A Study of Mobile Device Utilization

Cao Gao[1], Anthony Gutierrez[1], Madhav Rajan[2], Ronald G. Dreslinski[1], Trevor Mudge[1], and Carole-Jean Wu[2]

[1]University of Michigan, Ann Arbor, {caogao, atgutier, rdreslin, tnm}@umich.edu
[2]Arizona State University, {mrajan, carole-jean.wu}@asu.edu

***Abstract*—Mobile devices are becoming more powerful and versatile than ever, calling for better embedded processors. Following the trend in desktop CPUs, microprocessor vendors are trying to meet such needs by increasing the number of cores in mobile device SoCs. However, increasing the number does not translate proportionally into performance gain and power reduction. In the past, studies have shown that there exists little parallelism to be exploited by a multi-core processor in desktop platform applications, and many cores sit idle during runtime. In this paper, we investigate whether the same is true for current mobile applications.**

We analyze the behavior of a broad range of commonly used mobile applications on real devices. We measure their *Thread Level Parallelism* (TLP), which is the machine utilization over the non-idle runtime. Our results demonstrate that mobile applications are utilizing less than 2 cores on average, even with background applications running concurrently. We observe a diminishing return on TLP with increasing the number of cores, and low TLP even with heavy-load scenarios. These studies suggest that having many powerful cores is over-provisioning. Further analysis of TLP behavior and big-little core energy efficiency suggests that current mobile workloads can benefit from an architecture that has the flexibility to accommodate both high performance and good energy-efficiency for different application phases.

I. Introduction

Nowadays, mobile devices are gradually taking over the functions of traditional desktop applications. High-definition video playback, interactive games and web browsing are commonly supported by the latest smartphones and tablets. These performance-intensive tasks need powerful hardware support, which drives microprocessor vendors to continuously produce better mobile CPUs. Given the strict power budget of mobile devices, vendors reached the limits of frequency scaling quickly and turned to multi-processors. The first dual-core smartphones, such as Galaxy S II and HTC Sensation, came to market in 2011. Most of the high-end smartphones released in 2012 were dual-core or quad-core; in April 2013, the Samsung Galaxy S4 was released with the *Exynos 5 Octa*, which uses ARM's big.LITTLE architecture and has a total of eight cores. Mediatek shipped their octa-core SoC in late 2013, and Qualcomm announced their octa-core CPU with eight A53s in early 2014 as well.

However, some recent smartphones are still equipped with dual cores. Apple's new A8 Chip for the iPhone 6, released in September 2014, uses a dual-core CPU and still provides satisfactory performance. This leads to the question: How much of the computation potential residing in multi-core CPUs is actually being utilized? On the desktop end, Blake et al. [1] did a study on *Thread Level Parallelism* (TLP) on a suite of representative desktop applications. Their work was to measure the core utilization in modern multi-core CPUs, and they suggested that the number of cores that can be profitably used is less than 3 for most commonly used applications. It is possible that mobile device applications have similar characteristics and cannot effectively utilize a quad-core CPU, let alone hexa- and octa-core. Moreover, the GPU, DSPs, and ASICs in these systems already exploit much of the parallelism, leaving little for the CPU.

To make some observations about the benefit of multi-core, we performed two preliminary experiments on an up-to-date quad-core mobile device platform. First, we measured how many cores are actually activated by the Android OS when running an application. We found that the fourth core was only activated in 2 out of 21 apps, and the OS also shut off the third core for nearly half of the apps. Note that activation does not mean the core is in use; it only means the OS thinks that the core *might* be used. Second, we overrode system setup and manually set the number of activated cores in the system. Then we ran a browser benchmark [2]; we saw a significant performance improvement from single-core to dual-core, but negligible improvements from dual-core to triple- and quad-core. Both of these results show rather modest gains from high numbers of cores (here more than 2). In all, to measure how much parallelism actually exists is helpful to: a) inform vendors and prevent them from over-provisioning hardware that cannot be effectively used, b) highlight the need to find more parallelism, c) provide suggestions for a better design.

In this work, we analyze a broad range of popular mobile applications to determine how the growing number of cores are utilized. We measure the Thread Level Parallelism (TLP) of these applications. The results show that mobile apps are utilizing less than 2 cores on average, which means multiple cores are used rather infrequently. A small TLP scalability is observed for most applications, and increasing the number of cores has diminishing return on TLP. Even in heavy-load real-world scenarios with background applications or multi-tab browsing, there is still not enough work to keep utilization high. Due to the physical constraint and interactive user pattern, mobile applications tend to have less parallelism to exploit than desktop applications. The GPU and mobile co-processors on chip also reduce CPU load. All these factors, and the history of the slow pace of exploiting parallelism in desktop and mobile software environments [1, 3], indicate that having many powerful cores is over-provisioning. Further analysis suggests that current mobile applications can benefit from a system with the flexbility to satisfy high performance and good energy-efficiency for different application phases. We find that TLP behavior exhibits short peaks and long valleys rather than remaining constant. Peaks require high perfor-

mance, but not necessary good energy-efficiency because these peaks are usually short, meaning that power has less affect on overall energy consumption. Valleys, on the other hand, desire better energy-efficiency because they do not require high performance but usually dominate the application execution. There is also a number of other research opportunities that arise, such as building accelerators or customized hardware to further reduce the thread's TLP peaks with better energy efficiency, or building better OS infrastructure to utilize mobile heterogeneous systems.

To summarize, we make the following contributions:

- We construct a suite containing representative Android applications from a variety of categories, as well as their corresponding test actions.
- We measure the Thread Level Parallelism (TLP) of mobile applications on current mobile device platforms and show it is less than 2 on average.
- We observe diminishing returns of TLP when increasing the number of cores. Heavy-load test cases also show low TLPs, which suggests there is not a lack of hardware resources. Both demonstrate that having many powerful cores is over-provisioning.
- We make the case for the need of a flexible system that can accommodate both high performance and good energy-efficiency for different program phases.

II. Motivation

We perform two preliminary experiments on an Origen board, a current mobile device platform with a quad-core 1.4GHz ARM Cortex-A9 CPU. First, we measure how many cores are actually activated by the OS when running an application. The Android system employs a CPU governor which turns individual cores on or off and changes their frequencies based on CPU loads. When it finds that a core sits idle for most of the time, it will turn it off to save power. We run a suite of commonly used applications with the default governor—*ondemand*. We find that the fourth core is only activated in ***2*** out of the 21 apps we tested. For nearly half of the apps, the OS also shuts off the third core for most of the time. We show part of our results in Fig. 1. We plot a breakdown of the time percentage that each system configuration spends for these applications. Different colors represent system configurations with different numbers of cores activated. The only app in this graph that activates the fourth core is Google Maps. Browser and Jetpack (a game) do activate three cores for most of the time, but Facebook does so only for half of the time, and Email never. Note that activation does not mean utilization; it only means the OS thinks that this core *might* be utilized. The core could still sit activated and idle at the same time. We will show the core utilization in the results section.

