Energy-Discounted Computing and Supercapacitor Energy Buffering for Resource-Constrained Systems

by

Meng Zhu

Submitted in Partial Fulfillment of the Requirements for the Degree

Doctor of Philosophy

Supervised by

Professor Kai Shen

Department of Electrical and Computer Engineering Arts, Sciences and Engineering
Edmund A. Hajim School of Engineering and Applied Sciences

University of Rochester
Rochester, New York

2018
# Table of Contents

- **Biographical Sketch** vi
- **Acknowledgments** vii
- **Abstract** vii
- **Contributors and Funding Sources** ix
- **List of Tables** x
- **List of Figures** xii

## 1 Introduction

1.1 Background and Motivation ............................................. 1
1.2 Dissertation Statement ................................................... 2
1.3 Contributions .......................................................... 2
1.4 Dissertation Organization ............................................... 4

## 2 Background

2.1 Resource Constraints .................................................. 5
2.2 Platform and Application ............................................. 7
2.3 System Software Support for Energy Management and Optimization . . . . . 12
# 3 Energy Discounted Computing on Multicore Smartphones

3.1 Introduction .................................................. 17
3.2 Motivation ..................................................... 19
3.3 Energy Discounted Computing ............................... 23
3.4 Implementation ............................................... 26
3.5 Evaluation .................................................... 28
3.6 Related Work ................................................ 38
3.7 Conclusion .................................................... 39

# 4 Interactive Context for Mobile OS Resource Management

4.1 Introduction .................................................. 41
4.2 Motivations and Feasibility .................................. 43
4.3 Interactivity Context for Mobile Resource Management ................................. 47
4.4 Implementation ............................................... 57
4.5 Evaluation .................................................... 58
4.6 Related Work ................................................ 65
4.7 Conclusion .................................................... 68

# 5 Supercapacitor Energy Buffering for Self-Sustainable, Data-Intensive Systems

5.1 Introduction .................................................. 70
5.2 Prototype System Construction ............................ 72
5.3 Supercapacitor Energy Modeling ......................... 76
5.4 System Energy Management ............................... 81
5.5 Software Architecture and Implementation .......... 87
5.6 Evaluation .................................................... 87
5.7 Related Work ................................................ 98
5.8 Conclusion .................................................... 100
Biographical Sketch

Meng Zhu was born in Changzhou, a city located on the southern bank of the Yangtze River in the eastern part of China. He was an undergraduate student at Jiangnan University and received his Bachelor of Engineering degree there in 2011. He then moved to Rochester, New York to study Electrical and Computer Engineering at the University of Rochester. He worked as a summer intern for Broadcom Corp. in Sunnyvale, California in 2012. Afterward, he started his doctoral studies initially with Professor Tolga Soyata and later under the direction of Professor Kai Shen. He was awarded a Master of Science degree from the University of Rochester in 2013. He worked as an intern for Google Inc. in Mountain View, California in the summers of 2015 and 2016, and for FutureWei Technologies, Inc. in Santa Clara, California in the fall of 2015. In 2017 he joined Mesosphere Inc. in San Francisco as a software engineer.

The following publications were a result of work conducted during doctoral study:


Acknowledgments

First and foremost, I am genuinely grateful to my advisor Kai Shen for offering me an opportunity to participate in system research and helped set me on a wonderful path I never imagined. His constant support, patient guidance and technical prowess have been tremendously helpful in shaping this work. The experience of working with him showed me the virtue of optimism, diligence and methodicalness that helped me grow professionally and personally.

I have been fortunate to work with great professors. Tolga Soyata led me to start this journey and provided support in the field system construction. Michael Huang offered me a research opportunity during my master degree study and advised me towards the end of my doctoral research. My dissertation committee members, Wendi Heinzelman and Chen Ding, encouraged me and offered valuable advice.

I want to thank my primary school teacher Jufang Tan who taught me how to read and write and always gave me warm praise and encouragement that shaped my confidence. My high school classmates, Kai Xie and Li Guo, accompanied me during crucial times and from them I learned many good things.

Finally, I would like to thank my family: my fiancée Wenqin and my parents Yulan and Jianzhong. I am beyond words of gratitude for their infinite love.
Abstract

The past decade has witnessed an enormous growth and popularity of off-the-grid computer systems. From smartphones to field-deployed systems, self-contained computing devices that can operate anytime and anywhere have unleashed a wave of change in the society.

One of the biggest constraints of these systems is the limited energy buffering capacity. Comparing to the rapid advancement in the semiconductor industry, there has only been tepid improvement in the device energy buffering capacity over the years. Thus energy management and optimization techniques are vital to extend the usefulness of these systems in practice. As technology evolves, however, the emergence of new hardware platform, application usage pattern and energy buffering mechanism have rendered the existing techniques inadequate. In this dissertation, we study new techniques that capitalize on this technology trend to improve energy management for resource constrained systems.

First, we introduce energy discounted computing that is specifically tailored for multicore processors on smartphones. Multicore processors are not energy proportional: the first running CPU core that activates shared resources incurs much higher power cost than each additional core does. By non-work-conserving scheduling, we exploit energy-discounted co-run opportunities to process background smartphone tasks that involve no direct user interaction, improving energy efficiency without impacting user experience.

Second, we take a user-centric approach to develop application-transparent execution context for mobile operating systems that reflects the criticality of current execution on user interactivity. This interactive context enables differential resource scheduling for improved system interactivity and energy efficiency.
Finally, we construct a supercapacitor-sustained data-intensive field system that aims for continuous operation. The system leverages the voltage-to-stored-energy relationship in capacitors to enable precise energy buffer modeling. We demonstrate that the precise supercapacitor energy model allows model-driven system adaptation for optimal and stable operation quality-of-service.
Contributors and Funding Sources

This work was supervised by a dissertation committee consisting of Professors Kai Shen (advisor) and Chen Ding of the Department of Computer Science, and Professors Wendi Heinzelman and Michael Huang of the Department of Electrical and Computer Engineering at the University of Rochester. The work mentioned in Chapter 5 was done in collaboration with Professor Tolga Soyata who was a professor at the Department of Electrical and Computer Engineering at the University of Rochester and his student Moeen Hassanalieragh. Part of the work mentioned in Chapter 3 was done when the author was interning at FutureWei Technologies Inc. in 2015. This material is based upon work supported by National Science Foundation grants CNS-1217372, CNS-1239423, and CCF-1255729, and a Google Research Award.
List of Tables

3.1 CPU idle states available on the Huawei Mate 7 smartphone. A state’s target residency is a value defined in the corresponding cpuidle driver in the Linux kernel. It indicates the minimum time period during which the CPU expects to remain idle so that it is worthwhile to enter the state. ........................................... 20

3.2 The impact of L2 cache miss rate throttling threshold on system performance when doing power states preservation and contention-aware scheduling. We show differences in background task elapsed time and BBench slowdown ratio for two scenarios: (A) BBench co-running with FaceAnalysis and (B) BBench co-running with AppOpt. ................................................................. 35

3.3 Description of eight application scenarios used in the trace-based application study. ................................................................. 36

3.4 Results for the trace-based application study. Each usage scenario lasts for one minute. Abundance of discounted CPU cycles is the ratio of energy discounted CPU cycles to the active CPU cycles used by the corresponding interactive application. In the second column (marked with multicore), energy discounted cycles on all CPUs are counted, assuming the background task has perfect parallelism to utilize all idle CPUs. In the third column (marked with single-core), energy discounted CPU cycles are only counted on one of the eligible CPUs, assuming the background task has no parallelism. We use single-core cycles to calculate the equivalent work of background tasks in column four and five. . . . 37
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Application workloads, their average power consumption, and elapsed time under the default system setting.</td>
<td>60</td>
</tr>
<tr>
<td>4.2 Uncertain execution context coverage.</td>
<td>65</td>
</tr>
<tr>
<td>5.1 System power consumption and model coefficients for the traffic trajectory analysis application under different CPU configurations.</td>
<td>83</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A typical iPhone user’s home screen.</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>A snapshot of collected camera data streams used for the traffic trajectory analysis application. It shows the monitored campus street and parking lot with marked trajectories identified from the traffic.</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Objects floating on the river.</td>
<td>11</td>
</tr>
<tr>
<td>2.4</td>
<td>Picture of our prototype field system with a Nexus 7 tablet computer (without its internal rechargeable battery) sustained by eight Maxwell 3000 Farad supercapacitors (wrapped in black tapes), a custom-built controller board (at center of the picture), and a block of 21 Radio Shack Model 277052 solar panels.</td>
<td>12</td>
</tr>
<tr>
<td>3.1</td>
<td>Disproportional multicore power consumption on the number of active CPUs.</td>
<td>19</td>
</tr>
<tr>
<td>3.2</td>
<td>Experimental results of running interactive application browser loading BBench with various background tasks under different scheduling strategies. We show the background task energy discount (A), background task elapsed time (B), and impact on BBench’s interactivity (webpage rendering slowdown) (C).</td>
<td>31</td>
</tr>
</tbody>
</table>
3.3 Experimental results of running interactive application Angry Bird with various background tasks under different scheduling strategies. We show the background task energy discount (A), background task elapsed time (B), and Angry Bird’s frame rate (C).  

3.4 Current trace of BBench running alone and BBench+Spin co-run under alternative scheduling strategies. In the default co-run test (B), the spin task takes 9.07 seconds to finish. Under the power states preserving scheduling strategy (C), the same task takes 31.91 seconds.

4.1 An illustrative example of a user interaction session. The user touches a button to query the device status. Important data and control flows between threads are marked with arrows. Darkened portions of a thread timeline indicate active executions (while the rest represents blocking waits).

4.2 Illustrations of conventional and early binding based context propagation for the wake-up event where interactive (high priority) waker wakes up background (low priority) wakee. Dashed lines indicate thread blocking wait (while the solid lines represent active execution). Figure (a) illustrates conventional binding where the background wakee would not inherit the interactive context until it is picked up by the priority-aware scheduler. The wakee may experience a lengthy delay before it gets scheduled if the CPU is busy running interactive or even background tasks, (in the latter case) leading to priority inversion. Figure (b) illustrates early binding where the waker promptly propagates its interactive context to the wakee while the wakee is still blocking, causing the scheduler to pick the (now interactive) wakee early.

4.3 Code snippet of Java class WorkQueue. User space synchronization primitives such as synchronized monitor and notify function are implemented using generic OS facilities such as futex syscall. We use this snippet to illustrate different context propagation scenarios in Figure 4.4.
4.4 Illustrative examples of intra-process communications using `WorkQueue` presented in Figure 4.3. The left part shows the execution events timeline. The right part shows visible activities and the resulting context propagation in the OS. Dotted lines indicate thread blocking waits (while the rest represents active execution). In all three cases, Thread_A and Thread_C each sends one message to Thread_B.

4.5 Experimental platform setup including Monsoon power meter, logging PC and Nexus 5 smartphone. (A) The Monsoon power meter connects directly to the power supply pins of the battery-less smartphone, acting as an external power source and measuring the whole system current. (B) The Nexus 5 smartphone is not thermal stable. We put it on top of ice pack to get stable performance results. (C) No USB connection to the smartphone to avoid power disruption. The phone is controlled by the host through wireless channels. (D) Using the provided logging software, we are able to record the current at 5kHz.

4.6 Active energy consumption (A) and foreground application interactivity (B) under various system settings. We compare four different system settings: default, two variants of Android process importance based scheduling—treating system service tasks as either interactive or background, and our interactivity-aware scheduling. Both active energy and interactivity are normalized to that of the default system. When treating system tasks as background in the Android process importance based scheduling, benchmarks involving application launch (workload #5 to #12) usually end up with system hang or crash rendering the interactivity and energy measurement noninstructive.

4.7 Active energy consumption (A) and foreground application interactivity (B) under various context propagation strategies: propagate only through higher level dependency events (ignore low-level events such as `futex` and kernel block/wake-up), propagate through all dependency events and our uncertainty aware propagation scheme introduced in Section 4.3.2. Both active energy and interactivity are normalized to that of the default system.
4.8 Our interactivity-aware system with uncertain context propagation strategy runs with conventional and early context binding schemes. We show (A) active energy consumption and (B) foreground application interactivity. Both active energy and interactivity are normalized to that of the default system.

5.1 Picture of our deployed system on a seven-story building rooftop (center picture) that includes a camera, a block of solar panels, and a system box. The system box (left-side picture) contains a Nexus 7 tablet computer (without its internal rechargeable battery) sustained by eight Maxwell 3000 Farad supercapacitors (wrapped in black tapes), and a controller board (at the top of the box). The energy buffering capacity of the supercapacitor block is about $1.4 \times$ that in the original Nexus 7 battery.

5.2 System architecture including the supercapacitor block for energy buffering, Nexus 7 for energy consumption, and a 70mW controller board that regulates between solar power input and energy buffering/consumption.

5.3 Image snapshots of our camera sensing workloads.

5.4 Different topologies to build a block of multiple supercapacitors: a) fully-serial, b) fully-parallel.

5.5 Self-discharge (leakage) power for our 8-supercapacitor set at operating and overcharged voltages.

5.6 Effective capacitance at different terminal voltages. The capacitance is noticeably lower at low voltages.

5.7 Data capture+processing power consumption at a range of CPU configurations and application frame rates. For each CPU configuration, the experiments end at the maximum achievable frame rate. We also show a linear fitting of the frame rate to power each CPU configuration.

5.8 Agilent 34410A measurement setup to record simultaneous current and voltage values from the supercapacitor block to determine its power consumption ($P_{SC}$).
5.9 Accuracy of different supercapacitor energy models in predicting the time to reach a target voltage while system is in idle state with one CPU core turned on at 102MHz and an average power consumption of 0.55 Watt. The left plot illustrates the result of modeling the operational time from the full charge of 21.6 V for the supercapacitors while the right plot shows the result of modeling from 8 V to 7 V. 

5.10 Accuracy of different supercapacitor energy models in predicting the time to reach a target voltage while the system is operating ZoneMinder-based environmental camera traps with four CPU cores running at 1.2GHz and an average power consumption of 3.23 Watt. The left plot illustrates the result of modeling the operational time from the full charge of 21.6 V for the supercapacitors while the right plot shows the result of modeling from 8 V to 7 V.

5.11 Power budget evaluation—Effectiveness of supercapacitor energy models in guiding application operational frame rates to utilize the stored energy by a certain amount of time. The experiments were performed for the traffic trajectory analysis application. The left two plots illustrate the supercapacitor voltage and application frame rates under operations of consistent frame rates. The right two plots show the results when the system makes hourly assessment of remaining supercapacitor energy and adjustment of application frame rates.

5.12 Power consumption, supercapacitor voltage and application frame rate under model-driven static CPU configuration and delayed bursts.

5.13 System power proportionality under model-driven static CPU configuration and delayed bursts.

5.14 The CPU configurations for the capture-only and burst-processing phases under our delayed burst approach over a range of power budgets.

5.15 System duty-cycle ratio (indicating data capture quality-of-service) and supercapacitor voltage changes in 24-hour periods of our outdoor system deployment under sunny (A), overcast (B) and mixed (C) weather.

5.16 Supercapacitor charging curve and solar power variations during a bright period.
1 Introduction

1.1 Background and Motivation

Off-the-grid computer systems are becoming increasingly important nowadays. The ease of operating without the location constraints significantly extends their usages. For example, smartphone is among the most important inventions of the 21st century and has been reshaping entire industries and transforming societies ever since its inception. Also, field systems that collect and analyze data in areas where there is limited infrastructure help to monitor critical events and achieve a better understanding of the environment.

These systems, unlike their predecessors, are full-fledged computing platforms. They run resource intensive applications on top of conventional operating systems thanks to the legendary advances in the semiconductor industry. However, the development of energy buffering technologies that power these devices have been consistently lagging behind. It is well-known that the improvement trajectory of the battery capacity is far flatter than the transistor scaling curve projected by Moore’s Law. This has led to an ever widening gap between the buffered energy capacity and the device computing capability which significantly limits the system performance and usability in practice. As the amount of work such a system can perform is fundamentally limited by its buffered energy capacity, energy is undoubtedly a vital resource that deserves careful management.

Energy management has been an active area of work for computer systems. Numerous hardware techniques have been developed to form the foundation for higher-level control and
optimization. For example, processor dynamic voltage and frequency scaling (DVFS) enables agile power adjustment of active CPU power consumption at system runtime [75]. The ability to power off part of the silicon in many hardware components to enter more efficient idle states is also crucial to reduce energy consumption when the system is lightly loaded. In addition, various software systems, aided by the aforementioned hardware features, have also been proposed to manage the precious energy more efficiently. The software systems gather information about the hardware and application runtime status and try to devise the most appropriate energy management strategy at a given circumstance based on various policies [22, 31].

However, the advent of a new generation of hardware has rendered the existing techniques not optimal for low power, calling for new system development. The proliferation of multicore processors brings new features such as multi-level idle states and asynchronous DVFS among different cores, necessitating new system techniques. The birth of smartphones brings users and their devices ever closer and calls for a more efficient user-centric instead of application-centric approach to manage resources. Lastly, supercapacitor as a promising energy buffering mechanism poses new challenges as well as opportunities for the existing computer system.

1.2 Dissertation Statement

This dissertation develops new energy management and optimization techniques to harness new hardware platform, adapt to new application usage patterns and utilize new energy buffering mechanisms. We make the most of the new technologies and take a holistic approach to construct energy-ware systems based on deep understanding of the hardware as well as the application, along with tight system integration. Our approaches have shown significant improvement in system energy efficiency and quality-of-service.

1.3 Contributions

This dissertation presents the following energy management and optimization techniques for resource-constrained systems.
**Energy Discounted Computing on Multicore Smartphones** We introduce energy discounted computing that is specifically tailored for multicore processors on smartphones. Multicore processors are not energy proportional: the first running CPU core that activates shared resources incurs much higher power cost than each additional core does. By non-work-conserving scheduling, we exploit energy-discounted co-run opportunities to process background smartphone tasks that involve no direct user interaction. We show that, to achieve optimal co-run energy discount without impacting the user experience, the background execution must avoid elevating the overall system power state and prevent shared resource contention.

We evaluate our technique on Huawei Mate 7 smartphone with realistic workloads. The result shows significant energy discount (up to 63%) for background tasks with minimum impact on the interactive application execution (3.8% slowdown in the worst case).

**Interactive Context for Mobile OS Resource Management** We take a user-centric approach to develop application-transparent execution context for mobile operating systems that reflects the criticality of current execution on user interactivity. We track various OS-level events available on the Android/Linux that signal the initiation and dependency propagation of interactivity-related executions. This interactive context enables differential resource scheduling for improved system interactivity and energy efficiency. We devise novel techniques to consciously recognize and tolerate the inherent inaccuracies of OS-level tracking and minimize potential priority inversions.

