Don't Race the Memory Bus: Taming the GC Leadfoot

Ahmed Hussein[†] Antony L. Hosking[†] Mathias Payer[†] Christopher A. Vick[‡]
[†]Purdue University, USA [‡]Qualcomm, Inc., USA
[†]{hussein,hosking,mpayer}@purdue.edu [‡]cvick@qti.qualcomm.com

Abstract

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 %.

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

Keywords mobile, power, energy, Android, smartphones

1. Introduction

Shipped as systems-on-a-chip (SoC), mobile platforms feature heterogeneous multi-core hardware with on-die hardware peripherals such as WiFi and GPS. The fundamental user experience on such devices is driven by 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), which aim for energy savings while maintaining reasonable performance [29, 34]. The complexity of modern mobile platforms 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.

Our focus here is 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 explored this tradeoff for general-purpose platforms [15, 23, 40, 43], including surveying energy management across the stack [32]. Nevertheless, interactions across layers on mobile devices have not been directly addressed, even as such devices are more sensitive to energy and thermal conditions. Dalvik is the most widely used mobile managed run-time system, which we treat here essentially as an opaque black box, excepting that we observe and correlate significant memory management events with CPU power, performance, and responsiveness. Importantly, the CPU consumes from 20 % to 40 % of total device power [16], making it a significant component of power consumption that is worth managing effectively.

To obtain *in vivo* energy consumption for Android we modify a DragonBoard APQ8074 development kit [27] by interposing a Hall effect current sensor between the board's power management unit and its quad-core SnapDragon 800 SoC [42]. Using device performance counters to measure cycles, instructions, and elapsed time, we are able to correlate performance profiles with SoC power.

Moreover, by pinning Dalvik's concurrent garbage collection thread to one core and varying the frequency of just that core, we can isolate and understand the impact of garbage collection along these performance dimensions as power varies. We quantify the degree to which garbage collection is memory bound by showing how its utilization of the processor, measured in collector *cycles per instruction* (CPI), improves as the clock speed of the collector core is throttled back. We obtain these results using a modified *GC-aware* Linux power governor that responds by capping the frequency of only the collector core for the duration of each concurrent collection cycle. The upshot is that garbage collection work consumes less power, though at the cost of throughput. We show that the GC-aware governor allows tuning this tradeoff to match the needs of apps, to save save as much as 30 % of processor energy for at most 10 % reduction in execution time.

ISMM'15, June 14, 2015, Portland, OR, USA
Copyright 2015 ACM. ISBN 978-1-4503-3589-8/15/06. . . $15.00.
http://dx.doi.org/10.1145/2754169.2754182

2. Approach

Tracing garbage collectors traverse heap references starting from the mutator roots to determine all the *reachable* objects [30]. The collector reclaims memory occupied by non-reachable objects. As a result, memory operations to load and trace the references dominate collector instructions, and incur more memory cycles per instruction than compute-bound mutator workloads. Motivated by this specialized GC workload, several studies have explored offloading GC work to: (*i*) dedicated slow cores [15, 43], (*ii*) GPUs [33], and (*iii*) even specialized hardware [8–10]

Here, we explore the direct power impact of Dalvik's concurrent 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* [14] to influence DVFS decisions. Its 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 collector runs as a daemon thread in the Dalvik virtual machine. When triggered it can be scheduled on any available core at whatever frequency the governor sets for that core. The default Android governor has no special knowledge about the activity of the GC daemon; it applies the same workload feedback mechanisms for the GC daemon as it does for all other threads. To isolate and control the impact of the GC daemon we make the following modifications to Android and Dalvik:

1. Pin the Dalvik GC daemon to primary core 0 so that we know the precise core on which it will run. Importantly, this core is always online, so we do not affect decisions for onlining/offlining cores for other threads. Thus, we do not reserve a core and keep it online solely for the GC daemon, which runs only intermittently; otherwise we might consume power to keep a core online unnecessarily.
2. Modify the Dalvik virtual machine so that the `ondemand` governor knows when the concurrent GC daemon is active, by marking the beginning and end of each collection cycle.
3. Modify the `ondemand` governor to *cap* the frequency of 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 GC daemon is not active (i.e., outside the GC cycle), the governor is also free to adjust the frequency above the cap.

Remember that pinning is needed to prevent the GC daemon from migrating to other cores that are not *capped*. Otherwise, the daemon speed will vary depending on the core on which it is scheduled. Pinning to core 0 is the only way to control concurrent GC frequency while leaving other mutator threads to run at independently governed speeds on other cores. We pin to core 0 because it is always on; other cores can be offlined by the governor to save power when multi-core utilization is low.

Given the memory-bound nature of the GC daemon 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 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 GC daemon. Our results demonstrate how CPI improves for the GC daemon when the frequency of its core is capped.

Of course, slowing the collector core 0 also slows down mutator threads, both those direectlyt interleaved with the concurrent GC daemon on core 0, and those indirectly forced to wait for the GC daemon to finish the GC cycle if they try to allocate. 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.

3. Methodology

Our experimental platform comprises hardware (DragonBoard), software (Android, Dalvik) [3, 22], scripts for managing the device automatically, and a selection of standard benchmarks. We instrument both hardware and software to measure the behavior of the benchmarks, which we have ported from standard Java to Android. The instrumentation allows us to correlate events across the layers. We measure only *hot* runs of the benchmarks, to exclude the effects of Dalvik's dynamic "JIT" compilation. Note that early releases of the *Android Run Time* (ART) VM [6], intended to replace Dalvik as the standard VM for Android apps, have a similar memory management framework to Dalvik's. Thus, we expect our results to be reasonably predictive of ART's behavior. Unfortunately, the DragonBoard does not currently have support for the latest version of Android with ART.