In the second experiment, we override system setup and manually set the number of cores activated in the system. Since web-browsing is among the most commonly used features on mobile devices [4], we run a browser benchmark, BBench [2] on two browsers, and compare the performance of different CPU configurations. We plot the results in Fig. 2. The score is the time taken to render the complete set of webpages in BBench, and lower score means better performance. It is clear from the graph that a single-core system suffers from poor performance. However, the performance gain is negligible from dual-core to triple-core and quad-core. It may seem confusing why the OS activates 3 cores for BBench when there is such little performance improvement. Actually it again proves that activation does not mean utilization; the *ondemand* CPU governor in Android OS is performance-aware and will keep the core activated unless it is very unlikely to be utilized [5].

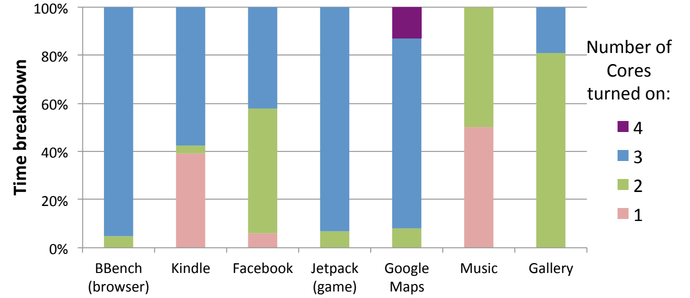


Fig. 1: Time breakdown of number of cores activated by the Android OS. For most of the applications, the fourth core is always shut down, For some of them, namely Email, Facebook, Music and Gallery in this graph, most of the time the third core is not activated as well.

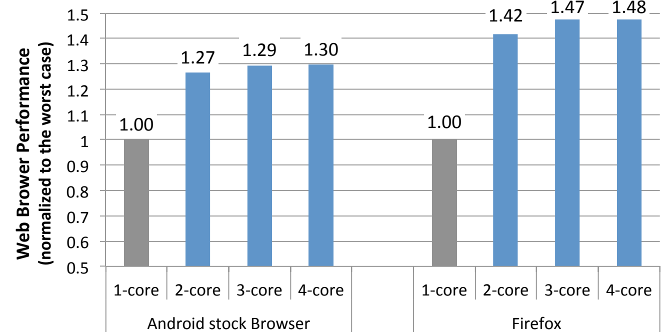


Fig. 2: BBench score for different number of cores. The scores here are webpage rendering time, so lower is better. Note the tiny difference in performance among 2 core, 3 core and 4 core configurations.

Both of these tests suggest a more thorough quantitive investigation of quad-core CPU utilization.

III. Background

A. TLP

To evaluate the utilization of a multi-core system, we need a metric for system profiling. A commonly used metric would be CPU utilization, which is simply the overall average CPU usage during runtime. However, it would underestimate the parallelism in mobile applications. Most of these apps are interactive, and there is a large portion of idle time for interactive applications. The program itself could be highly parallelized with a high utilization during busy time. However, it sits in the idle state waiting for user input for most of the total running time, which would drag down the average utilization number. To avoid this bias, we use *Thread Level Parallelism*

(TLP) [1, 6]. TLP is defined as the machine utilization over the non-idle portions of the benchmark's execution. The formula for TLP is given by Equation 1:

$$TLP = \frac{\sum_{i=1}^{n} c_i i}{1 - c_0} \tag{1}$$

where c_i is the fraction of time that i cores are concurrently running different threads, and n is the number of cores. Specifically, c_0 represents idle time fraction, which is excluded because it does not count towards the program's parallelism. Note that TLP is not a performance metric; the software could still spawn threads that do not perform useful work. Nevertheless, it is the natural metric to measure multi-core utilization, especially for interactive applications like the ones on a smartphone. The TLP serves as a good indicator of the number of processors needed to support the execution of a parallelized workload.

B. Early Studies on TLP

Flautner et al. [6] proposed the definition of TLP in 2000. At that time, multi-core was mostly exploited in research labs and appeared only in workstations and servers. They performed a study of TLP on desktop applications and found that a dual-core system improves the responsiveness of interactive programs. However, they also showed that desktop applications leveraged TLP very sparingly. This result was echoed 10 years later by Blake et al. [1] with a similar study of TLP of contemporary software and hardware, when multi-core had become the norm rather than the exception in home and office desktops. They reported that 2-3 cores were more than adequate for almost all but a few domain specific applications like Video Authoring. After observing low single-thread performance could have a small impact on the TLP, they claimed that software is lagging behind and is the main limiting factor in TLP.

Smartphones were already becoming popular during the time when Blake et al. presented their results, and they have continued to supplant desktops for many applications. To reflect this it is important to analyze TLP behavior on mobile devices because the original studies did not. Besides exploring a different hardware platform, we are also using a slightly different set of benchmarks from the original work. Some categories of desktop applications are rarely seen on mobile devices, such as Video Authoring and professional Image Authoring.

IV. Methodology

A. System Setup

We use the Odroid XU+E board. It has a Samsung Exynos 5410 SoC, which contains an ARM big.LITTLE octa core of four 1.6GHz A15s and four 1.2GHz A7s. Each core has its own 32KB/32KB L1 instruction and data cache; the four A15s share a 2MB L2 cache and the A7s share a 512KB L2 cache. Either four A15s or four A7s can be enabled at the same time, but not a mixture of them. The Odroid board has a PowerVR tri-core GPU running at 480MHz and with 2GB main memory. It also has an on-board current/power semiconductor sensor which measures the current/power consumption of CPUs, GPU and memory separately[1]. We run Android version 4.4.2 (Kitkat) and Linux kernel version 3.4.5. We choose the use the ART runtime instead of the older Dalvik.

[1]For CPU power, we measure the sum of power of big and little clusters.

B. Measurement

1) TLP: To get the TLP number, we track all the context switches that happen in the system, which reveals the information about the status of each core. For instance, a context switch from *SurfaceFlinger* to *swapper* on Core #0 indicates this core has turned from busy to idle. This information gives us the number of running cores at any time, which is sufficient to calculate TLP. Moreover, we can get information about which thread is running and filter out observation overhead threads. For example, we treat *adbd*, the Android Debug Bridge thread, as *swapper*. The core that is running *adbd* would then be treated as idle, preventing an overestimation of TLP. We use *ftrace*, a Linux kernel internal trace, to get context switches. The data we gathered contains task names, ids, CPU Number, and timestamp.

2) GPU utilization: For the PowerVR on the Odroid board, we directly read GPU utilization numbers from the sysfs interface provided.