We assemble a suite of representative foreground and background mobile applications and perform experiments on Nexus 5 smartphone. Its quad-core processor is capable of per-core frequency and voltage adjustment. Comparing to the default system, our interactivity-aware system can achieve 13% on average (up to 23%) energy saving. Meanwhile, it incurs only 1% on average (6% in the worst case) performance slowdown.

**Supercapacitor Energy Buffering for Self-Sustainable, Data-Intensive Systems** We construct a supercapacitor-sustained data-intensive field system that aims for continuous operation. The system leverages the voltage-to-stored-energy relationship in capacitors to enable precise energy buffer modeling. We find the supercapacitor self-discharge to be a minor issue in practice
where the operating power significantly exceeds the leakage power. However, accurate energy budgeting must account for the variation of effective capacitance—particularly lower capacitance at lower voltages nearing energy depletion. Our experiments show that the corrected supercapacitor model reduces the time-to-depletion prediction error from 79% to 23%.

We further recognize the particular importance of power proportionality to self-sustainable, continuous-sensing systems. By exploiting features in dynamic power systems such as Tegra3 in the Nexus 7 tablet and typical work patterns of continuous sensing applications, we significantly improve the power proportionality of our prototype system. This enhances our sensing application’s frames-per-second rate substantially compared to the optimal static CPU configuration (by over 25% at a wide range of power budgets).

1.4 Dissertation Organization

The rest of the dissertation is organized as follows:

Chapter 2 provides the background of the resource constraint systems studied in this dissertation.

Chapter 3 explores the technique of energy discounted computing for multicore smartphones.

Chapter 4 introduces interactive execution context for user-centric mobile OS resource management.

Chapter 5 presents our design of supercapacitor-sustained data-intensive field system.

Chapter 6 concludes and discusses future directions.
2 Background

2.1 Resource Constraints

Computer systems require various resources to perform computation. These finite resources are all subject to certain constraints. The resource constraint system studied in this dissertation is defined in relative terms—a computer system that operates under unconventional scenarios such that one or more types of the resources normally required and available are either very much limited or outright missing. As a result, these systems have to manage, optimize and somehow cope with these resource deficiencies to achieve required quality-of-service. We list two key constraint factors here that are related to the system studied in this dissertation.

2.1.1 Size

Size usually determines the mobility of the system. A small form factor makes the system easy to be carried around. It is now a common constraint, especially due to the rise of portable personal computers such as laptops and smartphones. Smartphone designers and manufacturers are trying every measure to make devices thinner and lighter to appeal to customers. The size constraint could limit system in many aspects, most notably power profile and buffered energy capacity.

Size and power limits usually go hand in hand. High power devices draw more current which converts to heat that needs to be swiftly dissipated to avoid system overheating. As
a result, accommodating high power hardware in the system requires large space for cooling mechanisms such as fans and air channels.

Thanks to the Moore’s Law, legendary advancement in the semiconductor industry not only phenomenally improved the hardware performance, but also brought down the power consumption. As a result, low-power computers nowadays boast comparable computing capabilities to that of a traditional desktop computer just several years ago. For example, Huawei Mate 7 smartphone is equipped with an octa-core processor with 1.8 GHz and 2GB of RAM. Even under peak system load, its system power consumption is below 4 Watt.

Still, without proper cooling mechanisms, small profile devices have to be equipped with specialized low-power processors that are less powerful than their contemporary desktop counterparts. Even with power-optimized processors, these systems usually need to make compromises. For example, we found out that, due to the thermal constraint, the Snapdragon 800 quad-core processor on LG Nexus 5 smartphone is not able to maintain at high frequency for a sustained period of time.

For computer systems with high mobility, energy buffering capacity is often a bigger concern than power consumption. Portable computer systems rely on batteries and other energy buffering mechanisms to operate while detached from the grid. Their usages are constrained between charges. The amount of computation an off-the-grid system can perform is fundamentally limited by its buffered energy capacity which is proportional to the system battery size. Yet, the advancement of the battery industry has been much less impressive compared to the chip industry. This poses a great challenge to system designers and manufacturers. Year after year, new iterations of devices equipped with more powerful and feature-rich hardware have to live with the more or less same energy budget. This dilemma stumps even the most advanced smartphone vendors, resulting in unsafe system design and terrible consequences [85]. Without major breakthroughs in the battery technology in the horizon, computer systems relying on buffered energy will have to live with such dilemma in the foreseeable future.
2.1.2 Access to Infrastructure

Like many other modern technologies, computer systems rely heavily on the existing infrastructure to operate properly. The internet provides data and services while the electrical grid supply power for computation. Normally, these infrastructure facilities are readily available. Yet, for computers operate in remote wilderness, connectivity to the internet or grid could become luxuries.

Access to the Internet is crucial for many computer systems. It empowers any systems that connect to it with limitless possibilities, from internet based applications and services to cloud computing and storage resources. However, field systems deployed in remote areas, for example to monitor highway traffic, usually have little to no connectivity. This requires the system to perform tasks mostly on their own or otherwise with minimal external support through the limited network bandwidth (e.g. using satellite).

While computer systems can still accomplish certain tasks without internet connectivity, access to energy resources is certainly indispensable in all cases. Most devices either are directly plugged into the electrical grid or can be charged on-demand with relative ease. However, field systems deployed to monitor natural environment such as volcano sites or wild life habitats often stationed in wilderness without power supply. They must live on ambient energy sources such as solar, wind or water current to generate energy to sustain their operations. Unlike electrical grid, these energy sources are intermittent and volatile in nature, thus careful energy management is crucial for continuous operation.

2.2 Platform and Application

This dissertation aims to cope with two energy related system constraints: energy scarcity on smartphones and energy sustainability of field systems. In this section, we introduce these two platforms and their applications.
2.2.1 Smartphone

A smartphone can be defined as a handheld personal computer with an integrated mobile broadband cellular network connection—a pocket-size device equipped with fast computing hardware and various sensors, running desktop level applications on top of conventional operating systems, and with easy access to the internet. This has proven to be a powerful and versatile combination that brought fundamental changes to the society. According to Statista, there are 2.32 billion smartphone users worldwide in 2017 [93]. In many developing regions, the smartphone is usually the only personal computer and internet access device. It serves as the primary access point to the web and a window to the outside world.

Smartphone Application The magic power of smartphone comes from its myriad of applications (apps). According to Statista, as of March 2017, there are 28 million apps in Google Play, the main Android application marketplace and 22 million in the iOS counterparts [92]. According to a research report from App Annie, smartphone owners, on average, use nine applications per day and 30 per month [96]. Figure 2.1 shows a typical iPhone user’s phone home screen with many popular applications. While smartphones continue to serve traditional functionalities that belong to phones such as voice call and texting. Increasingly, they are used for navigation, entertainment, personal wallet, fitness and many more, penetrating almost every aspect of people’s daily life.

Smartphone application usage has some unique characteristics. They are very interactive by nature. And due to the limited screen real estate, user usually only interacts with one application at a time. This has two implications. First, since users’ attention is solely dedicated to the application on the screen, any system sluggish responses would be immediately noticed, resulting in a bad user experience. Thus system performance and interactivity are of paramount importance. Second, single-user event-driven interactive applications usually have significant idle time (while users are consuming the content on the screen) and have limited parallelism, leaving smartphones mostly idling with occasional burst processing.

Multicore CPUs on Smartphone Smartphones nowadays are equipped with powerful multicore CPUs which enable them to run desktop level applications and operating systems. How-
ever, multicore processors are not power proportional: the first running CPU incurs much higher power cost than each additional core does. This disproportionality is mostly due to the aggressive hardware sharing. In order to drive down cost, reduce footprint and save power, modern multicore processors share substantial hardware components between cores. Cores on one socket usually share the oscillator, power rail and low-level cache together. As a result, multicore processors can achieve high energy efficiency during heavy parallel processing. However, when the system is lightly loaded, all the shared components still have to be kept on. Unfortunately, typical smartphone applications lack the parallelism to utilize the increasing number of cores available to them, resulting in a waste of energy. This creates opportunities for the mobile system to make use of the extra computing resources to complete certain tasks at an energy discount.

2.2.2 Field Systems

Industrial, societal, and environmental systems are rapidly recognizing the value of data collection and analysis in the wild. In particular, data intensive visual information is highly valuable
in areas such as transportation and environmental science. Many such systems reside in the field that are away from the wired infrastructure, necessitating self-sustainability through energy harvesting and buffering.

Field Sensing Application  Field data sensing and processing facilitates intelligent transportation [101]. These systems capture and analyze data streams of live road conditions for better traffic safety and efficiency [101]. In particular, a traffic trajectory analysis application [47] can monitor vehicle movements and extract their trajectories to enable the analysis of roadway conflicts [91], dangerous pedestrian violations [10], and the effectiveness of new traffic treatment [107]. We deployed the traffic trajectory application on a building rooftop-based field system to study the street and parking lot traffic condition in our campus. Figure 2.2 shows an image snapshot captured during our deployment with annotated vehicle trajectories.

![Figure 2.2: A snapshot of collected camera data streams used for the traffic trajectory analysis application. It shows the monitored campus street and parking lot with marked trajectories identified from the traffic.](image)

Environmental camera traps [72] are another application area. A field camera captures and monitors a live stream of images. Applications such as ZoneMinder [116] difference each newly captured image against a background scene computed by averaging a series of recent past images. Such differencing can detect motion or new objects like roaming animals in the park or floating logs in the river. Figure 2.3 shows a video snapshot collected by us on floating objects in a river near our campus.
Field System Energy Sustainability  Systems deployed in the natural environment or near highways are often away from infrastructure. They have to rely on ambient energy sources such as solar and wind. These energy sources are intermittent and volatile in nature, necessitating energy buffering capability to sustain continuous operation.

Rechargeable battery is a common choice for energy buffering. However, they are ill-suited for many field systems. They contain acids and other corrosive chemical compounds that are detrimental to the natural environment. Their poor thermal stability limits their usages in many natural conditions (typical lithium-ion battery has an operating temperature range between 0 to 45°C). Their limited lifetime also requires regular maintenance.

Supercapacitors, unlike ordinary capacitors, do not use the conventional solid dielectric. Rather, they use electrostatic double-layer capacitance and electro-chemical pseudo-capacitance, both contribute to their high capacitance. Thanks to the recent advance in the supercapacitor energy density, it has become a viable energy buffering alternative for field systems, Supercapacitors have practically infinite lifetime (≈ 10^6 charge cycles), can operate in a wide temperature range (-40 to 65°C), and are free of acids and other corrosive chemical compounds. These qualities imply low maintenance cost, high reliability and less environmental impact. All are compelling advantages for systems deployed in the field.

Figure 2.4 shows our supercapacitor-sustained filed system prototype. The off-the-shelf Nexus 7 tablet, sustained by solar-charged supercapacitors, is capable of running the data-intensive traffic trajectory application continuously.
2.3 System Software Support for Energy Management and Optimization

System software plays a crucial role in energy management and optimization. Sitting between the hardware and applications, the operating system possesses valuable information running applications and has direct control over hardware states. It naturally assumes the role of energy resource management, from the basics of energy accounting to enforcing sophisticated management policies. In addition, various software solutions have been proposed to reduce energy consumption and improve system efficiency.
2.3.1 Energy Accounting

Effective resource management solution requires accurate tracking of resource consumption, so that the system can make sound scheduling decisions and is able to enforce desired management policies.

However, tracking energy consumption and attributing it to the right entity are challenging for the operating system due to the lack of hardware support. For many computer systems, they are equipped with a single current sensor that is primarily used for signaling the battery level to the user. These primitive sensors are only capable of measuring energy consumption of the entire system at a slow rate. It is too coarse grained and far from enough to enable advanced energy management strategies. As a result, system software has to rely on other measures to achieve accurate, fine grained, online energy accounting.

A common solution is to conduct offline hardware power modeling beforehand and then combine it with online hardware usage statistics to approximate the actual energy consumption.

Hardware power models can be obtained by running calibration micro-benchmarks and use advanced multimeters to measure the power coefficients of the individual hardware component. The accuracy of the power model hinges on two factors. First, the benchmarks need to closely mimic the resource consumption pattern of the real world applications. Second, sophisticated hardware components such as CPU have many different operating states depending on the system load that can affect the power consumption. It is important to take these varying states into account.

The operating system is typically responsible for tracking hardware usage statistics and then correlate it with the offline hardware power model to estimate energy consumption. One way is to track the execution time of different entities (e.g. application processes) as a metric of usage [33]. However, in complex systems where different software components are closely interleaved with each other and hardware usage is heavily multiplexed, simply tracking application execution time may be inadequate. In these systems, event-based accounting can be more effective. For example, application requests made through kernel system calls can be intercepted so that the corresponding hardware activities can be properly attributed even when the application processes are not running (waiting for the response) [105]. In another case, perfor-
mance counters can be used to decouple the complex CPU sharing between different threads to achieve fine-grained usage accounting [87].

2.3.2 Energy Management

With the proper energy accounting infrastructure in place, energy resource can be effectively managed by the system software.

For many systems with scarce energy resource, it is necessary to manage energy as a first class operating system resource—similar to memory and disk space. Applications need to explicitly send request to the operating system to allocate energy for execution. In this way, the operating system is capable of limiting the energy consumption rate and share the limited resource among competing tasks according to user preferences. Various operating system constructs have been proposed. ECOSystem [108] uses a common unit “currentcy” to unify energy accounting over diverse hardware components. Applications need to pay “currentcy” while utilizing hardware devices or otherwise being halted. Cinder Operating System [83] introduces resource “reserve” and “tap” operations. It tracks entities responsible for resource consumption across inter-process communications and allows applications to delegate their resources for others. These system facilities enable various energy management policies. For example, fairness is a common objective for resource management. Malicious or resource-intensive applications should be promptly identified and their energy usage limited to avoid system wide starvation.

For field systems that rely on ambient energy sources, energy sustainability is another crucial target. For example, a continuous running system sustained by solar input should be able to survive the night period without shutting down prematurely. Given a fixed amount of energy budget and a target operation period, a system need to plan ahead and make necessary adaptations along the way to ensure continuous operation. Accurate energy resource accounting makes it possible for the system to calculate and allocate remaining energy properly and perform continuous corrections. A common adaptation strategy is to trade application quality-of-service for extended operation time. For example, video streaming applications can switch to black-and-white rather than full-color video to reduce data transmission and energy consumption [32].
2.3.3 Energy Optimization

Beside spending the scarce energy resource wisely, it is important to optimize the system to improve efficiency and lower energy consumption. Three optimization strategies are employed when appropriate: identifying and reducing unnecessary work, performing computation tasks using more efficient device states and offloading work to external channels.

A starting point to improve energy efficiency is to look for redundant and unimportant operations that can be optimized away without affecting application quality-of-service. For example, drowsy power management [54] avoids unnecessary device power-ons when push notification wakes up a smartphone in suspension state. It monitors device open, close, and read/write accesses to model a device state machine and infer task-device dependencies. This helps to construct a minimum set of hardware components that are necessary to serve the particular push notification. TAMER [61] provides an OS mechanism that interposes on events and signals that cause task wake-ups, and allows for detailed monitoring, filtering, and rate-limiting of noncritical smartphone background activities.

When the work is unavoidable, system energy consumption can still be optimized by leveraging the low-power hardware states to achieve higher energy efficiency. A common strategy is to bundle workload together to amortize the device wakeup energy cost. This requires workload to have soft quality-of-service constraints so that it can be scheduled with some flexibility. For example, Piggyback crowdsensing [53] showed that performing delay-able sensing work while a smartphone is already active by other applications saves substantial wakeup energy costs. Another approach is to identify slacks in applications so that tasks can be performed in low-power settings that are typically low-performing but more energy efficient without affecting the user experience. IAMEM [9] leverages slack in human perception to set memory to the most efficient state such that the performance degradation resulted from the increased memory access latency is within the human perception threshold. GRACE-OS [106] integrates dynamic voltage scaling into soft real-time scheduling to save energy while running multimedia applications, providing soft performance guarantee with high energy efficiency.

For devices with scarce energy resource, computation offloading, in which some portions of an application are migrated to a server, is a very tempting solution. The primary challenges
are to identify portions of the workload that can benefit from offloading and achieve seamless integration with the remote server. CloneCloud [17] utilizes virtualization technique to switch executions between local thread and clones in a computational cloud. MAUI [18] relies on the managed code environment in Windows applications to achieve fine-grained method level offloading. Its profiler makes offload decisions by continuously monitoring the program and network characteristics combining with the device power profile.
3 Energy Discounted Computing on Multicore Smartphones

3.1 Introduction

Energy remains the critical resource bottleneck for typical smartphone usage. Due to the slow progress on battery technologies and size restrictions of hand-held devices, battery capacity still limits effective smartphone usage between charges. At the same time, today’s popular smartphones are commonly equipped with quad-core or even octa-core processors. Powerful multicore processors put further pressure on the scarce energy resource of a mobile device.

Multicore processors are not energy proportional: the first running CPU incurs much higher power cost than each additional core does. This can be attributed to two reasons. First, modern processors are good at power gating. When the system is completely idle, most parts of the CPU can be shutdown resulting in minimum energy consumption. Second, the sharing of hardware resources on a multicore means that the first running core must activate the bulk of shared resources while additional cores can utilize the already activated resources at much lower cost. This power disproportionality suggests that a multicore processor is more energy-efficient when more of its cores are utilized at the same time.

Unfortunately, typical smartphone applications are built on event-driven, UI-centric framework and serve only a single user. They do not have sufficient parallelism to utilize multiple CPU cores simultaneously. Recent studies \cite{37,86} on Android applications show a lack of thread-level parallelism across applications and an over-provisioning of core resources across devices. This implies that smartphone multicore processors often operate at low core utilization
resulting in poor energy efficiency. At the same time, when one CPU core is being utilized, computing resources at other cores are available at a deep energy discount.

In this work, we propose to exploit such energy-discounted co-run opportunities to process background tasks that are useful on a smartphone but do not involve direct user interaction (and thus its time of execution is flexible). One example of background tasks is the file compression and encryption in preparation for backing up the user data to the cloud. Another example is the offline bytecode compilation into native code for optimized application execution in Android. The third example is background sensing and analysis of user’s facial expression or eye movements to improve user experience. We demonstrate that background tasks may be scheduled to co-run with interactive applications and realize significant energy discount.

The idea of saving smartphone energy by bundling tasks or piggybacking computation on other applications is not new [53, 70]. Unlike previous work, we recognize that optimal energy discount on multicores is only realized when the background task execution does not elevate the overall system power state. Specifically, the background task execution must not disrupt the multicore CPU idle state, increase the core frequency, or affect the smartphone’s suspension period. In other words, the smartphone’s multicore power states should experience no change due to the additional background task execution. We accomplish this objective through careful non-work-conserving CPU scheduling.

While co-execution of applications on multicore processors may improve the energy efficiency, it also risks significant interference on shared hardware resources, memory bandwidth and last-level-cache space in particular, and thereby leads to poor interactive application performance and degraded user experience. To mitigate such contention, we use available processor performance counters to monitor memory bandwidth usage during the co-execution, and throttle the background task when it interferes with the foreground application interactivity.

