux kernel (with 6K and 1K LoC, respectively) to allow direct access to hardware performance counters from the VM (to measure precise run-time statistics when running on the development board).

6.3.1 System startup

System boot follows the standard steps of loading hardware-dependent components and initializing subsystems and drivers. We extend these steps by launching the proxy to guarantee that the daemon is ready to receive requests. Then, the booting process forks the GC service which initializes the communication queues and the pool of worker threads. The server has a singleton *listener* thread that fetches tasks from the task queues and inserts them into local queues to be handled by the worker threads. Occasionally, the server updates the global statistics to adjust its decisions.

When forking a new VM instance, the native system agent daemon connects to the GC service. The agent inserts all GC events into the global queue, waiting for a message from the server that defines actions to be executed (i.e., GC or trimming).

6.3.2 Communication with applications

Although Android provides the binder as an Interprocess communication (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 [33; 79]; (*ii*) Binder restricts the maximum number of calls that can be handled concurrently (currently 16); and (*iii*) Shared memory makes our system portable and independent of platform-specific features.

IPC between the server and the *client* is based on asynchronous message queues to achieve efficient handshaking with a minimum overhead. IPC messages and signals are implemented using *futexes* [45] to synchronize in user space. The server utilizes a pool of work-stealing threads to reduce the overhead of thread creation.

IPC Overhead. IPC between the GC service and VMs is based on priority queues. This communication mechanism is an efficient way to support various heuristics—i.e., processing the foreground application with higher priority. The longest duration of time a request stays pending can be represented as a function $latency(IPC) = f(\upsilon, \eta, \rho)$, 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. In the GCService, latency is measured by the time difference between posting the request and responding to it. This value is used to add an extra heap *room* to allow enough time to respond to concurrent requests.

Although process context switching overhead is known to be high compared to thread switching, profiling of scheduling statistics on Android ART with the GC service shows this to be negligible. A possible explanation is that each GC loads completely different code into the caches and touches a lot of memory pages, so the cost of switching processes is in the noise.

6.3.3 Energy optimization

Power governors (e.g., *ondemand*) control the energy consumption of the multicore processor based on observed workload. The governor collects run-time statistics and applies heuristics in an attempt to meet optimization criteria. We integrate between the GC service and 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 [34].

Furthermore, 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 maximum frequency* as the median between the current core frequency and the governor *optimal_freq* (see Table 3.2). Following the collection cycle, the governor is free to adjust the frequency according to the observed workload and the default settings. GC service coordination with power managers differs from local power optimizations that may inherit conflicting GC scheduling decisions across concurrent VMs [62].

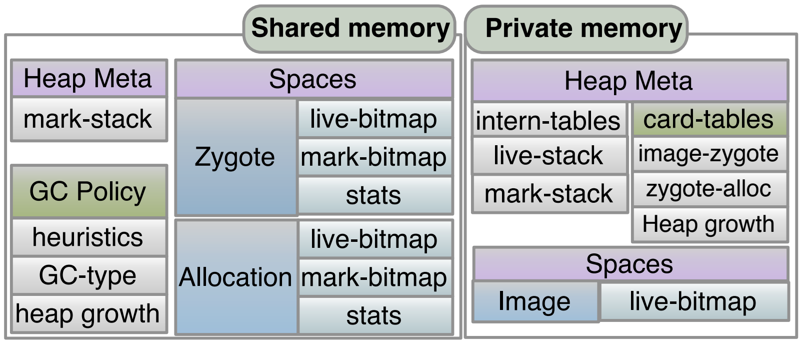


Figure 6.2.: Shared heap layout in the GC service

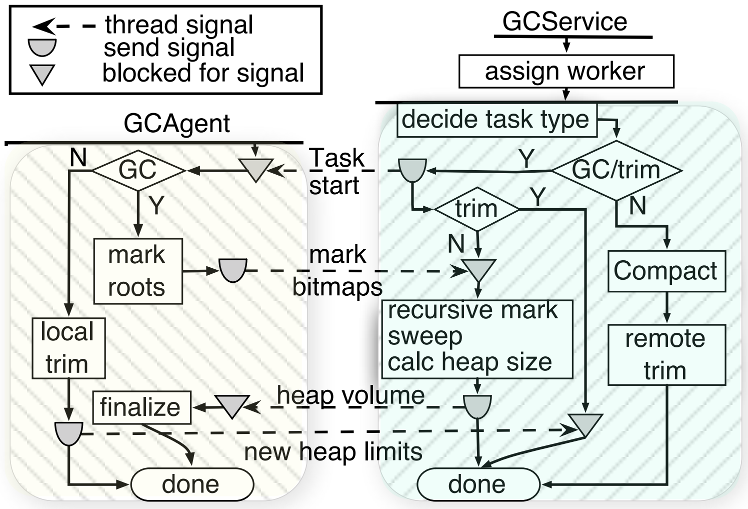


Figure 6.3.: Control flow of tasks in the GC service

6.3.4 Memory layout

The heap layout of a single ART 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, (*iv*) large-object-space (LOS): contains all objects larger than *LOS-threshold*. The space is a linked list of memory mapped pages.

ART keeps two bitmaps (live and mark) for each individual space, and a global allocation stack to keep track of newly allocated objects. *Card table* records dirty objects. Inter-space references are stored in internal tables to reduce the overhead of tracing the reachable objects in the active space. Finally, the *mark-stack* stores the marked objects that need to be scanned during the concurrent marking.

The client VM starts by creating a GC agent daemon, which registers the VM with the service by sending the VM process id (PID) and the continuous space addresses. Thus, a subset of heap metadata, zygote, and allocation spaces become accessible to the service. Figure 6.2 shows the shared memory layout in the new system.

Figure 6.3 shows how collection is handled in the GC service. The app-local agent first marks the heap roots including those from thread stacks and globals. The shared *mark-bitmap* is then used by the server to mark the reachable objects recursively. The agent takes control once more at the end of the mark phase to *stop-the-world* and revisit any remaining dirty objects pushed on the *mark-stack* due to concurrent updates by the client application threads (mutators).

The server sweeps the space to create the list of free objects, computes fragment distributions, and calculates the new size of the heap. Finally, the agent enforces the new heap threshold before finalizing the GC cycle. Since handling the dirty objects requires a stop-the-world step, we avoid IPC between the agent and the server to assure that the Java threads are resumed in a short period of time.

On the server side, the collector recalculates the reference fields based on the base address of the mapped heap. The server scans the reachable objects except for a small set allocated in the client's private memory range. If the adjusted address does not belong to the mapped range (shared space), the server decides to add the object to a "*delayed*" list. The GC agent processes the delayed list as a subset of the dirty objects.

Space Partitioning Tradeoffs. Android ART puts a new object in the LOS when its absolute size exceeds a predefined *large object threshold*. The current GC service implementation limits the heap to contiguous memory regions, causing a slightly higher overhead

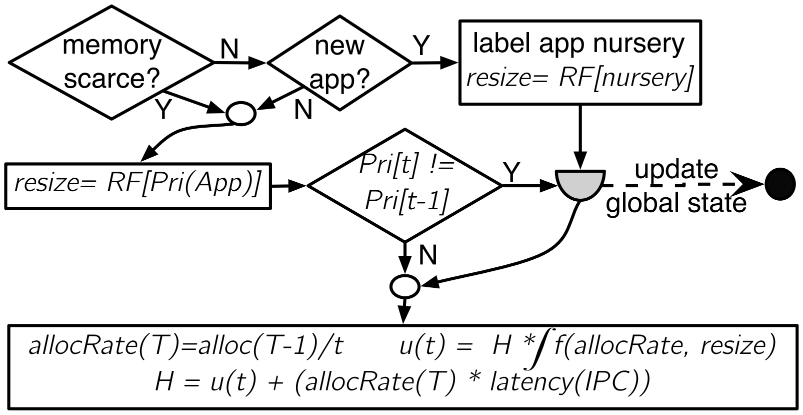


Figure 6.4.: Heap growth procedure following a GC task

when scanning the heap. Section 6.4.1 shows that very few object allocations (less than 1%) are affected by the removal of LOS.

6.3.5 Heap size management

Mobile apps often exhibit an execution pattern that makes static GC policies ineffective. For 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 as follows: (*i*) an app in start-up phase is able to grow the heap more aggressively; (*ii*) app priority is a factor for allowed heap growth, resize = Pri(app); (*iii*) for each VM, the GC service stores allocation and resizing statistics of the last 20 events; (*iv*) the GC service dynamically calculates the heap size using a predefined controller to prevent heap size [112] oscillation.

Figure 6.4 illustrates the steps followed by the GC service to resize the app's heap. The service uses the memory allocation/collection rates to auto-adjust the heap growth policy dynamically [112]. This allows for identifying the steady state of the heap volume in smooth steps that eliminate inefficient heap bounds. The GC service updates the global statistics when a new app is created, or when an important app changes status, possibly being replaced by another app. Whenever spare cycles are available, the server daemon picks a low priority app labeled to have a sparse heap (fragment ratio) from the process list to perform an offline compaction.

Offline 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 [95], 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, the GCService keeps statistics about empty slots following each full GC. When memory becomes scarce, the GCService *lazily* picks the VM with the highest fragmentation score in the list of low priority VMs. Once picked, the inactive app can undergo a safe *offline compaction*.

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, each VM needs to perform the compaction task, implying that the process changes its state from *inactive* to *running*. GC compaction also requires significant per-VM overhead to store the forwarding references (space overhead) and to synchronize attempts to access the moved objects [11; 27].

Taking advantage of the fact that the VM is already inactive, the GC service performs the compaction in an *offline* mode without the need to synchronize on forwarding references. Since the collector process is already *running*, the GC service avoids signaling an inactive process, which would otherwise decrease global CPU and memory utilization.

6.4 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. The default configuration appears in Table 3.2. For all applications, we use the *Monkeyrunner* tool to automate user inputs [3]. We leverage our prototype implementation described in Section 6.3.

We use the APQ8074 DragonBoard hardware Development Kit based on Qualcomm’s Snapdragon S4 SoC. The S4 uses the quad-core 2.3GHz Krait CPU. Caches are 4KiB + 4KiB direct mapped L0 cache, 16KiB + 16KiB 4-way set associative L1 cache, and 2MiB 8-way set associative L2 cache. The total memory available is 2GiB.

Consistent lightweight profiling. 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*.

To avoid perturbing application threads (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 because of the temperature.

Power Profiling. We measure the total physical energy consumed during the app execution and we correlate the results 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.