C. Benchmarks

In this work we test a diverse range of real-world Android applications. We prefer applications that are: a) most widely used by users; b) from a broad range of diversified categories. Based on these requirements, we choose 18 top-pick applications from the Google Play Android App store, and 4 native ones in the Android OS. This means they are the applications commonly used in their category and are thus representative of current mobile software. They come from 10 different categories: browser, video player, music player, image viewer, communication, games, social networking, navigation, office, and file browser. They make use of important hardware resources on a mobile device (CPU, GPU, co-processors, etc).

We then perform test actions on each of the testing applications. Three applications (browser, Adobe reader and MX Player) are so widely used that they have already been included in some benchmarks [2, 7, 8], therefore we leverage the existing work and use their implementation directly. For other applications, we design a series of actions that cover most typical functions of the application under test. We also refer to the study in [9] on mobile applications usages, including what the popular applications are and how long each session (from opening to closing) normally last. Test actions on the Odroid board are automated using android adb commands and RERAN, a record and replay tool for Android OS [10]. All experiments are repeated at least 5 times for more accurate results, and applications that require an internet connection are repeated for at least 10 runs. We observe a low standard deviation of TLP results as shown in Section. V-A.

It is also important to test TLP of scenarios with background applications, to reflect common daily usage. We also test three applications with a set of other applications running in the background. The three applications under test are Angry Birds, Adobe reader, and Chrome, while the background applications are Hangout, Spotify, and Email.

We briefly introduce each application, and its corresponding test actions in the following subsections.

1) Web browser: We use the Realistic General Web Browsing (R-GWB)[8], an automatic webpage rendering benchmark. It comprises offline pages of several most popular webpages, all of which utilize modern web technology such as CSS, HTML, ash, and multi-media. During the experiment, MobileBench uses JavaScript to load each webpage and then scroll over it with a pre-set speed. By doing so, it simulates an actual web browsing scenario. We run MobileBench on three popular browsers: the Android stock web browser, Firefox, and Chrome. For each test, we iterate through the MobileBench webpage set five times, and profile the third one. We do not impose any other user input as MobileBench is automatic itself.

2) Video Player: We use two applications: MXPlayer, a video playback application and Netflix, an online streaming application. We test both applications by playing a video for 30 seconds, with a 1 second pause at the 15th second of each of the tests.

3) Music Player: We use the Android stock music player and Spotify for testing music players. Spotify is a music player that supports online music streamling. We test both of the two apps by running a series of actions including open new song, jump to an arbitrary position of the song (not in spotify), and open another song.

4) Image Viewer: We use Android stock image viewer (Gallery) and Instagram for testing image viewers. We test it by opening images, scrolling between images, opening another image and new folders, and playing a slideshow for a couple of seconds. Instagram is mobile photo-sharing service. Users take pictures share them on a variety of social networking platforms. We test it by scrolling new feeds, opening a picture, applying Amaro filter and changing brightness to 75, and then sharing the picture.

5) Communication: We use Google Hangout and Skype here. We test both of these applications by initiaing a 1 minute video call, 30 seconds in foreground and 30 seconds in background (approximating an audio call).

6) Games: The three games we choose are Angry Birds, Fruit Ninja, and Jetpack. Angry Bird is a puzzle video game; in the game, players use a slingshot to launch birds at pigs stationed on or within various structures, with the intent of killing all the pigs on the playing field. We play this game by entering the first stage, firing two birds with one miss and one hit. Fruit Ninja is an action game with lots of floating objects and high-frequency user input. It represents a more intensive mobile game comparing to Angry Birds. We play this game in the "zen mode", where fruit keeps spawning for 90 seconds. During testing, the tester keeps sliding with a constant frequency horizontally in the upper middle of the screen. Jetpack Joyride is a game where the player tries to control the rider to avoid barriers and collect coins. We play this game by tapping the screen at a regular frequency for 45 seconds.

7) Social Networking: We choose the apps from two major Social networking service providers, Facebook and Twitter. When testing Facebook, we scroll feeds, open up the pictures in the feeds and browse the profile of the user. The action of testing Twitter includes clicks on tweets, checking out picture in the tweets and looking at the profile of the tweeter.

8) Navigation: The most popular navigation application on Android platform is Google Maps. We test Google Maps by searching directions between airports in New York city: we search driving directions from Newark to JFK, then public transportation from JFK to LaGuardia. We also save an offline map of New York city to avoid fetching map data from the internet during testing.

9) Office: The office category contains a broad range of commonly used apps. We test the following: 1) Android stock Email — we test it by writing an email and save that as a draft, sending it, checking and downloading new email. 2) Adobe reader — actions here include opening a pdf, zooming in and out, scrolling pages and searching for a keyword. 3) Amazon Kindle — actions here include opening a book, scrolling and jumping to arbitrary positions.

10) File Browser: We test Dropbox and ES File Browser. For Dropbox, we open and change folders, sort the content in the folder, do searching and editing new text file. For ES File Browser, we open the folders on the SD cards, scroll the images in it, sort and change the view of the folder.

11) Background: The background applications we choose are Google Hangout (video chatting), Spotify (playing music), Email (checking emails)[2]. With those applications we test Fruit Ninja, Maps, and Adobe Reader.

V. RESULTS

In this section, we present our experimental results and analysis of mobile device utilization, specifically on CPU and GPU. First, we show that current mobile applications have a rather low average TLP on modern mobile device platforms. We show that increasing the number of cores has diminishing returns on TLP. Even some heavy-load real-world scenarios do not use many cores. High GPU utilization also indicates that some of the parallelism is already offloaded from the CPU to the GPU. All these factors, and the history of the slow pace of exploiting parallelism in desktop environments [1], suggests that having many powerful cores is likely to be over-provisioning.

A. Overall Results

We list a summary of the results in Table I. Each row in the table shows the TLP and standard deviation (σ) for an application type. The first line, "System", refers to the plain Android OS testing environment without any application running; the last line, "*Average*", is the average of the statistics of all tested applications. The standard deviation of the TLP over runs for each application is low. This indicates the tests are reproducible and insensitive to user input variation.

We make two observations based on these results:

1) All the applications demonstrate some, but quite limited TLP.