The rest of this chapter is organized as follows. Section 3.2 elaborates on multicore power disproportionality and available smartphone background tasks that motivate our work. Section 3.3 presents our design of energy-discounted computing and resource contention mitigation on multicore smartphones. Section 3.4 describes our implementation on the Android platform. Section 3.5 evaluates the energy saving of background task executions and the impact of user interactivity under different scheduling strategies. We also perform trace-based application
3.2 Motivation

3.2.1 Multicore Power Disproportionality

CPUs have traditionally been the biggest energy consumer in the computer system and are not energy proportional. Thanks to many innovations, they are now much improved. Today, multicore processors have very sophisticated power states which can be dynamically adjusted to adapt to different workloads. Specifically, dynamic voltage and frequency scaling (DVFS) is used to achieve a wide range of performance/power settings when the system is active. During the idle period, clock gating and power gating are heavily utilized to power down various parts of the processor in order to achieve low power consumption. These techniques enable the CPU to scale their power consumption relatively well in relation to their utilization, making them probably the most energy proportional hardware component in the current computer system.
<table>
<thead>
<tr>
<th>State</th>
<th>Name</th>
<th>Power (mW)</th>
<th>Target residency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>Wait for interrupt (WFI)</td>
<td>403</td>
<td>1 nSec</td>
<td>Processor is clock gated but can respond to cache/TLB maintenance (e.g., L2 snoop) requests without exiting the WFI state.</td>
</tr>
<tr>
<td>C1</td>
<td>Individual powerdown</td>
<td>365</td>
<td>1 mSec</td>
<td>Processor is power gated. All state including L1 cache content is lost and the processor is removed from the coherency protocol.</td>
</tr>
<tr>
<td>C2</td>
<td>Cluster powerdown</td>
<td>214</td>
<td>4 mSecs</td>
<td>Can only be entered when all processors are in individual powerdown mode. All state including the L2 cache content is lost.</td>
</tr>
</tbody>
</table>

Table 3.1: CPU idle states available on the Huawei Mate 7 smartphone. A state’s target residency is a value defined in the corresponding cpuidle driver in the Linux kernel. It indicates the minimum time period during which the CPU expects to remain idle so that it is worthwhile to enter the state.

However, making good energy proportional hardware remains difficult and current CPU’s energy proportionality is far from perfect. This is particularly true for multicore processors. Figure 3.1 shows power consumption of several multicore smartphone/tablet platforms when different number of cores are active. On all platforms we can observe a disproportionate power jump when activating the first core of each multicore processor. Specifically, we can see that activating each additional cores typically consumes less than half of the power compared to that of the first core. This is substantial given the small profile of the mobile device.

This power disproportionality is mostly due to the aggressive hardware sharing. In order to drive down cost, reduce footprint and save power, modern multicore processors share many hardware components between cores. CPUs on one socket usually share the oscillator and power rail which forces each CPU to operate at the same frequency. As a result, multicore processors can achieve high energy efficiency during heavy parallel processing. However, if the workload can not scale to take advantage of available cores, the entire socket will have to be
kept at certain frequency and voltage level to accommodate a few cores’ performance needs, resulting in a waste of energy.

Besides limiting the capability of active performance/power scaling, hardware sharing also affects the processor idle state. Table 3.1 lists the available CPU idle states on the Huawei Mate 7 smartphone. As you can see, individual core idle state (C1) has only limited impact on the overall power consumption. Maximum power saving is only achieved through continuous and simultaneous CPU sleeps (C2). Again, hardware sharing plays an important role here. For example, L2 cache and related memory subsystem can only be shutdown when the whole CPU cluster is completely idle.

We are aware that various heterogeneous architectures have been proposed to improve the CPU energy proportionality. For example, chips with asymmetric clocking capabilities are able to set different frequencies for different cores, realizing more flexible performance/power scaling. Also, chips equipped with CPUs of different micro-architectures (e.g., ARM big-LITTLE) are used to mitigate the performance vs. power dilemma. However, these techniques do not completely eliminate the power disproportionality. In the pursuit of low cost, small footprint and peak energy efficiency, hardware sharing is and will continue to be one of the fundamental design principles of multicore processors. Consequently, CPU power disproportionality will remain a reality that computer systems have to live with in the foreseeable future.

To summarize, modern multicore processors can achieve high energy efficiency when doing heavy parallel processing or in complete idle states. But due to the aggressive hardware sharing, they are very inefficient when dealing with workloads of limited parallelism. Unfortunately, it is well known that typical smartphone applications lack the parallelism to utilize the increasing number of cores available to them. This creates opportunities for the mobile system to make use of the extra computing resources to complete certain tasks at an energy discount.

### 3.2.2 Smartphone Background Tasks

We define smartphone background tasks as application workloads that are meaningful to the user but do not involve direct interaction and thus have loose quality-of-service requirements.
As mobile phones are increasingly used for a variety of purposes, background tasks are becoming common in day-to-day uses. Here are a few examples.

**Upload and download** operations are common on smartphones. Syncing data with cloud storage services, posting on social websites and software installation/update are typical smartphone usages. Some of them can be delayed to a certain extent. Although significant energy consumption comes from the transmission module, CPUs also consume substantial energy during the process. For example, data compression/decompression requires heavy computation. And encryption/decryption are almost mandatory nowadays which also involve nontrivial CPU processing.

**System maintenance work** sometimes can also be treated as background tasks. For instance, Android/Linux uses *kswapd* daemon to scan for memory pages that can be swapped out to free up space. Another example is a system daemon called *dhd_dpc* which analyzes network packets and scans for Wi-Fi hotspots. In addition, during application installations, Android would optimize the downloaded packages by recompiling the bytecode for better native performance. All these require substantial CPU processing. Some of them may have timing constraint (e.g., memory management). However, their completions often only matter when the user is also actively using the phone, in which case discounted computation opportunities are likely to be abundant as we will demonstrate in Section 3.5.4.

**Smartphone sensing** is also a suitable background task. Previous work [53] has shown that delaying sensing activities to overlap with other application executions can be more energy efficient. With our technique, the bundled execution can reap even more energy discount. Recent trends also suggest more creative ways of sensing. For example, using camera sensors to analyze user’s facial expressions or eye movements [94] may improve user experience. Due to privacy issues, it is beneficial to perform these analysis locally which will put pressure on the device battery life. Since these sensing activities often overlap with user interactive tasks, our technique can be used to substantially lower the energy cost.
Proactive tasks are done predictively to improve user experience. As smartphones getting “smarter”, these tasks are becoming increasingly common. For example, smartphone virtual assistants like Siri can provide recommendations, news and applications proactively based on the user context. Previous work [104] also suggests to pre-launch applications to hide user perceived delay. These tasks often do not have hard deadlines and thus can benefit from our technique to save energy.

3.3 Energy Discounted Computing

Given the power disproportionality of smartphone multicore processors and the lack of parallelism in typical mobile applications, it is possible to get a deep energy discount by co-scheduling background tasks with the interactive application. However, achieving maximum energy discount without impacting the user experience requires careful system control.

3.3.1 Power State Preservation

During the execution of interactive applications, the CPU will dynamically adjust its power states to meet the application performance needs. The key principle of reaching the optimal energy efficiency is to utilize the additional (otherwise idle) processor resources without elevating the overall CPU power state.

- **CPU idle state, or ACPI “C” state [98]**: On smartphones, there are often long idle gaps between user interactions during which the user is consuming the content on the screen while all CPUs enter deep sleep state. As simultaneous and continuous sleeps can save a lot of energy [114], it is crucial to keep background tasks from disrupting these idle periods. On the other hand, during active application executions, due to lack of parallelism in the typical smartphone application, idle CPUs will often enter per-core idle states. These shallow sleep states, as we mentioned in Section 3.2.1, do not save much energy. Thus these idle cores can be utilized to run background tasks at an energy discount. To achieve this, the CPU scheduler needs to schedule background tasks opportunistically in accordance with interactive applications and therefore non-work-conserving CPU scheduling
may be necessary. Specifically, the system should schedule background tasks on idle cores only if there is at least one sibling CPU being actively utilized by the interactive application. Otherwise the scheduler should stop picking tasks even when background tasks are ready to run and let the core enter deep idle state.

- **Core frequency state, or ACPI “P” state:** Modern CPUs use DVFS to quickly adjust power levels to conserve energy and meet performance needs of different workloads. In our co-run scheme, the system should avoid raising the CPU frequency/voltage levels for background tasks. Otherwise, the extra energy consumption will negate the energy discount and the system may well consume more energy than running each task individually combined. At the same time, such caution should not affect the performance of interactive applications. In other words, the CPU frequency adjustment should only focus on the needs of interactive applications and ignore the presence of background tasks.

- **Smartphone suspension state, or ACPI “S” state:** Systems in the suspension state consume very little energy by shutting down most parts of the hardware, including the CPU and memory. On some platforms (notably Android), applications can prevent system suspension by making explicit requests to the operating system. It is important that, in our design, background tasks are not permitted to make such requests unless explicitly whitelisted by the user. The system should be able to enter the suspension state regardless of background tasks.

To summarize, realizing the maximum energy discount requires judicious control of various aspects of the system to prevent background tasks from elevating the system-wide CPU power states. In other words, background tasks should be *invisible* to the system when making CPU power state adjustments.

### 3.3.2 Resource Contention Mitigation

Carefully running background tasks along with interactive applications can bring significant energy savings. However, such savings should not sacrifice user experience. In particular, performance of interactive applications should not be affected.
Co-running tasks on a multicore can potentially slow down each other due to resource contention. This is further exacerbated in our system due to its scheduling strategy—background tasks are intentionally scheduled to run hand in hand with interactive applications, significantly increase the chance of resource contention.

One easy mitigation is to adjust the CPU scheduling priority. Various parameters are available for this purpose (e.g., nice values and CPU shares on Linux). In our design, due to the clear importance of interactive applications, we choose to grant absolute priority to them—they are always picked by the scheduler before background tasks. In other words, within one CPU, background tasks can only be scheduled when there is no interactive task waiting.

Absolute priority can eliminate contention on CPU time and mitigate private cache and TLB pollution. However, due to the hardware resource sharing on multicore processors, contention could also result from shared cluster resources like last-level-cache space and memory bandwidth between cores. To minimize its impact, our system adopts a simple contention identification approach. Specifically, we monitor the last-level-cache miss rate using the available performance counters. Contention is identified if the miss rate reaches a threshold that suggests memory bandwidth saturation. We acknowledge a limitation of our approach—last-level-cache space contention that does not lead to memory bandwidth saturation will not be identified. Comprehensively identifying cache space contention would be challenging and it generally cannot be accomplished by online monitoring of performance counters alone.

Once the contention is identified, the common approach is to throttle the antagonist (low priority tasks in the contention) executions. This can be done most efficiently on platforms that support certain hardware features such as CPU duty-cycle modulation. Unfortunately, such hardware features are not widely available on today’s smartphone processors.

Our system relies on the CPU scheduler to throttle the background tasks. Comparing to the above techniques, however, this is more coarse grained. We can only assert control in the granularity of a CPU quantum (otherwise risk extra scheduling overhead). Fortunately, closely monitoring the contention through performance counters could help time the throttling control more accurate, making this approach quite effective in practice.
3.4 Implementation

We have implemented our system on Huawei Mate 7 smartphone running Android 4.4 and Linux 3.10.30. Our entire modification resides in the Linux kernel.

We use Linux control groups (cgroup) to identify background tasks in the kernel. During system boot, a CPU control group named `best_effort` is created under the cgroup root hierarchy. Background tasks can then be easily added to this group by interacting with the cgroup virtual file system.

**Non-work-conserving CPU scheduling**  We modify the Linux complete fair scheduler (CFS) to maximize CPU idle state energy saving. Our scheduling policy requires coordination between sibling CPUs. To avoid expensive cross-CPU interrupts and synchronizations, we implemented these communications asynchronously. Specifically, a CPU that wants to schedule background tasks is responsible for checking its siblings’ task scheduling states. We have each CPU maintain a flag indicating its current scheduling state with the following four values:

- **BUSY** indicates the CPU is running normal tasks (e.g., interactive applications),
- **IDLE** indicates the CPU is in idle state (regardless of the level of idle state),
- **BACKGROUND** indicates the CPU is running background tasks,
- and **UNDEF** is a transient state (e.g., during context switches).

These per-core flags are cache line aligned to avoid possible false sharing. Although the asynchronous flag read may return stale value under data races, it is likely to be corrected at the next scheduling opportunity. When the scheduler is picking next task to run, normal tasks have absolute priority and are always picked before background ones. If there are only background tasks left in the run queue, the scheduler will first check its siblings’ state. If any of them is currently **BUSY**, it will proceed to schedule one of the background tasks. Otherwise, it will enter idle state directly.
Frequency preservation  Kernel cpufreq governor is responsible for adjusting the CPU frequency. It can come in different flavors but the process usually involves tracking the system load and making adjustments according to some fine-tuned parameters. The load is calculated on a per-core basis by looking at the CPU busy time during the past epoch. Governors typically raise the frequency according to the need of the most heavily loaded CPU (if per-core frequency setting is not available).

To prevent background tasks from affecting the CPU frequency, we track their CPU usage and subtract that from the total CPU busy time when calculating the load. This, along with the absolute priority modification in the scheduler, essentially make background tasks invisible to the CPU governor. While governors completely ignore background tasks, they can still respond to the need of interactive applications just like before.

These modifications reside in the generic governor framework thus individual governor change is not needed.

Suspension management  On Android, wakelock is used to govern the system suspension state. Applications that want to keep the system awake need to make explicit request to the kernel and grab a wakelock. According to our co-run policy, background tasks should not hold any wakelocks. We modify the wakelock kernel sysfs interface to reject any requests made from background tasks.

Contention-triggered throttling  Our performance counter based throttling strategy is implemented as a loadable kernel module. We assess the memory bandwidth usage by monitoring the L2 (last-level) cache miss rate. Specifically, we select two events in ARMv7 performance monitoring unit: ARMV7_A15_PERFCTR_L2_CACHE_REFILL_READ as L2 cache read miss and ARMV7_A15_PERFCTR_L2_CACHE_REFILL_WRITE as L2 cache write miss. Combined they approximate the total access to the main memory. The module triggers periodic interrupt every 20 ms to collect and update the counter statistics. We read the counter value directly from the registers and take care of the overflows. Before picking background task, the CPU scheduler is required to check the latest L2 cache miss rate. If the rate is above certain threshold, the background task will not be scheduled. Similar to our other scheduler modifications, the
counter maintenance and lookups are performed in an asynchronous way to avoid the overhead of cross-CPU interrupts and synchronizations.

Our modifications (including the periodic performance counter reading) incurs less than 1\% performance overhead for all our benchmarks described in Section 3.5.1.

3.5 Evaluation

In this section, we evaluate our techniques on a real device with realistic benchmarks. Section 3.5.1 introduces our evaluation setup. Section 3.5.2 evaluates the system energy efficiency. Section 3.5.3 assesses the effectiveness of our resource contention mitigation measures. Section 3.5.4 provides a trace-based application study to demonstrate the abundance of energy-discounted computing opportunities in various smartphone usage scenarios.

3.5.1 Evaluation Setup

Experimental device  We use Huawei Mate 7 smartphone. It was released in October 2014 and is equipped with a Hisilicon Kirin 925 SoC which contains an ARM big.LITTLE octa-core CPU. We use the big cluster in our evaluation. It has four 1.8 GHz ARM Cortex-A15 cores. Each core has its own 32 KB/32 KB L1 instruction and data cache and all cores share a 2 MB L2 cache. It has 2 GB LPDDR3 memory with a bandwidth of 12.8 GB/s.

Power measurement  In order to do precise power measurement, we remove the smartphone’s battery and connect its power pins to the Monsoon power meter [67] which acts as an external power source and measures the phone’s overall power consumption. The power meter is able to sample the current at 5 kHz. We turn off hardware components like GPS, cellular and dim the display to the minimum brightness. WiFi is kept on for the purpose of controlling the phone through the host machine without the USB connection (which will disturb the power measurement).

Interactive applications  We assemble a suite of benchmarks to represent typical interactive and background application co-run scenarios. Two representative interactive applications are
chosen. BBench [42], a widely used web browsing benchmark which automatically loads and renders locally cached popular websites. It measures the browser performance by tracking the JavaScript `onLoad` event which is triggered once a webpage is fully rendered. We run it using the Android default web browser. Another interactive application is Angry Bird, a popular mobile casual game.

For the co-run experiments, it is important that we are able to measure the interactivity of the interactive application. For applications like web browser, the interactivity can be defined as time needed to complete certain tasks. For BBench we use the aggregated webpage rendering time to measure its interactivity. On the other hand, for games like the Angry Bird, the interactivity is only defined by how responsive the application is. Frames-per-second (FPS) is a more relevant metric. We use GameBench [36] to measure its FPS.

**Background applications** We select five applications as background tasks.

*Spin*, a CPU intensive microbenchmark that calculates the $n$-th triangular number by summation. We choose this microbenchmark to illustrate the optimal co-run scenario—a CPU intensive workload with little memory activity thus no contention on shared resources.

*Compression* compresses a set of files using bzip and *Encryption* encrypts them using the AES encryption. These two are chosen to mimic user download and upload activities.

*AppOpt* optimizes Android application packages by recompiling them into native code. This is chosen to represent typical deferrable system work.

*FaceAnalysis* is an in-house developed application that analyzes input faces. It uses Stasm [65], an active shape model based library, to process images and extract positions of landmark features. This is particularly useful in facial expression analysis. We use it to represent emerging passive sensing applications. To make the experiment reproducible, we use locally cached face images as its input.

**Input Workload** Application workloads are carefully chosen such that the executions of the interactive application and the background task can mostly overlap with each other when using our co-run strategy. Specifically, BBench are configured to load 15 websites with two seconds
delay (to mimic user think time) between each website. The whole session takes roughly 44 seconds to complete. Angry Bird, on the other hand, is played for 42 seconds. Background tasks are launched in the background shortly after the interactive application starts and the amount of the work is configured such that they can finish right before the interactive application ends under the most strict (throttling-based) background task scheduling policy. To make experiments reproducible, we use RERUN \cite{38}, a record and replay tool for the Android operating system, to automate the test flow. User interaction sessions are recorded into a sequence of touch and system events. Later, these events are sent back to the phone to replay user interactions with precise timing and accuracy.

3.5.2 Energy Efficiency

To evaluate the energy efficiency of our system when running background tasks with interactive applications, we run BBench and Angry Bird with each of the five background workloads. We run each pair under two different scheduling strategies:

- default, where there is no change to the original system behavior;
- power-states-preservation scheduling, where our non-work-conserving scheduling techniques are used.