3.1 Platform

We measure a complete Android development platform *in vivo*, avoiding emulation. We use the DragonBoard™ APQ-8074 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 [27]. Krait allows cores to run *asymmetrically* at

Table 1. System defaults

Dalvik VM build properties			Governor: ondemand		
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	%	up_threshold	90	

different frequencies, or different voltages, controlled by software.

The board runs Android version 4.3 ("Jelly Bean") with Linux kernel version 3.4. We modified the kernel and Dalvik to: (*i*) allow direct access to hardware performance counters from Dalvik, (*ii*) control *enabling/disabling* of the cores, and (*iii*) expose the Dalvik profiler to other kernel-level events. We capture both events that enable/disable cores (hotplugging) and frequency transitions due to DVFS using a modified Android `systrace`. The default configurations for Dalvik and the Android governor appear in Table 1, as shipped in the APQ8074 Android distribution.

3.2 Consistent Lightweight VM Profiling

We measure the CPI of the GC daemon using hardware performance counters to collect the CPU cycles and number of instructions it executes [48, 49]. To minimize profiling overhead we use a separate profiler daemon to gather per-thread statistics such as heap demographics and performance counters and to correlate these values with GC events, triggering the profiler daemon every 64 KiB of allocation. To avoid I/O overhead and expensive synchronization with the mutator threads, the profiler stores its data in a lock-free cyclic buffer, which is drained only on overflow. The profiler daemon is disabled when performing timing-sensitive measurements, such as for execution time and energy.

3.2.1 Experimental Challenges

Measuring real apps running on an Android device has many challenges at both system and app levels. First, the device does not operate in single-user mode. At boot time Android launches several device-wide services as background processes that stay running in the background after the boot completes. These can affect the experiments by competing for CPU resources, so we control for these by repeating iterations of benchmarks and averaging results.

Second, consistent evaluation of performance is difficult on a system that has proprietary thermal and power management (such as Qualcomm's thermal engine and *mpdecision* userspace binary) and kernel-managed DVFS. Android (via Linux) supports a range of *CPUfreq governors* that implement different algorithms for setting core frequencies according to CPU usage [14]. The APQ8074 runs by default with the `ondemand` governor. This sets per-core frequencies depending on current usage. Moreover, the thermal engine and *mpdecision* proprietary components 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 proprietary thermal engine disabled, and instead apply external cooling to the SoC heat sink to prevent device failure.

Third, Android apps require user interaction both to start them running and to drive them. We initiate and control all of our benchmark runs via simulated user interaction using the *monkeyrunner* framework [35].

Finally, hardware counters are limited on mobile devices [49]. For example, L2 memory counters are not available on some ARM processors, including the APQ8074, and commercial devices often disable access to the performance counters. This limitation prevents importing existing analytical models relying entirely on hardware performance counters.

3.2.2 Taming VM Controls

Every Android app runs as a separate process in its own instance of the Dalvik VM. By default the Dalvik concurrent collector runs as a background native daemon. The 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 (once at the beginning and once at the end of the collector cycle). It operates as a *mark-sweep* collector, tracing references from roots, which include both thread stacks and other global variables, marking objects reachable via those references, and recursively through references stored in reachable objects. When all the reachable (live) objects have been marked it sweeps the heap to free up unmarked objects.

Heap sizing. We retain Dalvik's default heap sizing policies which are simple heuristics to balance the tension between frequency of garbage collection and heap size, similar to those described by Brecht et al. [13]. The primary parameter controlling heap size and garbage collection is the *target heap utilization* ratio (`targetutil`), used to resize the heap after each GC cycle. Resizing means computing a new allocation threshold for triggering the next collection cycle, called `softlimit`.

The value of `softlimit` is set to ensure that the ratio of the volume of live data to `softlimit` is equal to `targetutil`. In addition to `softlimit`, resizing also computes the *concurrent start bytes* (`CSB`) threshold, set at a delta of 129 KiB less than `softlimit`. When allocation would cause the heap to exceed `softlimit` then the allocating mutator directly performs a *foreground* concurrent GC cycle. Otherwise, when allocation succeeds without exceeding `softlimit`, but the new allocation exceeds `CSB`, then the allocating mutator signals the GC daemon to start a new *background* GC cycle. Of course, heap sizing decisions have an effect on energy consumption [18, 19, 25].

JIT compilation. Omitting the non-determinism introduced by dynamic "JIT" compilation is widely used in exper-

iments on general purpose Java platforms [5]. While Android does not offer *replay compilation*, the Dalvik dynamic "JIT" compiler is sufficiently simple [4] for its effects to be avoided by running multiple iterations of each benchmark within a single invocation of the Dalvik VM and discarding the first cold iteration. Execution times of these hot iterations vary in the range of 0.5 % to 1 % (for 90 % confidence intervals).

3.3 Benchmarks

Finding meaningful workloads to evaluate GC on mobile devices is difficult. Typical mobile apps are event-based, with behavior dependent on user actions. They also often interface to network and other peripheral devices in the normal course of their execution. These interactions result in behaviors that are difficult to control and difficult to script in a repeatable way. Android apps also typically rely heavily on native libraries for performance of computation-intensive functionality (such as audio and video processing), meaning that they rely very little on Dalvik and its managed run-time system. They are also often buggy and cannot reliably be run in a mode that allows evaluation.