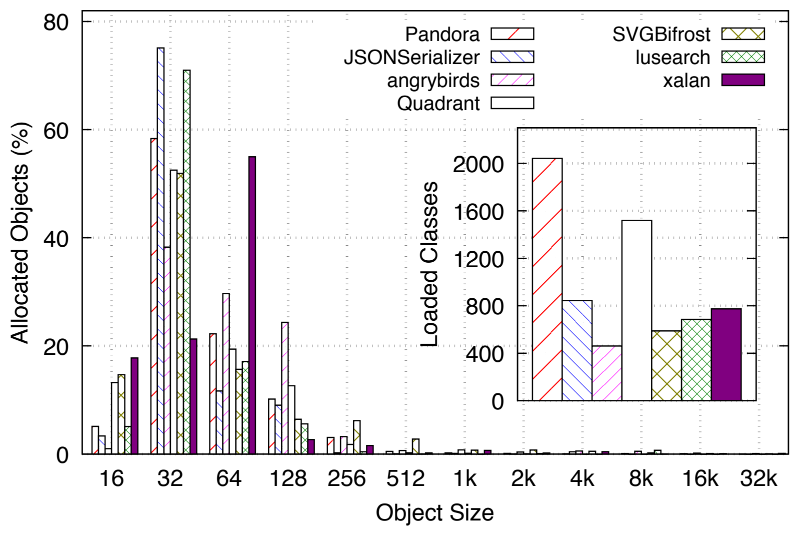


Figure 6.5.: Object size histograms and loaded classes

6.4.1 Workload

We developed a set of applications that wrap popular Android libraries allowing various workload sizes and number of iterations. We profile the object size demographics in the heap in a perfectly compacted heap (64KiB). Figure 6.5 plots the percentage of objects (*y*-axis) in each object size (*x*-axis) that each app allocates. The number of loaded classes reflects the variance in object types.

6.4.2 Global evaluation

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 3.2, the LMK starts killing processes from the lowest priority group.

Microtask Evaluation. The default Android memory system is tuned for single monolithic applications. First, after 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

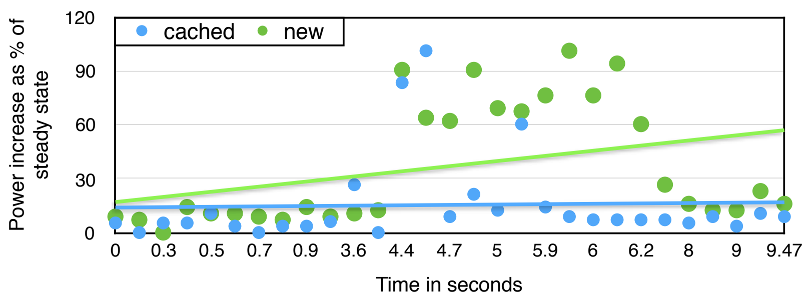


Figure 6.6.: Power trends when launching apps for first time (new) vs. cached apps

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 [48]. Killing processes is especially problematic for apps that are designed to run in the background like music players.

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. Killing 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 6.6 demonstrates that re-launching apps that were killed by the LMK has a large impact on energy consumption. In addition, our experiments reveal that local GC trimming tasks increase the power leaks for apps running in the background.

Case-A: Sequential App Execution. 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.

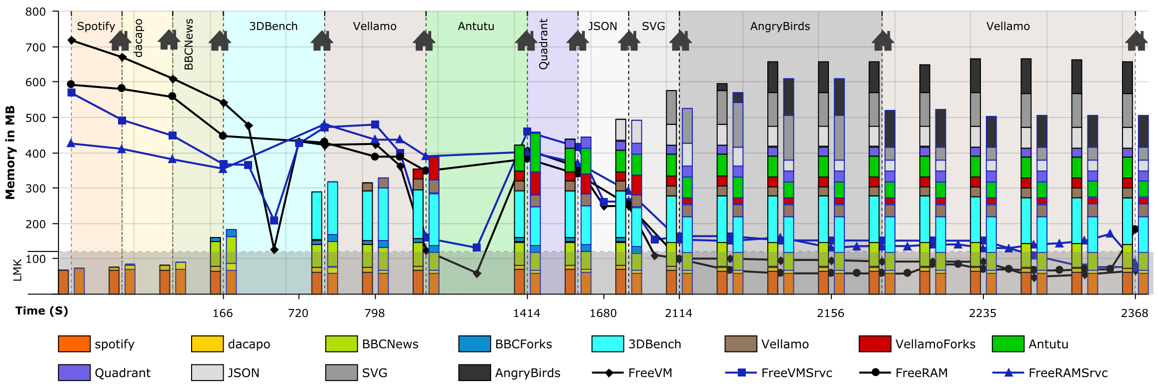


Figure 6.7.: Memory stats vs. time in seconds: Stacked bars are the total apps memory in ART (left) and the GC service (right); free VMStat (Free-VM/Srvc); available RAM (FreeRAM/Srvc); the LMK range; and the current foreground app (vertical guides)

This benchmark launches several apps, switching between them by pressing the *home* button. When an app $A_i a$ 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 with ART kills 27 processes including Browser, and BBCNews-Service. The GC service, on the other hand, reduces the number of killed processes to only 14–19 (depending on the individual run) without coordination with the *Android run-time manager*.

Figure 6.7 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 for running with and without the GC service. The service (right column) reserves more memory for the foreground app as a result of the heuristics allowing 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. Default Android ART tends to kill more apps

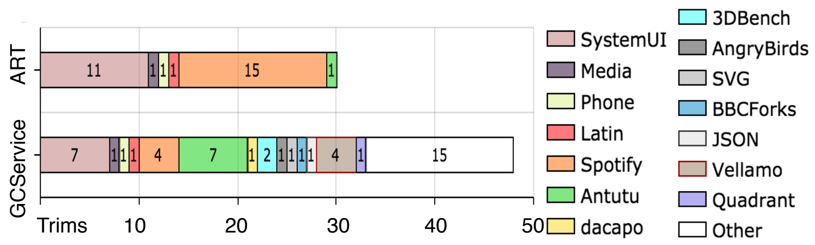


Figure 6.8.: Stacked trim counts per app during runtime: GCService trims more apps

Table 6.1.: App execution time in foreground and background modes (Background OFF/ON respectively)

Apps	VM	Background	Iterations	Average (s)	Geo. Mean (s)	Conf. Interval (ms, 5%)	Standard Dev. (ms)	Trim Count
lusearch	ART	ON	10	7.09	7.08	45.42	208.58	0
		OFF	10	6.65	6.65	19.85	50.65	36
	GCService	ON	10	6.76	6.76	34.42	118.58	0
		OFF	10	6.49	6.49	35.47	90.48	0
xalan	ART	ON	10	11.42	11.38	220.07	1,010.53	0
		OFF	10	10.27	10.26	155.05	395.54	48
	GCService	ON	10	9.60	9.59	65.69	301.64	0
		OFF	10	8.79	8.79	22.23	56.71	0
sqllite	ART	ON	6	155.01	155.01	71.46	182.30	0
		OFF	6	153.42	153.42	136.41	347.97	56
	GCService	ON	6	157.50	157.50	270.32	646.97	0
		OFF	6	156.60	156.60	212.59	542.32	0

as a result of not reclaiming memory from inactive apps. Figure 6.8 shows the number of trims performed by the apps (excluding `System` processes).

6.4.3 Local per-VM evaluation

Case-B: Sending Top App to Background. This experiment allows for assessing 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 6.9 shows the execution time and power results, with confidence interval (5%), of running each benchmark for a given number of iterations fol-

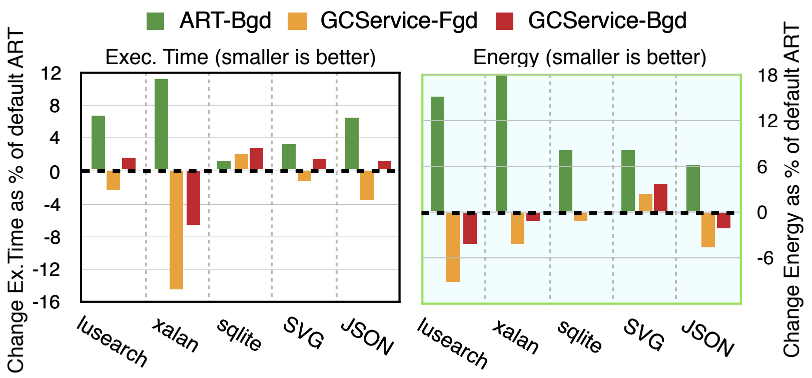


Figure 6.9.: App exec. time & energy in foreground and background modes compared to the default execution

lowing warmup under the two different Android systems. For Android ART, apps running in background exhibit considerable trimming, slowing down app execution and leaking more energy. Table 6.1 shows the execution time results, with confidence interval (5%), of running each benchmark for a given number of iterations following the warmup under the two different Android systems. For Android ART, apps running in background exhibit considerable trimming slowing down app execution. The *trim count* indicates the frequency of trimming done by each app during the execution time. Our experiments show that trimming phase may span up to 0.6s.

The GC service does not perform any trimming because the memory is not stressed, resulting in less GC overhead. Finally, LMK is triggered frequently on Android ART killing several apps. Using the same LMK configurations, the number of killed apps is reduced by 70% for the GC service. Note that sqlite exhibits a slow down as a tradeoff between responsiveness tuning and execution time as we address the responsiveness evaluation.

Responsiveness. GC pauses apps to mark live objects and free unused memory. For mobile devices, pauses greater than 50 ms can be perceived by the users and degrade the experience of animation-based frames [34; 41].

We instrumented, for each thread, the pause segments during execution. Figure 6.10 shows the GC pauses on the GC service, normalized to default Android ART. The *worst-*

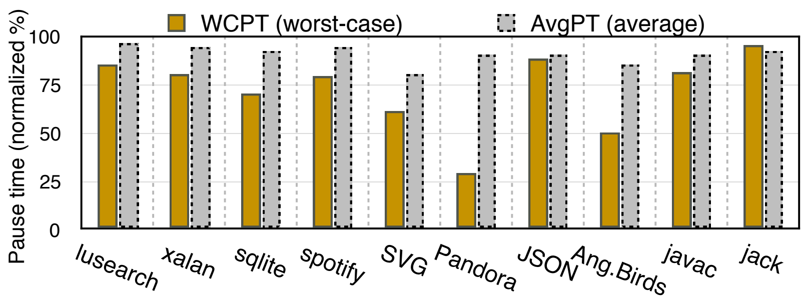


Figure 6.10.: Worst case and average pauses (WCPT and AvgPT) in GC service as % of respective ART pauses

case pause time (WCPT) is the maximum pause caused by GC operations that is recorded by any mutator thread during the execution time. The average pause (AvgPT) represents the average of all pauses by all the mutator threads.

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 on 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.

Case-C: Analyzing App Behavior. Users flag apps as battery and memory drainers when they cause issues on the device. We analyze the memory behavior of Spotify, frequently flagged by users as a drain on the battery [8], in order to explore possible causes of the memory and power leaks.

Our script launches Spotify, enters the login credentials, 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.