Figure 3.2 and Figure 3.3 show the result. Energy discount ($\sigma$) of the background task is calculated as

$$\sigma = 1 - \frac{E_{\text{co-run}} - E_{\text{interactive,alone}}}{E_{\text{background,alone}}}$$

where $E_{\text{background,alone}}$ is the amount of energy consumed by the background task running alone under the default system setting, $E_{\text{co-run}}$ is the total system energy consumption of the co-run execution and $E_{\text{interactive,alone}}$ is the total system energy consumption when running the interactive application alone. Each of our energy metrics measures the active energy—those consumed above the fixed system idle energy consumption. Energy discount is essentially the reduction percentage of the marginal energy cost of the best-effort task when it co-runs with interactive application.
Figure 3.2: Experimental results of running interactive application browser loading BBench with various background tasks under different scheduling strategies. We show the background task energy discount (A), background task elapsed time (B), and impact on BBench’s interactivity (webpage rendering slowdown) (C).

The result clearly shows that our system can realize deep energy discount in all co-run scenarios, ranging from 23% to 71%. We attribute this to the fact that the overall CPU power states are preserved—the execution of the background task is completely hidden behind the interactive application power profile.
Figure 3.3: Experimental results of running interactive application Angry Bird with various background tasks under different scheduling strategies. We show the background task energy discount (A), background task elapsed time (B), and Angry Bird’s frame rate (C).

This is further illustrated in Figure 3.4. When BBench running alone, the current trace shows the typical burst-then-idle pattern that is common on smartphones due to the long user think time between interactions. During these idle periods, the system is able to enter deep sleep states to conserve energy (trough in the current waveform). However, background tasks, under the default work-conserving scheduling strategy, will disrupt these deep sleep states.
addition, during the burst period, simultaneous executions of both tasks would increase the system load and drive up the CPU frequency. In both cases, the system overall power states are elevated, resulting in more energy consumption. These are shown in the current waveform of the default co-run execution. With our power-state-preservation scheduling, both kinds of disruption can be avoided. The background task is piggybacked by the interactive application execution during the burst period, resulting in improved energy efficiency.

Figure 3.4: Current trace of BBench running alone and BBench+Spin co-run under alternative scheduling strategies. In the default co-run test (B), the spin task takes 9.07 seconds to finish. Under the power states preserving scheduling strategy (C), the same task takes 31.91 seconds.

It is worth noting that, in both Figure 3.2 and Figure 3.3, the default co-run strategy can also provide some energy discount. This is mostly due to the fact that we intentionally overlap the two application executions by launching them roughly at the same time. Thus a portion of the background task execution happens to be able to utilize some computing resources without elevating the overall CPU power state. In other words, the resulted energy discount is undependable at best. It completely depends on the application characteristics and how the user interacts with them. In fact, in some of our tests such as Angry Bird co-running with Compress and FaceAnalysis, the default co-run strategy can actually result in more energy consumption than running each task individually combined. Given the typical burst-then-long-idle smartphone usage pattern, the default co-run strategy is likely to perform poorly in most practical scenarios. Our system, on the other hand, always preserves the CPU power states and thus is able to consistently provide high energy discount under all circumstances.

Our non-work-conserving scheduling strategy inevitably reduces the system resource utilization and leads to longer execution time of background tasks. Fortunately, background tasks
do not involve direct user interaction thus their time of execution is somewhat flexible. The saved energy, on the other hand, could extend the smartphone battery life and let user use their phones more freely which could greatly improve the user experience.

### 3.5.3 Throttling-based contention mitigation

As shown in Figure 3.2, there are non-negligible slowdown on the interactive application when doing power-states-preservation co-run scheduling. In this part of the evaluation, we assess the effectiveness of our throttling-based contention mitigation technique described in Section 3.3.2.

We first focus on two application pairs which experience large interactivity slowdown: BBench+AppOpt and BBench+FaceAnalysis with 6.77% and 8.66% slowdown respectively. Different L2 miss rate throttling thresholds are used to evaluate their impact on the system performance. Table 3.2 shows the result. The throttling technique, with properly set threshold, proves to be effective in resource mitigation and minimizing the interactivity slowdown. With L2 miss rate threshold set at 15 misses/$\mu$s, both the background task elapsed time and the interactivity slowdown remain similar to the non-throttling based scheduling strategy. This means that the throttling mechanism is probably not triggered and a lower threshold is needed. With lower L2 miss rate thresholds, the background task begins to see increased elapsed time while the interactive application performance is improved. This suggests that the system is throttling background tasks while the memory bandwidth is under pressure as it reaches the L2 miss rate threshold. This in turn helps to improve the interactive application performance by reducing the resource contention caused by background task.

However, the benefit is diminished above 10 misses/$\mu$s. Beyond that, there is very little improvement and even negative impact on the interactive application and the elapsed time of the background task increases dramatically. For a threshold of 6 misses/$\mu$s, the FaceAnalysis benchmark could not even finish within the 44 seconds BBench session. This implies that 10 misses/$\mu$s L2 miss rate is a good indicator of the memory bandwidth saturation and any lower thresholds can only bring unnecessary slowdown to the background task without benefiting the interactive application.
Table 3.2: The impact of L2 cache miss rate throttling threshold on system performance when doing power states preservation and contention-aware scheduling. We show differences in background task elapsed time and BBench slowdown ratio for two scenarios: (A) BBench co-running with FaceAnalysis and (B) BBench co-running with AppOpt.

Next, we apply the contention-aware scheduling technique to all BBench co-run pairs. The throttling threshold is set at 10 misses/µs. Figure 3.2 shows the result. Besides reducing the BBench slowdown, we can also see that the throttling technique does not affect the energy savings. In addition, for Spin and Encryption where there is no slowdown of the interactive application, we observe little changes on the background task elapsed time. This suggests that
the scheduler is truly contention-aware—it only activates the throttling when there is resource contention in the system.

3.5.4 Trace-based Application Analysis

In this section we conduct a trace-based application analysis to study the amount of energy discounted computing opportunities in typical smartphone usage scenarios.

Eight popular applications are selected, their detailed descriptions are listed in Table 3.3.

For fair comparisons, each usage scenario lasts for one minute.

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web Browsing</td>
<td>In Chrome, go to Yahoo.com and browse three top news.</td>
</tr>
<tr>
<td>Video Streaming</td>
<td>In YouTube, watch a short HD video for one minute.</td>
</tr>
<tr>
<td>Gaming</td>
<td>Play casual game Subway Surf for one minute.</td>
</tr>
<tr>
<td>Navigation</td>
<td>In Google Maps, search nearby attractions and get their directions.</td>
</tr>
<tr>
<td>Messaging</td>
<td>In Hangout, open two conversations, type and send two messages.</td>
</tr>
<tr>
<td>Social Network</td>
<td>In Facebook, load personal timeline, refresh for friend feeds and browse three posts.</td>
</tr>
<tr>
<td>Camera</td>
<td>Use the native camera app to take a minute long video.</td>
</tr>
<tr>
<td>Music Streaming</td>
<td>In Google Play Music, stream a song for one minute.</td>
</tr>
</tbody>
</table>

Table 3.3: Description of eight application scenarios used in the trace-based application study.

We use Linux debugfs-based kernel trace facility to collect CPU frequency and idle state transition events and then calculate the amount of energy discounted computing cycles based on our power-states-preservation scheduling policy. Specifically, if at least one CPU is actively running at certain frequency, other idle CPUs are counted to be able to provide energy discounted computing cycles at that frequency.

For each usage scenario, we measure the abundance of energy discounted computing opportunities in relation to the active CPU usage of the corresponding interactive application. Further, we convert the amount of discounted cycles into meaningful background task workloads.
by normalizing it to the amount of CPU cycles needed to finish a unit of work (e.g., analyzing one frame of face or encrypt one minute standard resolution video).

Table 3.4 shows the result. As you can see, there are substantial energy discounted computing opportunities in all application categories. This is consistent with the earlier observation that typical smartphone applications lack the parallelism to utilize multicore CPUs. This study demonstrates the potential of exploiting the opportunities enabled by the lack of parallelism in smartphone applications to process useful background tasks at a deep energy discount.

<table>
<thead>
<tr>
<th>Category</th>
<th>Abundance of discounted CPU cycles (multicore)</th>
<th>Abundance of discounted CPU cycles (single-core)</th>
<th>Equivalent work of FaceAnalysis (frames of face analyzed)</th>
<th>Equivalent work of Encryption (minutes of video encrypted)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web Browsing</td>
<td>1.63</td>
<td>0.66</td>
<td>30</td>
<td>21</td>
</tr>
<tr>
<td>Video Streaming</td>
<td>2.41</td>
<td>0.85</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Gaming</td>
<td>1.61</td>
<td>0.65</td>
<td>21</td>
<td>15</td>
</tr>
<tr>
<td>Navigation</td>
<td>2.42</td>
<td>0.85</td>
<td>13</td>
<td>9</td>
</tr>
<tr>
<td>Messaging</td>
<td>2.88</td>
<td>0.97</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Social Network</td>
<td>1.88</td>
<td>0.72</td>
<td>12</td>
<td>9</td>
</tr>
<tr>
<td>Camera</td>
<td>2.10</td>
<td>0.77</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Music Streaming</td>
<td>1.63</td>
<td>0.66</td>
<td>7</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 3.4: Results for the trace-based application study. Each usage scenario lasts for one minute. Abundance of discounted CPU cycles is the ratio of energy discounted CPU cycles to the active CPU cycles used by the corresponding interactive application. In the second column (marked with multicore), energy discounted cycles on all CPUs are counted, assuming the background task has perfect parallelism to utilize all idle CPUs. In the third column (marked with single-core), energy discounted CPU cycles are only counted on one of the eligible CPUs, assuming the background task has no parallelism. We use single-core cycles to calculate the equivalent work of background tasks in column four and five.
3.6 Related Work

Smartphone power characterization and energy management have received a great deal of research interests. Using an extensive smartphone power instrumentation platform, Carroll and Heiser [11] developed a power model of various smartphone components and identified promising directions to improve power management. They [12] further suggest that, on a multicore smartphone, CPU cores should be kept online as long as there is work for them. AppScope [105] monitors kernel as well as application activities and correlates them with the smartphone power usage. Song et al. [90] optimized smartphone energy efficiency by lowering CPU frequency when user facing tasks are completed (e.g., display updates finish). Martins et al. [61] monitor and intercept smartphone background activities while the system is in suspension state to extend the battery life. In this chapter, we present a new approach to improve smartphone energy efficiency by carefully running background tasks together with interactive applications on a multicore to realize deep energy discounts.

Previous research has recognized the efficiency benefit of piggybacking or co-running background work while the system is active with primary tasks. A classic example [60] is to perform disk work “for free” if such work happens to lie in the disk head rotation and seek path to serve the foreground requests. In the context of mobile systems, Lane et al. [53] showed that performing sensing work while a smartphone is already active saves substantial wakeup energy costs. Nikzad et al. [70] developed an annotation language that demarcates power-hungry executions for delayed execution when the device enters an active state. We make new contributions in this work on energy-efficient multicore piggyback execution by non-work-conserving scheduling that preserves the system power states as if the background task does not run.

Work-conserving schedulers that always utilize available resources when there is work to do is generally desirable for high resource utilization. However, previous research has found the benefits of non-work-conserving scheduling in particular contexts. In disk scheduling, the anticipatory scheduler [46] may keep the disk idling for a short period of time even when there are pending operations. It does so in anticipation of a new I/O operation from the process that issued the just completed operation, which often requires little or no seeking from the current disk head location. In the context of hardware multithreading processors, Fedorova et
al. [27] found that running fewer threads than the number of processors may reduce resource contention and improve performance. In this work, we use non-work-conserving scheduling to run background tasks only when it does not elevate the power states of a smartphone.

There exists a large body of prior work on characterizing smartphone workload behaviors. Gao et al. [37] analyzed a broad range of mobile applications and found that they exhibit little thread-level parallelism and thereby are unable to effectively utilize multiple CPU cores. Seo et al. [86] showed that the lack of thread-level parallelism also prevents mobile applications from effectively utilizing the heterogeneous (big and little) multicore processors. Shingari et al. [88] have identified that co-running multiple mobile applications may yield substantial contention on shared multicore resources. These identified characteristics of smartphone workloads motivate our work of improving execution parallelism and mitigating potentially resulted performance interference.

Multicore performance and power management has been an active area of work for general computer systems. In particular, many techniques were proposed to manage and isolate the shared multicore resources, by partitioning the shared on-chip cache [95, 112], contention-easing scheduling [29, 111], and execution timeslice adjustment [28]. Multicore power disproportionality was also recognized in server power modeling [87]. Mobile systems present new circumstances for multicore performance and power management. First, a lack of thread-level parallelism results in poor multicore energy efficiency on smartphones. Second, the co-existence of interactive and background applications presents differential quality-of-service requirements that complicate resource management.

### 3.7 Conclusion

This chapter demonstrates the feasibility and benefits in running background tasks on multicore smartphones for an energy discount. We are motivated by the power disproportionality of multicore processors and the lack of parallelism in typical smartphone applications. Due to hardware resource sharing, the first running CPU on multicore processors could incur high power cost while each additional core can be used at a much lower energy cost. On the other hand, smart-
phone applications exhibit little parallelism, leaving room for energy discounted computing on the additional cores.

We propose to exploit these opportunities by running background tasks—tasks that are useful to the user but do not involve direct user interactions. Thus their delayed execution would not hurt user experience. Our contribution lies in the recognition that maximum energy discount can only be realized when overall system power states are preserved. Specifically, the multicore CPU idle state, the processor frequency and the system suspension time should not be affected by the presence of background tasks. We apply careful non-work-conserving CPU scheduling to achieve this goal. In addition, to deal with the interactive application slowdown caused by the co-run activities. We use performance counter based throttling to mitigate the contention on the memory bandwidth. We evaluate our work on Huawei Mate 7 smartphone with realistic workloads. The result shows significant energy discount (up to 63%) for background tasks with minimum impact on the interactive application execution (3.8% slowdown in the worst case).
4 Interactive Context for Mobile OS
Resource Management

4.1 Introduction

Mobile processors and devices are increasingly heterogeneous and dynamically configurable. For instance, latest Snapdragon mobile CPUs allow per-core, dynamic frequency and voltage control that enables fine-grained (at a few dozen microseconds) adjustment on energy-performance trade-offs [77]. At the same time, smartphones run a variety of versatile, concurrent tasks with unequal importance toward user interactivity. Typically, a smartphone user focuses on one task at a time—e.g., a user browsing the web is only concerned with the rendering speed and scroll smoothness of the current webpage. The same browser application may contain other webpages loading in background tabs and user data like new bookmarks or browsing history might be syncing to the cloud. Furthermore, system activities like GPS tracking and software updates may be running at the same time. If properly identified, a substantial amount of execution in such a system may be delayed without affecting user experience.

We introduce an application-transparent interactive execution context for mobile operating systems. Specifically, the OS labels execution activities as interactive or background depending on its perceived importance of affecting user response latency. Awareness of interactivity opens up new optimizations in mobile OSes. In particular, CPU resources can be properly prioritized for performance, or dynamically configured as described in Chapter 3 for energy efficiency at no loss of user-perceived performance.
A user session may involve executions on a number of application and OS tasks. A process (or thread) may perform work on behalf of interactive user sessions and background jobs alternately. Thus tracking the intricate inter-dependencies between different tasks on the fly is necessary to maintain the correct interactive context. Fortunately, a mobile OS with unified UI, runtime, and kernel software stack already possesses a variety of information to infer the initiation and propagation of a user interactive session. In Android / Linux, for example, a UI session starts on well-defined events like touch-screen actions. Dependent executions of a UI session can be tracked by monitoring control/data flow system calls (such as futex, epoll and sockets), kernel-level block and wake-up events, and standard Android inter-process and intra-process communication facilities such as Binder, message queue and Java thread library.

Prior works [1,3,5,6,14,16,30,35,79,89] have presented techniques to track execution dependencies and causality, mostly for the purposes of resource accounting and performance analysis. Our system differentiates from prior works in leveraging dependency tracking to guide online resource scheduling and power state management. A key risk is that an interactive execution that is misidentified as background, or not identified soon enough, could be treated as low priority by a resource scheduler, or subject to lowered power state, leading to degradation of interactive performance. This work recognizes and addresses such challenges.

First, OS-level execution context tracking suffers from inaccuracies. For example, a pair of kernel-level block/wake-up events, while signaling clear control dependencies, may be due to incidental synchronization on a critical section, rather than inherent application semantic dependencies. We treat dependency events differently based on their trustworthiness in signaling true application-level semantic dependencies and propagate execution context selectively. In particular, we introduce a new uncertain context to cover executions whose importance for user interactivity cannot be confidently identified by the OS due to the semantically dubious events. Executions with uncertain context are prioritized behind interactive executions but otherwise run as fast as possible to minimize priority inversion.

Second, a resource scheduling system poses stricter requirements in the timing of context propagation. While in resource accounting, for example, it makes sense to let a successor inherit the execution context of the predecessor after the successor wakes up, this may lead to adverse performance impact in resource scheduling. Consider a successor, currently in a
background context, is to receive an interactive context. Since it remains in the background until being scheduled, it can be severely delayed in a long scheduling queue before it gets to run and inherits the interactive context. Our solution is an *early-binding* mechanism to promptly propagate execution context at the predecessor event (before the successor runs).

Our interactive context abstraction works well with the emerging asymmetric clocking multicore smartphone that is capable of adjusting frequency and voltage at per-core granularity. We utilize similar control techniques proposed in Chapter 3 by ignoring the load of background tasks during frequency adjustment to save energy. We further take advantage of the agility of the asymmetric clocking platform by consolidating the identified background executions to a subset of frequency capped cores to minimize resource contention with interactive workloads and maximize energy savings.

The rest of this chapter details the motivation, design, implementation, and experimental evaluation of our proposed OS construct and techniques. The chapter ends with a discussion of related work and concluding remarks.

The rest of this chapter is organized as follows. Section 4.2 motivates our work with the unique characteristics of smartphone workloads, emerging heterogeneous platforms and mobile OSes. Section 4.3 presents the design of our interactivity execution context and its usage in resource management. Section 4.4 describes our implementation on the Android platform. Section 4.5 evaluates the performance of our interactive-aware system in terms of energy efficiency as well user perceived interactivity with realistic smartphone benchmarks. We present related work in Section 4.6 before concluding the chapter in Section 4.7.

### 4.2 Motivations and Feasibility

#### 4.2.1 Versatile, Concurrent Mobile Workloads

As smartphones become more powerful, they are increasingly used for a variety of purposes. Applications grow in sophistication and more of them are running at the same time. Yet user attention at a given moment remains the same. Typically, a smartphone user focuses on one task at a time. Residual executions of the previous user interaction session and other background
tasks do not relate to the current user interaction and thus have little direct impact on user-perceived quality-of-service.

