Performance, Energy Characterizations and Architectural Implications of An Emerging Mobile Platform Benchmark Suite – MobileBench

Dhinakaran Pandiyan, Shin-Ying Lee, and Carole-Jean Wu
School of Computing, Informatics, and Decision Systems Engineering
Arizona State University
Tempe, Arizona 85281
Email: {dpandiya,lee.shin-ying,carole-jean.wu}@asu.edu

Abstract—In this paper, we explore key microarchitectural features of mobile computing platforms that are crucial to the performance of smart phone applications. We create and use a selection of representative smart phone applications, which we call MobileBench that aid in this analysis. We also evaluate the effectiveness of current memory subsystem on the mobile platforms. Furthermore, by instrumenting the Android framework, we perform energy characterization for MobileBench on an existing Samsung Galaxy S III smart phone.

Based on our energy analysis, we find that application cores on modern smart phones consume significant amount of energy. This motivates our detailed performance analysis centered at the application cores. Based on our detailed performance studies, we reach several key findings. (i) Using a more sophisticated tournament branch predictor can improve the branch prediction accuracy but this does not translate to observable performance gain. (ii) Smart phone applications show distinct TLB capacity needs. Larger TLBs can improve performance by an avg. of 14%. (iii) The current L2 cache on most smart phone platform experiences poor utilization because of the fast-changing memory requirements of smart phone applications. Using a more effective cache management scheme improves the L2 cache utilization by as much as 29.3% and by an avg. of 12%. (iv) Smart phone applications are prefetching-friendly. Using a simple stride prefetcher can improve performance across MobileBench applications by an avg. of 14%. (v) Lastly, the memory bandwidth requirements of MobileBench applications are moderate and well under current smart phone memory bandwidth capacity of 8.3 GB/s.

With these insights into the smart phone application characteristics, we hope to guide the design of future smart phone platforms for lower power consumptions through simpler architecture while achieving high performance.

Index Terms—smart phone benchmarks; smart phone architectures; energy and performance characterization.

1. INTRODUCTION

In recent years, there has been an explosive growth in the use of mobile phones – especially smart phones – for our everyday computing needs. The global shipment of smart phones has already surpassed the shipment of personal computers in 2011. In 2012, more main memory modules (DRAM) are shipped to smart phones and tablet computers than to traditional personal computers for the first time in history [1]. These trends indicate the increasing importance of studying the performance requirements of the applications that we run on these mobile computing platforms. We need to better understand the architectural implications that these applications have brought upon, in order to design architectures that can be as powerful and responsive as need be and, at the same time, be as efficient and energy-conserving.

From the performance perspective, the current architectural roadmap for the smart phone platforms has largely followed its desktop predecessors – cramming more and more architectural features such as branch predictors or cache prefetchers, and putting more and more cores on the same die, as exemplified by the recent Qualcomm S4 Snapdragon and NVIDIA Tegra 3 quad-core smart phone processors. Mobile phone computing has evolved in such a manner that it is time to re-evaluate and re-think the needs of the applications that we run on these mobile platforms. The processor cores in these modern smart phones have grown more complex with a goal to deliver high performance, since users expect feature-rich, responsive and interactive applications on their smart phones. However, there are some particular characteristics of the smart phone applications that have not been fully examined and exploited.

First, users typically do not use their smart phones for continuous computation. Typical smart phone usage reveals a pattern of a short-term use, e.g., texting, viewing pictures, or searching for information about a restaurant, followed by a long period of idle time during which the device is often put into sleep mode. Second, the main computation of many smart phone applications may be off-loaded onto the cloud, whereas the smart phone device is primarily responsible for displaying the content. Finally, smart phone applications are generally interactive applications that require user input and are expected to be responsive to handle a burst of data in a short amount of time.

In order to explore key characteristics critical to smart phone applications, examine the effectiveness of modern architectural features for these applications, and design architectural features specifically for smart phone devices, this paper develops a mobile platform benchmark suite, MobileBench. MobileBench is a collection of representative smart phone applications, including general-purpose interactive web browsing, education-oriented web browsing, photo browsing, and video playback, that constitute the majority of activities performed on today’s smart phone platforms.

This paper aims to answer the following questions:
1) What is the energy profile like for smart phone applications? What is the most energy-hungry processing element on a modern smart phone platform?
2) What are the key performance-critical components on the smart phone processors? E.g., how sophisticated does the branch predictor need to be for smart phone applications?
3) Is a large translation lookaside buffer needed to accelerate virtual-to-physical address translation?
4) For memory-bound smart phone applications, what is the appropriate cache memory hierarchy? How can we improve the cache memory utilization for smart phone applications?
5) Are smart phone applications prefetching-friendly? Is it needed to include a hardware prefetcher on the devices?
6) Is memory bandwidth a performance bottleneck due to the smaller size of the on-chip caches?