Free mobile apps often operate in modes that prevent a repeatable workload because their memory usage varies across different runs. Interestingly, the impact of a given workload is not limited to its innate memory profile: Pathak et al. [38] show that free mobile apps using third-party services to display advertising consume considerably more battery. For example, one app spends 75 % of its total power consumption on advertisements. App developers may also explicitly force components to stay awake, introducing more power drain [39].

Meanwhile, commercial Android benchmarks such as Quadrant lock the cores at their maximum frequency to measure peak performance. This makes them useless for meaningful evaluation of power-performance tradeoffs.

Moreover, both user apps and commercial benchmarks are essentially black boxes with opaque bahaviors, and no history of understanding and analysis in the GC literature. Many benchmarking apps are also synthetic, measuring a single system feature heavily, without modeling general-purpose executions.

For these reasons, our workloads are drawn from established Java benchmark suites that are already well understood, at least in the desktop and server space [11, 21, 31]. They also may exhibit behaviors (e.g., scalability and concurrency) that existing Android apps do not (yet) display.

We have faithfully ported several standard Java benchmarks from both the SPECjvm98 [46] and DaCapo [11, 12] benchmark suites. We have ported all eight of the SPECjvm98 applications to Android, but for space reasons focus here on the most "representative" [28] of those eight: javac and jack. Because the full set of Java APIs are not all fully supported on Android and some packages are completely omitted, our porting of DaCapo is restricted to two multithreaded applications from the DaCapo 9.12 Bach release, lusearch and xalan.

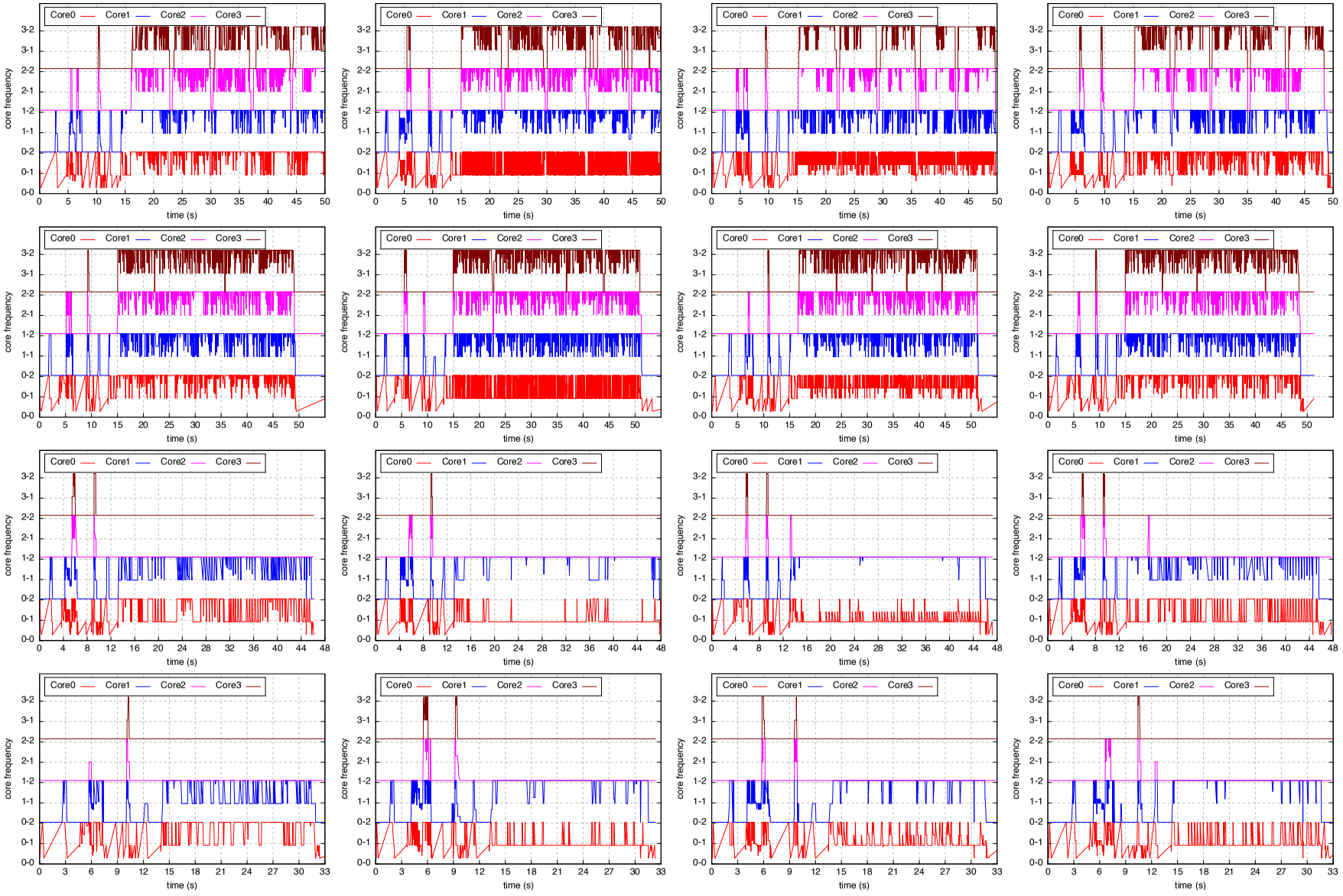
We run both SPECjvm98 and DaCapo apps using their standard benchmarking harnesses to obtain elapsed execution times. Our execution time experiments run six iterations of each benchmark on the small workload, discarding the first cold iteration, and taking the average of the warm iterations. Confidence intervals for these runs are tight, in the range 0.5 % to 1 %. The number of available hardware threads is automatically sampled by some of the benchmarks; there are four hardware threads on the quad-core APQ8074.