Multi-tasking on smartphones is increasingly common. Download or upload operations to sync data with cloud services, post to social media websites and install/update applications are typical smartphone usages. In addition, as smartphones become “smarter”, they are taking every opportunity to sense and understand the surrounding context preemptively in the background. GPS and motion sensors track user fitness activities; virtual personal assistants like Siri analyze user usage patterns to offer various recommendations; Google’s Federated Learning [39] trains machine learning models locally on the smartphone to minimize latency and ensure user privacy. Other more creative ways of sensing like eye tracking [94] are also emerging. These tasks often do not have hard deadlines and thus their executions could be optimized to avoid resource contention with interactivity or minimize energy cost.

However, existing system abstractions such as process, thread, and Linux cgroup (a collection of processes as a unit of resource isolation) do not match well with the user interactive context. An interactive execution often spans across a dynamic set of threads. A single thread may frequently switch between interactive and background contexts. This is especially common on Android where a single thread may host several application components that serve both foreground user interactions as well as background operations [40].

4.2.2 Heterogeneous, Configurable Platforms

Smartphone users continually desire new and powerful features as well as longer battery life. Facing such challenges, modern smartphones are equipped with heterogeneous, dynamically configurable hardware with different performance-energy trade-offs.

Mobile multicore processors are the best example. They have very sophisticated power states which can be dynamically adjusted to adapt to different workloads. Heterogeneous chips with cores of different micro-architectures (e.g., ARM big.LITTLE) are used to provide power-proportionality in a wide power-performance range. However, to utilize such platforms for energy/performance differentiation, threads of different priorities need to be migrated between different clusters for performance and power scaling. This incurs relatively high overhead in-
CLUDING THREAT PERSISTENCE ON THE CLUSTER IS REQUIRED TO AMORTIZE THE MIGRATION COST.

RECENT MOBILE CPUs ALLOW PER-CORE FREQUENCY AND VOLTAGE ADJUSTMENT. FOR EXAMPLE, EACH CORE’S FREQUENCY OF THE SNAPDRAGON 800 QUAD-CORE PROCESSOR ON THE NEXUS 5 SMARTPHONE CAN BE INDEPENDENTLY ADJUSTED BETWEEN 300 MHz AND 2.3 GHz WITH A POWER DIFFERENCE OF 1.06 W FOR EACH CORE, WHICH IS MORE THAN THREE TIMES THE ENTIRE PHONE’S ACTIVE IDLE POWER CONSUMPTION. PER-CORE ADJUSTMENT DOES NOT REQUIRE THREAD MIGRATION WHICH IS PARTICULARLY SUITABLE FOR PROVIDING INTERACTIVITY BASED DIFFERENTIAL CONTROL WHERE A THREAD MAY SWITCH BETWEEN INTERACTIVE AND BACKGROUND CONTEXTS FREQUENTLY. ON SNAPDRAGON 800, THE CORE FREQUENCY AND VOLTAGE ADJUSTMENT LATENCY IS LESS THAN 50 $\mu$s.

4.2.3 EXPOSED APPLICATION SEMANTICS

THANKS TO THE UNIFIED MOBILE APPLICATION AND RUNTIME SOFTWARE STACK, MUCH OF APPLICATION SEMANTICS ARE EXPOSED TO THE MOBILE OPERATING SYSTEM, MAKING IT FEASIBLE TO TRACK THE INTERACTIVITY CONTEXT TRANSPARENTLY WITH FAIR ACCURACY.

THE WELL-DEFINED APPLICATION AND LANGUAGE APIs MAKE IT MUCH EASIER TO TRACK TASK DEPENDENCIES. FOR EXAMPLE, MESSAGE PASSING BETWEEN SIBLING THREADS ARE USUALLY INVISIBLE TO A GENERIC KERNEL SYSTEM IF NO OS SUPPORT (E.G., THREAD BLOCK/WAKE) IS REQUIRED. MOREOVER, EVEN WHEN DEPENDENCY PROPAGATION EVENTS ARE OBSERVABLE AT THE KERNEL LEVEL, THEY OFTEN LACK THE HIGH-LEVEL APPLICATION SEMANTICS, LEADING TO IMPRECISE KERNEL INFERENCE. FORTUNATELY, MOBILE OSES PROVIDE APPLICATION DEVELOPERS WITH STANDARD INTRA-PROCESS AND INTER-PROCESS COMMUNICATION APIs. THUS LIGHT INSTRUMENTATION IN JUST A FEW KEY SYSTEM FACILITIES COULD EXPOSE MAJOR APPLICATION EXECUTION DEPENDENCIES TO THE OS. ON ANDROID, FOR INSTANCE, THE MESSAGE QUEUE INTERFACE IMPLEMENTS A THREAD-SPECIFIC EVENT LOOP THAT WAITS FOR AND DISPATCHES MESSAGES TO HANDLERS OF ITS SIBLING THREADS. ANDROID ALSO HEAVILY RELIES ON THE BINDER FRAMEWORK WHICH MANAGES A THREAD POOL FOR EACH PROCESS TO CARRY OUT INCOMING RPC REQUESTS, EXPOSING CLEAR APPLICATION DEPENDENCIES.

SMARTPHONE ALSO HAS A UNIFIED UI FRAMEWORK WHICH PROVIDES THE SYSTEM WITH INTIMATE KNOWLEDGE ABOUT APPLICATION RUNTIME STATES. ON ANDROID, ONLY ONE FOREGROUND APPLICATION
Figure 4.1: An illustrative example of a user interaction session. The user touches a button to query the device status. Important data and control flows between threads are marked with arrows. Darkened portions of a thread timeline indicate active executions (while the rest represents blocking waits).

Figure 4.1 illustrates a representative (but simplified) user interaction session on Android. A user touch event wakes up the application UI thread which belongs to the foreground process group. It then submits an asynchronous callback to the worker thread which in turn makes a request to the system server daemon to query the device status. Thanks to the standardized Android facilities such as Binder and message queue, important control and data flow between different entities are well exposed, beckoning system-level context tracking.
4.3 Interactivity Context for Mobile Resource Management

Given the diverse nature of mobile workloads and the emerging heterogeneous smartphone hardware, mobile OSes are increasingly expected to provide differential control by matching different types of workloads with appropriate hardware resource to improve performance and, above all, energy efficiency. In this section, we present such a system with the goal of minimizing smartphone energy consumption without impacting the user experience. To this end, our system maintains an interactive context that encapsulates user-perceivable executions in an application-transparent way, recognizes and tolerates inherent OS-level tracking inaccuracies, and leverages this interactivity awareness to minimize the CPU energy consumption of background executions through differential core frequency scaling and task consolidation.

4.3.1 Interactivity Context Maintenance

Context Initialization

Typical smartphone usage involves users inputting through the touchscreen and getting output from display updates. Ideally an interactive context should capture all and only executions that the final display update depends upon. However, this is impossible to do precisely without knowledge of the future. Our approach is to infer precedence to final display update by causality from the user input. Executions caused by the latest user input is likely semantically related to and thus depended upon by the next display update which is normally what users care about at the moment. On the other hand, executions that are not triggered by the recent user input (e.g., system maintenance work) or residue executions from the previous user input are unlikely related to what users expect to get next. They are categorized as background work.

An interactive context starts when events from user input devices (typically touchscreen interrupts) wake up the UI thread of the current foreground application. A mobile OS can capture these instances by recognizing the foreground application (e.g., through Android application life cycles [76]) and monitoring relevant kernel event interface `epoll_wait()`. Each time, two execution contexts are initialized—the interactive context contains the foreground threads (the UI thread and its siblings) and the background context includes all remaining threads.


Context Propagation

After the initiation, an execution context may propagate through additional threads and processes. We capture these propagation through various OS-visible data and control flows. Specifically, three sources of information signal potential execution dependencies—

- **Android framework calls** including Binder RPCs, thread message queue communications and Java thread pool library calls. We also make use of Android application life cycle information which guides the labeling of application threads according to their current visibility to the user;

- **Linux system calls** including process (or thread) forking, Unix domain sockets, `futex` synchronization and `epoll` event processing calls;

- **Low-level kernel** thread block / wake-up functions.

In each case, successors (child in forking, epoll event processor, message receiver in communication primitives such as Binder and message queue and wakee in kernel block / wake-up events) inherit the execution contexts of the predecessors (forking parent, event producer for epoll, message sender and waker in corresponding cases).

These three types of events reside in three system levels—Android framework, Linux syscalls, and low-level kernel that build on top of each other. For example, Android message queue utilizes kernel `futex` calls for synchronization. And `futex` further relies on the kernel block / wake-up function to actually stop and resume thread executions. The three levels have increasing distances from applications and therefore decreasing faithfulness in reflecting application-level semantics. We will explore its implication on context propagation inaccuracies in Section 4.3.2.

Early Context Binding

We exercise caution in controlling the timing of context propagation. In a typical dependency tracking system, the propagation often takes effect when the successor performs the context receiving action. For example, in the case of message passing, the sender tags the message with its execution context at the send time. The actual propagation to the receiver, however,
would not happen until the message receipt. Or in the case of block/wake-up relation, the wakee usually inherits waker’s context after it is woken up. Such practice makes sense for the purpose of resource accounting [5,6] or security tainting [23]. However, in our system where execution context is used to guide priority resource scheduling, this may lead to priority inversion. For instance, when an interactive sender submits a work request to a receiver that is currently labeled as background (idle or occupied with background work), the receiver will not inherit the interactive context until it is scheduled or finishes the background work at hand. Figure 4.2a further illustrates the issue in the case of wake-up event.

To guard against such hazard, we practice early binding when it is possible. Specifically, as soon as the successor is known, and if the predecessor has higher context priority (i.e., if the predecessor is interactive and the successor is not), we propagate the higher priority context immediately. Figure 4.2b illustrates how early binding could eliminate potential priority inversion. Our early context binding is reminiscent of the Lazy Receiver Processing [20] that binds an incoming network packet to an application socket at the early stage of interrupt handling.

The key to realize early binding is whether the successor can be identified at the time when the predecessor is initiating the context propagation. This is relatively straightforward in inter-thread communications such as futex, epoll and Android MessageQueue. Each such communication involves a deterministic pair of threads and therefore the predecessor can easily identify and manipulate the execution context of the successor at propagation initiation. In other cases like UNIX domain socket and Android Binder or multi-consumer scenarios such as the Java ThreadPoolExecutor interface, the identity of the successor is unknown until the successor performs receive (or equivalent) action. In such cases, further help from the successors is needed to realize early context binding. For example, successors may expose new interface to the predecessor such that the predecessor can designate a specific worker thread to handle its request.

**Context Closing**

There is no explicit context destroy operation in our system. Instead, each time a new user input is detected, we re-initialize the execution contexts. Strictly speaking, any execution after displaying user-expected content is not part of the interactive execution context. However, it
Figure 4.2: Illustrations of conventional and early binding based context propagation for the wake-up event where interactive (high priority) waker wakes up background (low priority) wakee. Dashed lines indicate thread blocking wait (while the solid lines represent active execution). Figure (a) illustrates conventional binding where the background wakee would not inherit the interactive context until it is picked up by the priority-aware scheduler. The wakee may experience a lengthy delay before it gets scheduled if the CPU is busy running interactive or even background tasks, (in the latter case) leading to priority inversion. Figure (b) illustrates early binding where the waker promptly propagates its interactive context to the wakee while the wakee is still blocking, causing the scheduler to pick the (now interactive) wakee early.
is difficult for the OS to recognize such ending points without the ability to look into future executions or receive a notification from the application. Fortunately those residual executions are often minor before user starts next UI action.

4.3.2 Recognizing and Managing Inaccuracies

Differentiation of Dependency Events

OS-level execution context tracking suffers from inherent inaccuracies. Lacking direct knowledge of the application semantics, it can only infer context propagation when applications make use of OS facilities that signal execution dependencies. However, program execution dependencies may result from incidental causes such as random scheduling decisions, that have no connection to the innate application behavior, leading to incorrect inferences.

To minimize context propagation errors, instead of blindly trusting all dependency events, we treat the three types of events mentioned in Section 4.3.1 differently depending on their distances to the application and trustworthiness in signaling true application-level semantic dependencies.

We completely trust events from the Android framework. These dependencies are mostly propagated through message passing or remote procedure calls (RPC) rather than opaque synchronizations. These events bear clear semantics as applications and the framework are closely knitted together.

On the other hand, we ignore kernel block/wake-up events which are semantically farthest from the application. Some of them are low level incarnations of higher level dependency events such as futex and epoll. Others may be products of unrelated scheduling decisions. For example, when handling interrupts, a thread may get interrupted and incidentally wake up another thread without any intrinsic inter-dependencies.

Lastly, for Linux system calls, we make further discretion. Thread fork, UNIX domain socket communications and epoll event processors carry clear semantics and are thus trusted. However, futex calls deserve extra caution when being used to infer context propagation.

Uncertain Synchronization
**futex** is the building block of Linux user space synchronization. Its opaque dependency semantic may result in both false negatives as well as false positives in context propagation.

False negative errors arise when context propagation happens through synchronization that does not require OS support and thus is effectively invisible to the OS. For instance, a thread may signal another waiting thread through a monitor. A synchronization operation must call into the OS kernel (yielding the CPU through a **futex** wait in Linux) when the waiter has to block while waiting for the synchronization signal. If such blocking is unnecessary (e.g., when the waiter can proceed immediately since the waited-for signal has already occurred), the entire synchronization would present no visibility to the OS. Figure 4.4 illustrates this scenario for the code snippet in Figure 4.3. When the synchronization between Thread_B and Thread_C is not necessary as Thread_B is already woken up by Thread_A earlier, the context propagation from Thread_C is missed by the OS.

```java
class WorkQueue {
    boolean enqueueWork(Work w) {
        synchronized (this) {
            // Insert work item.
            ... if (needWake) notify(ptr); 
        }
    }
}

Work next() {
    for (; ;) {
        // Block if queue is empty.
        ... check(ptr);
        ... if (needWake) notify(ptr);
        ... // Retrieve next work.
        ... }
    } // end of WorkQueue
```

Figure 4.3: Code snippet of Java class **WorkQueue**. User space synchronization primitives such as **synchronized** monitor and **notify** function are implemented using generic OS facilities such as **futex** syscall. We use this snippet to illustrate different context propagation scenarios in Figure 4.4.

False negative context propagation errors have long been recognized. A limited (and expensive) solution is to identify synchronization data structures in advance and invalidate relevant page accesses for OS traps [14].

While missing synchronization events may jeopardize context maintenance, some user space synchronization events, despite being perfectly visible to the OS, may not necessarily signal true context propagation. Trusting them blindly will lead to false positive propagation errors. Figure 4.4 illustrates such a case for the code snippet in Figure 4.3. The OS propagates an incorrect execution context through a user space synchronization that does not reflect
Figure 4.4: Illustrative examples of intra-process communications using WorkQueue presented in Figure 4.3. The left part shows the execution events timeline. The right part shows visible activities and the resulting context propagation in the OS. Dotted lines indicate thread blocking waits (while the rest represents active execution). In all three cases, Thread_A and Thread_C each sends one message to Thread_B.
innate application causality. The synchronization between Thread_A and Thread_C happens only to protect the critical section in the `enqueueWork()` function from concurrent accesses. It is the product of the incidental scheduling and execution order but not inherent causality in application semantics.

Comparing to false negatives, false positive propagation errors are mostly overlooked by previous dependency tracking systems \([5,6,23,30]\). This is not surprising. When tracking information leak in security systems, false positives are relatively innocuous. In resource accounting systems, coverage is often of higher importance and a segment of false positive would only lead to mis-accounting that is proportional to its execution segment size. However, when the execution contexts are used to guide resource scheduling, this may lead to disproportional harm.

**Resource Management with Uncertainty**

There could be two types of mis-labeling in our system—

- A *background→interactive* mis-labeling would result in executing the background segment unnecessarily early and at a higher energy cost, reducing energy efficiency and potentially impacting interactive performance (if computing resource is contended). The impact is proportional to the mis-labeled segment size.

- An *interactive→background* mis-labeling, on the other hand, will cause priority inversion and may lead to severe degradation in interactive performance that is disproportional to the mis-labeled segment size.

Our general goal of resource scheduling is to make scheduling performance (application response latency and energy saving) proportionate to the completeness and perfection of interactivity context tracking, and minimize the risk of priority inversion. To this end, when dealing with the semantic ambiguous *futex* calls, we raise alert when background context propagates to an interactive task. Instead of labeling the interactive task as background, we put it into a new *uncertain* context and expose the uncertainties directly to the resource scheduler.

Following the same guideline, our CPU scheduler first picks tasks of interactive context, then tasks of uncertain context, and last tasks of background context. Under this approach,
assuming that interactive and background contexts are correctly identified, priority inversion may only happen within the uncertain context, causing less severe performance degradation.

4.3.3 CPU Scheduling and Power State Control

Awareness of interactivity enables a variety of optimizations in OS resource management. We focus on improving the smartphone energy efficiency through CPU scheduling and power state control.

Energy remains the critical resource bottleneck for typical smartphone usage. And CPU is usually the predominant energy consumer. We take advantage of the increasingly popular heterogeneous CPU architectures. Specifically, we make use of the per-core frequency and voltage scaling capability of the Snapdragon 800 quad-core processor available on the Nexus 5 smartphone. Current mobile systems are not able to fully utilize this fine-grained performance-energy adjustment feature—the common practice is to set the core frequency based on the overall system load regardless of the load’s importance to the user. In contrast, our system possesses fine-grained interactivity context that well matches such agile hardware platform.

As mentioned in Section 4.3.2, our CPU scheduler picks tasks in each core’s run queue in three descending levels of context priorities: interactive, uncertain, and background. In addition to improving system interactivity and minimizing the possibility of priority inversion, this also consolidates executions of the same context and divides CPU executions into alternating phases, facilitating further differential frequency control for energy optimization.

Core frequency is generally controlled by the CPU governor based on the system load. To improve energy efficiency without impacting user interactivity, we utilize the similar technique described in Chapter 3: the CPU governor is modified to ignore the load of background tasks—a higher core frequency only depends on more interactive load. Therefore after a transition from interactive to background phases, the core frequency gradually drops to the lowest, resulting in best energy efficiency. When interactive tasks wake up, the governor responds quickly to satisfy their performance needs.

This approach works particularly well with the per-core adjustment hardware feature. Traditional multicore processors often share clocking circuitry among cores of the same cluster,
forcing each core running at the same speed. For our system, this would mean the core frequency would not drop unless all cores are running background tasks simultaneously. Fortunately, processors with asymmetric clocking capabilities do not have such restrictions. As soon as one core runs out of interactive task, its frequency is lowered, maximizing the energy benefits of interactivity awareness.

**Background Task Consolidation**

While our system works well with interactive and background executions alternating on a single core, it is still desirable to consolidate tasks of the same context on to separate cores as much as possible. Task consolidation would result in less frequent CPU power state adjustments (thus less switching overhead) and allow more aggressive core frequency drop for background tasks without perturbing the performance of interactive ones on the same core.