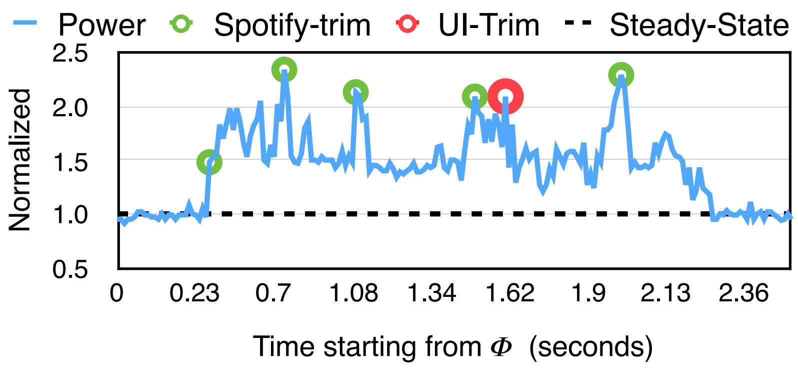


Figure 6.11.: The impact of trimming on power when running Spotify in the background normalized to the steady state

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 tasks in the background. This increases the GC overhead to 10% of the total CPU cycles. Although, the increase of GC overhead looks small, note that Spotify gets less CPU slots when it falls to the background based on Android scheduling policies.

We calculate the *average* steady state power consumption as a baseline and we correlate between the measurements and the GC events. Figure 6.11 shows a time window (starting at time ϕ) obtained when Spotify is pushed to the background, demonstrating the high cost of heap trimming. Occasionally, the *system-UI* executes GC trims following Spotify trimming events.

To explain the high frequency of trimming tasks, we profile the heap variables and the distribution of free slots following each concurrent cycle. The results reveal that the trim operations are not effective as the gaps after collecting small size objects do not form contiguous memory chunks that can be released to the system (see Figure 6.5). Figure 6.12 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.

Table 6.2.: Context switching overhead

	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.	43.41μs	Thread Ctx.	9.56μs

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 improved the resizing decisions by removing steps that reach a local maximum. *(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.

6.4.4 Interprocess overhead

It is important to get an estimate of the overhead of context switching in order to be able to do further tuning. For, e.g., our decision to avoid IPC communication while handling the dirty objects was based on the knowledge of the cost of process context switching. Table 6.2 shows the results of characterizing OS and HW strengths. The metrics include: TLB, Cache, and process context switch performance.

6.5 Summary

Mobile devices pose novel challenges to system designers as they juggle access to limited resources like battery usage against app performance and reactiveness to user requests. The Android system is running dozens of concurrent VMs, each running an app on a single

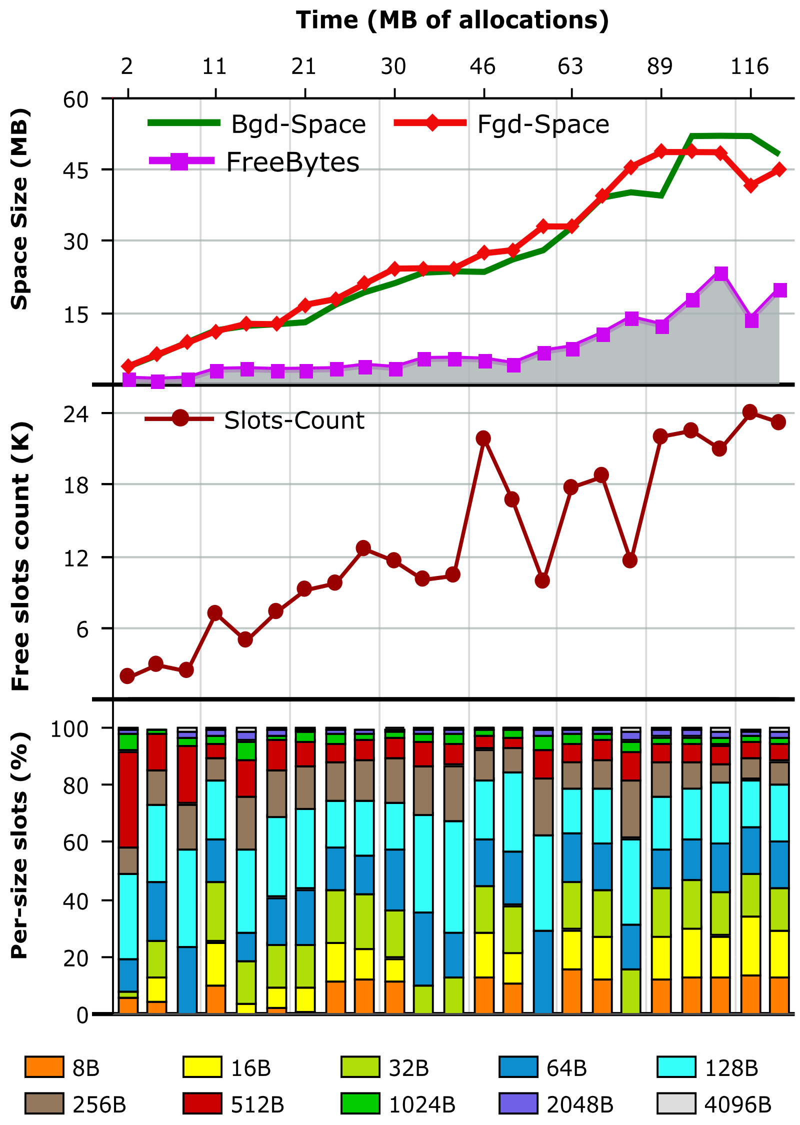


Figure 6.12.: Heap characteristics of Spotify vs. time measured in bytes allocated: (1) The heap size in background and foreground modes (Bgd/Fgd-Space) and the free slots volume (top). (2) The count of free slots (middle). (3) The stacked histogram of empty slots grouped by their size (bottom).

device in a constrained environment. Unfortunately, the mobile system so far treats each VM as a monolithic instance.

We have introduced a VM design that allows a central service to observe performance critical parameters of concurrent but independent VMs and carry out decisions optimized across the whole system instead of just locally. Our prototype addresses a major resource bottleneck, memory, by presenting a central GC service that evaluates GC decisions across all running VMs and optimizes for a global set of heuristics. Here, we present a first set of sensible heuristics but further research and tuning is necessary. 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 background apps, (*v*) it reduces the number of processes killed by the system LMK by returning more pages, (*vi*) it saves device resources during memory recycling, and (*vii*) it reduces the GC space overhead per VM—for example, instead of allocating internal data structures for each VM, heap structures are allocated by the global service.

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

7 FUTURE WORK

We show that GC behavior varies with workload. Our extensions serve as a platform to understand the implications of some major design decisions such as concurrency. Further tuning of all GC parameters is left for future work.

7.1 Tuning the GC Service

We described extensions to the default heuristics that we apply in our centralized GC service to demonstrate the system capabilities. We introduced a *sensible* first set of heuristics that take advantage of global statistics and improve the status quo. The GC service can be extended through a plugin-based system that allows for customization of its policies.

Although, the first set of heuristics are satisfying, there are some aspects that can be enhanced in the future:

1. Consider an efficient scheme to map the memory pages in the GC service. A possible approach is to directly insert the new pages into the service page table.

2. Analyze the app usage to assist the GC service. Our GC service is capable of providing several important and useful information about how 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).

3. Develop an adaptive formula for the optimal core frequency during garbage collection.

7.2 Code Optimization Service

While JIT is considered expensive as a standalone daemon, having a centralized compilation service that receives requests from other VMs and keeps a cache of the common used classes and libraries will be significantly powerful.

Moreover, our work can be extended beyond GC. One such example is to manage code optimization and share the jitted code efficiently between running VMs. A JIT service will allow sharing of native code between VMs which reduces per-application footprint and avoids repeated compilation of common code. A service JIT will work as an Ahead-of-Time (AOT) service for newly loaded applications. With Android ART adopting AOT, a JIT-service can be used to tune native code across several apps with minimum amortized cost.

7.3 Security

Security on mobile platforms is a critical point, and we believe that it is important to deeper evaluate the system from a security perspective. We considered security throughout the design and implementation phases as we explained earlier.

During the course of building our GCService we have carefully considered different designs, excluding the ones that comprise the security, e.g., not sharing zygote space over security. The design of the GC service also carries security implications, because it has access to all heaps. Thus, the code of the central GC service must be trusted. Each VM, on the other hand, has access to only its own local heap. Since Android allows for execution of native code, this becomes an important security property compared to previous approaches like *Multi-tasking VMs* (MVMs) that allow a single VM to run multiple applicationson a single heap [31; 117]. Our approach assures *reliability* by allowing for individual VM to revert back to the default standalone GC whenever the GC service fails to respond to memory requests.

The GCService needs extended system permissions to access the platform API and other system resources (i.e., power managers and process stats). However, having the GC-

Service running as an Android VM, the security and isolation mechanisms still apply to the new platform. There are two different approaches to manage the Zygote space: *(i)* to keep a zygote per application, or *(ii)* to share immuned objects between different applications.

The second option has an obvious advantage in memory optimization. However, we have decided to keep a zygote per application to create a platform independent from the memory layout of the system. Since Android supports layout randomization (ASLR), our objective has been to create a flexible system, without the assumption of fixed memory layouts. Hence, the proposed architecture meets the security principles defined for the stock Android.

8 CONCLUSIONS

Mobile devices pose novel challenges to system designers as they compete to limited resources like battery usage against app performance and responsiveness. The Android system runs dozens of VMs, each running a different app on a single device in a constrained environment. Unfortunately, the default Android configuration treats each VM as a monolithic instance. This work is a first step to analyze the GC within the system scope to serve as a guide to evaluating the coordination between design decisions across all the layers of the system stack (software and hardware).

8.1 GC Impact on Android devices

We show that different GC strategies have highly varying energy requirements that do not always correlate with the app throughput. Varying policies, such as heap growth or concurrency, can either significantly reduce the energy consumed or can reduce the worst-case pause time, but not at the same time [63]. Moreover, the app throughput is not necessarily correlated with power consumption. GC work is inherently memory-bound but current governor heuristics focus on the system load and do not incorporate the execution profile into their decisions. Our results imply that existing DVFS policies should be informed of GC events by the VM to make more informed hotplugging and frequency scaling decisions [62]. Similarly, app developers need a range of GC strategies to choose from, so they can tune for responsiveness, utilization, and power consumption.

8.2 GC As A Service

We introduce a VM design that allows a central service to observe performance critical parameters of concurrent yet independent VMs and carry out decisions optimized across the whole system, instead of just locally [64]. Our prototype then addresses a major resource bottleneck (i.e., memory) by presenting a central GC service that evaluates GC decisions across all running VMs and optimizes for a global set of heuristics. The new system aims at reducing the latency of app responses while assuring better performance and longer (battery) lifetimes. This is achieved without compromising the integrity of the platform.

We believe that the central management of VM services on a mobile device will open up other research topics like code optimization. Such a central removes redundancy and orthogonal conflicting local heuristics, replacing them with a global alternative.

8.3 Benchmarking and Evaluation Methodology

GC on mobile platforms is challenging due to the adaptive nature, the workload size, and the environmental restrictions of the programs. Thus, GC evaluations must consider management mechanisms across the stacks in order to obtain precise and relevant conclusions regarding the GC impact on user experience. Controlling GC strategies induces a large variation of the total on-chip energy consumed by the app, and worst-case pause times. This shows that GC has a significant impact on battery life and app responsiveness; in other words, GC directly affects the user experience.

