Impact of GC Design on Power and Performance for Android

Ahmed Hussein[†] Mathias Payer[†] Antony Hosking[†] Christopher A. Vick[‡]

[†]Purdue University, USA [‡]Qualcomm, Inc., USA

[†]{hussein,mpayer,hosking}@purdue.edu [‡]cvick@qti.qualcomm.com

Abstract

Small mobile devices have evolved to versatile computing systems. Android devices run a complete software stack, including a full Linux kernel, user land with several software daemons and a virtual machine to run applications. On these mobile systems energy is a scarce resource and needs to be handled carefully. Current systems rely on governors that adjust the frequency of individual cores depending on the system load.

We measure energy consumption of different components of this complex software stack, including *garbage collection* (GC) of the Android virtual machine. Here we propose several extensions to the default GC configuration of Android, including a generational collector, across established dimensions of heap memory size and concurrency.

Our evaluation shows that Android'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%). Our work quantifies the direct impact of GC on mobile system, enumerates the main factors and layers of this relationship, and offers a guide for analysis of memory behavior in understanding and tuning system performance.

Categories and Subject Descriptors C.1.4 [*Parallel Architectures*]: Mobile processors; C.4 [*Performance Of Systems*]: Measurement techniques; D.3.4 [*Programming Languages*]: Processors—Memory management (garbage collection), Runtime environments; D.4.8 [*Performance*]: Measurements

General Terms Algorithms, Experimentation, Languages, Measurement, Performance

Keywords mobile, power, energy, Android, smartphones

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

SYSTOR 2015, May 26–28, 2015, Haifa, Israel.

© 2015 ACM. ISBN 978-1-4503-3607-9/15/05. . . $15.00.

DOI: http://dx.doi.org/10.1145/2757667.2757674

1. Introduction

With the number of shipped Google Android devices projected to exceed a billion in 2014 [Gar 2014] the Dalvik virtual machine (VM) has become an ubiquitous computing platform. Mobile applications (apps) running on this computing platform rely on features of the underlying software stack, including automatic memory management (*garbage collection*, or GC). Due to their mobile nature, Android devices most commonly run on battery power. Therefore, energy is a scarce resource. Previous studies have established that individual components of managed language virtual machines can have a significant impact on energy consumption [Vijaykrishnan et al. 2001; Esmaeilzadeh et al. 2011; Sartor and Eeckhout 2012; Cao et al. 2012; Pinto et al. 2014].

Mobile platforms on the other hand face a set of new challenges regarding the impact of different VM components as they must optimize between the conflicting targets *performance*, *responsiveness*, and *power consumption* when running apps. In addition, mobile platforms are more 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. To address power and thermal issues, the operating system (OS) relies on dynamic voltage and frequency scaling (DVFS) in software controlled by *governors* [Brodowski]. These governors collect runtime statistics (e.g., system load and core temperature) and then apply complex heuristics to meet optimization criteria [Iyer and Marculescu 2002; Miyoshi et al. 2002; Carroll and Heiser 2014].

We measure the impact of individual adaptive VM components like the GC across all layers of the software stack in a live Android Dalvik VM identifying GC as a component that will profit from further optimization. We show that evaluation of GC on mobile devices must consider (and control for) different forms of adaptive behaviors. Moreover, adaptive layers of the system might better meet energy and performance goals with additional knowledge of the underlying workload, including phase changes in the workload such as those manifested by GC, which exhibits different memory access patterns than the app itself.

We propose and evaluate several extensions of the default memory management configuration of Android systems that allow tunning of the performance/power consumption

tradeoff. In addition, we use performance counters to collect counts of L1 cache misses and other memory events.

Several studies evaluated the energy consumption on mobile devices by dividing the battery capacity by the total power consumed by the device subsystems (e.g., CPU, display, GPU, or GPS) [Carroll and Heiser 2010; Pathak et al. 2012; Jindal et al. 2013; Chen et al. 2013]. This approach is useful to understand the hardware, and the impact of the I/O on the battery lifetime. However, understanding the VM services design tradeoffs cannot be refined based on that approach because the VM is not linked to the I/O expenses. Hence, the experimental environment focuses on the energy consumed by the CPU which ranges between 20–40% of the total device consumption [Carroll and Heiser 2010].

Our results 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. Varying the GC strategy can reduce total on-chip energy (by 20–30%) with low impact on application throughput (by 10–40%) and worst-case pause times (by 20–30%). We propose and discuss strategies that allow power savings with minimal impact on performance.

The contributions of this paper are:

- A proposal for a measurement methodology to capture precise energy consumption and performance on a real Android device in vivo.
- 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 evaluation of our measurement methodology for 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.

2. Background and Methodology

Each Android app runs as a separate process in its own instance of the Dalvik VM. Figure 1 illustrates the Android software stack, consisting of the Linux kernel, native libraries, and the Android Runtime which includes the supporting Java

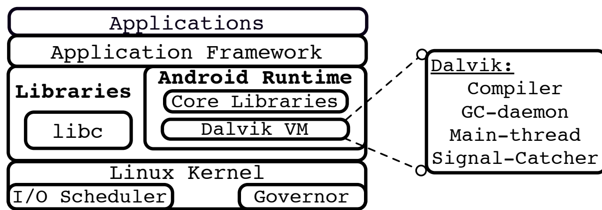


Figure 1. Android and the main components of Dalvik

VM (Dalvik VM [Ehringer 2010]) and core Java libraries. Active components of Dalvik that run in separate threads include the GC daemon, the just-in-time (JIT) compiler, the signal catcher, main thread, and the application threads (called *mutator* threads).