3.4 Power Measurements

Carroll and Heiser [17] model the power consumption of an n-core CPU as $P_{\text{CPU}} = P_{\text{uncore}} + n(P_{\text{dynamic}} + P_{\text{static}})$. P_{static} is the workload-independent power consumed by a core that is online but otherwise idle, varying only with core voltage. P_{dynamic} is the additional workload-dependent power consumed by an active core based on voltage/frequency: $P_{\text{dynamic}} = V^2 \times C_{\text{eff}} \times f$ [50], where V is the core voltage, C_{eff} is the effective switching capacitance of the core, and f is the frequency of the core. The remaining CPU power consumption, P_{uncore}, is independent of the number of online cores, typically contributed by last-level caches, buses, etc.

Since the governor reacts to workload by adjusting core speeds, dynamic energy is affected by its decisions. Hence, power consumed by an active core is dependent on the workload (instructions executed), i, and the core frequency f: $P_{\text{dynamic}} \propto \frac{i}{\text{cycles}}$. Furthermore, Carroll and Heiser [17] reveal that transitioning between different core frequencies and online/offline states (hotplugging) has significant power impact on total energy. They measure transition costs in the range 317 to 1 926 mW for the Snapdragon™ 600 (8064T). Thus, on-chip consumed energy depends on the cost of the frequency transitions in addition to the explicit cost in instructions at a given core frequency. Further, measuring energy can only be achieved by measuring the power at the circuit level as the product of measured current I and the the voltage drop V across the CPU. However, measuring total AC current to the device with a clamp ammeter is not precise enough to measure the effects of workload on CPU power [15]. Evenso, measuring power on the SoC level does not account solely for workload on the cores since it also includes power consumed by other on-chip components (modem, GPU, sensors, etc.).

We measure overall current flow at the circuit level using a Pololu-ACS714 Hall-effect linear current sensor [1], positioned between the CPU and the voltage regulator. We measure voltage using a National Instruments NI-6009 data acquisition device [36]. From these we calculate instantaneous power and thence energy over time. The Pololu-ACS714 has total output error of ± 1.5 % at room temperature with factory calibration. The NI-6009 DAQ allows 48 kS/s sampling rate with typical absolute accuracy 1.5 mV (error 0.9 %). We read



(a) default Dalvik **(b)** 0.96 GHz **(c)** 1.5 GHz **(d)** 2.1 GHz

Figure 1. Timeline of frequency transitions for lusearch (top), xalan, javac and jack (bottom), respectively

the voltage across the voltage regulator and the sensor output at sampling rate 2 kS/s using the differential method and we take a simple moving average for each set of 10 points.

4. 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. 1. Figure 1a plots the frequency transitions of the apps running on the default Dalvik system.

For DaCapo benchmarks lusearch and xalan, cores 2 and 3 are often offlined between iterations, while the second core 1 is mostly offline outside the main control loop for the iterations. The single threaded benchmarks (i.e., SPECjvm98) use only two cores. Figures 1b to 1d 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 2.15 GHz, respectively). Notice how capping affects not only the transitions for 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.

4.1 Energy and Throughput

The collector is a memory bound task so we measured the per-thread CPI of the concurrent GC daemon for each GC cycle. Figure 3 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 the 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. 2 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 the GC daemon. Thus, the GC daemon is a good candidate for targeted frequency capping to improve its efficiency.

The energy impact of capping GC core frequency by the GC-aware governor is clear. Figure 4 plots the effect on total energy consumed (left-hand vertical axis) and execution time (right-hand vertical axis) over a range of GC core frequency

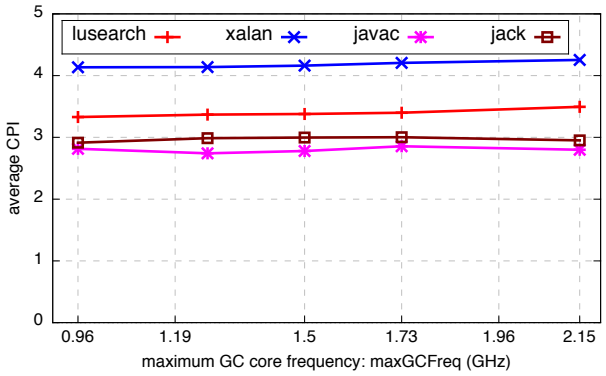


Figure 2. Average overall CPI

caps for each benchmark. Energy consumed with the default Dalvik governor for one execution is shown as a dashed horizontal line (default-energy) while the default Dalvik execution time (default-time), is shown as a solid horizontal line. The trend lines (trend-energy), 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).

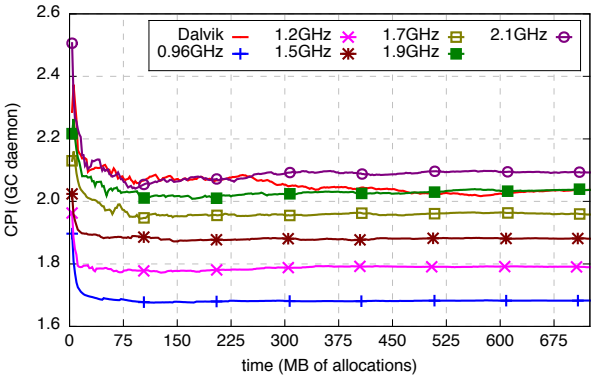
For lusearch we see best energy consumption at 0.96 GHz which is approximately 20 % lower than energy for the cap at the highest frequency (2.15 GHz). Although both xalan and lusearch are multithreaded apps, xalan shows less energy savings (around 10 %) than lusearch. The differences are due to the characteristics of the workloads. For example, xalan is known to perform more frequent memory operations [31], borne out by the higher overall CPI for xalan in Fig. 2.