On the other hand, from the perspective of energy-efficiency, modern smart phones are limited by their energy usage, due to the constrained battery capacity. To understand the energy profile of smart phones, we implement EnergyUsageCollector, a background service application within the Android Jelly Bean framework. EnergyUsageCollector collects energy usage for critical platform components, e.g., application cores, WiFi and radio antenna, LCD screen, etc., for running applications. When the brightness of the LCD screen is at 25%, EnergyUsageCollector shows that application cores consume most of the total energy capacity. This motivates us to delve deeper into understanding the architecture of the application cores with detailed full-system simulation. To identify the performance-critical components for the baseline architecture, we run MobileBench on the Android ICS operating system on an ARM-based processor and reach several important findings.

- With the detailed energy profile, we show that the application cores, executing the main application, the library functions, and the kernel source codes, dominate the energy consumption of the smart phone device.
- Using a more sophisticated tournament branch predictor can improve the branch prediction accuracy but this does not translate to observable performance gain.
- Smart phone applications show distinct TLB capacity needs. Although not all MobileBench applications see performance benefit with a larger TLB, using a larger TLB improves MobileBench performance by as much as 50% and by an average of 14%.
- The current L2 cache on the smart phone platform generally experiences poor utilization because of the fast-changing application memory requirements. Using a more effective cache management scheme mitigates the problem and can increase the cache utilization by as much as 29.3%.
- Prefetching can also mitigate the long memory latency between the processor and the memory. Overall, smart phone applications are prefetching-friendly. With a simple stride prefetcher, the application performance is improved by an average of 14%.
- Lastly, the memory bandwidth requirements of MobileBench applications are moderate and are well under the current smart phone memory bandwidth capacity. This work provides a benchmark suite of frequently-used smart phone applications which is used to explore the best suitable architectural designs for mobile platforms. With the detailed performance analysis and architectural insights, we hope to guide the design of future smart phone platforms for lower power consumptions through simpler architecture while achieving high performance.

2. Constructing MobileBench Benchmark Suite:

A key aspect of this work is to run each MobileBench application on a smart phone platform and to identify key microarchitectural characteristics of these smart phone applications with real-system effects. The MobileBench suite includes a diverse range of real-world smart phone applications. In addition to the publicly available BBench [2] that is used to represent simple web browsing behavior, we include four additional commonly-used benchmarks – realistic web browsing, education-oriented web browsing, photo rendering, and video playback. Next, we discuss each MobileBench application in more detail.

General Web Browsing (GWB): One of the most important smart phone applications is web browsing. In fact, the web browser is one of the most commonly-used interactive applications on smart phones, and many other smart phone applications are also browser-based. To study the behavior of general-purpose web browsing, Gutierrez et al. [2] constructed BBench which is an offline, automated benchmark to assess the performance of a web browser when rendering a collection of 11 popular websites on the web, including Amazon, BBC, CNN, Craigslist, eBay, ESPN, Google, MSN, Slashdot, Twitter, and YouTube. BBench traverses the collection of the websites repeatedly by loading the webpage and scrolling down to the bottom of the webpage before proceeding to the next website. In this paper, we refer to BBench as GWB since it is a benchmark which focuses on simple general web browsing behavior.

Realistic General Web Browsing (RealisticGWB): The always-scroll-down browsing pattern in GWB does not reflect a realistic browsing pattern. In order to model a more realistic user web browsing behavior, we instrument index.html for each webpage to include additional movement patterns. Specifically, the RealisticGWB benchmark introduces vertical up-and-down, horizontal right-and-left movements with random delays between each movement. This models the browsing pattern where users spend more time reading web contents located on specific parts of a webpage and skim through the rest of the page.

Education Web Browsing (EWB-Blackboard): As technology advances, students today are able to use their smart phones to read course announcements and get started with assignments by accessing course websites on smart phone devices. These educational websites, however, exhibit significantly different types of contents than those included in general-purpose web browsers. Unlike general-purpose websites where web contents are more sophisticated, e.g., with images, audio/video streams, or advertisement clips, web contents on these educational websites are mostly text or document formats, where documents are often embedded in web links. To capture this set of browsing behaviors in addition to RealisticGWB, we develop a benchmark that focuses on browsing educational websites.

To model EWB-Blackboard, we use a popular education-
oriented web platform commonly used in universities to host course materials and tools, Blackboard. Students access Blackboard webpages for course information, announcements, assignments, discussions, etc. In addition to the Blackboard webpage browsing, EWB-Blackboard often involves viewing assignment documents that are not directly displayed in a web browser. For example, course assignments from Blackboard webpages are often made accessible in Portable Document File (PDF) format. This means that in-between several Blackboard webpage browsing sessions, students often need to switch from web browsing to document viewing.

To understand the interaction between Blackboard webpage browsing and document viewing, we model the behavior of first browsing through Blackboard webpages and opening an assignment file embedded in a link on the Blackboard webpage. To open the assignment file in PDF format, we use a third-party PDF viewer, ezPDF Reader, on our target.