However, the execution context of a consolidated thread must be stable enough to amortize the migration cost. Fortunately, foreground and background application executions, although overlapping in numerous system daemon services, typically spend most of the time in a few application threads and system processes. This is not surprising given the different nature and functionalities of distinct applications. Threads with relatively stable execution context, once identified, are ideal targets for task consolidation.

We allocate cores with capped (lower than normal) peak frequency exclusively for background tasks. Tasks with a relatively long execution time and stable background context are migrated to these cores to gain maximum energy efficiency. This also helps to improve interactive performance thanks to reduced contention on shared core resources such as L1 cache and TLB.

Keeping interactive tasks away from the frequency capped cores improves energy efficiency, but it also incurs the risk of impacting the interactive performance due to reduced CPU availability. Fortunately, typical interactive applications do not have sufficient parallelism to utilize multiple CPU cores simultaneously. Recent studies [37, 86] on Android applications show a lack of thread-level parallelism across applications and an over-provisioning of core resources across devices.
4.4 Implementation

We have implemented our context tracking facility in the Android 4.4 mobile OS and Linux 3.4.0 kernel. Most of our modifications reside in the kernel with some light instrumentation in the Android framework and Java library.

We added a 72-byte execution context data structure associated with each task (as a part of `task_struct` in Linux kernel) that includes an identifier and execution statistics such as total and individual context execution time, context propagation/inherent counts and etc. They are exposed through the `/proc` file system. These statistics are re-initialized and tracked for each UI transaction. Specifically, each interactive context is identified by a generation number incremented at every new user input event. Each time a thread switches in, it first checks whether the local execution context data structure has expired (i.e., not matching the current interactive context generation number) and re-initialize if needed.

We added two new system calls—`get_context()` and `set_context()` for user-space code to query and propagate execution contexts. These are intended for instrumentation in the Android framework `MessageQueue` and Java library `ThreadPoolExecutor` class (through the Java native interface). Potential abuse from third party applications can be easily detected through code analysis at pre-installation time.

We modified the pick-next-task logic of the Linux complete fair scheduler (CFS) to realize priority scheduling of interactive tasks. Tasks are picked in three descending priority levels: interactive, uncertain and background. Within each level, however, tasks still follow the original CFS execution order for fairness. For frequency adjustment, we keep track of the total background execution time of the last epoch during the periodic scheduling tick. This time is then subtracted from the load calculation in the `cpufreq` governor to avoid raising frequency for the background workload.

We implemented task consolidation on the quad-core LG Nexus 5 smartphone. We dedicate one of the four cores to heavy background tasks. Its peak frequency is capped at 800 MHz (2.3 GHz is the default peak frequency). Initially, all tasks except kernel threads are restricted from using this core. Tasks are migrated and pinned to the low frequency core if it has more than 10 millisecond execution time and if more than 90% of it is background execution. On
the other hand, if a task’s background execution falls below 90%, it is migrated back to normal cores. The maximum number of migrations for a task during each UI session is capped at four to avoid thrashing.

Our interactive context tracking incurs minimum overhead. A context propagation essentially only involves an atomic flag update in the `task_struct`. The most significant overhead would come from the user-space instrumentation in the Android message passing and Java thread pool interfaces. These communications need to be trapped to the OS to ensure correct context maintenance. Fortunately, these operations are infrequent. On average, our system incurs less than 1.1% overhead.

## 4.5 Evaluation

### 4.5.1 Evaluation Setup

#### Experimental Device

We use an LG Nexus 5 smartphone running Android 4.4 KitKat. It is equipped with a Qualcomm Snapdragon 800 quad-core processor with a peak frequency of 2.3 GHz. It is capable of asymmetric clocking—each core’s frequency as well as voltage can be individually adjusted.

One thing we noticed is that the CPU on the Nexus 5 is very thermal sensitive. Its cores can not sustain a high frequency simultaneously for an extended period of time. Thus, for lengthy experimental benchmarking, it is necessary to keep the device cool to obtain a stable result. We accomplish this by putting the device on top of an ice pack.

#### Power Measurement

For precise power measurement, we remove the internal battery and connect its power pins to a Monsoon Power Meter [67] which acts as an external power source and measures the smartphone’s overall power consumption. The power meter is able to sample the current at 5 kHz. We turn off hardware components like GPS, cellular and dim the display to the minimum brightness. WiFi is kept on but only connected to the local area network for the purpose of controlling the phone through the host machine without the USB connection (which would
disturb the power measurement). Internet connection is cut off to avoid power disturbances (e.g., random push notifications). Figure 4.5 shows our experimental platform setup.

Figure 4.5: Experimental platform setup including Monsoon power meter, logging PC and Nexus 5 smartphone. (A) The Monsoon power meter connects directly to the power supply pins of the battery-less smartphone, acting as an external power source and measuring the whole system current. (B) The Nexus 5 smartphone is not thermal stable. We put it on top of ice pack to get stable performance results. (C) No USB connection to the smartphone to avoid power disruption. The phone is controlled by the host through wireless channels. (D) Using the provided logging software, we are able to record the current at 5 kHz.

**Benchmarks**

We assemble a suite of benchmarks to represent typical smartphone usage scenarios.
### Table 4.1: Application workloads, their average power consumption, and elapsed time under the default system setting.

<table>
<thead>
<tr>
<th>#</th>
<th>Interactive application</th>
<th>Background application</th>
<th>Average power (Watt)</th>
<th>Elapsed time (Second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BBench</td>
<td>AppInstall</td>
<td>2.53</td>
<td>21.5</td>
</tr>
<tr>
<td>2</td>
<td>BBench</td>
<td>SensorService</td>
<td>2.20</td>
<td>22.3</td>
</tr>
<tr>
<td>3</td>
<td>BBench</td>
<td>FaceReco</td>
<td>2.28</td>
<td>21.8</td>
</tr>
<tr>
<td>4</td>
<td>BBench</td>
<td>AppInstall</td>
<td>3.12</td>
<td>21.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FaceReco</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Shazam</td>
<td>AppInstall</td>
<td>3.74</td>
<td>43.2</td>
</tr>
<tr>
<td>6</td>
<td>Shazam</td>
<td>SensorService</td>
<td>3.44</td>
<td>34.1</td>
</tr>
<tr>
<td>7</td>
<td>Shazam</td>
<td>FaceReco</td>
<td>3.91</td>
<td>33.8</td>
</tr>
<tr>
<td>8</td>
<td>Shazam</td>
<td>AppInstall</td>
<td>4.63</td>
<td>43.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FaceReco</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Kindle</td>
<td>AppInstall</td>
<td>2.58</td>
<td>42.8</td>
</tr>
<tr>
<td>10</td>
<td>Kindle</td>
<td>SensorService</td>
<td>2.59</td>
<td>41.4</td>
</tr>
<tr>
<td>11</td>
<td>Kindle</td>
<td>FaceReco</td>
<td>3.06</td>
<td>35.4</td>
</tr>
<tr>
<td>12</td>
<td>Kindle</td>
<td>AppInstall</td>
<td>2.92</td>
<td>38.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FaceReco</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Three interactive applications are chosen. BBench [42], a widely used web browsing benchmark which automatically loads and renders locally cached popular websites. It measures the browser performance by tracking the JavaScript `onLoad` event which is triggered once a webpage is fully rendered. We load it using the Chrome mobile browser. Popular music recognition application Shazam and e-book reader application Kindle are the other two interactive benchmarks. We measure their launch time performance through the Android internal logging which calculates the amount of time elapsed between launching the application and finishing drawing the corresponding activity on the screen [41].
We select three background applications. AppInstall installs the downloaded Facebook application. Installing application is a common background workload. SensorService is an Android application that periodically queries and logs various device sensors such as gyroscope, gravity sensor and etc. at 100 Hz. This represents common smartphone sensing activities like step counting and speed tracking. FaceAnalysis is an in-house developed Android application that analyzes input faces. It uses Stasm\[65\], an active shape model based library, to process images and extract positions of facial features. This is particularly useful in facial expression analysis. We use it to represent emerging passive sensing applications. To make the experiment reproducible, we use locally cached face images as input.

Each interactive application co-runs with one or more background applications. Table \ref{tab:appcombination} lists application combinations used in our evaluation. We carefully marshal these benchmarks such that the background executions are mostly overlapped with foreground applications. In the case of short interactive sessions (launching Shazam and Kindle), we repeatedly launch, wait for a short period of time, and then kill the application. To make UI interactions reproducible, we use RERUN \[38\], a record and replay tool for the Android operating system, to automate the test flow. User interaction sessions are recorded into a sequence of touch and system events. Later, these events are sent back to the phone to replay user interactions with precise timing and accuracy.

### 4.5.2 Overall System Evaluation

We evaluate the overall performance—foreground application interactivity and energy efficiency of our system. We first compare our interactivity-aware system with the default smartphone setting. Results in Figure \ref{fig:energy} show that our interactivity-aware system can significantly improve energy efficiency, achieving 13\% on average (up to 23\%) energy saving. Meanwhile, it incurs small interactivity degradation—1\% on average and 6\% in the worst case. In a few cases such as workload #6 and #10, it even improves the performance by 8\% and 7\% respectively. This demonstrates the value of application-transparent interactivity awareness—scarce mobile resources are optimized for user-centric activities, improving user interactivity and prolonging battery life.
Figure 4.6: Active energy consumption (A) and foreground application interactivity (B) under various system settings. We compare four different system settings: default, two variants of Android process importance based scheduling—treating system service tasks as either interactive or background, and our interactivity-aware scheduling. Both active energy and interactivity are normalized to that of the default system. When treating system tasks as background in the Android process importance based scheduling, benchmarks involving application launch (workload #5 to #12) usually end up with system hang or crash rendering the interactivity and energy measurement noninstructive.

We then compare our system with other potentially viable differential resource scheduling schemes. In particular, we are interested in the possibility of leveraging the Android process importance hierarchy [76] in guiding CPU resource scheduling.

Android process importance hierarchy assigns different values to application processes based on their perceived importance to the user. It does so by monitoring the application life cycle callbacks. This importance hierarchy helps the memory management in selecting victim processes to kill when memory pressure arises. It is, however, not designed to guide scheduling other resources. For our purpose in inferring interactive context, the Android importance hierarchy either marks an application process as interactive (with kernel oom_adj value equals to zero) or non-interactive (with positive oom_adj values). Non-application processes such as An-
droid service daemons that are shared by multiple applications have highest importance (with negative `oom_adj` value) and are last to kill. Our experimental usage of importance hierarchy in directing CPU scheduling is straightforward—interactive application processes are prioritized over non-interactive ones. There are two variants depending on treating non-application processes as either interactive or non-interactive.

Figure 4.6 shows the results. For the variant of treating non-application processes as interactive, it brings only 5% energy saving on average, far less than the 13% energy saving of our system. This is because non-application processes often constitute a large chunk of the background workload such as optimizing application binaries before installation or managing sensor components in the background sensing applications. On the other hand, treating non-application processes as non-interactive, while supposedly could bring more energy saving, does not appear to be a viable strategy. As all benchmarks involving application launch, a common user action, experience severe performance degradation, resulting in system hang or even crash. This is not surprising given that launching application usually involves non-trivial amount of system processes support. Treating them as non-interactive would lead to priority inversion. This shows the limitations of primitive system constructs such as the Android process importance hierarchy and demonstrates the necessity of full-system dependency tracking in maintaining the interactive context.

### 4.5.3 Evaluation of Uncertain Context Propagation

This section evaluates our technique in dealing with the inherent context tracking inaccuracies discussed in Section 4.3.2.

We compare three context propagation strategies regarding the low-level dependency events. These events are either ignored, completely trusted or partially trusted. In the last case, low-level kernel block/wake-up events are ignored but uncertain context is propagated when a low priority task wakes up a high priority one through the semantically ambiguous `futex` call. Figure 4.7 shows the results.

Ignoring or partially trusting the low-level dependency events can both bring high energy efficiency, 15% and 13% energy savings respectively. But the strategy of blindly propagat-
Figure 4.7: Active energy consumption (A) and foreground application interactivity (B) under various context propagation strategies: propagate only through higher level dependency events (ignore low-level events such as futex and kernel block/wake-up), propagate through all dependency events and our uncertainty aware propagation scheme introduced in Section 4.3.2. Both active energy and interactivity are normalized to that of the default system.

Table 4.2 further presents the coverage of the uncertain context—percentage of CPU executions that are labeled as uncertain. As you can see, uncertain executions are relatively small which opens up opportunities in resource optimization. A large amount of uncertain execu-
tion, on the other hand, shows lack of confidence in execution labeling and limited resource optimization space.

<table>
<thead>
<tr>
<th>Workload #</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncertainty</td>
<td>1.48%</td>
<td>1.91%</td>
<td>0.89%</td>
<td>1.84%</td>
<td>1.66%</td>
<td>1.11%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Workload #</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncertainty</td>
<td>0.78%</td>
<td>1.76%</td>
<td>1.32%</td>
<td>0.98%</td>
<td>1.17%</td>
<td>1.95%</td>
</tr>
</tbody>
</table>

Table 4.2: Uncertain execution context coverage.

### 4.5.4 Evaluation of Early Binding Context Propagation

In this section, we evaluate the benefit of early binding context propagation compared to the conventional binding approach. As shown in Figure 4.8, both approaches save 13% energy on average. They also deliver comparable interactivity except for workload #4, #8 and #12 where the early binding approach has a noticeable edge on performance by 4%, 6%, and 7% respectively.

The benefit of early binding is determined by how long the to-become-interactive threads need to wait before inheriting interactive labels. This mostly depends on the system load. When there are multiple background workloads co-running at the same time (e.g., workload #4, #8 and #12), this time gap is much more significant, leading to interactivity degradation. As a result, these are the scenarios that can benefit most from the early binding strategy.

### 4.6 Related Work

Prior work has characterized smartphone workload behaviors. Studies by Gao et al. [37] and by Seo et al. [86] found that mobile applications exhibit little thread-level parallelism and thereby are unable to effectively utilize multiple CPU cores. Fan and Lee [25] showed it is critical to exploit workload diversity and parallelism within an application that requires the mobile cores to share cache. These identified characteristics of smartphone workloads motivate increased execution parallelism and resulted opportunities for performance and energy optimizations.
Figure 4.8: Our interactivity-aware system with uncertain context propagation strategy runs with conventional and early context binding schemes. We show (A) active energy consumption and (B) foreground application interactivity. Both active energy and interactivity are normalized to that of the default system.

TaintDroid [23] performs information flow taint tracking on mobile devices to protect sensitive data. It does not distinguish execution contexts or support resource scheduling and optimization. Panappticon [109] logs mobile system event and identifies causality offline to support performance analysis and debugging. It does not maintain an online execution context necessary for CPU scheduling and power state control. ShuffleDog [45] identifies delay-critical threads in a mobile system and elevates their scheduling priority for high responsiveness. It does not maintain fine-grained interactive/background execution contexts. AppInsight [79] and Timecard [80] instrumented mobile app binaries to identify user transaction critical path and perform response-aware server adaptation. Their approach depends on the use of a specific application development framework (Microsoft Silverlight) which eases the tracking of dependencies among application methods and callbacks.

al. [1] employed signal processing techniques to derive causal relationships among communication events. Other work including Chen et al. [15], Google’s Dapper [89], X-Ray [3], and the Mystery Machine [15] record and process request, path, or transaction traces to perform offline performance analysis and debugging. Nightingale et al. [69] collect and delay I/O in an online transaction to optimize I/O performance. Whodunit [14] showed that some user-level request context transfers may be observed at the OS by trapping accesses to critical synchronization data structures. The X-Trace [35] and Quanto [34] frameworks monitor data flows in network packets and propagate activity labels across a distributed system. Context and path tracking in these systems target resource accounting or performance analysis. In comparison, interactive context tracking in this chapter is designed to support online scheduling and resource control. Our contribution lies in the recognition and tolerance of OS-level context tracking inaccuracies to support effective resource scheduling and mitigate the risk of priority inversion.

Aside from direct control and data flows, the interactivity dependency may manifest in more subtle ways. For instance, drowsy power management [54] monitors device open, close, and read/write accesses to model a device state machine and infer task-device dependencies. In another example, quasi-asynchronous I/O [48] recognized that file system functions sometimes depend on the completion of asynchronous I/O (that was issued earlier on blocks needed by later synchronous functions). They call for more comprehensive dependency tracking in the system, or exposing the uncertain context when precise tracking is impossible or too expensive.

Power profiling and modeling of mobile systems has received a great deal of research interests. Early work by Flinn and Satyanarayanan [32] coordinated the external power measurement with interrupt-triggered program sampling to profile the energy usage of application processes and procedures. Using an extensive smartphone power instrumentation platform, Carroll and Heiser [11] developed a power model of various smartphone components and identified promising directions to improve power management. Eprof [74] addressed the accounting of asynchronous power behaviors and concurrent accesses to I/O devices for smartphone systems. AppScope [105] monitors kernel as well as application activities and correlates them with the smartphone power usage. Smartphone energy optimization in our work does not require an elaborate power model but may benefit from it if quantitative tradeoffs between performance and energy metrics have to be made.
Song et al. [90] optimized smartphone energy efficiency by lowering CPU frequency when user facing tasks are completed (e.g., display updates finish). It leverages the mobile phone UI centric model to improve energy efficiency: monitor the UI thread activity (and its dependencies) to determine when UI transaction ends and lower the CPU configuration. However, no interactive context is maintained and energy saving only occurs after a UI transaction completes. Nikzad et al. [70] developed an annotation language that demarcates power-hungry executions and notifies the OS for delayed execution when the device enters an active state. Martins et al. [61] monitor and intercept smartphone background activities while the system is in suspension state to extend the battery life. Piggyback CrowdSensing [53] and Energy Discounted Computing [113] piggyback low-priority (or best-effort) work by non-work-conserving scheduling that preserves the system power states as if the best-effort work is not performed. Our work distinguishes mobile executions in interactive and background contexts (even when they run concurrently) so that interactive performance is not degraded in energy optimization.

4.7 Conclusion

This chapter presents an application-transparent, interactive context in the mobile OS that reflects the criticality of current execution on user interactivity. This interactive context enables new optimizations in CPU scheduling and power state management. At the same time, we face the unique challenge of relying on the imperfect OS-level context tracking to direct resource scheduling. Our contribution contains new techniques in minimizing the risk of priority inversion during context propagation.

Specifically, we recognize that certain OS-visible events such as block / wake-up events and futex system calls, while signaling clear control dependencies, may carry ambiguous semantics that could induce false context propagation. Our solution is to treat dependency events differently based on their trustworthiness in signaling true application-level semantic dependencies. We maintain an uncertain context to cover tasks that inherited lower priorities from semantically ambiguous dependency events. They are properly prioritized by the scheduler to minimize priority inversion. We further recognize the importance of early context binding—propagating
higher priority context at the earliest moment such that the successor, if it is currently occupied with low priority work, can be promoted in time to avoid priority inversion.