2.1 Dalvik Concurrent Mark-Sweep (CMS)

The garbage collector traverses all the references starting from the *root* to reclaim memory occupied by non-reachable objects [Jones et al. 2011]. This makes the GC tasks memory bound compared to the compute-bound mutators.

Dalvik GC typically runs concurrently in its own native Linux thread as a C-coded background daemon, with the application-level Java mutator threads also scheduled as native Linux threads. It operates as a *mark-sweep* collector: first marking reachable objects from roots like thread stacks and global variables and then sweeping up and freeing any unmarked objects.

We refer to the default Dalvik collector as the *concurrent mark-sweep* (CMS) collector. This collector suspends all the mutator threads 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, marks any remaining unmarked objects that are reachable from the dirty objects, and then restarts the mutator threads. It then safely (and concurrently) sweeps up and frees the remaining unmarked objects as garbage.

CMS uses simple heuristics to balance the tension between frequency of GC and heap size, similar to those described in by Brecht et al. [Brecht et al. 2001]. The primary parameter controlling heap size 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 *targetUtil*. Thus, the bigger the *targetUtil*, the tighter the heap allocated to the app. The available space (*room*) is constrained to the range 2–8MiB. The threshold *concurrentStartBytes* (*CSB*) is set at some delta ($\delta = 128$KiB) below the *softLimit*. The relationship among these parameters, at time t, is given by the following:

$$\begin{aligned} room(t) &= (1 - targetUtil) \times live(t) \\ softLimit(t) &= live(t) + \min(\max(room(t), 2\text{MiB}),\ 8\text{MiB}) \\ CSB &= softLimit(t) - \delta \end{aligned} \quad (1)$$

Dalvik GC is triggered for several different reasons:

`GC-CONC`: When 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 exceeds the *softLimit* threshold, or when allocation fails, then the mutator boosts its

priority and performs a *foreground* GC cycle, so long as the GC daemon is not already active in which case it waits for it to finish. If an allocation fails then the mutator retries the allocation after the GC cycle ends.

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

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.

2.2 Device Steady State

We can only measure precise numbers for VM components if the system under measurement is in a steady state (i.e., there is no *noise* from concurrent processes). During the boot sequence resources like CPU and memory are highly utilized for different tasks, leading to noise and imprecision. We disable any unneeded services to further reduce jitter.

In addition to the hard out-of-memory (OOM) killer, Android also introduces a low-memory-killer (LMK) that terminates apps that are no longer in the foreground. We must ensure that an app is not evaluated while the memory utilized by that app (or other concurrent apps) causes the system to hit a pre-defined LMK or OOM threshold.

Figure 2 shows an example of lowmemory regions in MB. When the free memory (RAM) in a system is less than 120MB, 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 120MB, 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.

2.3 Benchmarks

Mobile apps often operate in modes that prevent a repeatable workload because their memory usage varies across different runs (e.g., due to ads). Furthermore, many benchmarking apps are synthetic, measuring a single system feature heavily.

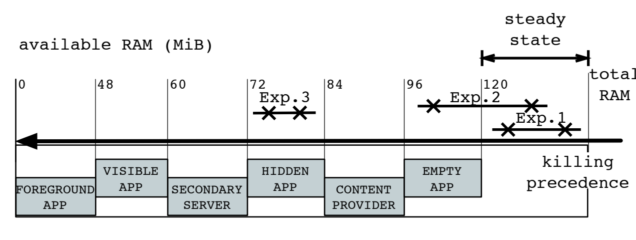


Figure 2. Low memory configurations

While benchmarking on desktop and server platforms is standardized [Dieckmann and Hölzle 1999; Dufour et al. 2003; Kalibera et al. 2012], standard mobile benchmarks have yet to emerge given the young age of mobile platforms. To address this constraint, we have faithfully ported all eight SPECjvm98 applications [Standard Performance Evaluation Corporation]. 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 benchmarks [Blackburn et al. 2006, 2008] to the multi-threaded lusearch and xalan benchmarks. For space reasons we only report on javac and jack of the SPECjvm98 benchmarks.

We run both SPECjvm98 and DaCapo apps using the standard benchmarking harness to obtain execution times using five iterations on the small workload, discarding the first iteration, and averaging all remaining iterations. In addition, we measure two Android benchmarks: (*i*) Quadrant version 2.1 Professional Edition is an industry-standard benchmark, and (*ii*) Pandora version 5.4 is the popular internet radio app for discovering and recommending music based on the Music Genome Project.

We invoke the benchmarks directly from the Android Runtime, which spawns each Dalvik VM instance from the pre-initialized *zygote* VM using the *monkeyrunner* framework [monkeyrunner] (as opposed to spawning a new Dalvik VM process from the command line).