**Photo Viewing** (PhotoView): With the increasing number of pixel counts for the camera on modern smart phones (as high as 8 mega pixels), high resolution photos are prevalent on these mobile platforms. To view high resolution photos smoothly, modern smart phones must be capable of displaying high resolution photos on the screen timely for a satisfactory user experience. To represent this class of applications, we automate high resolution photo rendering for the Android platform using a picture viewing application: QuickPic. Our PhotoView benchmark includes consecutive photo rendering of five high resolution images, each with a resolution of 4912x3264 and is of size between 4 to 6 MB.

**Video Playback** (VideoPlayback): In addition to PhotoView, an important class of applications for modern smart phones is high definition video streaming. With popular video sharing, users today frequently view video/movie clips on their smart phones and expect high performance delivery in this application class. We prepare VideoPlayback to evaluate only the rendering performance for our target mobile platform, excluding any network issues that might affect our results. We use the same QuickPic application to play a high-definition (720p) MPEG-4 video of size 80 MB and 1 minute in length.

### 3. EXPERIMENTAL SETUP

This section introduces our experimental methodology for both real-system measurements and full-system simulation.

#### 3.1. Real-Device Energy Measurement Infrastructure

To characterize the energy consumption for the various important hardware components on a smart phone platform, we implement a background service application, called EnergyUsageCollector, that runs within the Android framework perioically to collect energy usage behavior. EnergyUsageCollector calculates the energy consumption of a running application based on two pieces of important information. First, by reading the energy specification sheet (power_profile.xml in framework-res.apk) provided by smart phone vendors, EnergyUsageCollector obtains the energy consumption specific to the various hardware components. Figure 1 shows the amount of energy consumed by various hardware component on our smart phone target, Samsung Galaxy S III. Then, EnergyUsageCollector measures the amount of time an application spends utilizing the different hardware components, e.g., application CPU cores, Wifi, LCD screen. To calculate the total energy consumed for each hardware component, EnergyUsageCollector simply multiplies the amount of time spent at each component with the energy constant from power_profile.xml. This approach is similar to how the Android framework estimates for its remaining battery capacity for the operating system level power management activities. Because our EnergyUsageCollector implementation requires changes to the default Android framework, we flash the device ROM for the service application to take effect. For MobileBench, we run and collect the application energy profile for a duration of 30 minutes. Note, because power_profile.xml is available in most of the modern smart phones, our EnergyUsageCollector can be easily ported to other smart phone devices to provide energy profile characterization.

To validate EnergyUsageCollector’s measurement, we measure the device energy consumption with a Watt’s Up RC watt meter (part# B000X4MSVW) with which the current measurement resolution is 0.01A and the energy measurement resolution is 0.1Wh. The source of the power meter is connected to a constant DC power source whereas the load is connected to the smart phone device.

#### 3.2. Full-System Performance Simulation

We evaluate the performance of MobileBench applications with full-system simulation using gem5 [3]. We model a smart phone platform based on the ARMv7 processor architecture running the Android Ice Cream Sandwich (ICS) operating system. The browser-based applications, GWEB, RealisticGWEB, and EWB-Blackboard, render webpages off-line and focus on webpage rendering performance.

In order to identify the microarchitectural components that are performance-critical for smart phone applications, we model the processor pipeline with the influence of instruction and data translation lookaside buffers (ITLBs), a tournament branch predictor, L1 instruction and data caches as well as an unified L2 cache, and a hardware prefetcher. The simulated processor has a 64-entry ITLB, a 64-entry DT LB, and an 1KB page table walker cache. In addition, the core has 4-way set-associative private L1 instruction and data caches. The L1 caches are both 64KB with 64B cache lines. There is also a 16-way set-associative unified L2 cache that is 1MB with 64B cache lines. The configuration for the processor core and the

![Fig. 1. The amount of energy consumed by various hardware component on Samsung Galaxy S III smart phone.](image-url)
memory subsystem is roughly based on the ARM Cortex-A9 processor core [4]. Table I summarizes these parameters used in our simulation infrastructure.

### 3.3. Branch Predictors

To investigate the performance impact of branch predictors, we include two different branch predictors in our study: a local and a tournament branch predictor [5]. The local branch predictor represents a simple predictor which uses local information to predict branch outcomes, whereas the tournament branch predictor uses a more sophisticated algorithm by combining local and global information to reach a prediction.

### 3.4. Memory Subsystem

Technology trends have shown that main memory speeds significantly lag behind processor speeds. Therefore, the performance of the memory subsystem often has a significant impact on the overall system performance. While many smart phone processors adopt the traditional multi-level memory hierarchy for TLBs and caches, e.g., ARM Cortex-A9 or Intel Atom, it is, however, unclear how well this memory architecture performs for the commonly-used smart phone applications in MobileBench.