Energy consumed for jack varies least. Referring back to the frequency transition diagrams for jack in Fig. 1 we note 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.

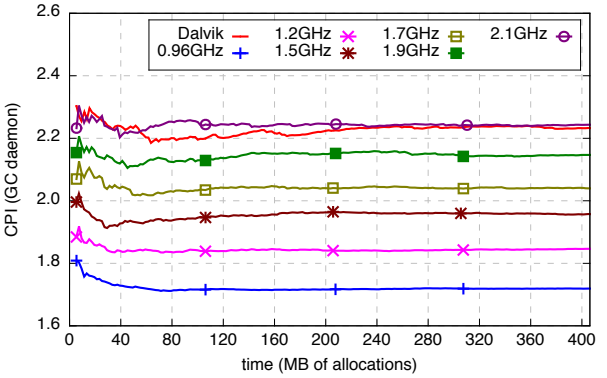
Figure 5 summarizes the effect of dynamic GC core frequency capping on the energy (normalized to the smallest value *per benchmark*). The clear trend is that higher frequency caps (faster collector thread and higher CPI) implies more energy consumption.

We now explore the *tradeoff* between power and throughput while varying the GC core frequency cap. We expect 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 6 shows the performance tradeoff with varying GC core frequency, normalized to the execution time of the default system.

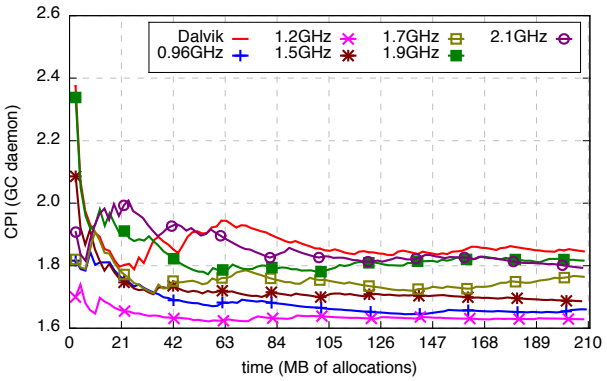
For three benchmarks (lusearch, xalan and javac), the throughput slowdown is at worst 10 %. As noted earlier, jack



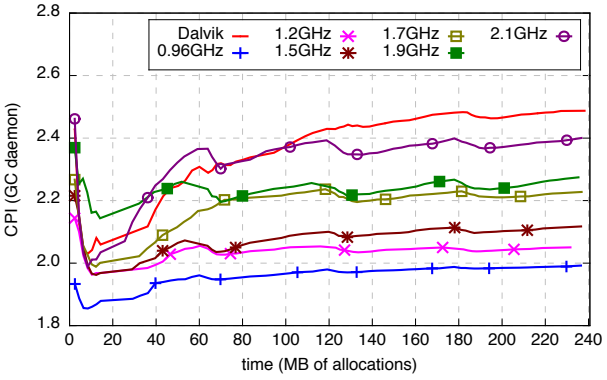
(a) lusearch



(b) xalan

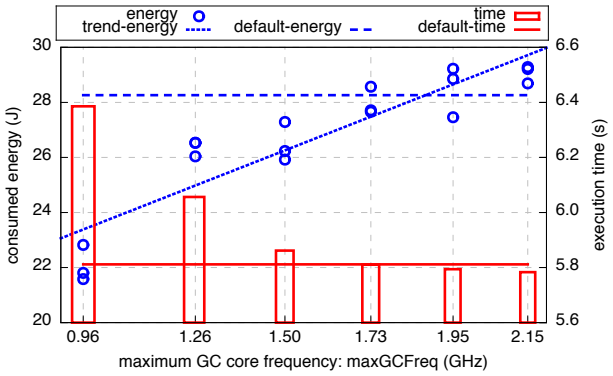


(c) jack

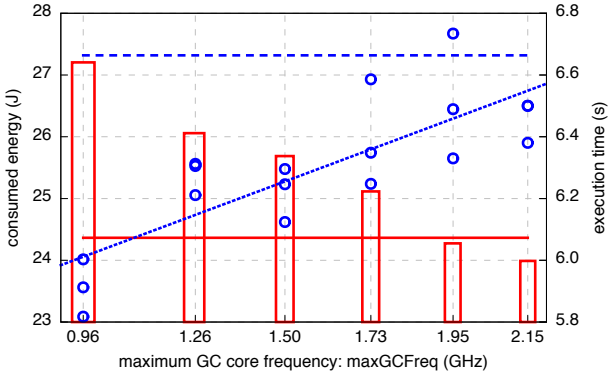


(d) javac

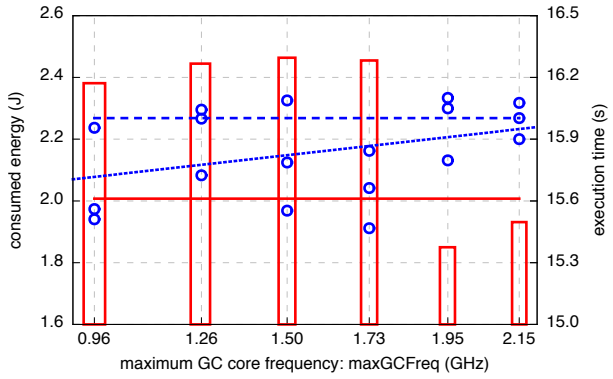
Figure 3. Cumulative average GC daemon CPI