We take advantage of the interactivity awareness to achieve background workload consolidation and differential per-core frequency control to improve energy efficiency. We assembled a suite of representative foreground and background mobile applications and performed experiments on a smartphone capable of per-core frequency and voltage adjustment. Comparing to the default system, our interactivity-aware system can achieve 13% on average (up to 23%) energy saving. Meanwhile, it incurs only 1% on average (6% in the worst case) performance slowdown.
5 Supercapacitor Energy Buffering for Self-Sustainable, Data-Intensive Systems

5.1 Introduction

Continuous sensing (particularly of high-data-rate visual information [4]) in the field enables a broad range of emerging applications. For example, visual images and videos captured on a street or highway [101] may enable the analysis of traffic conditions, and consequently improve transportation safety and efficiency. In the domain of wildlife monitoring [72, 102], sensors continuously capture large volumes of data in the physical environment that is used to identify moving objects and activity patterns. Intelligent feature recognition and filtering at the data sources allow systems to operate at high data processing capacities without requiring high-speed network connections. Local processing also permits faster response to dynamic, emergent events.

Self-sustainability is critical for successful data sensing and processing in the field. Wire-free field systems are easy to install and maintain. A self-sustainable field node must live on ambient energy sources (e.g., solar cells) and energy buffering to power the system when ambient energy is unavailable (e.g., no solar power at night). Conventional rechargeable batteries have been blamed for negative environmental impact from their chemical compounds, limited lifetime (≈10³ charge cycles) and poor thermal stability (typical temperature limits for lithium-ion battery is between 0 to 45°C).

As an energy buffering alternative, supercapacitors have practically infinite lifetime (≈10⁶ charge cycles) and much wider operating temperature range (-40 to 65°C) [62]. Supercapacitors
are also free of acids and other corrosive chemicals. These qualities imply low maintenance cost, high reliability and less environmental impact. All are compelling advantages for systems deployed in the field.

Supercapacitors have traditionally lagged far behind on energy storage density and therefore have only been used in ultra-low-power sensor motes \cite{49,81,100,115} that run limited-function custom software and low-intensity operations. However, the past decade has witnessed major progress in supercapacitor manufacturing technology, translating to substantial energy density improvements. Today, a commodity 3,000 Farad supercapacitor \cite{63} at its rated voltage of 2.7 Volts stores about 11 KJoules of energy. Multiple 3000 Farad supercapacitors can store sufficient energy to sustain 12-hour continuous data-intensive operations at 1 Watt. In the mean time, mobile processors have advanced to levels that permit computation intensive operations at low power consumption. These two simultaneous progresses have reached an inflection point such that it is now possible for a field system to use supercapacitors as its sole energy buffer and perform high-data-rate sensing and analysis.

This brings new challenges as well as opportunities. The relatively small energy buffering capacity of supercapacitor calls for careful energy management to support system sustainability. It requires precise supercapacitor energy modeling and system adaptation. Fortunately, the stored energy in a capacitor can be theoretically modeled as \( E = \frac{1}{2} CV^2 \), where \( C \) and \( V \) are the capacitance and terminal voltage respectively. This model facilitates precise energy budgeting and enables a field-deployed system to operate intelligently at optimal stable quality-of-service while maintaining energy sustainability. In contrast, precise energy budgeting is much more challenging for rechargeable batteries \cite{19,110} due to their complex energy storage mechanisms and relatively fast aging.

This chapter presents our design and construction of a software/hardware platform (pictured in Figure 5.1) for continuous data collection and processing on supercapacitor-sustained systems. We shed new insights into the self-discharge (or leakage) and voltage-dependent effective capacitance issues in supercapacitor modeling. We further recognize the importance of power proportionality to self-sustainable, continuous sensing systems and exploit features in low-power computer systems and typical sensing applications to improve it. Our working pro-
Figure 5.1: Picture of our deployed system on a seven-story building rooftop (center picture) that includes a camera, a block of solar panels, and a system box. The system box (left-side picture) contains a Nexus 7 tablet computer (without its internal rechargeable battery) sustained by eight Maxwell 3000 Farad supercapacitors (wrapped in black tapes), and a controller board (at the top of the box). The energy buffering capacity of the supercapacitor block is about $1.4 \times$ that in the original Nexus 7 battery.

totype has been successfully deployed at a campus building rooftop where it analyzes nearby traffic patterns continuously.

### 5.2 Prototype System Construction

We design and construct a supercapacitor-sustained system platform for continuous sensing. Figure 5.2 illustrates our system architecture. Key components of our system are a Nexus 7 tablet computer running Ubuntu Linux, a supercapacitor block for energy buffering, and a controller board to regulate power and supply critical power/energy statistics for software system management. The controller board sends the supercapacitor voltage, solar power input, and Nexus power consumption statistics to the Nexus tablet through a Bluetooth connection once every 10 minutes. The Bluetooth communication consumes about 10 mW.
Figure 5.2: System architecture including the supercapacitor block for energy buffering, Nexus 7 for energy consumption, and a 70mW controller board that regulates between solar power input and energy buffering / consumption.

### 5.2.1 Computing Platform

High-speed processing of high-frame-rate images requires substantial computational capability that is not available on simple mote-based wireless sensor nodes. Data processing applications (e.g., OpenCV [73]) and associated tools also desire a conventional software environment such as Linux. We selected the Nexus 7 tablet computer containing a Tegra3 quad-core mobile processor [71] in our prototype system. We disabled the display and other unnecessary components to minimize its power consumption. Tegra3 allows multiple CPU speed options from a single active core at 102 MHz to four active cores at 1.2 GHz with a wide dynamic power range. Total idle system power is 0.54 Watts using a single active core at 102 MHz. With four active cores at 1.2 GHz, 35 frames/sec operation of ZoneMinder, the system consumes 3.35 Watts and a CPU-intensive micro-benchmark consumes 5.13 Watts.

We support field operations for the Nexus 7 on solar energy harvesting. Some applications only work during the day when there is sufficient light for camera operations. Others operate continuously through the night when collecting data over night is possible. We target the support
of at least 14 hours of system operation on buffered energy (with no power harvesting). This can be a dark night or an overly cloudy day.

5.2.2 Supercapacitor Provisioning

We explain four critical issues in supercapacitor provisioning.

Energy buffered in the supercapacitor block ($E_{SC}$): A supercapacitor can store energy in the amount of $E = \frac{1}{2} C \times V^2$, where $C$ is the capacitance of the supercapacitor, and $V$ is the voltage at its terminals. We choose a block of eight Maxwell BOOSTCAP BCAP3000E [63] supercapacitors in our system, with a rated voltage of 2.7 Volts, and a rated capacitance of 3,000 Farads for each. Therefore, this block of eight BCAP3000E supercapacitors can store a total energy in the amount of

$$E_{SC} = 8 \times \frac{1}{2} \times 3000 \times (2.7)^2 = 87,480 \text{ Joules} \quad (5.1)$$

This block occupies a volume of $8" \times 7" \times 7"$ and weighs roughly 11 lbs.

Serial vs. parallel multi-supercapacitor layout: Multiple supercapacitors can be connected in different topologies [52] as depicted in [5.4]. A serial setup connects multiple supercapacitors in a way resembling batteries in a flashlight, yielding a high terminal voltage—$8 \times 2.7 = 21.6V$ for eight fully-charged supercapacitors. In a parallel setup where the terminals of all supercapacitors are connected together, the terminal voltage is only 2.7 V at full charge.
We choose the serial topology due to more efficient operation at higher voltage and lower currents. Higher operating currents are undesirable since they cause higher energy losses on the copper wires connecting the circuit ($P_{\text{loss}} = I^2R$ denotes the power consumption on a copper wire with a resistance of $R$).

![Figure 5.4: Different topologies to build a block of multiple supercapacitors: a) fully-serial, b) fully-parallel.](image)

However, the serial setup risks imbalanced voltages between supercapacitors due to the manufacturing tolerances in supercapacitor capacitance. When serially connected, voltages of smaller capacitors rise faster, thereby reaching and exceeding the rated 2.7 V while voltages of other capacitors are still below 2.7 V. To prevent overcharging of any individual capacitor, a passive voltage limiter, such as a precision Zener diode, can be connected to each supercapacitor [52].

**Maximum supercapacitor block voltage** ($V_{SC_{max}}$): Rated terminal voltage of 2.7 V should not be exceeded for each BCAP3000E supercapacitor. If any supercapacitor exceeds 2.7 V, this overcharged region would cause extreme aging in the supercapacitor and overcharging above 2.85 V for more than a minute could result in safety concerns. Therefore, the maximum allowed voltage for eight serially connected supercapacitors is $V_{SC_{max}} = 2.7 \times 8 = 21.6$ V.
Minimum supercapacitor block voltage ($V_{SC_{min}}$): The Nexus 7 system requires a constant 5 V supply. We limit the minimum supercapacitor voltage to 7 V through our software control but allow a safety margin down to 3.5 V.

5.2.3 Custom Controller Board

As depicted in Figure 5.2, we have used a custom controller board which buffers solar energy on supercapacitors, delivers power to the tablet, measures solar power and supercapacitor voltage and communicates the data to the tablet. The custom controller board is built upon the design introduced in [43], consumes $\approx 70$ mW and has the following features:

Energy Harvester: We use a standard switching DC-DC converter design called SEPIC (Single-ended primary-inductor converter) [97]. It is able to harvest solar energy in the voltage range of 7-20 V and store the energy into supercapacitor block in the voltage range of 3.5-21.6 V.

PIC $\mu$Controller: We use PIC16F1783 $\mu$Controller to control the duty cycle of the harvester based on the well-known maximum power point tracking (MPPT) technique [26] to extract the maximum amount of energy from the solar panels. It also sends the supercapacitor voltage and other power/energy related information to the Nexus7 tablet.

Bluetooth Module: Our custom controller board communicates with the Nexus 7 through the Bluetooth connection. The average power consumption of the Bluetooth unit is 10 mW.

Voltage Regulator: We use a SEPIC DC-DC converter controlled by LTC1624 switching controller [59] to provide a 5 V regulated voltage to the Nexus 7 tablet. The regulator is able to achieve high efficiency with low quiescent power consumption.

5.3 Supercapacitor Energy Modeling

The limited energy storage on the supercapacitors requires careful energy budgeting and utilization. Motivated by the simple voltage-to-energy relationship of $E(V) = \frac{1}{2}CV^2$, we enable precise energy buffer modeling and time-to-depletion prediction on supercapacitor-supported systems. Specifically, assuming an incoming power supply of $P_{\text{supply}}$, a portion of which is
consumed by the system without being buffered by the supercapacitors (denoted as $P_{\text{load}}$), we can calculate the time it takes to charge or discharge a supercapacitor block—

$$t = \frac{E(V_{\text{start}}) - E(V_{\text{target}})}{P_{\text{load}} - P_{\text{supply}}} \quad (5.2)$$

where $V_{\text{start}}$ and $V_{\text{target}}$ are the terminal voltage values of the entire supercapacitor block at the beginning and the targeted times, respectively. We call this the time prediction model.

We can also use the model to identify the proper power load such that the supercapacitor block can reach the target voltage in a specific time $t$—

$$P_{\text{load}} = P_{\text{supply}} + \frac{E(V_{\text{start}}) - E(V_{\text{target}})}{t} \quad (5.3)$$

We call this the power budget model. The accuracy of these supercapacitor models is critically affected by the following two issues.

### 5.3.1 Leakage Modeling

A supercapacitor is made out of activated carbon particles immersed in an ionic solution (electrolyte). When the terminal voltage of the supercapacitor increases, due to the impurities in the carbon-electrolyte interface, undesired redox operations take place in the electrolyte, causing the charge to decrease. This phenomenon manifests itself as decreased voltage at the supercapacitor terminals even without external load (i.e., self-discharge, or leakage) [82]. The leakage affects the supercapacitor energy modeling because the energy is consumed by the system load as well as the leakage.

We measured the leakage of our supercapacitors at different voltage levels. The leakage can be computed by subtracting the power load from the loss rate of the stored supercapacitor energy. To accelerate this measurement process, we applied a small power load of about 0.1 W during the measurement. The experiment lasted for more than 10 days to cover the full range (22.8 V to 7 V) of operating and overcharged voltages.

Figure 5.5 illustrates the results of our leakage measurements. The leakage is significant (0.69 W) at the overcharged level of 22.8 V. Overcharged supercapacitors are unsafe to operate and we do not use them for remaining experiments in this work. Within normal operating
voltages, the leakage is about 0.07 W at the peak 21.6 V and decreases rapidly to negligible levels as the voltage approaches 19 V and lower. Without the additional 0.1 W load to accelerate the test, we observed another identical block of eight BCAP3000 supercapacitors to reach $\approx 16$ V after about a year. Therefore, we conclude that, leakage is only a minor concern at the high end of the rated voltage of the supercapacitors for a field system (e.g., 19 to 21.6 V for our 8-supercapacitor block).

![Figure 5.5: Self-discharge (leakage) power for our 8-supercapacitor set at operating and overcharged voltages.](image)

Using the profiled leakage power at each voltage level, we augment our supercapacitor energy models to include leakage effects. For the time prediction model that predicts the time to reach a target voltage under a certain power load, we compute it through a discrete-time simulation. Specifically, we assume that, under a normal power load, the leakage power $P_{leakage}$ during a short time duration $\Delta t$ (e.g., 1 second) is constant and can be looked up through our profiled leakage power at each voltage level. Let $V_t$ be the supercapacitor block voltage at time $t$, then the voltage at time $t + \Delta t$ can be computed by—

$$V_{t+\Delta t} = E^{-1}\left( E(V_t) - \Delta t(P_{load} + P_{leakage} - P_{supply}) \right)$$  \hspace{1cm} (5.4)
Our discrete-time simulation incrementally repeats this computation to predict the reachable voltage at an arbitrary future time, and can also predict the time to reach a target voltage.

Although profiling is needed for building an accurate leakage model, it is only necessary to profile each supercapacitor manufacturer model once. In various lab/outdoor experiments, we saw little variations when using different supercapacitor units of the same model.

Next we consider the leakage-aware power budget model that estimates the proper power load to reach the target voltage in a specific time. Built on the above time prediction model, we compute the power budget model using an approach similar to the Newton-Raphson method in numerical analysis. We start from two inaccurate bounding power load estimates—one too high (leading to shorter time to target voltage) and one too low (resulting in longer time to target voltage). We then take the middle of two bounding voltages and, depending on whether its time-to-target-voltage is too long or too short, decide on two new bounding voltages with a smaller gap. We repeat this process until the two bounding voltages are close enough to produce a low-error estimate.

5.3.2 Voltage-dependent Effective Capacitance

Our energy model is also affected by an important operational characteristic of supercapacitors: the energy storage capacity (i.e., the capacitance) increases at higher terminal voltages. According to the three-branch model [68][117], supercapacitor energy is conceptualized to reside in three branches or storage buckets. The first branch consists of a fixed capacitor ($C_0$) and a voltage-dependent capacitor ($C_{0\,\text{var}} = a \times V$), where the term $a$ signifies the degree of voltage dependence in terms of Farad/V. The first branch also has a very small equivalent-series resistor ($R_0$). The other two branches are represented by two additional series resistor-capacitor pairs (namely, $C_1$, $R_1$, $C_2$ and $R_2$).

Our detailed analysis and measurements show that, for the power levels (up to 5 W) and the 8×3000 F supercapacitor block, lumping every fixed capacitor into a single fixed capacitor ($C_0 + C_1 + C_2$) causes a negligible error, while the effect of the voltage-dependent capacitor must be handled as a separate phenomenon. In other words, for a supercapacitor operating at a
Figure 5.6: Effective capacitance at different terminal voltages. The capacitance is noticeably lower at low voltages.

While $C_0$ is the fixed storage capacity of the supercapacitor block, the parameter $a$ is a direct implication of the three-branch supercapacitor model and cannot be ignored without risking noticeable modeling errors. We call $C_{eff}$ the effective capacitance of the supercapacitor block.

To demonstrate how noticeable this effect is, we have performed constant-current discharge experiments on our supercapacitor block. The effective capacitance at each short interval (where the capacitance is assumed to be a constant) is calculated by $C_{eff} = \frac{\Delta V}{\Delta t}/I$. Results in Figure 5.6 show that, the nominal capacitance $C = 3000/8 = 375 \, F$ deviates substantially from the effective capacitance, especially at the critical, low-voltage range where the stored energy is near depletion. We also found that the effective capacitance for our supercapacitor block can be accurately modeled by $C = 310 + 3.8 \, V$, where $C_0 = 310 \, F$ and $a = 3.8 \, F/V$ in Equation 5.5.

The voltage-dependent effective capacitance requires changing the original supercapacitor energy model of $E(V) = \frac{1}{2}CV^2$ where the capacitance is assumed to be a constant. Specif-

\footnote{due to our serial supercapacitor layout as described in Sec. 5.2.2}
ically, we model the stored energy as the integral of power over time while the supercapacitor voltage decreases from \( V \) to zero:

\[
E(V) = \int P \, dt = \int v \cdot I \, dt = \int_0^V v \cdot C \, dv \\
= \int_0^V v \cdot (C_0 + av) \, dv
\]

\[
E(V) = \frac{1}{2} C_0 V^2 + \frac{a}{3} V^3
\]  

(5.6)

Using \( C_0 \) and \( a \) determined above, the total energy stored in our supercapacitor block can be computed from Equation \ref{5.7} as \( E(V) = 155 V^2 + 1.2667 V^3 \).

\section{5.4 System Energy Management}

Our supercapacitor energy buffering mechanism provides limited energy storage with precise budgeting. This enables efficient system management and adaptations under specific energy constraints. In this section, we demonstrate how to utilize this model to maximize the application quality-of-service while maintaining energy sustainability. We then point out the importance of power proportionality to continuous sensing systems and exploit features in low-power computer systems and sensing applications to improve it.

\subsection{5.4.1 Energy Sustainability and QoS Stability}

Our system targets continuous field operations. Given a limited energy budget, we must plan and adapt its operation to avoid depleting the buffered energy prematurely. It is also desirable to maintain high, stable quality-of-service in continuous data sensing and processing. In particular, it would be poor service if the system maintains a high-frame-rate operation over some periods of time but has to substantially degrade the frame rate at other time.

Our supercapacitor models presented in Sec. \ref{5.3} enable us to apply feed-forward control to determine an appropriate operational frame rate. Specifically, we set a target tuple of time (when the system can resume charging) and supercapacitor voltage (minimum voltage that can keep system running plus a small margin). Based on Equation \ref{5.3} the operating power load
(and consequently the operational sensing frame rate) can then be determined by the current supercapacitor voltage and the estimated solar input power.