To have an in-depth understanding of memory characteristics for MobileBench applications, we compare the memory subsystem performance using different TLB sizes, various state-of-the-art cache insertion and replacement techniques, an inclusive or an exclusive cache hierarchy, and a hardware prefetcher. For the purpose of the cache performance characterization, we modify the cache module implementation of a publicly available simulation framework released in the First JILP Workshop on Computer Architecture Competitions [6]. The modified cache simulator models a two-level cache hierarchy with the same parameters listed in Table I. Next, in order to create the input memory traces for the modified cache simulator, we execute each MobileBench application individually in gem5 (under the environment described earlier in this section). Therefore, the input memory traces also include full-system effects.

To study the performance effects of more advanced cache management techniques, we implement three high performing cache insertion/replacement policies, i.e., DIP [7], DRRIP [8], and SHiP [9], in addition to the default Least-Recently-Used (LRU) and random cache replacement schemes. We give the parameters of the cache management techniques in Table II.

#### Stride prefetcher: In addition to investigating the performance impact of using more effective cache management techniques, we are also interested in memory prefetching characteristics of MobileBench applications. To do so, we model a simple stride prefetcher for the L2 cache. The stride prefetcher trains on L2 cache accesses for learning about the stride lengths and can fetch up to 8 prefetch requests ahead.

### 4. Platform Energy Characterization with EnergyUsageCollector

This section shows the MobileBench energy profile with EnergyUsageCollector application. Figure 2 shows the MobileBench application’s energy profile. When at the brightest level (100%), the LCD screen is undoubtedly the energy hog among all platform components. However, when the brightness of the LCD screen is lower to 25%, the energy consumption of the application cores starts dominating.

The second important observation based on the MobileBench energy profile is that, except for general web browsing (GWB and RealisticGWB), commonly-used smart phone applications spend a significant amount of energy at executing library function calls (by as much as 36% for PhotoView at the screen brightness of 25% and by an average of 21% for all
MobileBench applications).

For media-content based applications, e.g., Video Playback and PhotoView in MobileBench, the application cores consume 30% (at 100% LCD brightness) to 70% (at 25% LCD brightness) of total energy. Given that users spend a significant amount of time executing media-content based applications and the auto-brightness setting runs on most of today’s smart phone platforms (by advanced power management), the application core energy consumption becomes increasingly dominant.

Validation. As Section 3.1 presents, we validate the total energy consumption obtained by EnergyUsageCollector with the Watt’s Up power meter measurement. EnergyUsageCollector estimates the energy consumption of the device with a minimum error of 3.6% and an average error of 14.5% for VideoPlayback and the web browsing applications. However, this error increases sharply by 3.6X for PhotoView. The energy consumption estimate given by EnergyUsageCollector is much higher than that by the power meter. This is because EnergyUsageCollector does not consider the RGB components of the pixels which constitute the photo images. Previous study [10] has shown that the power consumption of a white pixel in comparison to that of a black pixel can be as high as 5 times. This change in energy consumption based on the color composition becomes particularly significant for PhotoView, which spends the majority of the time displaying the images on the screen. To improve the accuracy of EnergyUsageCollector, both EnergyUsageCollector and the default battery estimation application in the Android framework need to account for the color profile of images being displayed on the screen.

Overall, when the screen display brightness is at a reasonable, 25% brightness level, the application cores executing the main application, the library functions, and the kernel source codes, become the dominating energy-hungry component on the Samsung Galaxy III platform. This motivates a deeper understanding of the application core’s architecture – what are the performance-critical microarchitectural features on the application cores? Next section presents our performance analysis with a detailed full-system simulation for MobileBench.

5. Performance Analysis for MobileBench

In order to glean insights into the inherent properties of these smart phone applications, we study in detail the characteristics of each application in MobileBench: (1) instruction type distribution (Section 5.1), (2) TLB capacity requirement (Section 5.2), (3) average memory access time (Section 5.3), (4) working set size (Section 5.4), (5) access pattern analysis (Section 5.5), and (6) program phases of memory accesses (Section 5.6). With the understanding of the applications’ behaviors, we discuss the performance results which provide insights into potential improvement for mobile platform processor designs. Specifically, we look at the impacts of the following critical components: (1) branch predictors (Section 5.7), (2) state-of-the-art cache memory management techniques (Section 5.8), (3) prefetchers (Section 5.9), and (4) inclusive vs. exclusive cache hierarchy (Section 5.10).

Fig. 3. Performance comparison for using various TLB sizes from 32 to 256 entries. MobileBench applications show distinct TLB capacity needs. Note, performance is normalized to the baseline 64-entry TLBs.

5.1. MobileBench applications exhibit distinct dynamic instruction type distributions

To identify the critical microarchitectural features that affect the performance of an application, we need to first understand the composition of the applications [11]. On average, 61% of the instructions in the MobileBench applications are integer instructions and only 1% of the instructions use the floating point functional unit. 16% of instructions are control instructions and about 25% of instructions are memory load and store instructions. Compared to other MobileBench applications, VideoPlayback has the largest proportion of memory loads and stores among all instructions it has executed. This is because, over the program execution, VideoPlayback streams large amount of data residing in the high definition frames continuously. As a result, it has a relatively larger portion of memory load and store instructions in its program. Furthermore, VideoPlayback has a relatively small number of branch instructions in its program. This is because of its inherent program behavior. Video playback usually contains fewer number of control instructions in its execution [12].