Table 1 summarizes the collected execution characteristics of these benchmarks. We obtain the GC events and overhead columns when running the default CMS collector. The allocation statistics (`Heap`, `Objects`, `Threads`) are obtained by running the 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. The `Heap` and `Objects` results for the ported Java benchmarks are similar to those reported by others using different VMs [Dieckmann and Hölzle 1999; Blackburn et al. 2006]. 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 `Heap-Contentions` shows the number of times a thread fails to acquire the heap lock. This helps as an indication of the intensity of concurrent activities on the heap (allocation and collection).

The `Max-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

Table 1. Benchmark characteristics for Dalvik CMS (ignoring activity of the *zygote* process)

Benchmark	**Heap** (MiB)		**Objects** (M)		**Lifetime** (%)			**Threads**		**GC events**				**GC overhead**		**Code**				**Compilations**	
	Alloc	Live	Alloc	Live	128 KiB	256 KiB	512 KiB	Total	Heap Contentions	CONC	ALLOC	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	20.5	40.7	16	448	6	4	42	3	30.5	2.8	1,721	11,891	895	2,582	3,961	41.8
Pandora	48.91	18.76	0.28	0.06	13.0	24.9	46.0	77	829	7	18	0	4	33.1	6.2	1,596	14,419	4,402	3,701	6,302	97.1
SPECjvm98																					
javac	217.47	10.19	6.15	0.27	7.60	15.8	32.7	7	276	55	42	6	1	99.0	19.7	227	1,464	674	320	5,308	68.9
jack	180.22	0.87	5.52	0.02	11.7	23.8	48.2	8	4,133	105	0	2	1	24.0	8.0	131	717	275	199	2,018	41.8
DaCapo																					
lusearch	686.75	1.22	11.65	0.01	10.3	22.5	47.2	26	2.63e6	356	0	5	1	35.0	5.4	326	3,016	615	781	3,473	56.2
xalan	395.06	2.26	4.14	0.02	9.46	19.3	39.0	26	4.38e5	199	1	5	1	37.2	3.5	489	5,287	915	1,029	5,449	67.6

Table 2. System defaults

Dalvik build properties			Governor: ondemand		
VM parameter	value		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	%	`up_threshold`	90	

unit in the code cache (*count*), and the size of the compiled code (in KiB).

2.4 Platform

We measure a complete Android platform *in vivo* using the APQ8074 DragonBoard™ Development Kit based on Qualcomm's Snapdragon™ S4 SoC running the quad-core 2.3GHz Krait™ CPU [Intrinsync]. Importantly, Krait allows cores to run *asymmetrically* at different frequencies, or different voltages. Software governors can adjust the frequency/voltage of each core individually depending on the workload.

The board runs on Android version 4.3 ("Jelly Bean") with Linux kernel version 3.4. We have instrumented the Dalvik VM and the kernel to record statistics on demand. In addition, we allow direct access to hardware performance counters from the VM, to control *hotplugging* (onlining/offlining) of the cores, and to expose the VM profiler to other kernel-level events. The default Dalvik VM configuration for our board leverages the build properties shown in Table 2.

2.4.1 Dalvik VM Profiling

We run a VM profiler as a daemon thread (implemented in C) inside Dalvik. Profiling is only enabled to gather execution statistics, never when capturing measurements that are sensitive to timing or scheduling such as total execution time or OS context switching The profiling daemon is signalled after every 64KiB of allocation to gather 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 periodically dumped 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.

By default APQ8074 runs Android with the ondemand Linux *CPUfreq* governor. This governor sets per-core frequencies depending on current workload. Moreover, the thermal-engine and *mpdecision* (proprietary component) can also affect CPU frequencies and hotplugging. To avoid perturbation by these services we run experiments that are sensitive to time and scheduling with the thermal-engine disabled, and apply external cooling to the SoC heat sink to prevent failure. We measure steady state execution time by allowing the application to run multiple iterations to warm-up. This reduces the non-deterministic impact of the JIT and avoids any GC overhead during the startup phases. We report the average execution of hot runs with 95% confidence intervals.

2.4.2 Responsiveness

The responsiveness of embedded systems was thoroughly studied and evaluated by estimating the *Worst-Case Execution Time* (WCET) of individual tasks leading to the existence of several commercial tools and research prototypes [Wilhelm et al. 2008]. However, worst case and average mutator pause times do not adequately characterize the impact of GC on responsiveness because of the complexity of the system stacks.

GC pauses can prevent threads that service user interface tasks from giving timely responses to user input events. Humans typically perceive interaction pauses greater than 50ms [Efron 1973], so any greater pause is likely to be noticed by users. Thus, we use *minimum mutator utilization* (MinMU) [Cheng and Blelloch 2001; Printezis 2006; Jones et al. 2011] over a range of time intervals yield a better understanding of the distribution and impact of pauses.

We divide mutator pauses into three different categories: (*i*) safepoint pauses, when a mutator stops in response to a suspension request (e.g., for marking mutator roots), (*ii*) foreground pauses, when a mutator performs a foreground GC cycle, and (*iii*) concurrent pauses, when a mutator waits for a concurrent GC cycle to finish. To compute MinMU for a multithreaded app having a total execution time T and

M mutators $m_1, \ldots, m_M$, each experiencing p_i GC pauses $\delta_1, \ldots, \delta_{p_i}$, we define MinMU for a window of length w as the MinMU (for all mutators) over all time slices of length w in the execution. Our profiler records the values $\delta_1, \ldots, \delta_{p_i}$ for each mutator, with time stamps. To compute MinMU for a given window size w, we take the worst-case MinMU for each mutator in each window of size w sliding over the entire execution T. Only pauses that overlap the window contribute to that mutator's utilization for that window.