For Android, even in a case where a developer writes code with no awareness of multi-threading, a number of threads are still created for external I/O, garbage collection, graphics rendering, etc. This means that even a programmer

[2]During the test we automatically send an email to the account on board from the host machine every half minute

Category	App	TLP	σ (TLP)
System	[None]	1.03	0.00
Browser	Stock Browser	1.47	0.03
	Firefox	1.31	0.02
	Chrome	1.66	0.01
Video Player	MXPlayer	1.34	0.01
	Netflix	1.53	0.07
Music Player	Stock Music	1.29	0.03
	Spotify	1.23	0.05
Image Viewer	Stock Gallery	1.46	0.03
	Instagram	1.31	0.03
Communication	Google Hangout	1.82	0.15
	Skype	1.55	0.13
Games	Angry Birds	1.31	0.08
	Fruit Ninja	1.40	0.12
	Jetpack	1.54	0.09
Social Network	Facebook	1.43	0.04
	Twitter	1.32	0.04
Navigation	Google Maps	1.59	0.06
Office	Stock Email	1.52	0.04
	Adobe Reader	1.30	0.05
	Kindle	1.45	0.01
File Browser	Dropbox	1.33	0.02
	ES file browser	1.22	0.02
Background	Back_Fruit	1.65	0.12
	Back_Maps	1.91	0.11
	Back_Adobe	1.60	0.14
Average		1.46	0.06

TABLE I: TLP results for the Odroid board — using ondemand governor.

with no idea about multithreading could benefit from parallel processing on different CPU cores. Moreover, many software developers are aware of the multi-core hardware they are using and write applications explicitly with multiple threads.

However, the parallelism we observed is generally still quite low. On average, we see a TLP of 1.46. The application with the highest TLP, Google Hangout, has a TLP of just 1.8. Applications like Music and File Browser have rather low TLP, around 1.2 to 1.3. This result shows, on average, the system is using less than 2 cores.

2) Multi-core is utilized infrequently.

We present a time breakdown of how the multi-core system is utilized in Fig. 3. The big pie chart on the right shows an average result of all application categories we tested. The smaller pie charts show the breakdown of three representative application categories. We observe a very low 4-core and 3-core utilization; on average only 0.68% of all the non-idle time is the system fully utilized (all four cores running), and 5.81% of the time when three cores are used — this means there is only a small amount of time that four or three cores are being utilized at the same time.

B. Core Scaling

Microprocessor vendors put more cores on a chip to exploit more parallelism in the system. Therefore, it is important to consider the change of TLP as the system scales from 2 to 3 to 4 cores.

We show our TLP results in Fig. 4a. We change the number of active cores in the system and repeat the same experiments

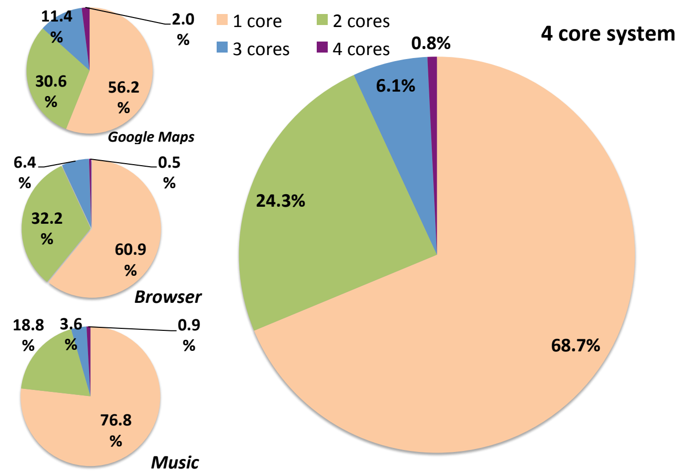


Fig. 3: Time breakdown of how the multi-core is utilized. The big pie chart on the right shows an average result of all application categories we tested. The smaller pie charts show the breakdown of three representative kinds of apps: Google Maps with the highest TLP, stock Browser, and stock Music with a low TLP.

for TLP. For each category, the leftmost bar shows the TLP when 4 cores are kept activated. The middle one shows the TLP when the fourth core is shut down and the remaining 3 cores are kept activated. Similarly, the rightmost bar shows the system configuration with 2 cores. The results demonstrate:

1) Increasing the number of cores has little impact on TLP.

On average, TLP increased by 7.03% when we switch from a 2-core system to a 3-core system, and only 3.47% from a 3-core system to a 4-core system.

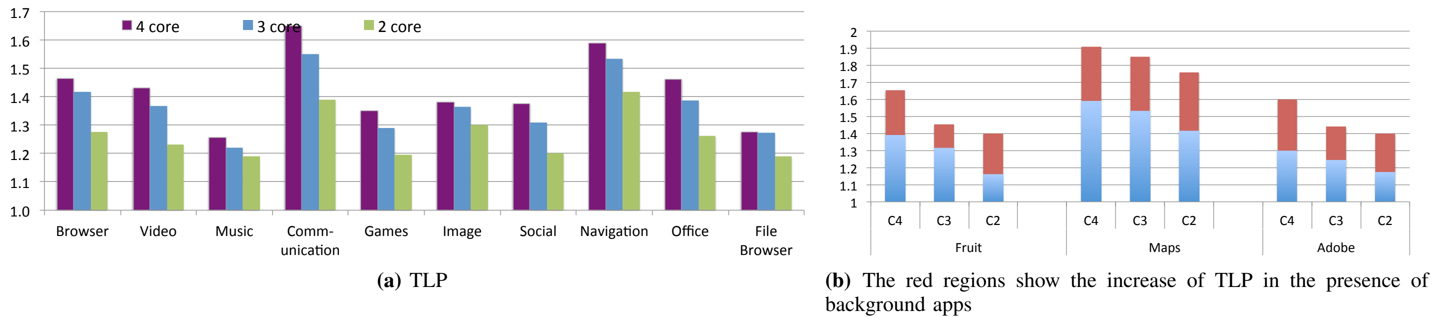
2) Most applications show some scalability, but not much.

Particularly, Games, Navigation, Office and Social apps show over a 10% increase of TLP from a 2-core to a 4-core system. File manager only shows a 5.6% increase. Most apps show small increases in TLP from 3-core to 4-core which are below 4%. This indicates that the software does not generate many concurrently parallel threads during its execution.

C. Heavy Load Scenarios

Intuitively, a multi-core system is beneficial when the CPU load is high. In this section we test the TLP of a couple of heavy load scenarios.

1) Background Applications: It is now common to have several applications like music or email checking running in background concurrently with a foreground application. One argument that favors having more cores is that they can boost performance of such scenarios. We measure the TLP of several applications with a set of background applications running concurrently (described in Section. IV-C), and present our result in Table I and Fig. 4b. The results demonstrate that background applications only lead to limited increase in TLP, and we are still not fully utilizing all four cores with background activities.



(a) TLP

(b) The red regions show the increase of TLP in the presence of background apps

Fig. 4: Overall TLP result

2) Multi-tab Web Browsing: We test multi-tab scenarios to see how mobile browser applications exploit parallelism under high load circumstances. We measure the TLP and performance of MobileBench on different CPU configurations. We run one, two and three MobileBench tabs concurrently and measure the average of the metrics. We manually switch between tabs constantly to make them appear in the foreground for similar amount of time. Fig. 5 shows the results. The maximum TLP is still below 2 for big cores and 2.2 for little cores, and there is significant performance degradation when increasing the number of tabs.