We urge researchers and industry workers to develop a common platform with a transparent access to different system layers. Such a design simplifies the task of evaluating new techniques.

Writing power-aware source code is not a feasible option due to a widely heterogeneous hardware and software. Our results show that code optimizations are specific to the default system configurations (i.e., heap size, and concurrency). Energy optimizations can be achieved by simple modifications to both run-time and system layers. For example, extending the VM to dynamically enable/disable the GC daemon to balance between synchronization overhead and the mutator utilization can lead to an adaptively tuned performance. Similarly, heap growth policies need to be integrated with DVFS decisions to achieve a better energy consumption than heuristics based on only memory footprint.

To address these challenges, we describe a systematic approach that tames individual feedback systems, reducing variations across experiments by disabling thermal throttling, adaptive governors, and unneeded system services [64]. In addition, we ensure stable conditions by controlling the system image and parameters the experiment runs in. We capture the discussed metrics by collecting *(i)* fine-grained measurements of the power at a microprocessor level, *(ii)* detailed performance counters data (on demand), *(iii)* system events, and *(iv)* VM events, correlating all of them across the experiment.

There were no standard benchmarks available for mobile platforms. Therefore, we presented a benchmark suite (*Etalon*). The apps provided by Etalon are helpful for system developers interested in evaluating VM components (e.g., garbage collection or compiler optimizations). In addition, the apps may exhibit behaviors (e.g., scalability and concurrency) that existing Android apps do not (yet) display.

REFERENCES

REFERENCES

[1] *ACS714: Automotive Grade, Fully Integrated, Hall effect-based linear current sensor IC with 2.1 kVRMS Voltage isolation and a low-resistance current conductor*. Allegro MicroSystems, LLC, 2016. URL http://www.pololu.com/product/1185.

[2] A. Anantaraman, K. Seth, K. Patil, E. Rotenberg, and F. Mueller. Virtual simple architecture (VISA): Exceeding the complexity limit in safe real-time systems. In *Proceedings of the International Symposium on Computer Architecture*, ISCA'03, pages 350–361, June 2003. doi: 10.1109/ISCA.2003.1207013.

[3] *monkeyrunner tool*. Android, 2016. URL https://developer.android.com/studio/test/monkeyrunner/index.html.

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

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

[6] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI'14, pages 259–269, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594299.

[7] I. Assayad, A. Girault, and H. Kalla. Tradeoff exploration between reliability, power consumption, and execution time. In *Proceedings of the International Conference on Computer Safety, Reliability, and Security*, pages 437–451, Naples, Italy, 2011. doi: 10.1007/978-3-642-24270-0_32.

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

[9] D. F. Bacon, P. Cheng, and S. Shukla. And then there were none: A stall-free real-time garbage collector for reconfigurable hardware. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI'12, pages 23–34, Beijing, China, June 2012. doi: 10.1145/2254064.2254068.

[10] D. F. Bacon, P. Cheng, and S. Shukla. Parallel real-time garbage collection of multiple heaps in reconfigurable hardware. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management*, ISMM'14, pages 117–127, Edinburgh, Scotland, June 2014. doi: 10.1145/2602988.2602996.

[11] 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, November 2012. doi: 10.1145/2362389.2362392.

[12] 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 *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'06, pages 169–190, Portland, Oregon, October 2006. doi: 10.1145/1167473.1167488.

[13] S. M. Blackburn, R. Garner, C. Hoffman, A. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, S. Z. Guyer, A. Hosking, M. Jump, J. E. B. Moss, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Widerman. Wake up and smell the coffee: Evaluation methodology for the 21st century. *Communications of the ACM*, 51(8):83–89, August 2008. doi: 10.1145/1378704.1378723.

[14] *LoganSquare source code*. blueline labs, 2015. URL https://github.com/bluelinelabs/LoganSquare.

[15] T. Book, A. Pridgen, and D. S. Wallach. Longitudinal analysis of Android Ad library permissions. *Mobile Security Technologies*, abs/1303.0857, 2013. URL http://arxiv.org/abs/1303.0857.

[16] T. Brecht, E. Arjomandi, C. Li, and H. Pham. Controlling garbage collection and heap growth to reduce the execution time of Java applications. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'01, pages 353–366, Tampa, Florida, November 2001. doi: 10.1145/504282.504308.

[17] 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, September 2006. doi: 10.1145/1152649.1152652.

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

[19] C. Cameron, J. Singer, and D. Vengerov. The judgment of Forseti: Economic utility for dynamic heap sizing of multiple runtimes. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management*, ISMM'15, pages 143–156, Portland, Oregon, June 2015. doi: 10.1145/2754169.2754180.

[20] 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 *Proceedings of the International Symposium on Computer Architecture*, ISCA'12, pages 225–236, Portland, Oregon, June 2012. IEEE Computer Society. doi: 10.1109/ISCA.2012.6237020.

[21] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen. EdgeMiner: Automatically detecting implicit control flow transitions through the Android framework. In *Proceedings of the ISOC Network and Distributed System Security Symposium*, NDSS'15, 2015.

[22] A. Carroll and G. Heiser. Mobile multicores: Use them or waste them. In *Proceedings of the Workshop on Power-Aware Computing and Systems*, HotPower, November 2013. doi: 10.1145/2626401.2626411.

[23] A. Carroll and G. Heiser. Unifying DVFS and offlining in mobile multicores. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS'14, pages 287–296, Berlin, Germany, April 2014. doi: 10.1109/RTAS.2014.6926010.

[24] *mongoose*. CESANTA Embedded Communication, 2015. URL https://www.cesanta.com/products/mongoose.

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

[26] 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, November 2002. doi: 10.1145/581888.581892.

[27] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970. doi: 10.1145/362790.362798.

[28] P. Cheng and G. E. Blelloch. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI'01, pages 125–136, Snowbird, Utah, June 2001. doi: 10.1145/378795.378823.

[29] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *Proceeding of the ACM International Conference on Mobile Systems, Applications, and Services*, MobiSys'11, pages 239–252, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0643-0. doi: 10.1145/1999995.2000018. URL http://doi.acm.org/10.1145/1999995.2000018.

[30] *Power Choke Coil PIFE20161T Type*. CYNTEC Co., Ltd., 2014. URL http://www.cyntec.com/product/hp_choke/download/PIFE20161T.pdf.

[31] G. Czajkowski and L. Daynés. Multitasking without compromise: A virtual machine evolution. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'01, pages 125–138, Tampa Bay, FL, USA, 2001. doi: 10.1145/504282.504292.

[32] S. K. Datta, C. Bonnet, and N. Nikaein. Usage patterns based security attacks for smart devices. In *Proceedings of the IEEE International Conference on Consumer Electronics*, ICCE'14, pages 284–287, September 2014.

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

[34] U. Degenbaev, J. Eisinger, M. Ernst, R. McIlroy, and H. Payer. Idle time garbage collection scheduling. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI'16, San Jose, California, 2016.

[35] S. Dieckmann and U. Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP'99, pages 92–115, Lisbon, Portugal, July 1999. doi: 10.1007/3-540-48743-3_5.

[36] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978. doi: 10.1145/359642.359655.

[37] T. M. T. Do and D. Gatica-Perez. Where and what: Using smartphones to predict next locations and applications in daily life. *Pervasive and Mobile Computing*, 12: 79 – 91, 2014. ISSN 1574-1192. doi: 10.1016/j.pmcj.2013.03.006.

[38] T. Domani, E. K. Kolodner, E. Lewis, E. E. Salant, K. Barabash, I. Lahan, Y. Levanoni, E. Petrank, and I. Yanorer. Implementing an on-the-fly garbage collector for Java. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management*, ISMM'00, pages 155–166, Minneapolis, Minnesota, October 2000. doi: 10.1145/362422.362484.

[39] X. Dong, S. Dwarkadas, and A. Cox. Characterization of shared library access patterns of Android applications. In *proceedings of the IEEE International Symposium on Workload Characterization*, IISWC'15, pages 112–113, October 2015. doi: 10.1109/IISWC.2015.19.

[40] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'03, pages 149–168, Anaheim, California, October 2003. doi: 10.1145/949305.949320.

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

[42] H. Esmaeilzadeh, T. Cao, Y. Xi, S. M. Blackburn, and K. S. McKinley. Looking back on the language and hardware revolutions: Measured power, performance, and scaling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 319–332, Newport Beach, California, March 2011. doi: 10.1145/1950365.1950402.

[43] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin. Diversity in smartphone usage. In *Proceeding of the ACM International Conference on Mobile Systems, Applications, and Services*, MobiSys'10, pages 179–194, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-985-5. doi: 10.1145/1814433.1814453.

[44] *Jackson Project*. FasterXML, 2015. URL https://github.com/FasterXML/jackson.

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

[46] A. Gautham, K. Korgaonkar, P. Slpsk, S. Balachandran, and K. Veezhinathan. The implications of shared data synchronization techniques on multi-core energy efficiency. In *Proceedings of the USENIX Conference on Power-Aware Computing and Systems*, HotPower, Hollywood, California, October 2012. URL https://www.usenix.org/system/files/conference/hotpower12/hotpower12-final40.pdf.

[47] *Multitasking the Android Way*. Google Inc. URL http://android-developers.blogspot.kr/2010/04/multitasking-android-way.html.

[48] *AOSP Issue Tracker*. Google Inc., 2015. URL https://code.google.com/p/android/issues/detail?id=98332.

[49] *google-gson*. Google Inc., 2015. URL https://github.com/google/gson.

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

[51] *ART and Dalvik*. Google Inc., 2016. URL https://source.android.com/devices/tech/dalvik/art.html.

[52] M. Graa, N. Cuppens-Boulahia, F. Cuppens, and A. Cavalli. Detecting control flow in smartphones: Combining static and dynamic analyses. In *Proceedings of the International Conference on Cyberspace Safety and Security*, CSS'12, pages 33–47, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-35361-1. doi: 10.1007/978-3-642-35362-8_4.

[53] P. Griffin, W. Srisa-an, and J. M. Chang. An energy efficient garbage collector for Java embedded devices. In *Proceedings of the ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES'05, pages 230–238, Chicago, Illinois, June 2005. doi: 10.1145/1065910.1065943.

[54] *The Mobile Economy*. GSMA, 2015. URL http://gsmamobileeconomy.com/global/GSMA_Global_Mobile_Economy_Report_2015.pdf.

[55] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE International Workshop on Workload Characterization*, WWC'01, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7803-7315-4. doi: 10.1109/WWC.2001.15.

[56] A. Gutierrez, R. G. Dreslinski, T. F. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver. Full-system analysis and characterization of interactive smartphone applications. In *proceedings of the IEEE International Symposium on Workload Characterization*, IISWC'11, pages 81–90, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-4577-2063-5. doi: 10.1109/IISWC.2011.6114205.