2.4.3 Power Measurements

We build our analysis of GC impact on an app running on n cores for total execution time T based on the CPU energy model (E_{CPU}) defined by [Carroll and Heiser 2014]. Measuring energy can only be achieved by measuring the power at the circuit level as the product of measured current I and the voltage drop V across the CPU. Similar to [Cao et al. 2012], we measure current flow at the circuit level as shown in Fig. 3, using a `Pololu-ACS714` Hall-effect linear current sensor between the Krait application microprocessors and the voltage regulator. We use the data acquisition device [National Instruments] with 48kS/s sampling rate and typical absolute accuracy 1.5mV (error 0.9%). We filter the analog noise using two bias resistors 50kΩ to satisfy the bias current path requirement of the instrumentation to the ground.

As an example, Fig. 4a plots power measurements obtained during the execution of five full iterations of lusearch. Figure 4b plots the frequency of each core over time. Between iterations the governor offlines unused cores and lowers the frequency of the remaining online cores in order to reduce power consumption. Together, the plots show how power impulses correlate with frequency transitions. Note that the time axes are offset because they are measured differently: we measure power against real elapsed time and frequencies against CPU cycles recorded using performance counters and converted to time.

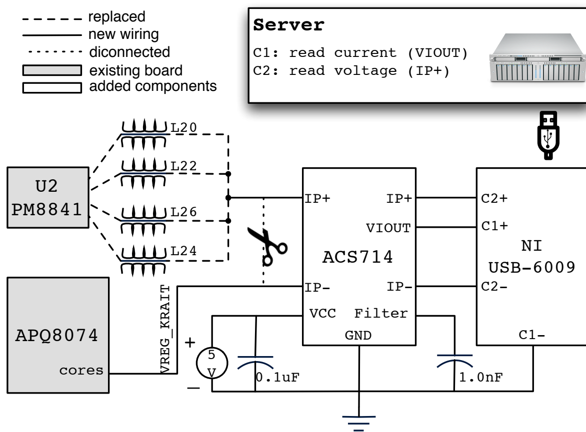


Figure 3. Circuit-level power measurement on the APQ8074

3. GC Extensions

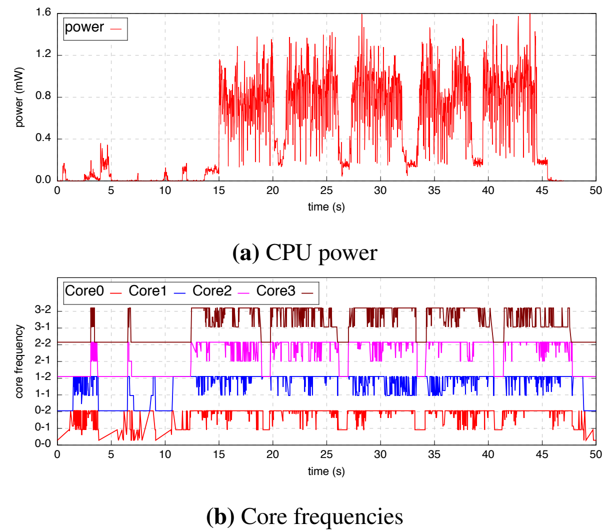
We consider both generational and on-the-fly variants of the default Dalvik CMS collector. These allow us to compare tradeoffs among different GC variants for mobile devices.

3.1 Generational CMS

We implemented a *generational* variant of the CMS collector (GenCMS) to study its effect on app performance. Generational collectors [Lieberman and Hewitt 1983; Ungar 1984] 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*



(a) CPU power

(b) Core frequencies

Figure 4. Power and frequency over time for lusearch

plus the *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 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 performs a minor GC, noting that the next GC-CONC should be major.

3.2 On-the-fly

The CMS collector has brief stop-the-world phases in which all Java are stopped: (*i*) while marking the thread stack roots, and (*ii*) while re-scanning 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 [Domani et al. 2000; Dijkstra et al. 1978] avoid the stop-the-world phase during the marking phase.

We have extended the CMS collector to address the first of these pauses, dubbed "*CMSFly*". The second kind of pauses 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.

3.3 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 allocation exceeds the *softLimit* or fails then the mutator instead forces allocation, and signals the GC daemon to start a 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.

4. Evaluation

We evaluate our methodology along a range of metrics: scheduler preemptions, responsiveness, and power versus both heap size and collector variant, and performance counter statistics. The default heap growth follows Eq. (1) using *targetUtil* of 75% (Table 2); recall that larger *targetUtil* means tighter heap.

4.1 Energy and Throughput

There are a number of ways in which the GC workload can affect power consumption, not the least of which is its effect on hotplugging and DVFS decisions.