The reason for low TLP here is that even for these two cases, we do not see a real "multi-tasking" scenario; instead, we see a main task and several light-load tasks, and that does not exhibit a high TLP. For instance, for multi-tab browsing, only the visible web pages will be loaded at regular speed, and all background pages will be given much less priority and use less CPU. Energy and thermal constraints are tight in mobile devices. It might be that the developers realize there is not enough need for implementing a fast but energy-hungry multi-tab browser for mobile phones. Reasons may include small display size or typical user behavior. In other words, the physical constraints and use pattern could reduce the amount of parallelizable work of mobile applications. In the future, we may have phones with bigger screens and higher resolutions, but human user perception will not change. On the other hand, the desktop TLP study by Blake et al. [1] showed the TLP of desktop applications remained relatively low, even after 10 years of effort writing parallelized software. Similarly, we have not seen a significant increase in TLP compared work from one year ago [3]. Clearly, parallelizing software is an extremely challenging problem, particularly for desktop/mobile applications.

D. Little Cores

We also perform the same set of TLP tests on the little cores in order to see how a less powerful CPU would affect TLP. We present our results in Fig. 6. Both the TLP and the average percentage of time that four or three cores are utilized has increased when using little cores compared to big cores. One reason is that tasks on more powerful cores run faster and finish earlier, reducing the overlap between them. This result suggests that as CPU architecture designs improve, exploiting TLP will be harder. The CPUs will be less utilized if software developers fail to produce better parallelized program.

Nevertheless, we still have TLP less than 2. Further suggesting that software is still the main limiting factor in exploiting TLP.

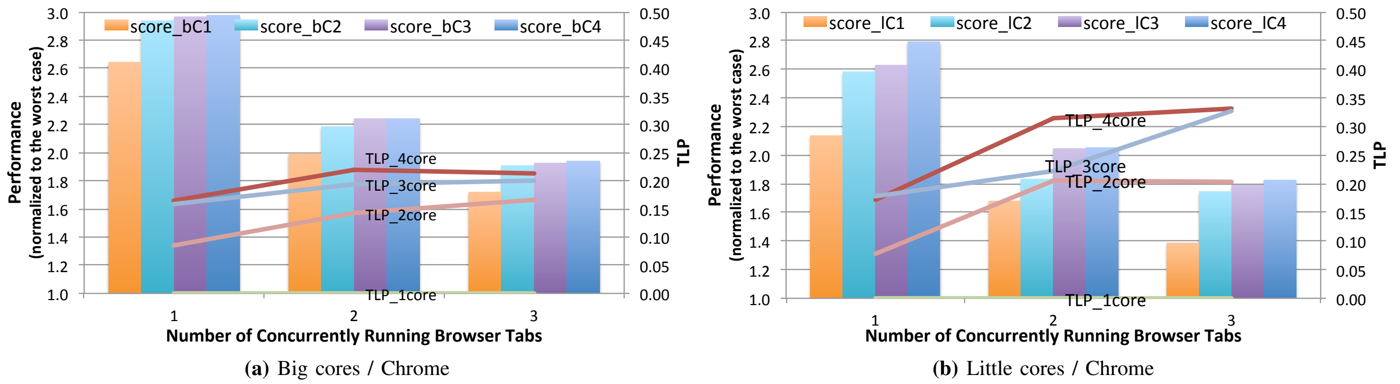
E. GPU

Applications like games and web browsers require large amount of graphical computation. On the hardware side, almost all mobile device SoCs now contain their own GPU units. On the OS side, the Android 2D rendering pipeline has started to support hardware acceleration in Android 3.0 (*Honeycomb*). Hardware acceleration is enabled by default from Android 4.0 (*Ice Cream Sandwich*). Therefore, it is important to analyze the actual utilization of mobile device GPUs.

We measure the GPU utilization of the same suite of applications on the Odroid board. We show our experimental result of GPU usage in Fig. 7. For each category, the leftmost bar shows the average GPU utilization when 4 big cores are kept activated, then 3 big cores, 2 big cores, and little cores. We have not found much variance when we change the number and type of CPU cores. Average GPU utilization is 24.1%, and some specific applications such as games, communcation (chats in the graph) and navigation utilize a considerable amount of the GPU. This is an indication that a part of the parallelism is already offloaded from the CPU to the GPU, which reduces the amount of parallelism that the CPU can exploit.

Given the availability of programmable GPUs, an increasing amount of general-purpose, non-graphics work can be offloaded from the application cores to the GPU or other accelerators for performance and energy efficiency. This led us to examine the energy efficiency of computation offloading for mobile platforms. We analyze several applications on both the CPU and the GPU and present the results in Fig. 8. We run the OpenMP and OpenCL versions of three machine learning algorithms – kmeans, backpropagation (BP), and nearest neighbor (NN) as well as a streaming algorithm Daxpy on a Qualcomm Snapdragon board, one of the few development boards which support offloading for general-purpose GPU applications. We evaluate the machine learning algorithms because these are the important building blocks of application domains such as audio recognition, image recognition/processing, and recommendation algorithms.

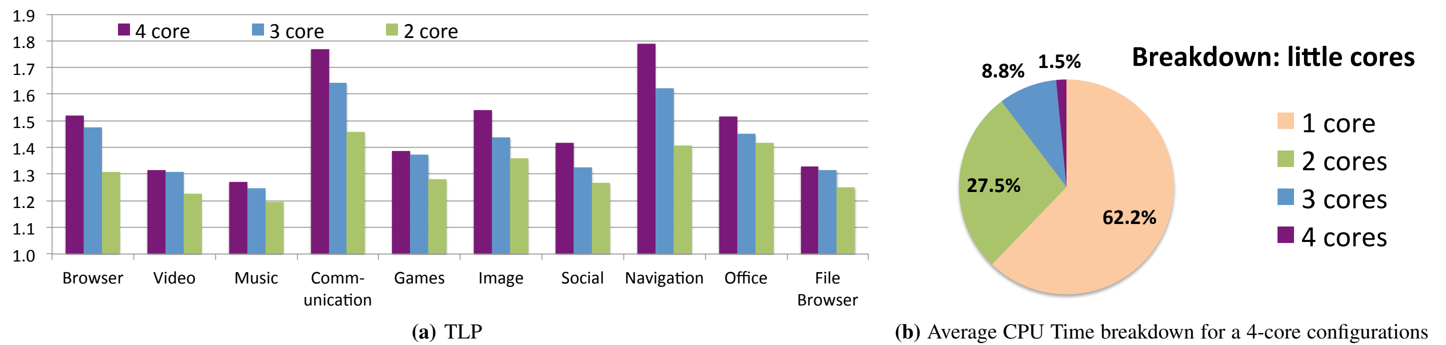
The programs running on the Krait CPU are written in OpenMP whereas the programs running on the Adreno GPU use OpenCL to exploit the heterogeneous GPU compute unit. We find that for Daxpy, the Krait CPU achieves higher energy



(a) Big cores / Chrome

(b) Little cores / Chrome

Fig. 5: Performance and TLP results for browser. Performance are shown in columns and TLP in lines. Performance scores are calculated by taking the inverse of MobileBench rendering time then normalize against the worst score in each graphs. For instance, score_bC4 stands for performance of four big cores, lC2 for two little cores, etc. For each CPU configuration, we test three scenarios: 1, 2 and 3 tabs running MobileBench on Chrome.