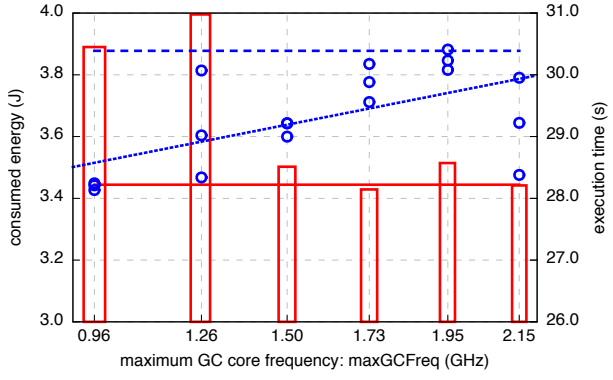
(a) lusearch



(b) xalan



(c) jack



(d) javac

Figure 4. Total consumed energy and execution time

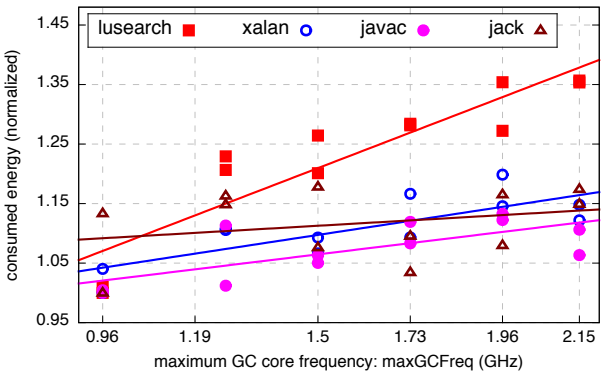


Figure 5. Energy consumed relative to default Dalvik

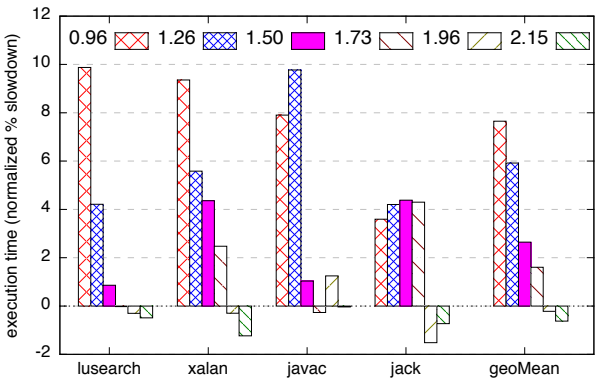


Figure 6. Execution time slowdown relative to default Dalvik

is less sensitive to the value of the GC core frequency cap; it has throughput penalty at worst 4 %.

Importantly, it is possible to obtain significant energy savings for modest reductions in throughput. For example, at the 1.5 GHz 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 %!

4.2 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 main 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 timeframes yield a better understanding of the distribution and impact of pauses [20, 30, 41]. 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 7 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).

Table 2. Scheduling statistics normalized to default Dalvik

GC core 0 frequency cap	Migrations		Switches	
	lusearch	xalan	lusearch	xalan
0.96 GHz	1.23	1.01	0.97	0.99
2.15 GHz	0.94	0.95	0.98	0.98

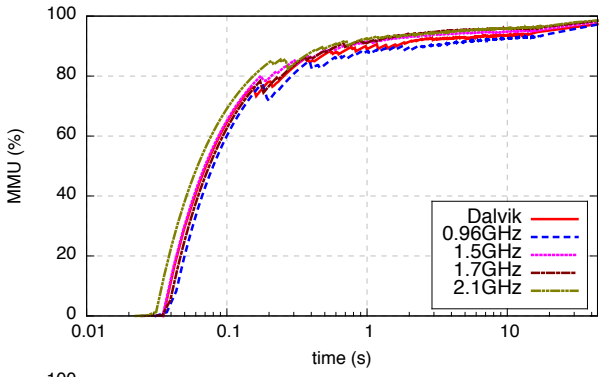
The GC-aware governor with 2.15 GHz cap has the best MMU curve on the DaCapo benchmarks lusearch and xalan (Fig. 7a). This can be explained by the fact that pinning works as a hint to the scheduler to improve its scheduling decisions. As circumstantial evidence Table 2 shows some key scheduling statistics:

- *migrations*: the number of times threads migrate from one core to another; and
- *switches*: the number of times cores switch from one thread to another, including both *voluntary* and *involuntary* switches.

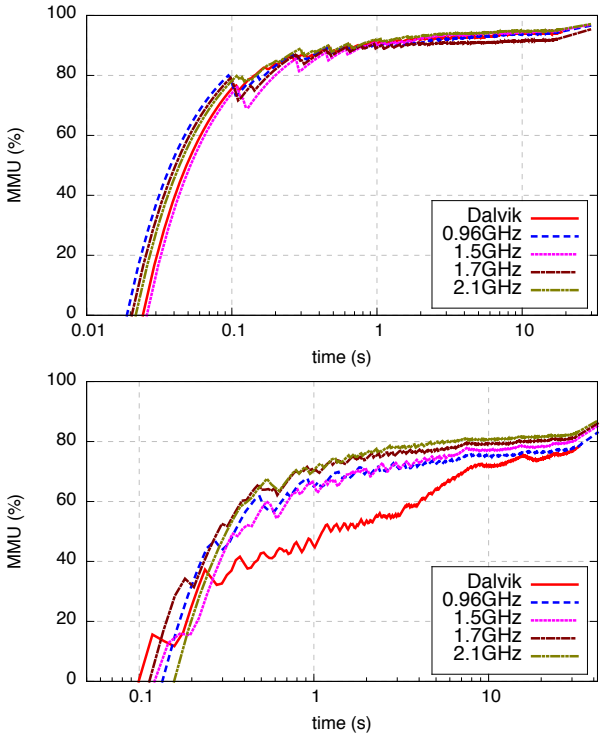
The GC-aware governor at 2.15 GHz reduces the number of task migrations on lusearch and xalan by 6 and 5 %, respectively.