4.1.1 Effect of Heap Size

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 total work, though the smaller heap can have second order effects on app locality. Figure 5 (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 5 (top) illustrates how *targetUtil* affects the number of frequency transitions (normalized to CMS numbers *per benchmark*) imposed on the cores.

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) and the app energy to increase as GC overhead increases with *targetUtil*. Figure 6 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.

COROLLARY 1. *Increased throughput (larger heap) does not always correspond to better energy consumption.*

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 and therefore limited by the memory access speed. Thus, choosing a higher CPU frequency to perform the 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 7 shows the distribution of core

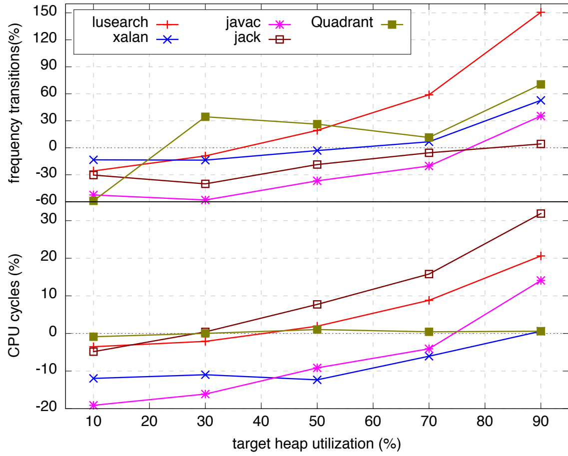


Figure 5. Effect of *targetUtil* on CPU cycles (bottom) & frequency transitions (top) normalized to default CMS

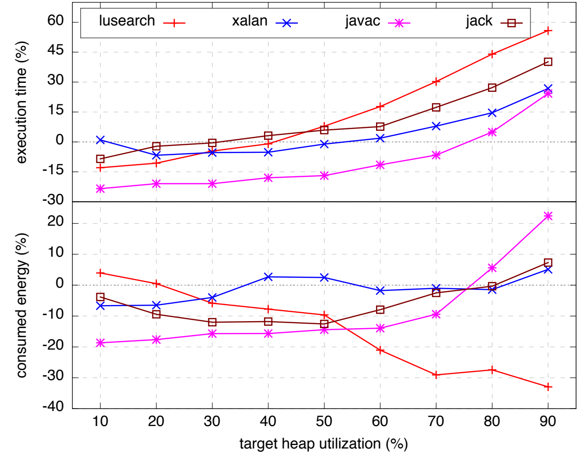


Figure 6. Effect of *targetUtil* on energy (bottom) & throughput (top) normalized to default CMS

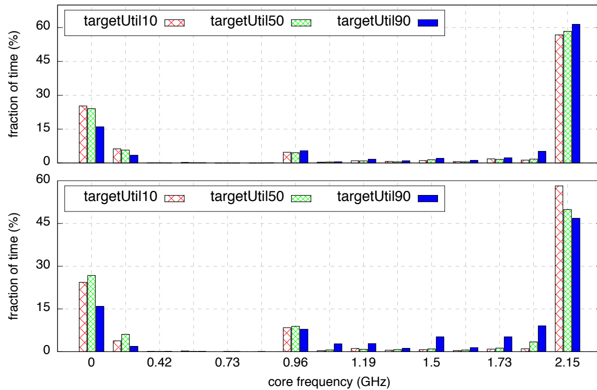


Figure 7. xalan (top), lusearch (bottom): Core frequency distribution (as fraction of time)

Table 3. Schedule statistics normalized to Dalvik CMS

GC Variant	Migrations					Delayed Time				
	lusearch	xalan	javac	jack	**g.mean**	lusearch	xalan	javac	jack	**g.mean**
bg	0.97	1.02	1.36	1.00	1.05	1.14	1.06	0.90	1.09	1.04
fg	0.44	0.79	0.71	0.38	0.55	0.63	0.90	1.93	1.20	1.05

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 higher frequencies (less efficient) with tighter heaps.

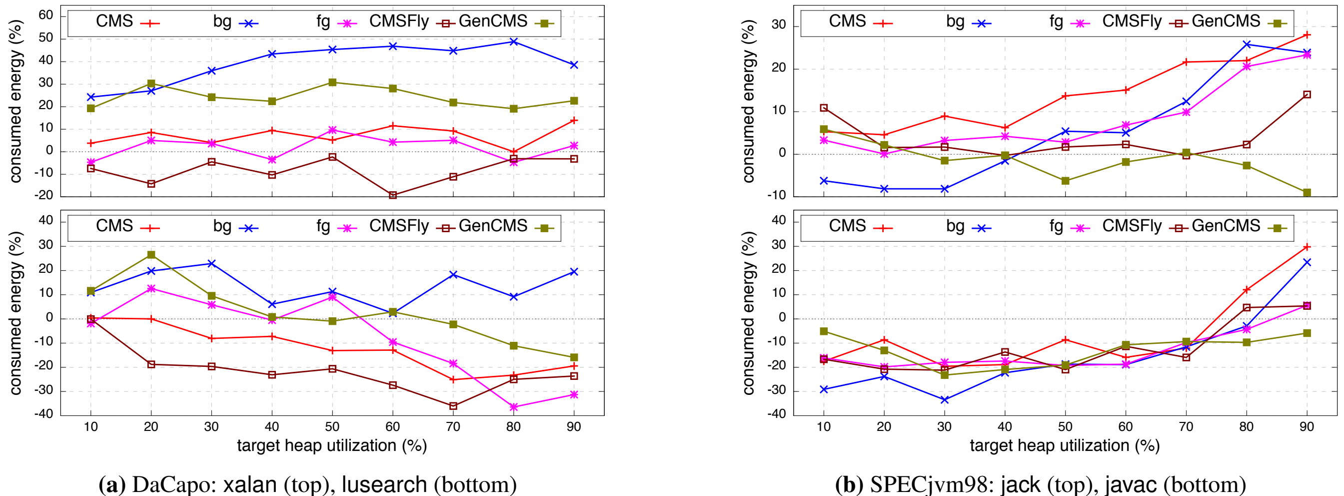
COROLLARY 2. *GC events have significant impact on DVFS decisions.*