(a) TLP

(b) Average CPU Time breakdown for a 4-core configurations

Fig. 6: TLP result for little cores

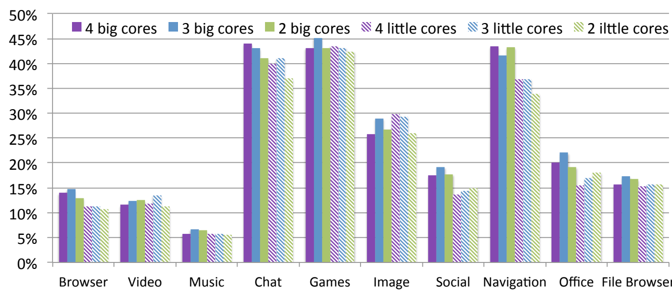


Fig. 7: GPU utilization of different category of apps. Columns with different colors represent system configuration with different number and kind of cores activated.

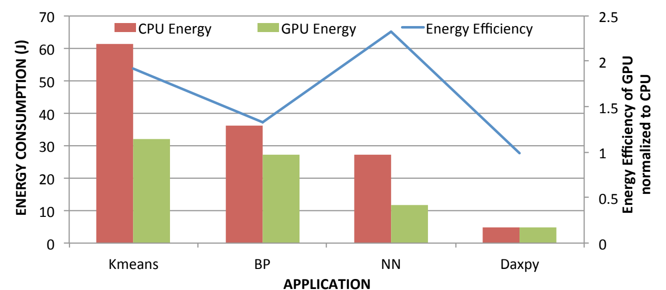


Fig. 8: Energy consumption and energy efficiency (defined as performance per watt) comparison for Krait CPU vs. Adreno GPU.

efficiency than the GPU (less than 1). This means that when the parallelism can be well exploited by the software, Daxpy, and when the instructions are simple enough for the CPU, the energy efficiency of the multicore CPU can be equivalent or slightly higher than that of the GPU. On the other hand, for the machine learning algorithms, GPU offers higher energy efficiency of varying degrees. This suggests that, for software where there is an ample amount of thread- and data-level parallelism, e.g., the machine learning algorithms, and where there are instructions that can be accelerated by the GPU, e.g., the multiplication, sqaure root mathematical functions, it is more beneficial to offload the computation to the GPU or other accelerators for performance and energy efficiency. For workloads with more branch-divergence, or with a small amount of parallelism, it is more efficient to run them on the CPU. In short, the variety of mobile workloads suggests that we should look deeper into building a suitable heterogeneous system to take advantage of the different types and the varying degree of parallelism and better utilizing the existing hardware real estate.

VI. SUGGESTIONS

In the previous section, we demonstrate that current mobile applications are not fully utilizing mobile devices, and simply adding more cores can be over-provisioning. However, it is not clear yet what kind of system is more desired for mobile devices. In this section, we try to shed light on this question by further analyzing the TLP behavior and energy efficiency of current CPUs. We make the following two observations:

a) TLP behavior exhibits short peaks and long valleys rather than staying constant. This suggests that during peak TLP times, higher performance is desired: the system needs to be kept responsive for better user experience, also these peaks are short so the extra computation power will not have a big impact on total energy consumption. During low TLP times when the performance requirement is low, better energy-efficiency is required as any extra power will be a waste and will affect battery life.

b) For current systems, there is a distinct energy-efficiency difference between the big and little cores.

Based on these observations, we argue that current mobile applications can benefit from a system that has flexibility to accommodate both high performance and good energy-efficiency under different applications as well as different program phases. Architectures including heterogenous multi-cores[11, 12] and flexible core architectures[13–15] might be among the possible solutions.

A. TLP vs. Time

We record the TLP over time for applications and present the results in Fig. 9. We choose the 20 seconds[3]of the test starting from launching the applications. For Browser, pronounced peaks can be seen in TLP (Fig. 9a). In MobileBench these occur when the application is launched and new browser webpages are opened (these actions are labeled in circled numbers in Fig. 9a). We also see a shift of peaks towards the right from a 4 core system to a 2 core system corresponding with the actions, which reflects a quicker webpage load time in a 4 core system. For MXPlayer, there are more significant peaks during application launch and when starting, pausing and resuming a video. For Angry Birds, there are less pronounced peaks during runtime but it still shows one when the application launches as well as few others during the game.

These results show that the interactive nature of most mobile application can cause TLP to fluctuating above 2, but the average TLP still remains low. The peaks do suggest a need for multiple cores for quicker response time, which is critical for better user experience. However, multiple cores are used only during brief bursts, mostly at the application launch time. Moreover, even during these peaks, we do not observe a constant high peak TLP (above 3). This suggests that the idea of keeping many big cores (four or even more) for short bursts may not be a good choice for mobile devices. Instead, a system that can provide high performance during peaks and good energy-efficiency fits better with the fluctuate TLP pattern of mobile applications.

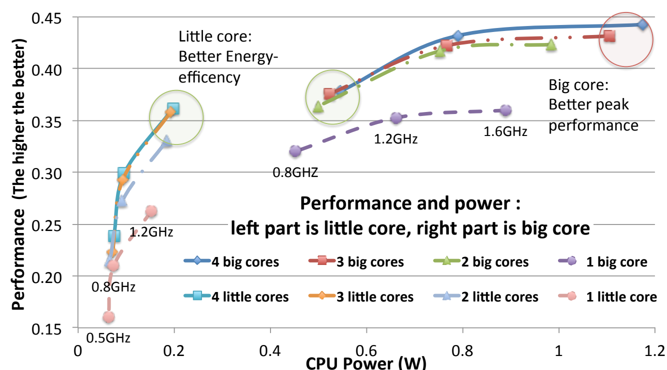


Fig. 10: Performance and power under different frequency and cores (tested using MobileBench). Lines represent different cores configurations; dots on lines represent different frequencies, with lower frequencies (thus poor performance) on the left. Error bars are drawn to a show range of scores for each dot.