5.2. MobileBench shows distinct TLB capacity requirements

Since every memory reference requires virtual-to-physical address translation, TLBs cache frequently accessed page table entries to accelerate these translations. When there is a hit in the TLB, the virtual-to-physical address translation can be used directly; otherwise, page table walks are needed.

To characterize the performance impact of TLBs for MobileBench, we vary the size of the instruction and data TLBs from 32 to 256 entries. Figure 3 shows the corresponding performance changes in IPC with respect to the baseline 64-entry TLBs. In general, increasing the TLB sizes helps to improve MobileBench application performance. Figures 4 (a) and (b) compare the number of page table walks needed to be performed for resolving ITLB and DTLB misses respectively.

The performance of web browsing applications in MobileBench, GWB and RealisticGWB, is significantly affected by the size of the ITLB. This is because web browsing applications involve more shared library functional calls referencing a larger instruction address space. Figure 4 (a) shows that, compared with the baseline 64-entry ITLB, using a larger ITLB of 256 entries reduces the number of page table walks required by the two applications by 87% and 86% respectively. The miss rate reductions correspond to 13.3% and 8.2% performance improvement for GWB and RealisticGWB.
In addition to the web browsing applications, VideoPlayback sees even more performance improvement when the DTLB size is increased from 32 entries to 128 entries: a performance gain of 75%. This is because VideoPlayback accesses a large data working set spanning over multiple memory pages. The number of virtual-to-physical address translations could not fit well into either the 32-entry nor the baseline 64-entry DTLB. Therefore, page table walks need to be performed frequently to resolve the address translations that are not cached in the DTLB. When the number of DTLB entries are increased from the baseline 64 to 128 entries, the number of page table walks performed are reduced significantly by 91% which corresponds to the IPC performance improvement of 52%.

In contrast to the large TLB size requirement for GWB, RealisticGWB, and VideoPlayback, EWB-Blackboard and PhotoView while experiencing lower ITLB and DTLB miss rates, do not see much performance gain with larger ITLB or DTLB. Overall, the smart applications in MobileBench have distinct TLB capacity requirements. This motivates design optimization for the TLB structures to maintain high performance for applications that are sensitive to the TLB sizes.

5.3. A large portion of average memory access time is spent at handling L2 cache misses

In addition to the TLB structure, around half of the processor die area is dedicated to the memory subsystem. By including the multi-level cache hierarchy on chip, frequently-used data are kept close to the processor, enabling fast accesses.

Figure 5 (a) shows the breakdown for the average memory access time of the L1 instruction and data caches, as well as the unified L2 cache in the multi-level cache hierarchy. The L1 caches retain the most frequently accessed instruction and data working sets close to the processor so it can reduce the processor stall time. While the L1 cache access latency is only one cycle, it still contributes to over half of the total average memory access time across all MobileBench applications (L1 Inst. Cache Access Time and L1 Data Cache Hit Access Time in Figure 5 (a)). This is because the L1 caches are also one of the most frequently-accessed structures.

Furthermore, as Gutierrez et al. pointed out in their interactive smart phone application studies [2], smart phone applications often generate more function calls to the shared libraries and device drivers. In addition to the fact that there are more instruction references (one for every instruction issued) than data references, the frequent function calls to the shared libraries contribute to more instruction cache references as well. As a result, while the L1 instruction cache hit rate is slightly higher than that of L1 data cache, the total L1 instruction cache hit latency still dominates.

In contrast to the high temporal locality access pattern seen in the L1 instruction and data caches, most references that miss in the L1 caches are likely to miss in the L2 cache as well (L2 Cache Hit Access Time of Figure 5 (a) is insignificant). Consequently, a significant amount of overall memory access latency, 41%, is spent at accessing the main memory. This low utilization of the L2 cache causes not only performance overhead but also increases bandwidth requirement.

Figure 5 (b) shows the off-chip memory bandwidth utilized by the smart phone applications. PhotoView is the application that has the highest memory bandwidth utilization in MobileBench. The high bandwidth requirement is because PhotoView needs to perform high definition photo rendering for five distinct high resolution photos consecutively. On the other hand, one might think that VideoPlayback should also show a similar memory bandwidth requirement. However, because most video encoding schemes only encode the differences between frames for most video frames, VideoPlayback does not have to fetch a lot of data when transitioning from one frame to another. As a result, within the same time period, its bandwidth requirement is not as much as that of PhotoView.