[57] S. Hao, D. Li, W. G. Halfond, and R. Govindan. SIF: A selective instrumentation framework for mobile applications. In *Proceeding of the ACM International Conference on Mobile Systems, Applications, and Services*, MobiSys'13, pages 167–180, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1672-9. doi: 10.1145/2462456.2465430.

[58] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating mobile application energy consumption using program analysis. In *Proceedings of the International Conference on Software Engineering*, ICSE'13, pages 92–101, San Francisco, California, May 2013. IEEE Press. doi: 10.1109/ICSE.2013.6606555.

[59] R. Hay, O. Tripp, and M. Pistoia. Dynamic detection of inter-application communication vulnerabilities in Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA'15, pages 118–128, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3620-8. doi: 10.1145/2771783.2771800.

[60] K. Huang, C. Zhang, X. Ma, and G. Chen. Predicting mobile application usage using contextual information. In *Proceedings of the ACM Conference on Ubiquitous Computing*, UbiComp'12, pages 1059–1065, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1224-0. doi: 10.1145/2370216.2370442.

[61] Y. Huang, Z. Zha, M. Chen, and L. Zhang. Moby: A mobile benchmark suite for architectural simulators. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS'14, pages 45–54, March 2014. doi: 10.1109/ISPASS.2014.6844460.

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

[63] A. Hussein, M. Payer, A. Hosking, and C. A. Vick. Impact of GC design on power and performance for Android. In *Proceedings of the International Systems and Storage Conference*, SYSTOR'15, pages 13:1–13:12, Haifa, Israel, 2015. doi: 10.1145/2757667.2757674.

[64] A. Hussein, M. Payer, A. Hosking, and C. A. Vick. One process to reap them all. In *Submission to the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'17, 2017.

[65] A. Iyer and D. Marculescu. Power efficiency of voltage scaling in multiple clock, multiple voltage cores. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, ICCAD'02, pages 379–386, San Jose, California, November 2002. doi: 10.1145/774572.774629.

[66] N. Jacek, M.-C. Chiu, B. Marlin, and E. Moss. Assessing the limits of program-specific garbage collection performance. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI'16, pages 584–598, Santa Barbara, California, June 2016. doi: 10.1145/2908080.2908120.

[67] M. R. Jantz, F. J. Robinson, P. A. Kulkarni, and K. A. Doshi. Cross-layer memory management for managed language applications. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'15, pages 488–504, Pittsburgh, Pennsylvania, 2015. doi: 10.1145/2814270.2814322.

[68] *JSR 121: Application Isolation API Specification*. Java Community Process, 2016. URL https://jcp.org/en/jsr/detail?id=121.

[69] Y. Jing, G.-J. Ahn, A. Doupé, and J. H. Yi. Checking intent-based communication in Android with intent space analysis. In *Proceedings of the ACM on Asia Conference on Computer and Communications Security*, ASIA CCS'16, pages 735–746, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4233-9. doi: 10.1145/2897845.2897904.

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

[71] T. Kalibera and R. Jones. Rigorous benchmarking in reasonable time. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management*, ISMM'13, pages 63–74, Seattle, Washington, June 2013. doi: 10.1145/2464157.2464160.

[72] T. Kalibera, M. Mole, R. Jones, and J. Vitek. A black-box approach to understanding concurrency in DaCapo. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'12, pages 335–354, Tucson, Arizona, October 2012. doi: 10.1145/2384616.2384641.

[73] M. Kambadur and M. A. Kim. An experimental survey of energy management across the stack. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'14, pages 329–344, Portland, Oregon, October 2014. doi: 10.1145/2660193.2660196.

[74] J. M. Kang, S. S. Seo, and J. W. K. Hong. Usage pattern analysis of smartphones. In *Proceedings of the Network Operations and Management Symposium*, APNOMS, pages 1–8, 2011. doi: 10.1109/APNOMS.2011.6077030.

[75] M. Karami, M. Elsabagh, P. Najafiborazjani, and A. Stavrou. Behavioral analysis of Android applications using automated instrumentation. In *Proceedings of the IEEE International Conference on Software Security and Reliability-Companion*, SERE-C'13, pages 182–187, June 2013. doi: 10.1109/SERE-C.2013.35.

[76] S.-H. Kim, S. Kwon, J.-S. Kim, and J. Jeong. Controlling physical memory fragmentation in mobile systems. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management*, ISMM'15, pages 1–14, Portland, Oregon, 2015. doi: 10.1145/2754169.2754179.

[77] P. Lengauer and H. Mössenböck. The taming of the shrew: Increasing performance by automatic parameter tuning for Java garbage collectors. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ICPE'14, pages 111–122, Dublin, Ireland, 2014. ISBN 978-1-4503-2733-6. doi: 10.1145/2568088.2568091.

[78] J. Levin. *Android Internals — Volume I: A Confectioner's Cookbook*. Jonathan Levin, 2014. ISBN 9780991055524. URL http://newandroidbook.com/index.php.

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

[80] H. Li, X. Lu, X. Liu, T. Xie, K. Bian, F. X. Lin, Q. Mei, and F. Feng. Characterizing smartphone usage patterns from millions of Android users. In *Proceedings of the ACM Conference on Internet Measurement Conference*, IMC'15, pages 459–472, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3848-6. doi: 10.1145/2815675.2815686.

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

[82] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. Mining energy-greedy api usage patterns in Android apps: An empirical study. In *Proceedings of the Working Conference on Mining Software Repositories*, MSR'14, pages 2–11, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2863-0. doi: 10.1145/2597073.2597085.

[83] M. Maas, P. Reames, J. Morlan, K. Asanović, A. D. Joseph, and J. Kubiatowicz. GPUs as an opportunity for offloading garbage collection. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management*, ISMM'12, pages 25–36, Beijing, China, 2012. doi: 10.1145/2258996.2259002.

[84] M. Maas, K. Asanović, T. Harris, and J. Kubiatowicz. Taurus: A holistic language runtime system for coordinating distributed managed-language applications. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'16, pages 457–471, Atlanta, Georgia, 2016. doi: 10.1145/2872362.2872386.

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

[86] M. Moghimi, J. Venkatesh, P. Zappi, and T. Rosing. Context-aware mobile power management using fuzzy inference as a service. In *Proceedings of the EAI International Conference on Mobile Computing, Applications, and Services*, MobiCASE'13, pages 314–327, Berlin, Heidelberg, 2013. Springer. ISBN 978-3-642-36632-1. doi: 10.1007/978-3-642-36632-1_18.

[87] *NI USB-6008/6009 user guide and specifications: Bus-powered multifunction DAQ USB device*. National Instruments, February 2012. URL http://www.ni.com/pdf/manuals/371303n.pdf.

[88] S. Park, W. Jiang, Y. Zhou, and S. Adve. Managing energy-performance tradeoffs for multithreaded applications on multiprocessor architectures. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 169–180, San Diego, California, June 2007. doi: 10.1145/1254882.1254902.

[89] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang. Fine-grained power modeling for smartphones using system call tracing. In *proceedings of the ACM European Conference on Computer Systems*, EuroSys'11, pages 153–168, Salzburg, Austria, April 2011. doi: 10.1145/1966445.1966460.

[90] 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 *proceedings of the ACM European Conference on Computer Systems*, EuroSys'12, pages 29–42, Bern, Switzerland, April 2012. doi: 10.1145/2168836.2168841.

[91] G. Pinto, F. Castor, and Y. D. Liu. Understanding energy behaviors of thread management constructs. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'14, pages 345–360, Portland, Oregon, October 2014. doi: 10.1145/2660193.2660235.

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

[93] *Vellamo Open*. Qualcomm Innovation Center, INC., 2012. URL https://www.codeaurora.org/projects/all-active-projects/vellamo-open.

[94] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. AppInsight: Mobile app performance monitoring in the wild. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, OSDI'12, pages 107–120, Hollywood, CA, 2012. USENIX. ISBN 978-1-931971-96-6. URL https://www.usenix.org/conference/osdi12/technical-sessions/presentation/ravindranath.

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

[96] J. B. Sartor and L. Eeckhout. Exploring multi-threaded Java application performance on multicore hardware. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'12, pages 281–296, Tucson, Arizona, October 2012. doi: 10.1145/2384616.2384638.

[97] S. Schwarzer, P. Peschlow, L. Pustina, and P. Martini. Automatic estimation of performance requirements for software tasks of mobile devices. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ICPE'11, pages 347–358, Karlsruhe, Germany, 2011. doi: 10.1145/1958746.1958796.

[98] S. Seneviratne, A. Seneviratne, P. Mohapatra, and A. Mahanti. Predicting user traits from a snapshot of apps installed on a smartphone. *ACM SIGMOBILE Mobile Computing and Communications Review*, 18(2):1–8, June 2014. ISSN 1559-1662. doi: 10.1145/2636242.2636244.

[99] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. *IEEE Micro*, 23(6):84–93, November 2003. doi: 10.1109/MM.2003.1261391.

[100] R. K. Sheshadri, I. Pefkianakis, H. Lundgren, D. Koutsonikolas, A. K. Pietilainen, A. Soule, and J. Chandrashekar. Characterizing mobile user habits: The case for energy budgeting. In *Proceedings of the IEEE Conference on Computer Communications Workshops*, INFOCOM'15, pages 306–311, April 2015. doi: 10.1109/INFCOMW.2015.7179402.

[101] C. Shin, J.-H. Hong, and A. K. Dey. Understanding and prediction of mobile application usage for smart phones. In *Proceedings of the ACM Conference on Ubiquitous Computing*, UbiComp'12, pages 173–182, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1224-0. doi: 10.1145/2370216.2370243.

[102] J. Singer, G. Brown, I. Watson, and J. Cavazos. Intelligent selection of application-specific garbage collectors. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management*, ISMM'07, pages 91–102, Montreal, Canada, 2007. doi: 10.1145/1296907.1296920.

[103] *Moshi Project*. Square, Inc., 2015. URL https://github.com/square/moshi.

[104] V. Srinivasan, S. Moghaddam, A. Mukherji, K. K. Rachuri, C. Xu, and E. M. Tapia. Mobileminer: Mining your frequent patterns on your phone. In *Proceedings of the ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp'14, pages 389–400, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2968-2. doi: 10.1145/2632048.2632052.

[105] *SPECjvm98 Benchmarks*. Standard Performance Evaluation Corporation, release 1.03 edition, March 1999. URL http://www.spec.org/jvm98.

[106] D. Sunwoo, W. Wang, M. Ghosh, C. Sudanthi, G. Blake, C. Emmons, and N. Paver. A structured approach to the simulation, analysis and characterization of smartphone applications. In *proceedings of the IEEE International Symposium on Workload Characterization*, IISWC'13, pages 113–122, September 2013. doi: 10.1109/IISWC.2013.6704677.

[107] A. Tongaonkar, S. Dai, A. Nucci, and D. Song. Understanding mobile app usage patterns using in-app advertisements. In *Proceedings of the 14th International Conference on Passive and Active Measurement*, PAM'13, pages 63–72, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-36515-7. doi: 10.1007/978-3-642-36516-4_7.