One might consider MMU for jack to be quite unintuitive as 0.96 GHz has both smallest maximum pauses and best overall utilization. However, note that applying the GC-aware governor with a GC core cap of 0.96 GHz, 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. 1 (bottom). For javac (single threaded), the mutator spends more time waiting for collecting a relatively large heap (maximum heap size 14 MiB). 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) DaCapo: lusearch (top); xalan (bottom)



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

Figure 7. Minimum mutator utilization

Table 3. Total heap volume over time relative to default

app	GC core 0 frequency cap (GHz)				
	0.96	1.26	1.50	1.73	2.15
	total heap volume (% of default)				
lusearch	0.06	−0.57	−0.06	0.12	−0.15
xalan	0.04	0.12	0.13	0.14	−0.13
javac	−0.28	−0.92	−0.10	0.30	1.45
jack	0.92	1.00	−0.14	−0.80	−0.06

4.3 Impact on Heap Volume

Slowing down the core of the GC daemon affects mutator thread scheduling, and so also the timing of collector cycles with respect to mutator execution. The Dalvik concurrent collector does not permit threads to allocate beyond `softlimit`; any thread attempting to do so during the collector cycle will be forced to wait. As a result we do not expect to see significant variation in heap size over time. To demonstrate that the speed of the GC daemon has little impact on total heap volume Table 3 reports the integral (i.e., sum) for the `softlimit` as it varies over time (measured in bytes allocated): $space(t) = \int_0^t \texttt{softlimit}_t \, dt$, relative to the default collector. It is clear that heap volume is unaffected by pinning and slowing down the concurrent collector.

5. Discussion

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

5.1 Choice of CPI 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 still correlate well with CPI, at least for GC in our benchmarks.

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 Dalvik 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 resulting from the most recent (full) heap GC. Thus, both the benchmark and the `targetutil` affect the GC workload,

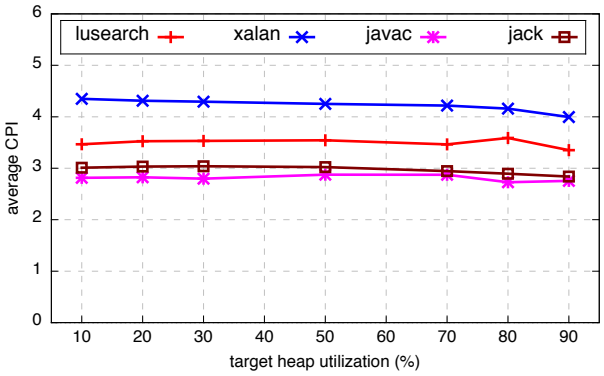


Figure 8. Average overall CPI varying `targetutil`

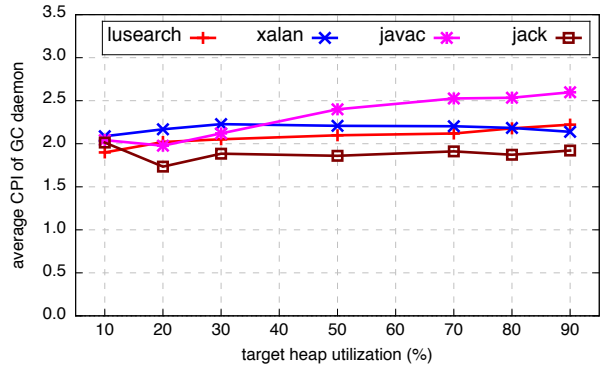


Figure 9. Average GC daemon CPI varying `targetutil`

in the number of instructions executed, in the mix of those instructions, and in the scheduling of the collector. Figure 8 shows the average CPI over 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 Figs. 9 and 10. 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.

5.2 Android Run Time (ART)

Recently, Google announced ART, a next-generation run-time system for Android 4.4 "Kitkat" that relies on somewhat aggressive ahead-of-time compilation of apps [6], and which will is to replace Dalvik. We regard exploring ART's behavior as an important continuation of our study. Given that our analysis methodology requires covering a large set of metrics and configurations across system layers, we preferred not to include conclusions based on the subset of our experiments generated with ART before we fully cover all possible configurations. ART also implements several run-time system improvements that will affect our results:

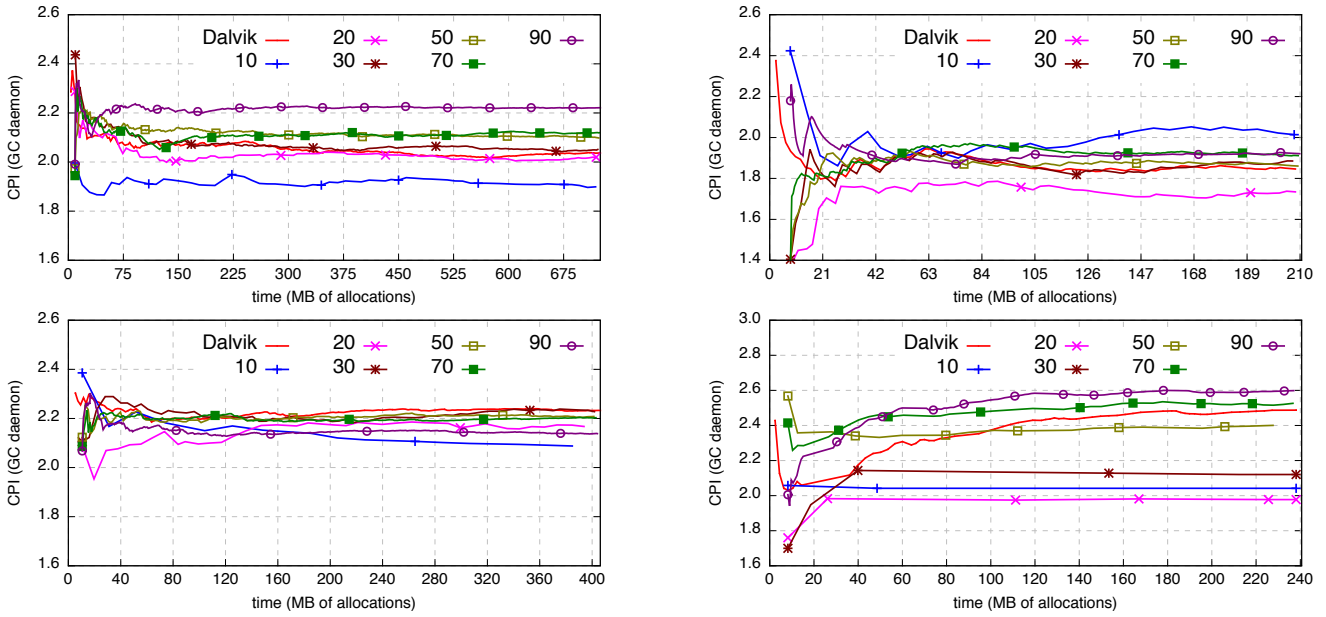
- the marking phase has one stop-the-world phase for marking the roots instead of two (no longer stop-the-world for weak references);
- introducing a pseudo-generational *sticky* collector to deal with short-lived objects;
- dedicating a separate heap for large objects; and
- enabling parallel processing during marking of mutator roots.

We do not expect the CPI of the GC daemon to change significantly as its work is dependent on 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 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.

6. Related work

Several studies have addressed GC requirements when deployed in restricted environments. Chen et al. [18, 19] tune the collector to enable shutting down memory banks that hold only garbage objects. Griffin et al. [25] implement a hybrid mark-sweep/reference-counting collector to reduce power consumption. Sartor and Eeckhout [43] 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 [15, 47]. Occasionally, GC has been evaluated as an asymmetric activity that can be isolated on a separate core [15, 43]. However, their methodology relies on dedicated hardware which is impractical 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 [38, 39]. Hao et al. [26] presented an approach for power estimation based on offline 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 [51]. However, the relation between the WCET analysis and power consumption is less understood, because of the challenge in assuming a direct



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

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

Figure 10. Cumulative average GC daemon CPI varying `targetutil`

correlation between execution bounds that involve different components such as compiler, scheduler, and hardware specifications [2, 7, 51].

In this paper, 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. [44] suggest methods to estimate performance requirements of software tasks using simulation, our approach is based on the observation made by Sherwood et al. [45] 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 [29], 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 [32]. We show that energy for GC can be reduced by simple integration across system layers (i.e., managed runtime system and governor). Our results differ from the work of Hao et al. [26] in fitting the run-time performance within the whole system stack (i.e., hardware, kernel, and power management). The results generated in this paper reflect real executions involving synchronization overhead, induced by spin-locks and context-switching as noted by others [24, 37].

7. Conclusions

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).

Acknowledgments

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

References

[1] Allegro MicroSystems, LLC. ACS714: Automotive grade, fully integrated, Hall effect-based linear current sensor IC with 2.1 kVRMS voltage isolation and a low-resistance current conductor. 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 *International Symposium on Computer Architecture*, pages 350–361, June 2003. doi: `10.1109/ISCA.2003.1207013`.

[3] Android. URL http://source.android.com.

[4] Android Performance Tips. URL http://developer.android.com/training/articles/perf-tips.html.

[5] 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.

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

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

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

[9] D. F. Bacon, P. Cheng, and S. Shukla. And then there were none: a stall-free real-time garbage collector for reconfigurable hardware. *Commun. ACM*, 56(12):101–109, Jan. 2013. doi: 10.1145/2534706.2534726.

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

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

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

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

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

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

[28] C. Isen, L. John, J. P. Choi, and H. J. Song. On the representativeness of embedded Java benchmarks. In *IEEE International Symposium on Workload Characterization*, pages 153–162, Seattle, Washington, Sept. 2008. doi: 10.1109/IISWC.2008.4636100.

[29] 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.

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

[31] 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.

[32] 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.

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

[34] 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.

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

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] Qualcomm. *Snapdragon S4 processors: System on chip solutions for a new mobile age*, Oct. 2011. URL http://tinyurl.com/q2yzn9r.

[43] 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.

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

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

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

[47] 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.

[48] V. M. Weaver. Linux perf_event features and overhead. In *International Workshop on Performance Analysis of Workload Optimized Systems*, FastPath, 2013. URL http://researcher.watson.ibm.com/researcher/files/us-ajvega/FastPath_Weaver_Talk.pdf.

[49] 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.

[50] N. H. E. Weste and K. Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective*. Addison-Wesley, 1985.

[51] 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.