Finally, today’s smart phone chips are equipped with memory bandwidth capacity a couple times larger than what MobileBench applications need. For example, Apple’s A6 chips in the iPhone 5 released in 2012 utilizes a memory bandwidth that has the peak capacity of 8,528 MB/s. Unless multiple bandwidth-hungry smart phone applications are executing simultaneously, current memory bandwidth capacity should be sufficient for fast data delivery. Although memory bandwidth on today’s smart phone chips is not a performance bottleneck, we believe that the memory bandwidth requirement can be significantly reduced via more effective L2 cache management, so fewer data accesses go to the main memory.
5.4. The active working set of MobileBench applications is much larger than the last-level cache capacity

In addition to the memory bandwidth resource, a crucial on-chip resource for achieving high performance is the multi-level cache hierarchy, in particular the L2 cache. The last-level L2 cache bridges the long latency gap between the processor and the memory and, therefore, its efficiency can directly impact an application’s performance. This and the next two sections focus on the study of the last-level L2 cache. We first investigate the L2 cache memory requirement of MobileBench. To do so, we measure the working set sizes.

Figure 6 shows the cache sensitivity study for MobileBench applications by varying the cache sizes. With the baseline 1MB L2 cache, high cache miss rates are experienced for MobileBench. This is because the active working sets of the smart phone applications under study simply do not fit in the 1MB L2 cache. While increasing the cache capacity helps retain an application’s working set in the cache eventually, this comes with the expense of additional area and power overhead. An alternative solution is to deploy a more effective cache management policy for the L2 cache. Section 5.8 will show that using more advanced cache management techniques can significantly reduce the L2 cache miss rate for MobileBench.

5.5. MobileBench exhibits heterogeneous cache accesses

To delve deeper into understanding the degree of data locality in the L2 cache for MobileBench applications, we analyze the reuse distance of all L2 cache references. The reuse distance of a memory reference is defined as the number of distinct interleaving memory references until its successive reuse. Figure 7 illustrates the reuse distance characteristics for MobileBench applications. The x-axis represents reuse distances and the y-axis represents the cumulative density function (CDF) of reuse distances. For all smart phone applications in MobileBench, only less than 20% of memory references have reuse distances smaller than the L2 cache set associativity. These are the memory references which are guaranteed to be retained in the cache and be able to receive future cache hit reuses. For memory references with reuse distances larger than the set associativity, they will result in misses under the baseline LRU replacement policy.

With a more effective cache management technique, memory references that will be reused but with reuse distances larger than the cache set associativity have a higher chance to stay in the cache and to receive future cache reuses. This is because cache access outcomes (a cache hit or a cache miss) depends not only on a reference’s reuse distance but also on the ordering of the memory references.

Through detailed cache access pattern analysis, we observe a frequent streaming and mixed access patterns\(^2\) in MobileBench applications. With a more effective cache management technique, streaming cache references can be filtered out from the cache more quickly, so data with relatively better locality will be retained in the cache and continue to receive cache reuses. Consequently, the poor cache utilization problem caused by the streaming and mixed access patterns can be much reduced. Section 5.8 will later show that, compared with the baseline LRU replacement, EWB-Blackboard performs much better under a more advanced cache management technique, e.g., DIP, DRRIP, or SHIP, because references that will not receive cache reuses are identified and quickly filtered out from the cache. As a result, the L2 cache miss rate of EWB-Blackboard is reduced significantly by 29.3%.

5.6. MobileBench shows fast-changing program phases

In addition to the prevalent mixed access pattern in MobileBench applications, memory access characteristics within an application show fast phase changes as well. Over one program phase, an application can mostly have a streaming memory access pattern resulting in poor cache utilization. During another phase, the application can start seeing a locality-friendly memory access pattern.

To illustrate this fast changes in memory access characteristics as programs run, we measure which insertion policy – LRU-insertion or MRU-insertion – performs better for the running application under DIP. Figure 8 shows the preferred insertion policy as programs execute at runtime. The x-axis plots time in the unit of L2 cache accesses and the y-axis plots the policy selector value. The policy selector is a 10-bit saturating counter and its most significant bit (MSB) determines the preferred cache insertion policy over the program execution. If the MSB of the selector is 1, the L2 cache performs better under the LRU-insertion policy so it will use the LRU-insertion policy. On the other hand, if the MSB of the selector is 0, the L2 cache follows the MRU-insertion policy.

All MobileBench applications show at least three phase changes in cache memory access patterns. In particular, for

\(^2\) In a mixed access pattern, frequently-accessed data are interleaved with less-frequently-accessed data.
the educational web browsing application, EWB-Blackboard, memory phase changes are the most frequent. For example, in EWB-Blackboard, users focus on a small, specific range of web contents or a specific course document for a period of time, which often generates data accesses with good locality. But, for another period of time, users quickly skim through lots of different web contents that can generate data accesses exhibiting a streaming access pattern. The different activities users perform in EWB-Blackboard translate to more heterogeneous cache memory access patterns seen by the L2 cache. Furthermore, we simulate different activities in EWB-Blackboard. In addition to the education webpage browsing, we model the scenario where users open a course assignment file in the PDF format. This also contributes to the phase shifts in cache memory access patterns seen by the memory subsystem. Because cache memory access patterns for smart phone applications are more dynamic, a high performance cache management scheme targeting smart phone chips must take this into account and be able to adapt to the fast-changing memory phases.