[108] H. F. Unelsroed, P. C. Roeine, and F. Ghani. Power Guru: Implementing smart power management on the Android platform.

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

[110] N. Vijaykrishnan, M. Kandemir, S. Kim, S. Tomar, A. Sivasubramaniam, and M. J. Irwin. Energy behavior of Java applications from the memory perspective. In *Proceedings of the USENIX Java Virtual Machine Research and Technology Symposium*, Monterey, California, April 2001. URL https://www.usenix.org/legacy/events/jvm01/full_papers/vijaykrishnan/vijaykrishnan.pdf.

[111] V. M. Weaver, D. Terpstra, and S. Moore. Non-determinism and overcount on modern hardware performance counter implementations. *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 215–224, April 2013. doi: 10.1109/ISPASS.2013.6557172.

[112] D. R. White, J. Singer, J. M. Aitken, and R. E. Jones. Control theory for principled heap sizing. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management*, ISMM'13, pages 27–38, Seattle, Washington, 2013. doi: 10.1145/2464157.2466481.

[113] 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.

[114] *The Power-Choke WE-TPC 8012*. Würth Elektronik eiSos GmbH & Co. KG, 2009. URL http://www.farnell.com/datasheets/1645832.pdf.

[115] Y. Xu, M. Lin, H. Lu, G. Cardone, N. Lane, Z. Chen, A. Campbell, and T. Choudhury. Preference, context and communities: A multi-faceted approach to predicting smartphone app usage patterns. In *Proceedings of the International Symposium on Wearable Computers*, ISWC'13, pages 69–76, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2127-3. doi: 10.1145/2493988.2494333.

[116] K. Yaghmour. *Embedded Android: Porting, Extending, and Customizing*. O'Reilly Media, 2013. ISBN 9781449327972. URL https://books.google.com/books?id=PsN9nq4WGB0C.

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

[118] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras. DroidMiner: Automated mining and characterization of fine-grained malicious behaviors in Android applications. In *Proceedings of the European Symposium on Research in Computer Security*, ESORICS'14, pages 163–182. Springer International Publishing, 2014. ISBN 978-3-319-11203-9. doi: 10.1007/978-3-319-11203-9_10.

[119] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev. Static control-flow analysis of user-driven callbacks in Android applications. In *Proceedings of the International Conference on Software Engineering*, ICSE'15, pages 89–99, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4799-1934-5. URL http://dl.acm.org/citation.cfm?id=2818754.2818768.

[120] C. Zhang, X. Ding, G. Chen, K. Huang, X. Ma, and B. Yan. Nihao: A predictive smartphone application launcher. In *Proceedings of the EAI International Conference on Mobile Computing, Applications, and Services*, MobiCASE'13, pages 294–313, Berlin, Heidelberg, 2013. Springer. ISBN 978-3-642-36632-1. doi: 10.1007/978-3-642-36632-1_17.

APPENDICES

A HARDWARE SCHEMATICS

The DragonBoard 8074 provides a quick reference or evaluation platform for Qualcomm's 800 series Snapdragon 8074 processor. The Snapdragon 800 SOM (System On Module) measures 70x70mm, 230 pin MxM with BtB connector to support other SOC interfaces. The SOM includes the following features:

- Snapdragon 800 main application processor
- PM8941 Power Management Integrated Circuits (PMIC) for Peripheral LDOs, Boost Regulators
- PM8841 PMIC for Processor Core regulators
- LPDDR3 up to 800Mhz 2GB RAM.

Figure A.1 shows the two major functional blocks of the PM8441: *(i)* output power management, and *(ii)* IC-level interfaces. The PM8841 device, integrates all power manage-

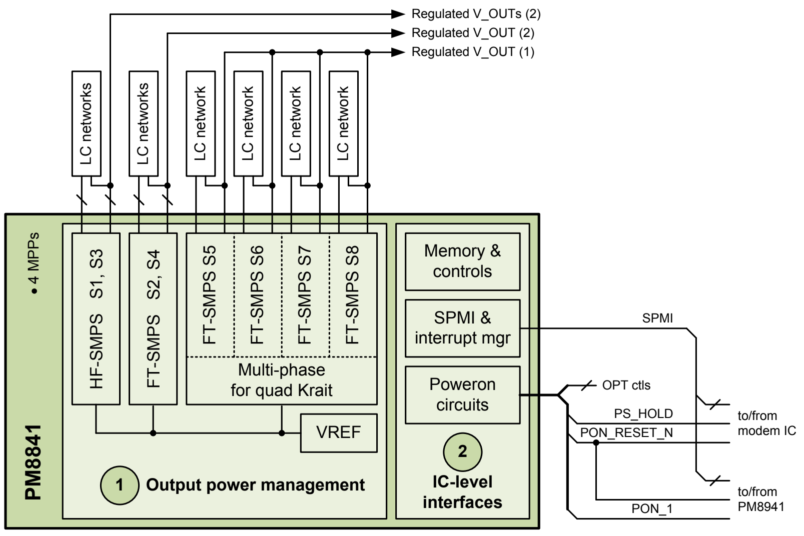


Figure A.1.: High-level PM8841 functional block diagram, ©Intrinsyc 2013

ment, general housekeeping, and user interface support functions into two mixed-signal ICs. The PMIC includes poweron circuits that provide the proper power sequencing for the entire APQ8074 chipset.The default poweron sequence is defined as follows:

1. VDD_MEM (on-chip memory)
2. VDD_CORE (digital core circuits)
3. VREF_SDC (SDC reference voltage)
4. VDD_P3(I/Os), VDD_P7 (SDC1), VDD_DDR_CORE_1P8 (DDR core 1.8 V)
5. VDD_USB_1P8 (USB 1.8 V circuits)
6. VDD_P1 (EBI and DDR I/Os), VDD_P4 (HSIC), VDD_DDR_CORE_1P2 (DDR core 1.2 V)
7. EBIx_VREF_CA2, EBIx_VREF_DQ (EBI0/1 CA and DQ LPDDR3 reference voltage)
8. VDD_USB_3P3 (USB 3.3 V circuits)
9. VDD_PLL2 (PLL circuits), VDD_QFPROM_PRG (QFPROM programming), VDD_P2 (SDC2)
10. VDD_KRAIT (Krait applications microprocessor)

VDD_KRAIT pins are dedicated to the power for quad-krait applications microprocessors. These pins are listed as: J25, J27, J29, J37, J39, J41, N25, N27, N29, N37, N39, N41, U25, U27, U29, U37, U39, U41, AC25, AC27, AC29, AC37, AC39, and AC41. The power supply maximum rating on all VDD_KRAIT is 1.8 V.

Figure A.3 shows the voltage regulators and controls connected to the PM8841. We replace the inductors L20, L22, L24 and L26 to measure the voltage drop on the microprocessors as described in Chapter 3. The Pololu-ACS714 Hall-effect linear current sensor [1] is positioned between the CPU and the voltage regulator. We read the output voltage using a National Instruments NI-6009 data acquisition device [87]. From these we calculate instantaneous power and thence energy over time. On the board we replaced the four inductors L20, L22, L24 and L26 PIFE20161T power-choke inductors (0.24 μH, 20%, DCR = 19 mΩ, ISAT = 4.7 A ROHS) by four power-choke WE-TPC-8012 shielded inductors (0.24 μH, DCR = 19 mΩ, ISAT = 5.8 A) in series with the hall-effect sensor [30; 114].

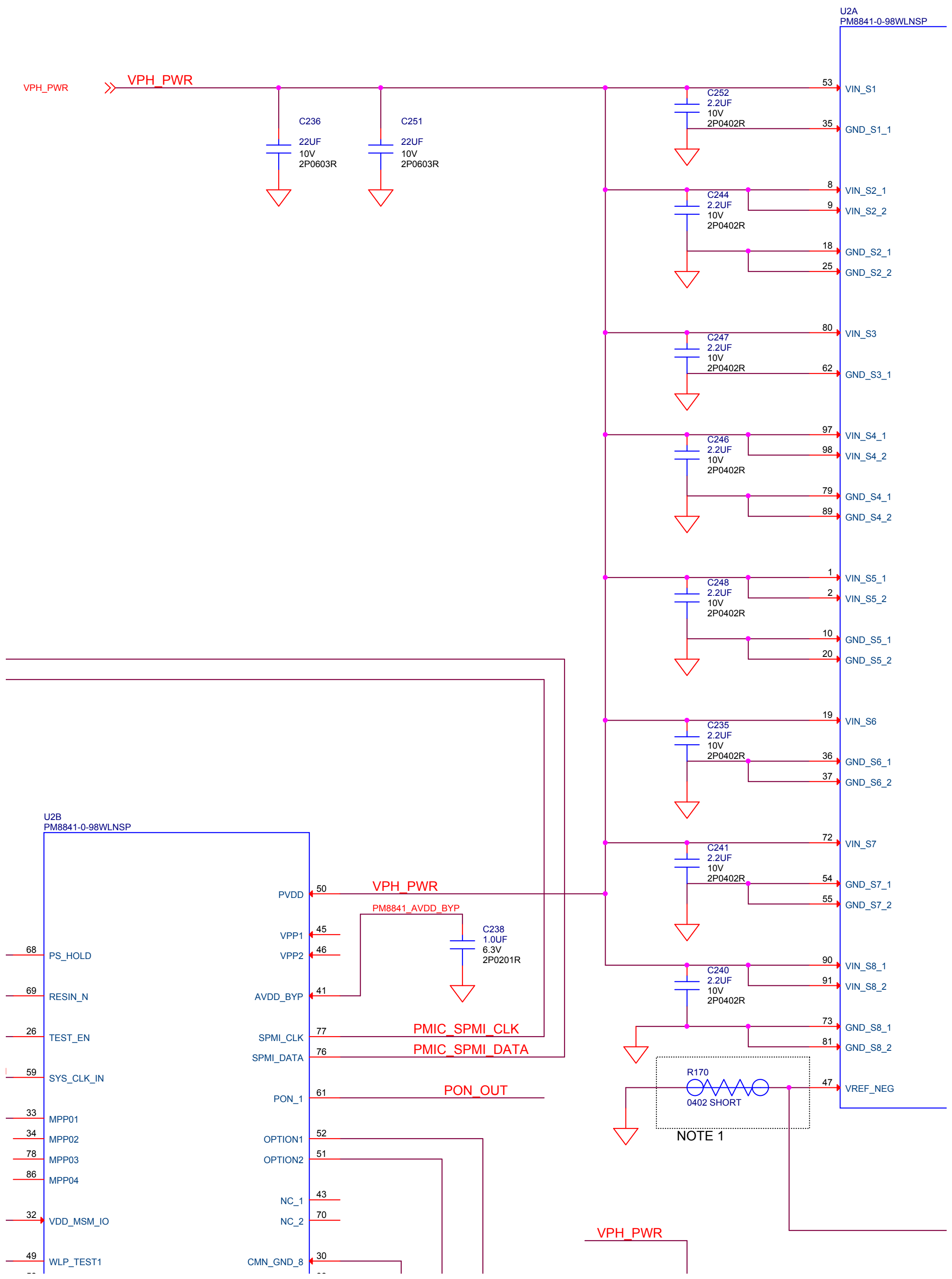


Figure A.2.: PM8441 Schematics Control and VREG: ground and input pins; ©Intrinsyc 2013