4.1.2 Effect of Design Choices

Figure 8 shows the effect of the heap footprint on total energy consumption when running under the default CMS, CMSFly, fg, bg, and GenCMS, each of which is normalized to the default CMS value *per benchmark*).

On single threaded benchmarks, having all GC performed in foreground by mutators (fg) results in better throughput than for collectors that use a background GC daemon. The reason is that fg boosts the collector priority and holds the heap locks for the duration of the GC cycle. As a result, the app threads suffer less preemptions on fg compared to the configurations. For multithreaded applications, the throughput depends on the tradeoff between the workload and the number of cores (i.e., lusearch versus xalan). The generational variant does not perform better than its non generational counterparts on any benchmark. This behavior could have different reasons: (*i*) micro-architectural differences to desktop CPU could favor full collections (i.e., due to different memory timings), (*ii*) the changes between Dalvik and Java VMs, or (*iii*) our extension of the default collector to a generational collector is not optimized enough.

When the GC daemon performs all work in the background, the energy consumption increases. This is due to heap synchronization (i.e., context switches) and trimming operations triggered when the heap is under utilized. On the other hand, a mutator performing a foreground GC boosts its priority while keeping the heap locked. Table 3 shows the scheduler stats for the Java benchmarks running under the bg and fg variants normalized to the default CMS. The left column `Migrations` shows the number of the events during which a thread is is resumed to a new core. Higher numbers imply that a collector imposes more overhead on the scheduler (more cache misses) [David et al. 2007; Park et al. 2007; Gautham et al. 2012]. The column `Delayed-Time` shows the



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

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

Figure 8. Energy vs. *targetUtil* with GC variants normalized to default CMS

total delay that tasks saw; time from the point a task got on the `runqueue` to the point it actually executes the first instruction. These numbers show that the bg variant causes more delays to the scheduler.

COROLLARY 3. *Having all GC performed in background (bg) consumes more energy.*

4.2 Responsiveness: MinMU

MinMU 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 implementations, 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).

4.2.1 Effect of Design Choices

Figure 9 shows the MinMU results for each benchmark under a range of GC variants. The fg configuration shows the best MinMU on lusearch and xalan, with both smallest maximum pauses and best overall utilization. Recall that a mutator performing a foreground GC boosts its priority for the duration of the GC cycle. Also, the GC daemon increases the synchronization overhead. As a result, the fg causes mutators to have less scheduler preemptions for lusearch and xalan compared to the bg variant (because lusearch and xalan have a high memory allocation rate, triggering frequent GC pauses). Conversely, fg performs worst for Pandora, which has few GC events, allocates very little (so allocations need not wait for GC so often), yet has many mutator threads. Live heap size is relatively large for Pandora, Quadrant, and javac, which also experience a large number of foreground `GC-ALLOC` events, so GC cycle times are longer. Thus, CMS and fg both suffer poor utilization as mutators must wait for the longer GC cycles (whether by the GC daemon or in foreground) to complete. Similarly, pushing whole heap collections to background (bg) results in much better MinMU on these benchmarks.

Only javac has maximum GC pauses in the observable range (beyond 50ms). Again, bg offers best overall utilization for javac. The latter shows the worst MinMU for the default CMS, because the high number of `GC-ALLOC` events (Table 1) causes the mutator either to wait for the GC daemon to finish the current GC cycle, or to perform a full foreground GC.

COROLLARY 4. *For multithreaded apps, while foreground GC often improves MinMU, concurrent collectors (background & generational) are better for large heaps.*

4.2.2 Effect of Heap Size

The relation between responsiveness and heap size is not always clear as it depends on how often mutators perform foreground collections. This varies due to the synchronization between mutators and GC daemon. Small heaps have better throughput for small window sizes (worst pause times) compared to large heaps. However, the latter increases the utilization for larger time windows.

5. Related Work

Several studies addressed the GC requirements when deployed in restricted environments. [Griffin et al. 2005] implemented a hybrid scheme between the standard Mark-Sweep Compact collector and reference counter.[Chen et al. 2002b,a] proposed an adaptive GC strategy to optimize the power consumption by shutting down memory banks that hold only garbage objects.