5.7. A simple branch predictor leaves room for further performance improvement

After our detailed instruction and memory behavior characterization for MobileBench from Section 5.1 to Section 5.6, we next discuss microarchitectural optimization techniques we apply to the mobile platform processor design and present the corresponding performance impacts.

The branch predictor is one of the most performance-critical component in today’s pipelined processors. Without a branch predictor, the processor has to wait until the branch instruction is resolved (often at the execution stage in the pipeline) before the next instruction can enter the processor pipeline. By providing a prediction for the branch instruction, the processor can proceed with speculative execution. If the branch prediction is correct, the processor avoids the stall time caused by branch instructions. However, if a branch misprediction occurs, the processor needs to squash instructions in the pipeline. To understand the complexity of the branch predictor needed to achieve high performance for MobileBench applications, we explore two different branch predictors: a local branch predictor and a tournament branch predictor. The detail of each predictor can be found in Section 3.3.

Figure 9 (a) shows the prediction accuracy for the two branch predictors used in our studies. In general, the more sophisticated tournament branch predictor performs better than the simple local branch predictor. In particular, the tournament predictor outperforms the local branch predictor for RealisticGWB and EWB-Blackboard by a larger margin. This is because, in both RealisticGWB and EWB-Blackboard, realistic web browsing patterns are simulated. The benchmarks model various movement patterns by browsing webpages up-and-down, left-and-right with some random delay. This means, in both applications, there are more if-else statements for the different movements than the simplistic always-scroll-down browsing pattern in GWB. We believe this is the reason the more sophisticated tournament branch predictor achieves much better accuracy than the simple local branch predictor.

For other applications in MobileBench, the simple local branch predictor is sufficient and can provide similar prediction accuracy as the tournament branch predictor. Figure 9 (b) shows the corresponding IPC performance comparison for using the two different branch predictors. The y-axis plots the IPC performance of using the tournament branch predictor normalized to the performance of using the local branch predictor. For the two applications, RealisticGWB and EWB-Blackboard, that benefit the most from using the tournament branch predictor, RealisticGWB receives 1% performance improvement whereas EWB-Blackboard receives 2% performance improvement. The speedup is not impressive due to the fact that the pipeline stages for smart phone processors are relatively short, as compared to their desktop counterpart. As a result, the performance impact of using an accurate branch predictor is not significant.
5.8. There is a need for using an effective cache management technique for the L2 cache

As previously presented in Sections 5.5 and 5.6, fast-changing, heterogeneous cache access patterns are commonly in smart phone applications. However, under the default LRU replacement policy, the L2 cache cannot effectively retain the more frequently-used data of smart phone applications and, as a result, experiences poor utilization. To investigate the performance impacts of using more advanced cache management techniques for MobileBench applications, we implement several state-of-the-art cache insertion and replacement policies for the last-level L2 cache: random, DIP, DRRIP, and SHiP, and compare the L2 cache miss rate reduction with respect to the default LRU replacement scheme. The parameters of each cache management techniques can be found in Table II.

Figure 10 illustrates the L2 miss rates for MobileBench using the various cache management techniques. Among all cache management schemes under study, SHiP performs the best and can reduce the L2 miss rate by as much as 29.3% for EWB-Blackboard and by an average of 12% for all MobileBench applications. Other state-of-the-art cache management schemes perform relatively well also and can reduce the L2 cache miss rate for MobileBench effectively.

As previously shown in Section 5.5, under the default LRU replacement scheme, cache lines with temporal locality cannot always be identified and retained in the cache effectively because of other streaming memory references. With a more effective cache management technique however, the streaming and mixed access patterns in the smart phone applications can be quickly detected. For example, DRRIP and SHiP cache management schemes dynamically predict the reuse pattern for an incoming cache line and adjust the cache line’s insertion position according to the reuse pattern prediction. When a mixed cache access pattern is encountered, DRRIP and SHiP can distinguish cache lines that are less likely to receive future cache reuses from other lines exhibiting good data temporal locality. By giving streaming references less time in the cache, the interference between the streaming references and cached data is reduced. Compared to DRRIP and SHiP, cache management techniques, DIP can only address the streaming access pattern but cannot intelligently differentiate cache lines that are likely to be reused from cache lines that are less likely to be reused upon insertion. So, it generally performs less well. Overall, with a more effective cache management scheme, the L2 cache utilization for MobileBench applications can be significantly improved.

5.9. MobileBench applications are prefetching-friendly

In addition to using an effective cache management scheme to bridge the memory latency gap, another popular approach is to prefetch data into the cache hierarchy before the actual demand reference. While prefetching can hide memory latency and improve performance significantly, it can severely degrade performance in the event of untimely and/or inaccurate prefetch requests. To investigate how the performance for MobileBench applications can be improved (or degraded) in the presence of hardware prefetching, we measure the MobileBench performance in the presence of a simple stride prefetcher and compare it against the performance in the absence of the prefetcher. The prefetch parameters are presented in Table I and its algorithm is described in Section 3.4.