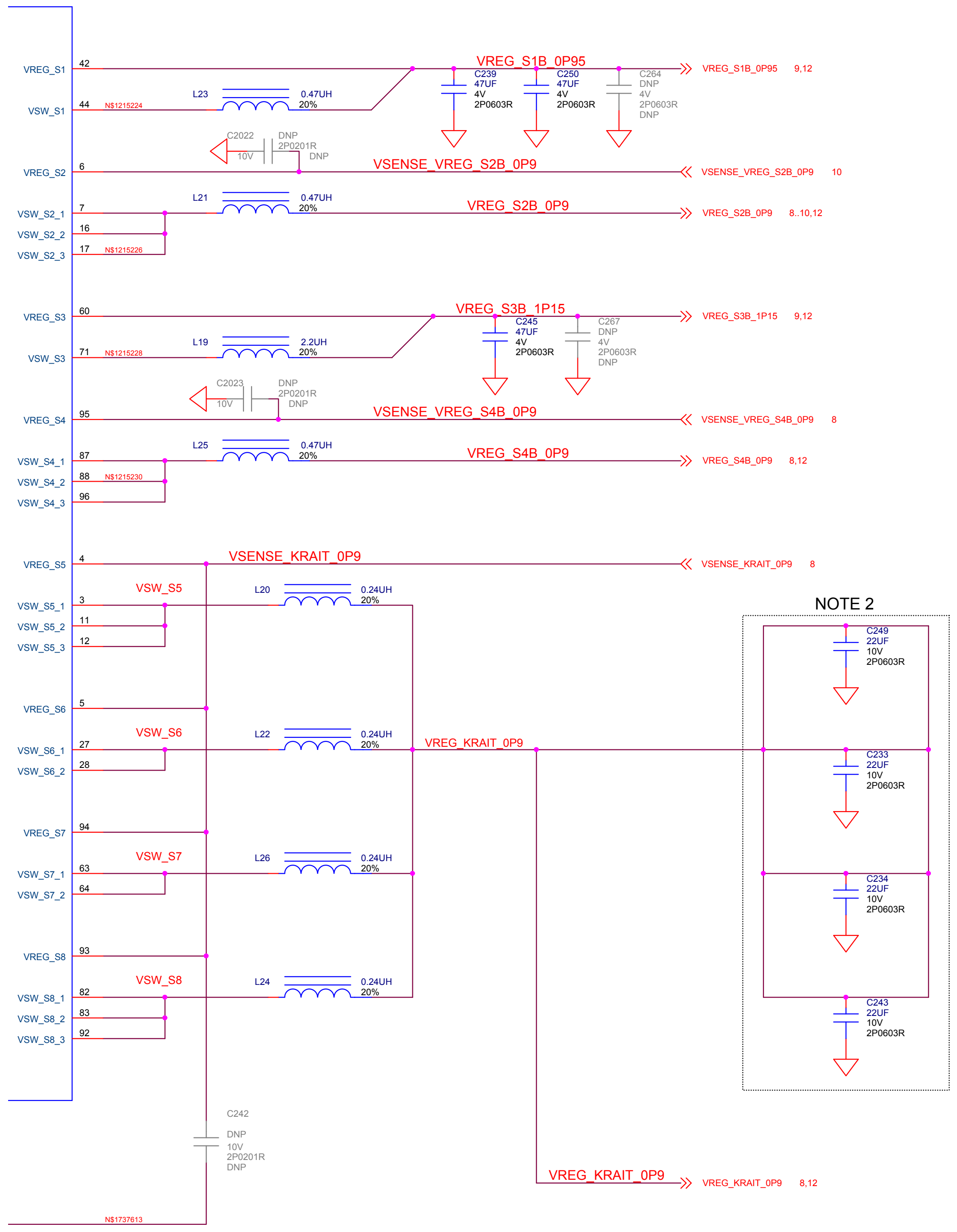


Figure A.3.: PM8441 Schematics Control and VREG: switch and regulator pins; ©Intrinsyc 2013

B IMPACT OF POWER MANAGER ON THE VM

Measuring performance on mobile systems is challenging due to the complex hardware and software stacks. Different feedback mechanisms continuously adapt system parameters, resulting in changed response time, power consumption, and time performance. These metrics are deeply entangled and must be evaluated in unison in a controlled environment.

Here, we consider metrics for evaluating Android apps running on a real device (the Snapdragon multi-core platform), and steps needed to obtain controlled results for those metrics, like limiting interference from non-salient layers, and controlling variability due to adaptive components that perturb the target metric. Understanding results requires correlating metrics with underlying platform (e.g., hardware, OS, run-time, and application) events.

B.1 Power Measurements

Measuring total AC current to the device with a clamp ammeter is not precise enough to measure the effect of the VM components on the CPU power. Nevertheless, measuring the power on the SoC level shadows the GC contribution on the CPU power since it accounts for the total power consumed by individual components (e.g., modem, GPU, or sensors). We measure overall current flow at the circuit level as shown in using a `Pololu-ACS714` Hall-effect linear current sensor [1], positioned between the CPU and the voltage regulator. We read the output voltage using a National Instruments NI-6009 data acquisition device [87]. From these we calculate instantaneous power and thence energy over time. We eliminate noise for analog signals using two bias resistors 50KΩ to satisfy the bias current path requirement of the instrumentation to the ground. At sample rate 2 kS/s, we read the voltage across the voltage regulator and the sensor output using the differential method and we take simple moving average for each 20 points. As an example, Figure B.1 shows

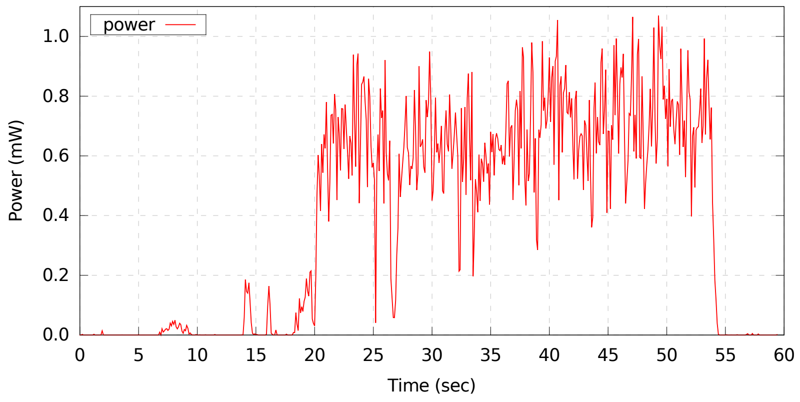


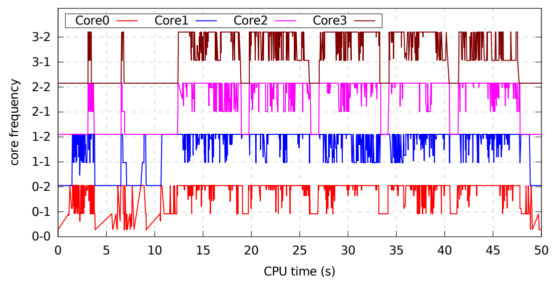
Figure B.1.: xalan: Power readings over execution time

power measurements obtained during the execution of five full iterations of xalan. Between iterations the system (governor) disables unused cores and lowers the frequency of the remaining online cores in order to reduce power consumption.

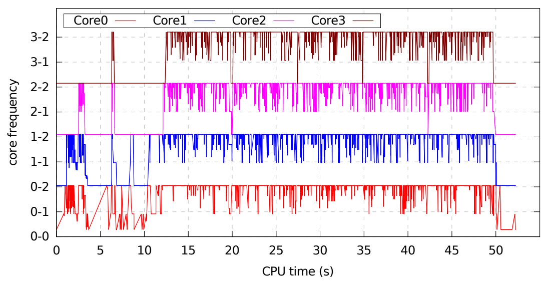
We note that the core frequencies are important to measure and analyze power consumption during the execution of programs. By extending systrace to show core frequency updates and events that enable or disable cores, we can correlate the power expenses of the system configurations and their impact on the speed of individual cores. Figure B.2 plots the frequency for several benchmarks of each core over time. For DaCapo benchmarks, cores 2 and 3 are disabled between iterations, while the second core is disabled outside the main control loop for the iterations. The single threaded benchmarks (i.e., SPECjvm98) utilize only two cores.

For Quadrant, Figure B.2(e), all cores run at their maximum frequency which makes Quadrant inappropriate for experimental evaluation since any change to the system will not be reflected on the CPU speed nor the power consumption.

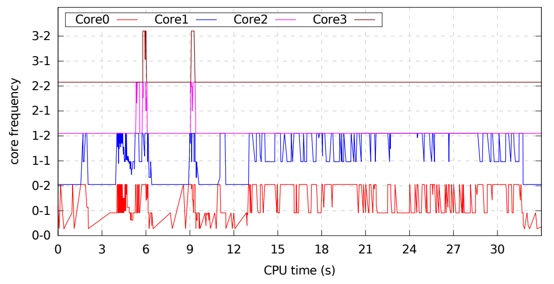
The governors manage the frequency to meet power and performance constraints. We note that during the evaluation of the GC, applying different GC strategies generates different workloads on the CPU. Hence, the governor reacts in different ways according to the current strategy. Figure B.3 shows the distribution of core frequency as a percentage of total execution time for lusearch and xalan, for two different governors. Although, the