B. Energy Efficiency of Big and Little Cores

The processor takes up a substantial portion of power consumption in mobile devices [16]. It is meaningful to analyze the energy efficiency between big cores and little cores. We run MobileBench on the Odroid board for different types of cores, different number of cores, and three different frequencies[4] on each cluster (big and little). We show the results on two different clusters in Fig. 10. For every core/frequency combination, we did four repeated runs. The results show a distinct energy-efficiency difference between the big and little cores: big cores have better performance, but little cores only use roughly a quarter of the power consumption than big cores. Though big cores have approximately 25% less execution time, their power consumption is 3$\times$ more than little cores so they consume more total energy. In a system with flexbility, we can use little cores as much as possible, and big cores in situations that are both computational intensive.

Additionally, the importance of user experience makes such architecture more desirable. Human users want to deliver a response within a user acceptable timeframe rather than finishing the task as fast as possible. For instance, literature shows that a latency less than 0.1s is not perceivable by a human [17]. Any extra resources that are used in accelerating the program to finish faster than 0.1s is unnecessary. In the case of browser performance, as shown in Fig. 10, we may or may not need to switch to a big core depending on the workload of the webpage and the quality of experience demanded by the user. More flexibility in such scenarios will be beneficial for both performance and energy-efficiency.

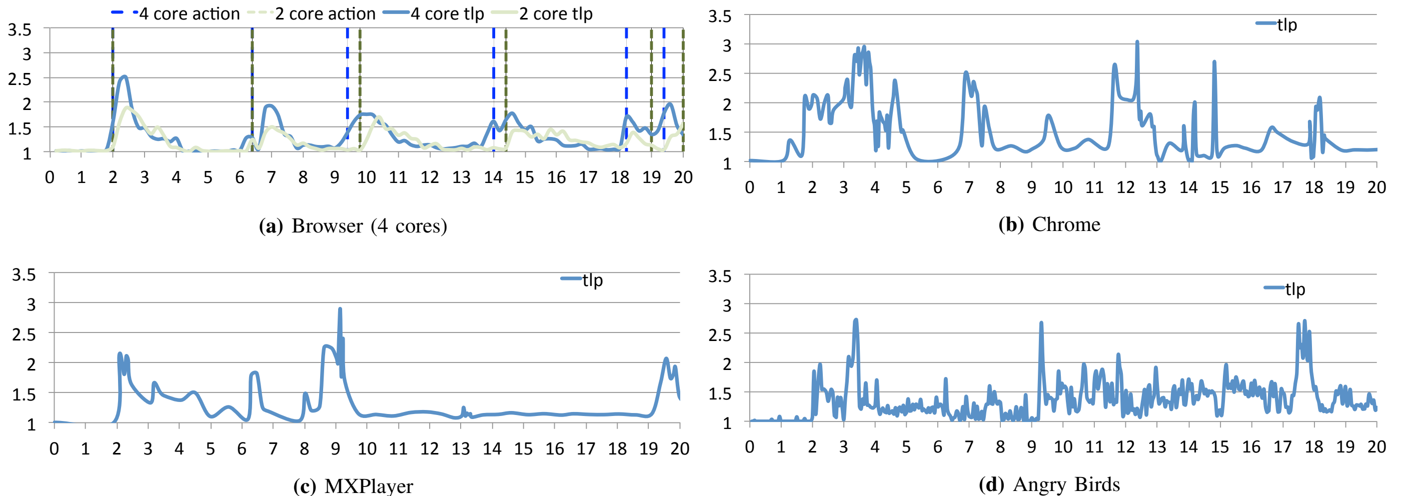
VII. RELATED WORKS

A. Mobile Workload Characterization

Extensive research has been done on characterizing mobile workloads. Gutierrez et al. [2] measure the microarchitectural behavior of a set of mobile applications. Hayenga et

[3]20 seconds is enough for apps to reach steady state.

[4]We use 1.6, 1.2 and 0.8GHz for big cores, and 1.2, 0.8, 0.5GHz for little cores.



(a) Browser (4 cores)

(b) Chrome

(c) MXPlayer

(d) Angry Birds

Fig. 9: TLP vs. time in seconds for mobile applications. For Browser, the solid lines represent TLP, and the dashed vertical lines show when there is an action, such as application startup or opening a new webpage. Actions are labeled by circled numbers.

al. [18] present a workload characterization and memory-level analysis of internet and media-centric applications for an embedded system used for mobile applications. Sunwoo et al. [19] propose a methodology to tractably explore the processor design space and to characterize applications in a full-system simulation environment. Zhang et al. [20] study the performance of mobile applications using multicore CPUs and develop a new CPU power model with a high accuracy. Their results also show that even large applications like web browsers with multi-threading acceleration cannot fully utilize the multicore CPUs. Narancic et al. [21] evalute the memory system behavior of smartphone workloads and show that many workloads are memory throughput bound, especially with specialized compute engines providing enough compute performance. We directly measure multi-core utilization of mobile applications and analyze its implications.

B. Mobile Benchmarks

There has been literature which argue traditional desktop benchmarks such as PARSEC or SPEC are not suitable for mobile devices [2]. Several mobile benchmarks have been proposed. MobileBench [8] is a collection of applications, including a revised version of BBench, Adobe PDF reader, Photo viewer and video playback. Mobybench [7] is also comprised of popular applications, which has already been ported to the gem5 simulator [22]. AM-Bench [23] is an open source based mobile multimedia benchmark for Android platform. Some more application-specific benchmarks have also been proposed [24, 25]. Compared to these benchmarks, we have a boarder coverage of common categories with more applications such as social networking or communication.

C. Mobile CPU Architectures

Zhu and Reddi [4, 26] propose two specialized hardware and an event-based scheduling for mobile web applications. Several papers have presented work on addressing the *Dark Silicon* problem [27–30]. They propose specialized, energy-efficient hetergeneous co-processors which provides much better energy-efficiency. By providing quantitive analysis of mobile workload CPU utilization, we provide a written record of information that can benefit researchers pursuing better mobile CPU designs.

VIII. Conclusion and Discussion

In this paper, we considered how multi-core processors in mobile devices are being used. We have shown that current mobile applications cannot effectively use a large number of cores. Instead, we suggest that a flexible system that can accommodate both high performance and good energy-efficiency is a more preferable choice for current mobile applications.

We have analyzed a wide range of common mobile applications, and calculated the *Thread Level Parallelism* (TLP) of these applications. The average TLP across all categories is 1.46, which shows that mobile apps are utilizing less than 2 cores on average. The applications with the highest TLP, Google Hangout, only has a TLP of just 1.8. We have also evaluated a number of different CPU configurations, including different numbers of cores, core frequencies, and CPU types. We observe a diminishing return on TLP when the number of cores increases. Even in those heavy-load real-world scenarios with background applications or multi-tab browsing, there is still not enough work to keep utilization high. Both these results suggests that having many powerful cores is over-provisioning. Due to physical constraint and interactive user patterns, mobile applications tend to have less parallelism to exploit than desktop applications. The GPU and mobile co-processors on chip also takes off work from CPU. Historically, the desktop TLP study by Blake et al. [1] showed the TLP of desktop applications remained relatively low, even after a 10 year gap. It indicates that parallelizing software is an extremely challenging problem. All these contribute to the low TLP for current mobile applications.