In the presence of the stride prefetcher, the L2 miss rate for MobileBench applications is significantly reduced by an average of 15.1%. Figure 11 (a) shows the L2 miss rate when using the L2 cache stride prefetcher, normalized to when not using the prefetcher. Across all MobileBench applications, the L2 miss rate reduction is consistently observed. Furthermore, Figure 11 (b) illustrates the corresponding IPC performance improvement for each of MobileBench applications. Among all MobileBench applications, while VideoPlayback has the most L2 miss rate reduction, its corresponding IPC performance gain is only 2.8%. This is because, compared to other MobileBench applications, VideoPlayback is also the application that suffers the least from the low L2 cache utilization (see Figure 5 (b)). Therefore, although the prefetcher can reduce the L2 miss rate by 23.3%, this miss rate reduction is not reflected in a similar degree of performance gain for VideoPlayback. Overall, MobileBench applications benefit significantly from hardware prefetchers. The simple prefetcher increases MobileBench performance by an average of 14%.

5.10. MobileBench applications see performance gains from using an exclusive cache hierarchy

Another important design issue regarding a multi-level cache hierarchy is the property of cache inclusion versus...
exclusion. In an inclusive cache hierarchy, data in the upper-level caches (e.g., L1 caches) must also exist in all of its lower-level caches (e.g., L2 caches). The advantage of having an inclusive cache is that when a clean L1 cache line is evicted, this line can simply be discarded. In contrast, for an exclusive cache hierarchy where there is no duplicate in the hierarchy, any line evicted from the L1 cache always incurs an exchange of cache lines between the L1 and L2 caches. The L1 victim is written back to the L2 cache and the requested line is inserted in the L1 cache. While the procedure for handling L1 victims is more complicated for an exclusive cache hierarchy than for an inclusive one, the overall cache capacity using an exclusive cache hierarchy is larger by the size of the L1 cache.

To compare the performance between the two different cache hierarchies for MobileBench, we implement an exclusive cache hierarchy using the same cache parameters described in Table I. Across all applications, the L2 miss rate is improved by 4.2% on average using the exclusive cache hierarchy. Although smart phone applications benefit from the larger effective cache capacity with the exclusive hierarchy, the memory controller design for supporting the exclusive cache hierarchy will become much more complicated as we move to multicore chips with cache coherence.

6. RELATED WORK

The availability of representative smart phone benchmarks enables the development of future smart phone processors. Unfortunately, to date, there is a only limited number of benchmarks for evaluating modern smart phone chip designs. Existing smart phone benchmarks (e.g., The Embedded Microprocessor Benchmark Consortium [13]) are either proprietary or are not freely accessible. BBench [2], a recently released benchmark, attempted to address the problem of the lack of smart phone benchmark resources for researchers. With BBench, we can now evaluate a web browser’s performance when it is rendering some of the most popular sites on the web. In fact, it is the most related work to this paper. While BBench provides access for evaluating a web browser’s rendering performance, its algorithm only focuses on the rendering performance but does not take into account user browsing patterns. In reality, users perform more heterogeneous browsing patterns. To address this, we design RealisticGB that considers various additional movement patterns with random delays. This simulates the browsing pattern where users spend more time reading a specific section of a webpage and quickly skim over the rest. In addition to the web browser based benchmarks, MobileBench includes other workloads which represent the commonly used smart phone applications. The accessibility of MobileBench will enable mobile platform designers to perform characterization and performance analysis for a variety of applications.

In addition to benchmarks that model interactive user behaviors on smart phone platforms, there are several stressmarks that allow smart phone users to evaluate the CPU or memory system performance, e.g., GeekBench2, Quadrant, or Vellamo. These stressmarks typically assess the integer, floating-point CPU performance, memory speed, or bandwidth performance individually. However, the evaluation performed by these stressmarks does not model realistic user behaviors on smart phones and therefore does not directly translate to user-visible performance. Similarly, several other stressmarks aim to evaluate the graphics system performance of modern mobile computing platforms, e.g., An3DBench, Basemark, or CF-Bench. Again, these stressmarks do not model realistic user behaviors and are used solely for the purpose of comparing the graphics system performance between different smart phone architectures.

7. CONCLUSION

In summary, this paper presents detailed performance and energy characterizations for MobileBench, a collection of smart phone applications. MobileBench includes benchmarks for realistic interactive web browsing, educational web browsing, photo rendering, and video playback, representing a diverse variety of applications commonly-run on mobile platforms. With the performance and energy characterization, the architectural insights provided by this paper, and the release of MobileBench, we hope to inspire innovative designs that lower power consumptions through simpler architecture while maintaining high performance for smart phone devices.

ACKNOWLEDGMENT

The authors would like to thank Lingjia Tang and the anonymous reviewers for their feedback. This work was partially supported by the NSF I/U CRC Center for Embedded Systems, and from NSF grant #0856090.

REFERENCES