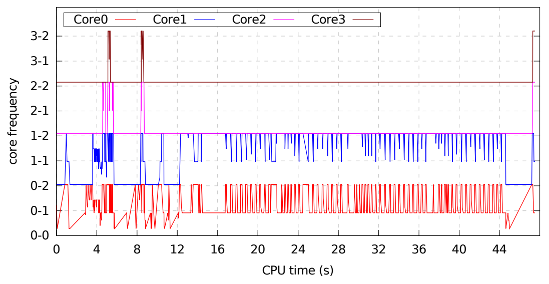
(a) lusearch



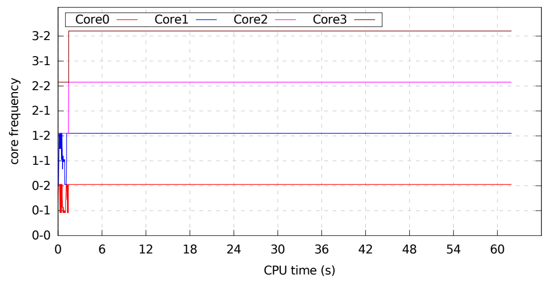
(b) xalan



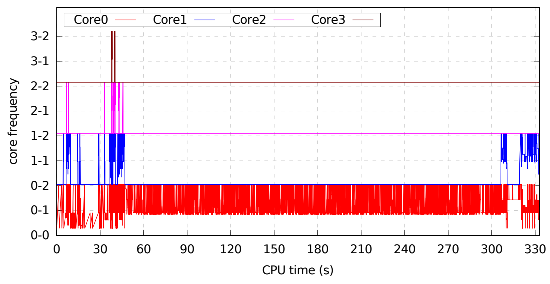
(c) jack



(d) javac



(e) Quadrant

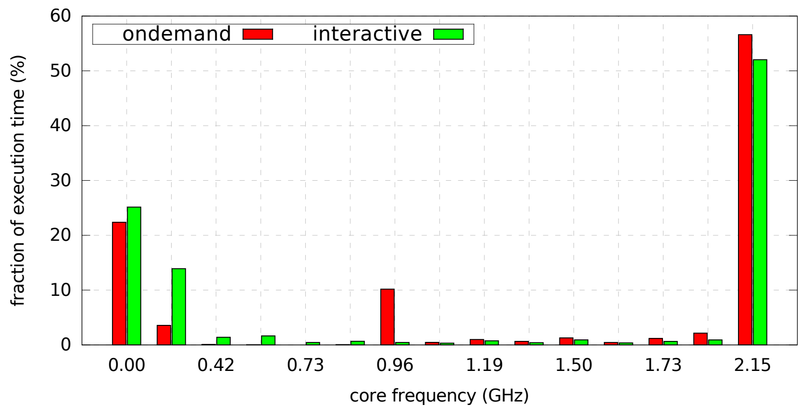


(f) Pandora

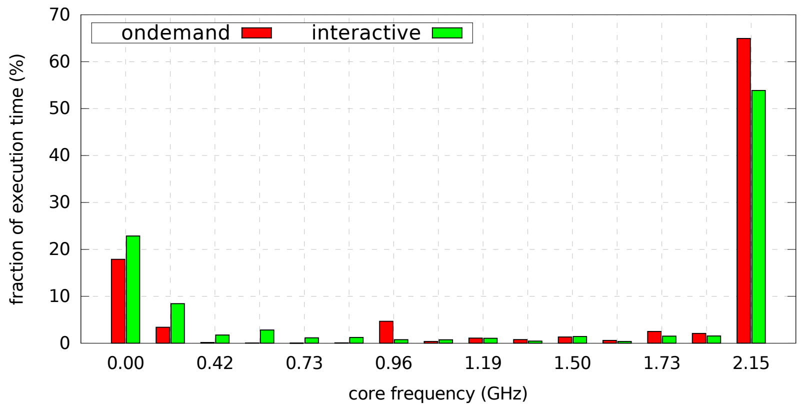
Figure B.2.: Core frequencies during execution time

ondemand governor does not allow a core to stay at the maximum frequency more than a configurable threshold, the cores spend a higher percentage of time on higher frequencies. This observation suggests that the GC evaluation has to consider carefully the ways the GC changes the governor decisions leading to different power expenditures.

Smaller heap sizes naturally cause more frequent GC collections. With a tighter heap, apps consume more energy as the GC is triggered more frequently during execution. All the benchmarks showed this effect except for lusearch which exhibits less energy consumption with tighter heaps. Figure B.4 shows the effect of the heap footprint on total energy



(a) lusearch

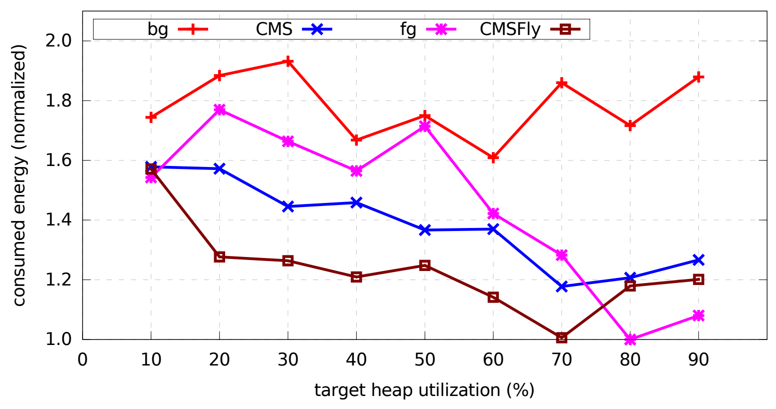


(b) xalan

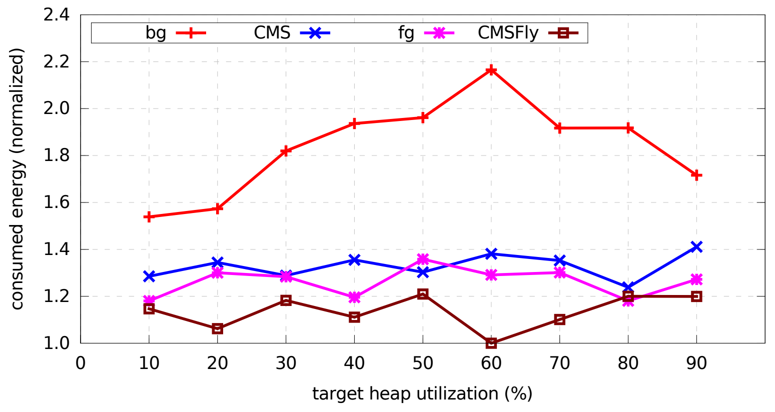
Figure B.3.: Frequency transitions trade-offs with governor

consumption when running under the default Dalvik GC (CMS), CMSFly (on-the-fly extension of CMS), fg (CMS with no GC daemon), and bg (CMS performing GC exclusively through the GC daemon) (normalized to the smallest number *per benchmark*).

To understand the lusearch power trends, we analyze the frequency transitions and the trade-offs between static and dynamic power as the execution time varies with different heap utilization. lusearch benefits from the system making more effective frequency transition decisions than the other benchmarks. With target utilization 10%, the cores spend 60% of the execution time on the maximum frequency. For a heap utilization of 90%, the cores spend 47% on maximum frequency. On the other hand xalan spends 56% and 61% of execution time on the maximum frequency for 10% and 90% target utilization, respectively.



(a) lusearch



(b) xalan

Figure B.4.: Energy versus target heap utilization with GC variants

The energy behavior of lusearch demonstrates that it is not sufficient to evaluate the GC impact on the power consumption based on only the CPU cycles. Instead, correlation between other non-adjacent system layers such as scheduler and governor is necessary to tune on-chip power expenses.

Figure B.4 also shows that concurrent GC (i.e., the GC daemon runs in the background) consumes more energy. The performance counters (i.e., L1 misses and CPU cycles) suggest that a background GC results in a higher workload across all the benchmarks due to heap synchronization (i.e., context switches) and trimming operation triggered when the heap is under utilized.

To control for variability across GC variants for the effects of enabling and disabling cores and frequency scaling for the GC daemon, our experiments pin the GC daemon on a dedicated throttled core to prevent it running faster than a fixed maximum frequency, and consider each GC variant as the frequency varies. Still, running GC in the background has the worst power efficiency across all the benchmarks. This suggests that the GC daemon must be further integrated into frequency scaling decisions (to adapt the system to the GC workload) in order to conserve energy.

B.2 Time Execution

We explore the trade-off between power and throughput, versus heap size. As we mentioned, apps consume more energy with larger heap except for lusearch. For execution time, all benchmarks show that a larger heap increases the throughput.

To control variability across GC variants due to the effects of enabling or disabling cores and frequency scaling, our experiments pin the GC daemon to run on a dedicated core that is throttled to prevent it running faster than a fixed maximum frequency, considering each GC variant as the frequency is adapted. A foreground GC can still have an effect on governor decisions regarding the mutator threads, which run only on the other three cores. Also, when the GC daemon is idle, the only threads that can execute on that core are OS and daemon threads of other processes. A surprising outcome is that having all GC performed in foreground by mutators (fg) results in better throughput than for collectors that use a background GC daemon.

Table B.1 shows the execution time, geometric mean, confidence interval 5%), and the standard deviation of running each benchmark for a given number of iterations following the warmup under two different governors. All benchmarks score better confidence interval when the thermal-engine is disabled, demonstrating that the thermal engine affects the accuracy of the experiments with the continuous increase in temperature during execution. Quadrant has an exceptional pattern because the cores are locked to the highest frequency (Figure B.2(e)).

Table B.1.: Impact of the governor and the thermal-engine on the throughput experiments

Benchmarks	governor	Thermal	Warmup	Iterations	Average (s)	Geo. Mean (s)	Conf. Interval (ms, 5%)	Standard Deviation (ms)
DaCapo								
lusearch	ondemand	ON	1	4	5.68	5.68	14.00	36.00
		OFF	1	4	5.18	5.18	10.80	27.60
	interactive	ON	1	4	5.68	5.68	25.00	51.20
		OFF	1	4	5.66	5.66	18.50	37.70
xalan	ondemand	ON	1	4	6.71	6.71	36.10	92.10
		OFF	1	4	6.69	6.69	24.00	62.70
	interactive	ON	1	4	6.82	6.81	50.20	102.40
		OFF	1	4	6.83	6.83	30.10	61.50
SPECjvm98								
javac	ondemand	ON	1	3	27.53	27.53	162.20	165.50
		OFF	1	3	27.46	27.46	111.70	114.00
	interactive	ON	1	3	30.62	30.62	57.30	117.00
		OFF	1	3	30.90	30.90	190.10	194.00
jack	ondemand	ON	1	3	16.68	16.68	13.72	14.00
		OFF	1	3	15.83	15.83	7.35	7.50
	interactive	ON	1	3	18.84	18.84	60.00	120.90
		OFF	1	3	18.77	18.77	66.64	68.00
compress	ondemand	ON	1	3	25.51	25.51	151.90	155.00
		OFF	1	3	25.77	25.77	29.40	30.00
	interactive	ON	1	3	25.73	25.73	112.10	228.80
		OFF	1	3	25.66	25.66	93.60	95.50
jess	ondemand	ON	1	3	19.66	19.66	41.65	42.50
		OFF	1	3	18.85	18.85	7.35	7.50
	interactive	ON	1	3	21.81	21.81	52.80	107.80
		OFF	1	3	19.66	19.66	41.65	42.50
db	ondemand	ON	1	3	21.61	21.61	164.15	167.50
		OFF	1	3	21.29	21.29	112.20	114.50
	interactive	ON	1	3	26.49	26.49	119.70	244.50
		OFF	1	3	26.72	26.72	4.41	4.50
Android								
Quadrant	ondemand	ON	1	3	47.62	47.62	24.00	36.80
		OFF	1	3	47.23	47.23	43.40	66.40
	interactive	ON	1	3	47.61	47.61	16.30	24.90
		OFF	1	3	47.38	47.38	43.40	66.50

VITA

VITA

Ahmed Hussein received a Bachelor of Science in Computer Science from Alexandria University, Egypt, in July 2005, and the Master of Science in Computer Science from Purdue University, USA, in December 2013. His research interest spans memory management, implementation of compilers, and runtime systems for high-level languages.