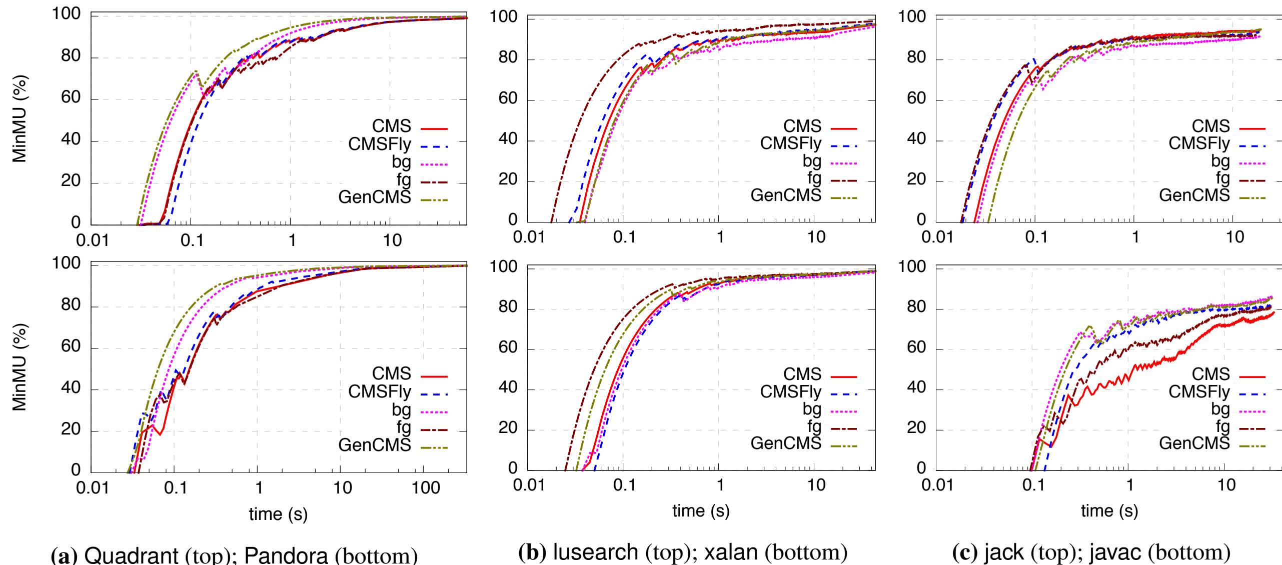


Figure 9. Minimum mutator utilization

Analyzing the concurrency tradeoffs of multithreading in managed runtime systems on desktop [Vijaykrishnan et al. 2001; Pinto et al. 2014; Cao et al. 2012] prove the impact of the GC on the JVM energy requirements. Occasionally, the GC was evaluated as an asymmetric activity that can be isolated on a separated core [Sartor and Eeckhout 2012; Cao et al. 2012]. This approach is not practical for mobile devices due to the high cost of keeping a core online.

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 app blocks that leak large amounts of energy [Pathak et al. 2011, 2012]. [Hao et al. 2013] suggested a program analysis and per-instruction modeling to estimate energy consumption of an Android app at the code level.

Here, we evaluate GC across non-adjacent system layers. We demonstrate that the GC requirement is not solely dependent on memory size. Instead, it is necessary to define GC requirements as a function of system mechanisms such as the governor and scheduling policies. [Kambadur and Kim 2014] similarly show that energy evaluation on server platforms has more significance when it considers the full system stack. Our results differ from the work done by [Hao et al. 2013] in fitting the runtime performance within the whole system stack (i.e., hardware, kernel and power management). Thus, the results generated in this paper reflect real execution involving synchronization overhead, induced by spin-locks and context-switching [Park et al. 2007; Gautham et al. 2012].

6. Discussion

We show the degree by which system performance (power, time, and response) is affected by GC design strategies and enumerate the main impacting factors. This work is a first step to better understand mobile GC behavior and interactions between power and performance on mobile systems. Further tuning of all GC parameters is left for future work.

Suggestions for the community. GC on mobile platforms has novel challenges 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 get 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 significant impact on battery life and app responsiveness; GC directly affects the user experience.

Researchers and industry should develop a common platform with transparent access to different system layers. This approach allows researchers to evaluate the side effects of their implementation on all other system components. We also encourage the creation of *standard benchmarks* in order to refine evaluation methodologies on mobile platforms and provide a first step towards this goal by porting existing Java benchmarks to the Android platform.

Writing power-aware source code is not a feasible option due to widely heterogeneous hardware and software. Our results show that code optimizations are specific to the default system configurations (i.e., heap size, and concurrency). This leads us to believe that energy optimizations can be achieved by simple modifications to both runtime 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 a

adaptively tuned performance. Similarly, heap growth policies need to be integrated with DVFS decisions to achieve better energy consumption than heuristics based only on memory footprint.

6.1 Methodology Restrictions

Evaluation methodology on general purpose computing devices is a well-studied problem [Georges et al. 2007; Kalibera and Jones 2013; Blackburn et al. 2008]. Unfortunately, profiling the mobile platform in real time context has not yet been studied as vigorously.