On the other hand, we find out that TLP behavior exhibits peaks and valleys rather than remaining constant. User experience, which is critical for mobile applications, also varies by different application scenarios and different users. A system

with the flexiblity to satisfy both high performance and good energy-efficiency for different program phases is a good choice for mobile devices.

We believe this work can motivate new research directions. TLP is a utilization metric rather than a performance metric. We have only used web browser benchmarks [2, 8] as performance metrics in this paper; the research community can benefit from having benchmarks that quantitatively measure the performance for popular applications of various categories such as games, social, office, etc. Responsiveness is another metric worth noting, especially for user experience of interactive mobile applications [5]. This is also where the peak TLP mainly comes from. Building an accelerator or specific processor architecture tailored for such phases may be an interesting avenue of research into.

IX. Acknowledgement

We would like to thank our shepherd, Vijay Janapa Reddi, and the anonymous reviewers for their valuable feedback. This work is supported in part by a grant from ARM Ltd and NSF I/UCRC Center for Embedded Systems (NSF grant #0856090).

References

[1] G. Blake, R. G. Dreslinski, T. Mudge, and K. Flautner, "Evolution of thread-level parallelism in desktop applications," in *Proceedings of the 37th annual International Symposium on Computer Architecture (ISCA)*, 2010.

[2] A. Gutierrez, R. G. Dreslinski, T. F. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver, "Full-system analysis and characterization of interactive smartphone applications," in *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, 2011.

[3] C. Gao, A. Gutierrez, R. Dreslinski, T. Mudge, K. Flautner, and G. Blake, "A study of thread level parallelism on mobile devices," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, 2014.

[4] Y. Zhu and V. J. Reddi, "Webcore: architectural support for mobileweb browsing," in *Proceedings of the 41th annual International Symposium on Computer Architecture (ISCA)*, 2014.

[5] L. Yang, R. Dick, G. Memik, and P. Dinda, "Happe: Human and application driven frequency scaling for processor power efficiency," in *Mobile Computing, IEEE Transactions on*, 2013.

[6] K. Flautner, R. Uhlig, S. Reinhardt, and T. Mudge, "Thread-level parallelism and interactive performance of desktop applications," in *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems (ASPLOS)*, 2000.

[7] Y. Huang, Z. Zha, M. Chen, and L. Zhang, "Moby: A mobile benchmark suite for architectural simulators," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, 2014.

[8] D. Pandiyan, S.-Y. Lee, and C.-J. Wu, "Performance, energy characterizations and architectural implications of an emerging mobile platform benchmark suite: Mobilebench," in *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, 2013.

[9] M. Böhmer, B. Hecht, J. Schöning, A. Krüger, and G. Bauer, "Falling asleep with angry birds, facebook and kindle: A large scale study on mobile application usage," in *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services*, 2011.

[10] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "Reran: Timing- and touch-sensitive record and replay for android," in *Software Engineering (ICSE), 2013 35th International Conference on*, 2013.

[11] big.little processing with arm cortex-a15 & cortex-a7. [Online]. Available: http://www.arm.com/files/download/big_LITTLE_Final_Final.pdf

[12] Variable smp a multi-core cpu architecture for low power and high performance. [Online]. Available: http://www.nvidia.com/content/PDF/tegra_white_papers/tegra-whitepaper-0911b.pdf

[13] K. Khubaib, M. A. Suleman, M. Hashemi, C. Wilkerson, and Y. N. Patt, "Morphcore: An energy-efficient microarchitecture for high performance ilp and high throughput tlp," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012.

[14] P. Petrica, A. M. Izraelevitz, D. H. Albonesi, and C. A. Shoemaker, "Flicker: A dynamically adaptive architecture for power limited multicore systems," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[15] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. F. Wenisch, and S. Mahlke, "Composite cores: Pushing heterogeneity into a core," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012.

[16] A. Carroll and G. Heiser, "An analysis of power consumption in a smartphone," in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, 2010.

[17] R. B. Miller, "Response time in man-computer conversational transactions," in *Proceedings of Fall Joint Computer Conference, Part I*, 1968.

[18] M. Hayenga, C. Sudanthi, M. Ghosh, P. Ramrakhyani, and N. Paver, "Accurate system-level performance modeling and workload characterization for mobile internet devices," in *Proceedings of the 9th workshop on MEmory performance: DEaling with Applications, systems and architecture*, 2008.

[19] D. Sunwoo, W. Wang, M. Ghosh, C. Sudanthi, G. Blake, C. D. Emmons, and N. C. Paver, "A structured approach to the simulation, analysis and characterization of smartphone applications," in *Workload Characterization (IISWC), IEEE International Symposium on*, 2013.

[20] Y. Zhang, X. Wang, X. Liu, Y. Liu, L. Zhuang, and F. Zhao, "Towards better cpu power management on multicore smartphones," in *Proceedings of the Workshop on Power-Aware Computing and Systems*, 2013.

[21] G. Narancic, P. Judd, D. Wu, I. Atta, M. Elnacouzi, J. Zebchuk, J. Albericio, N. Enright Jerger, A. Moshovos, K. Kutulakos, and S. Gadelrab, "Evaluating the memory system behavior of smartphone workloads," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on*, 2014.

[22] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," in *SIGARCH Comput. Archit. News*, 2011.

[23] E. K. Chayong Lee and H. Kim, "The am-bench: An android multimedia benchmark suite," in *Technical Report, School of Computer Science, Georgia Institute of Technology*, 2012.

[24] Aurora softworks: Quadrant. [Online]. Available: http://www.aurorasoftworks.com/products/quadrant

[25] Gfxbench: unified graphics benchmark based on dxbenchmark (directx) and glbenchmark (opengl es). [Online]. Available: http://gfxbench.com/result.jsp

[26] Y. Zhu, M. Halpem, and V. J. Reddi, "Event-based scheduling for energy-efficient qos (eqos) in mobile web applications," in *Proceedings of the 42th annual International Symposium on Computer Architecture (ISCA)*, 2015.

[27] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: Reducing the energy of mature computations," in *Proceedings of the 15th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, 2010.

[28] G. Venkatesh, J. Sampson, N. Goulding-Hotta, S. K. Venkata, M. B. Taylor, and S. Swanson, "Qscores: Trading dark silicon for scalable energy efficiency with quasi-specific cores," in *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture*, 2011.

[29] A. Raghavan, Y. Luo, A. Chandawalla, M. Papaefthymiou, K. P. Pipe, T. F. Wenisch, and M. M. K. Martin, "Computational sprinting," in *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA)*, 2012.

[30] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th annual International Symposium on Computer Architecture (ISCA)*, 2011.