Coarse-grained profiling. Ideally, we would like to measure the fraction of GC time, work, and power to assess the maximum impact of GC (and to report how close we get to this theoretical limit in practice for each of the metrics). This is surprisingly hard due to interferences and dependencies between concurrent components in the software stack. We argue that considering the overall system performance is the best way to evaluate the GC. Figure 6 showed that the increased throughput and reduced GC work for lusearch do not correspond to lower energy consumption. Hence, breaking down the attribution of the GC in each metric alone does not reflect the actual GC overhead.

Profile tools. Hardware counters are limited on mobile devices. For example, L2 memory counters are not available on many ARM processors [Weaver et al. 2013]. This prevents porting analytical methodologies which are relying on hardware performance counters. Commercial devices disable access to performance counters and power rails.

Taming non-determinism. While techniques such as *replay compilation* are widely used to omit the level of non-determinism caused by JIT optimizations [Arnold et al. 2000], Android does not offer similar features. Instead, we run our measurements within 95% of confidence interval by calculating the average of several iterations after excluding the cold ones.

6.2 Android Run Time (ART)

Recently, Google announced ART, an ahead-of-time compiler framework for Android 4.4 [ART and Dalvik] which will soon replace Dalvik. Our GC extensions for Dalvik mirror many of the new features of ART including generational, and on-the-fly collection. We regard exploring ART's behavior as an important continuation of our study.

7. Conclusion

Our results show that different Dalvik GC strategies have highly varying energy requirements that do not always correlate with app throughput. Varying policies, such as heap growth or concurrency can significantly reduce the energy consumed or can reduce the worst-case pause time, but not at the same time. 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. 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. Similarly, app developers need a range of GC strategies to choose from, so they can tune for responsiveness, utilization, and power consumption.

This paper 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).

Acknowledgments

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

References

M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the jalapeño JVM. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–65, Minneapolis, Minnesota, Oct. 2000. doi: 10.1145/353171.353175.

ART and Dalvik. URL https://source.android.com/devices/tech/dalvik/art.html.

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

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. *Commun. ACM*, 51 (8):83–89, Aug. 2008. doi: 10.1145/1378704.1378723.

T. Brecht, E. Arjomandi, C. Li, and H. Pham. Controlling garbage collection and heap growth to reduce the execution time of Java applications. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 353–366, Tampa, Florida, Nov. 2001. doi: 10.1145/504282.504308.

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

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

A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. In *USENIX Annual Technical Conference*, pages

271–284, Boston, Massachusetts, June 2010. URL https://www.usenix.org/legacy/event/atc10/tech/full_papers/Carroll.pdf.

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

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

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

X. Chen, A. Jindal, and Y. C. Hu. How much energy can we save from prefetching ads?: Energy drain analysis of top 100 apps. In *ACM Workshop on Power-Aware Computing and Systems*, HotPower, pages 3:1–3:5, Farmington, Pennsylvania, 2013. doi: 10.1145/2525526.2525848.

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

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

S. Dieckmann and U. Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, pages 92–115, Lisbon, Portugal, July 1999. doi: 10.1007/3-540-48743-3_5.

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. *Commun. ACM*, 21(11):966–975, Nov. 1978. doi: 10.1145/359642.359655.

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 *ACM SIGPLAN International Symposium on Memory Management*, pages 155–166, Minneapolis, Minnesota, Oct. 2000. doi: 10.1145/362422.362484.

B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 149–168, Anaheim, California, Oct. 2003. doi: 10.1145/949305.949320.

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

D. Ehringer. *The Dalvik Virtual Machine Architecture*, Mar. 2010. URL http://davidehringer.com/software/android/The_Dalvik_Virtual_Machine.pdf.

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 *ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 319–332, Newport Beach, California, Mar. 2011. doi: 10.1145/1950365.1950402.

Market Share: Ultramobiles by Region, OS and Form Factor, 4Q13 and 2013. Gartner, Feb. 2014. URL https://www.gartner.com/doc/2672716/market-share-ultramobiles-region-os.

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

A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 57–76, Montréal, Canada, Oct. 2007. doi: 10.1145/1297027.1297033.

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

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

Intrinsync. *DragonBoard development board based on the Qualcomm Snapdragon 800 processor (APQ8074)*. URL http://mydragonboard.org/db8074.

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

A. Jindal, A. Pathak, Y. C. Hu, and S. Midkiff. Hypnos: Understanding and treating sleep conflicts in smartphones. In *ACM European Conference on Computer Systems*, EuroSys, pages 253–266, Prague, Czech Republic, 2013. ISBN 978-1-4503-1994-2. doi: 10.1145/2465351.2465377.

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

T. Kalibera and R. Jones. Rigorous benchmarking in reasonable time. In *The International Symposium on Memory Management*, pages 63–74, Seattle, Washington, June 2013. doi: 10.1145/2464157.2464160.

T. Kalibera, M. Mole, R. Jones, and J. Vitek. A black-box approach to understanding concurrency in DaCapo. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems,*

Languages, and Applications, pages 335–354, Tucson, Arizona, Oct. 2012. doi: 10.1145/2384616.2384641.

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

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

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

monkeyrunner. URL http://developer.android.com/tools/help/monkeyrunner_concepts.html.

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

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

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

A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In *ACM European Conference on Computer Systems*, pages 29–42, Bern, Switzerland, Apr. 2012. doi: 10.1145/2168836.2168841.

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

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

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

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

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

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

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

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.