Our system primarily relies on the model-driven feed-forward control but also includes feedback adjustments to prevent the buildup of modeling errors. We periodically monitor the supercapacitor voltage and adjust the system operating state accordingly. Note that both precise energy modeling (feed-forward control) and periodic adjustment (feedback control) are necessary to ensure high-quality, stable operations. Without precise energy modeling, periodic adjustment alone may still prevent the system from shutting down prematurely, but high energy modeling errors would require large QoS fluctuation for correction.

5.4.2 Model-Driven CPU Configuration Adaptation

At a given power budget, we need to identify the maximum quality-of-service (data capture frame rate) in operational management. CPU is the dominant factor in determining the system performance and power consumption. Modern CPUs offer a range of configurations with different performance-power trade-offs: while a lower CPU configuration (fewer cores at a lower frequency) is generally more energy-efficient—consuming less power for the same rate of work, it suffers from a lower maximum data frame rate—possibly unable to fully utilize the power budget. An intelligent system must consider multiple possible CPU configurations to select the one producing the highest quality-of-service at a given power budget.

We build a power consumption model, characterizing the relationship between data frame rate and system power consumption, under each candidate CPU configuration. Intuitively, the data rate is proportional to the resource utilization (CPU and memory activities) which is then linearly correlated with the system power consumption [8]—

\[
P = P_{idle} + c_{capture} \times r_{capture} + c_{process} \times r_{process}
\]

where \( P_{idle} \) is the idle power, \( r_{capture} \) and \( r_{process} \) are data rates (frames per second) for data capture and processing respectively, \( c_{capture} \) and \( c_{process} \) are coefficients in the linear model.

Work by McCullough et al. [64] pointed out that dynamic resource contention on multicore processors may result in inaccuracies of the linear power model. Such resource contention has
not manifested in a significant way on our system for two reasons. First, continuous data sensing and processing applications exhibit stable resource usage over time. Second, their data stream processing only reuses data with small distances (processing current or nearby frames) and therefore make more uses of the smaller, core-private caches than the larger, shared last-level cache. In a specific experiment, we run our traffic trajectory analysis application (described earlier in Section 2.2.2) on the Tegra3 quad-core processor at 1.0 GHz CPU frequency with varying data rates from 1 fps to the peak of 19 fps. The processor performance counters report very stable Instructions-per-Cycle values (mean 0.510; standard deviation 0.013, or 2.6% of the mean) which suggests little variation of resource contention.

The Tegra3 chip in our Nexus 7 system allows a range of CPU configurations with different CPU core counts and core frequencies. We select four representative configurations in our study: I) 4 active CPU cores at 1.0 GHz, II) 4 active CPU cores at 640 MHz, III) 1 active CPU core at 475 MHz, and IV) 1 active CPU core at 102 MHz. We have collected data on the power consumption under different data capture and processing rates and calibrated the power model for each of our four chosen CPU configurations. The calibrated model parameters for our traffic trajectory analysis application are listed in Table 5.1. Figure 5.7 shows our measurement and modeled power consumption for data capture+processing experiments.

To meet a power budget $P_{load}$, our model-driven CPU configuration adaptation works as follows. Under a specific CPU configuration, the power model in Equation 5.8 allows us to compute the application frame rate to meet the target power load—

$$r_{target} = \frac{P_{load} - P_{idle}}{c_{capture} + c_{processing}}$$  \hspace{1cm} (5.9)
Figure 5.7: Data capture+processing power consumption at a range of CPU configurations and application frame rates. For each CPU configuration, the experiments end at the maximum achievable frame rate. We also show a linear fitting of the frame rate to power each CPU configuration.

We perform such frame rate computation for each candidate CPU configuration and then choose the one supporting the highest data frame rate.

5.4.3 Delayed Bursts for Power Proportionality

Beyond the model-driven control, we explore techniques to improve the system energy efficiency and adaptability. In particular, earlier work has argued for power proportionality—the system power consumption changes proportionally on the application load or QoS—for data centers [7] and smartphones [13]. In this work, we recognize the importance and challenge of realizing power proportionality in a continuous sensing system.

First, power proportionality is important for energy efficiency. Modern hardware is most efficient when running at peak load level or completely idling. However, continuous sensing systems operate continuously with little to no idle time, making it difficult to utilize the low power idle states without incurring significant performance penalty. And due to the often-tight energy budget, self-sustainable systems rarely venture to peak load levels. Consequently, a poor power-proportional system would spend most time operating in the worst efficiency region of the hardware.
Second, power proportionality is crucial for adaptive system energy management. It indicates the amount of the leverage a system has in controlling its power consumption through load adjustment. Self-sustainable systems often face power budget variations due to the volatility of ambient energy sources. A poor power-proportional system with narrow dynamic power range would have to drastically adjust its load to compensate for these variations, resulting in large QoS fluctuation.

We propose delayed burst processing to improve the power proportionality of continuous sensing systems. The CPU configuration adaptation presented in Sec. 5.4.2 assumes a constant rate of data capture and processing. While a constant data capture rate is necessary for the continuous sensing purpose, the processing of captured data can be delayed. In particular, we let application alternate between two phases— a low-power capture-only phase that stores the captured data without processing, and a burst processing phase that processes all previously captured data at a high CPU configuration. Such heterogeneous, two-phase computing has a wider dynamic power range and exhibits stronger power proportionality than the static configuration.

Under a pair of CPU configurations (a low configuration for the capture-only phase and a burst configuration for the processing phase), we compute the application frame rate $r_{\text{target}}$ to meet the target power load $P_{\text{load}}$. Another important variable in this computation is the time ratio of the two phases through the execution. We use $(1 - \delta) : \delta$ to represent the ratio between the capture-only time and burst processing time.

Let the power model parameters under the low and burst configurations be $P_{\text{idle}}^{\text{low}} / c_{\text{capture}}^{\text{low}} / c_{\text{processing}}^{\text{low}}$ and $P_{\text{idle}}^{\text{burst}} / c_{\text{capture}}^{\text{burst}} / c_{\text{processing}}^{\text{burst}}$ respectively. Further let the burst processing rate be $r_{\text{burst}}$. We can then compute the target frame rate $r_{\text{target}}$ and burst processing time ratio $\delta$ by solving the following two-variable quadratic equations—

$$
\begin{align*}
&\begin{cases}
    r_{\text{target}} = \delta \times r_{\text{burst}} \\
    P_{\text{load}} = (1 - \delta) \times (P_{\text{idle}}^{\text{low}} + c_{\text{capture}}^{\text{low}} \times r_{\text{target}}) + \\
    \delta \times (P_{\text{idle}}^{\text{burst}} + c_{\text{capture}}^{\text{burst}} \times r_{\text{target}} + c_{\text{processing}}^{\text{burst}} \times r_{\text{burst}})
\end{cases}
\end{align*}
$$

(5.10)
We perform such target frame rate computation for each candidate pair of low/burst CPU configurations and then choose the pair supporting the highest data frame rate.

Our delayed burst processing bears resemblance to computational sprinting [78] in terms of an elevated CPU configuration for bursty work. However, our different objectives (long-term energy efficiency for delayed burst processing vs. short-term fast responses for computational sprinting) demand different policy decisions. In particular, since the CPU over-clocking hurts the long-term energy efficiency, burst processing at the highest CPU configuration is often the wrong choice for our approach. Instead, the optimal burst processing CPU configuration is typically the lowest configuration to fully utilize the power budget while the optimal capture-only CPU configuration is even lower to gain the maximum energy efficiency.

5.4.4 Duty-Cycle Management

CPU configuration adaptation is effective when the CPU power consumption dominates the overall system power. When this is not true (e.g., large static power of peripherals), we would need a more general approach to improve the system power proportionality.

We take advantage of the suspension feature of low-power devices which typically consume minimum power during suspension (60 mW for our Nexus 7 tablet). Therefore the overall energy consumption of a device operation is approximately proportional to the amount of its active (non-suspension) time. This motivates a simple duty-cycle management that adjusts the active/suspension ratio to control the device power consumption. Let the power under active execution and system suspension be $P_{active}$ and $P_{suspend}$ respectively. Let $\alpha$ be the active ratio. Then the average system power consumption is—

$$P_{load} = \alpha \times P_{active} + (1 - \alpha) \times P_{suspend}$$ (5.11)

While being simple and effective, the drawback of duty-cycle management is that the system would not be able to collect and process data during the suspension period.
5.5 Software Architecture and Implementation

In our software architecture, the operating system only provides the basic mechanisms for necessary information reporting and CPU configuration control (all exposed through the Linux \texttt{sysfs} interface). All energy modeling and system adaptation optimization work is performed by the application at the user space. The application-controlled approach eases the integration of application semantics with energy modeling and optimization.

We assume single-application operations in typical field system deployments and therefore OS-level protection and isolation is unnecessary.

Specifically, we install Ubuntu 13.04 on the Nexus 7 and modify the Linux kernel to allow application privilege of directly controlling the CPU configuration. A target application is linked with a monitoring/control thread. It monitors the supercapacitor terminal voltage and determines the appropriate power load according to the energy model and desired operational condition (e.g., reaching a target voltage at a specific time). The application profile is then utilized to determine the appropriate CPU configuration (Sec. 5.4.2) or duty-cycle policy (Sec. 5.4.4). Accordingly, the control thread exerts appropriate control—injecting delays into the worker threads or suspending/waking up the system.

5.6 Evaluation

We built a measurement platform to acquire important metrics for our evaluation. Specifically, we connected an Agilent 34410A Digital Multimeter as shown in Figure 5.8 to measure the supercapacitor voltage and current with high precision. The product of the voltage and current measurements yields our system power consumption. We logged the voltage and current readouts to an external machine once every 10 seconds.

Our field device is capable of coarse-grained, internal logging of application processing performance (frames per second rate) and controller-reported voltage/power statistics. This coarse-grained logging, at the frequency of once every 10 minutes, consumes very little power. Therefore, for the field system’s production run, we did not use the Agilent multimeter or the data logging machine to eliminate the need for additional power sources for these two devices.
Figure 5.8: Agilent 34410A measurement setup to record simultaneous current and voltage values from the supercapacitor block to determine its power consumption ($P_{SC}$).

Our evaluation targets two distinct goals of i) focusing on our core contribution of supercapacitor-sustained computing and ii) demonstrating the practical impact at the full-system scale. We take an incremental evaluation approach: Sec. 5.6.1 starts with an evaluation of supercapacitor energy modeling. Sec. 5.6.2 evaluates CPU configuration adaptation in a laboratory setup using only core system components (no solar panels or peripheral camera, using video as the input data source) under night-time conditions (eliminating the need for solar power modeling). Finally, Sec. 5.6.3 presents results from a real system deployment with a full set of peripherals in a 24-hour operational period. Applications and workloads were explained earlier in Sec. 2.2.2.

5.6.1 Supercapacitor Energy Modeling

We evaluate the accuracy of our supercapacitor energy models using the two energy model utilization scenarios described at the beginning of Sec. 5.3. First, the *time prediction model* predicts the time to reach a specific target voltage (e.g., effective depletion of usable energy) under a given power load. Second, the *power budget model* estimates the power budget that would utilize the stored energy in a certain amount of time. The power budget is then used to determine a proper application quality-of-service (QoS) in operations.
Evaluation of Time Prediction Model  We compare the accuracy of three supercapacitor energy models—

1. the *base* model in Equation 5.2 without considering supercapacitor leakage or voltage-dependent capacitance,

2. the *leakage-aware* supercapacitor energy model presented in Sec. 5.3.1

3. and the additional management of voltage-dependent *effective capacitance* presented in Sec. 5.3.2.

Figure 5.9 and 5.10 shows the accuracy of the three models in predicting the time to reach certain voltages under two load levels (left/right columns)—0.55 W Nexus 7 idle load and 3.23 W peak operational load for the ZoneMinder-based environmental camera traps. We also show results in two modeling durations (top/bottom rows)—from the full charge of 21.6 V and from 8 V. The shorter-duration modeling accuracy is particularly valuable for systems that make periodic model adjustments in production.

![Figure 5.9: Accuracy of different supercapacitor energy models in predicting the time to reach a target voltage while system is in idle state with one CPU core turned on at 102MHz and an average power consumption of 0.55 Watt. The left plot illustrates the result of modeling the operational time from the full charge of 21.6 V for the supercapacitors while the right plot shows the result of modeling from 8 V to 7 V.](image-url)
Figure 5.10: Accuracy of different supercapacitor energy models in predicting the time to reach a target voltage while the system is operating ZoneMinder-based environmental camera traps with four CPU cores running at 1.2GHz and an average power consumption of 3.23 Watt. The left plot illustrates the result of modeling the operational time from the full charge of 21.6 V for the supercapacitors while the right plot shows the result of modeling from 8 V to 7 V.

We find that the base and leakage-aware models produce almost identical results under all evaluation cases. This validates our leakage profiling result in Sec. 5.3.1 that the supercapacitor leakage is insignificant during its normal operating voltages (in fact almost completely absent at 19 V and lower voltages). It produces no observable effect for practical system energy management purposes.

On the other hand, there is a clear effect of modeling the voltage-dependent effective capacitance, particularly at low voltage levels when the system energy is near depletion. Specifically for modeling the idle load from 8 V to 7 V, the effective capacitance model exhibits 3.3% error in predicting the time to energy depletion, compared to 7.2% error under the leakage-aware model without considering the voltage-dependent effective capacitance. For the peak load operation, the effective capacitance model reduces the modeling error from 8.8% to 1.9%.

The interesting curvy voltage patterns in Figure 5.9 and 5.10 are due to the charge redistribution phenomenon at the supercapacitor block [117] which are observable when the power consumption levels exhibit some periodic fluctuation.
**Evaluation of Power Budget Model**  We assess the effectiveness of supercapacitor energy models in guiding application frame rates to utilize the stored energy by a certain amount of time. In this evaluation, we run the traffic trajectory analysis application to assure operation continuation during the night time while achieving the best application QoS. Our particular objective is that, starting a 14-hour dim period (part of the day without solar supply) with a fully-charged supercapacitor unit, we arrive at a target voltage by the end. We choose 7 V as our target voltage since, at 7 V, the supercapacitor can continue to support the system for another hour of processing after the dim period in case there is no solar input at all.

We compare two models: the base model and the effective capacitance model. For the base model, the above constraint yields an energy budget of 78,321 J which corresponds to an average power consumption of 1.554 W. For the effective capacitance model, the energy budget is 76,809 J and the average power consumption is 1.524 W. According to the static model driven CPU configuration described in Sec. 5.4.2, the configuration with 4 active CPU cores at 640 MHz yields the highest application frames rate, which is 6.9 fps for the base model and 6.4 fps for the effective capacitance model.

We consider two application operating approaches. The first approach uses the supercapacitor energy budget to determine the application frame rate at the beginning of the 14-hour dim period and maintains the frame rate consistently until the end. The second approach makes periodic (hourly in our experiments) assessment of supercapacitor voltage and buffered energy, and readjust the application frame rate accordingly. Such periodic model adjustments prevent error buildup and ensure on-time energy depletion, but they run the risk of producing unstable quality-of-service.

Figure 5.11 shows that the precise energy budgeting yields benefits in both application operating approaches. For operations with consistent QoS, the base supercapacitor model exhausts the energy 51 minutes earlier than planned while the effective capacitance model reduces the error to 22 minutes. For operations with hourly adjustments, while both models can support continuous operation for 14 hours as expected, they deliver very different quality-of-service. For the base model, its frame rate plunges in the last hour to 2.87—a 66% reduction comparing to the average. We attribute this to two reasons: 1) the voltage-dependent capacitance deviation is particularly pronounced at low voltages, as explained in Sec. 5.3.2 and 2) the cumulative ef-
ffects of erroneous energy budgeting and control have to be reversed in a short amount of time. In contrast, system under the effective capacitance model maintains very stable quality-of-service with the lowest frame rate at 6.25—91% of the average.

![Graphs showing voltage and frame rate over time]

Figure 5.11: Power budget evaluation—Effectiveness of supercapacitor energy models in guiding application operational frame rates to utilize the stored energy by a certain amount of time. The experiments were performed for the traffic trajectory analysis application. The left two plots illustrate the supercapacitor voltage and application frame rates under operations of consistent frame rates. The right two plots show the results when the system makes hourly assessment of remaining supercapacitor energy and adjustment of application frame rates.

### 5.6.2 CPU Configuration Adaptation

A precise energy budget, together with a parametrized power consumption model, allow intelligent selection of the CPU configuration and dynamic application adaptation to achieve higher quality-of-service. In our experimental setting, the system tries to sustain continuous operations during a 14-hour dim period while providing the best sustainable QoS.

We compared two different system/application management approaches presented in Sec. 5.4.2 and Sec. 5.4.3:

- The first approach used the model-driven CPU configuration to select the best static configuration that optimized for the QoS while ensuring continuous operation. Based on
Equation 5.9 with a 1.52 W target power, the configuration with 4 active CPU cores at 640 MHz yielded the highest application frame rate, which was 6.4 fps.

- The second approach employed the delayed burst processing to improve the system power proportionality. Using Equation 5.10 to compute the CPU configurations for the two phases of the delayed burst model (capture-only and burst processing), we derived 1 active CPU core at 475 MHz and 4 active CPU cores at 640 MHz respectively. We also computed that the ratio of (capture only)-to-burst time should be 0.15:0.85 ($\delta = 0.85$) and the application’s achieved QoS was 9.3 fps.

While both approaches can ensure continuous operations through the 14-hour dark period, they exhibit different power consumption and application frame rate patterns. Figure 5.12 illustrates these results. The pulse-shaped power consumption pattern in delayed bursts is due to the periodic switches between low-power capture-only and high-power burst-processing phases. The spikes are due to burst Bluetooth communication between the Nexus 7 tablet and the PIC controller. The result shows that the delayed burst optimization (achieving 7.8 fps and higher) produced 30% enhancement over the static configuration (achieving 6.2 fps and higher).

We attribute this QoS enhancement to the improved system power proportionality. Figure 5.13 further illustrates the difference under the two CPU configuration adaptation approaches in terms of power proportionality. As you can see, system under the delayed bursts approach exhibits a better power proportionality. The resulting benefits are twofold. For efficiency, the improvement on the frames/sec rate for the same power budget is at least 25% for power targets of 0.7–0.9 W, 1.4–1.5 W, and 2.2–2.5 W. More importantly, the system QoS would be more stable—the overall smoother slope of the delayed bursts approach means the same amount of power variation leads to less QoS fluctuation in the worst case.

Figure 5.14 further illustrates the CPU configurations under our delayed burst approach. We find that, for optimal energy efficiency, the chosen burst processing CPU configuration is not always the highest available configuration on the CPU. This is because at some power budget levels a lower-than-peak CPU frequency can more efficiently utilize the available power budget. On the other hand, the capture-only phase’s CPU configuration moves away from
1core@102MHz (lowest configuration) at the power budget of 1.0 W and higher. This is due to the limited data capture rate at the lowest CPU configuration.

5.6.3 Real System Deployment

We deployed our system on the rooftop of a seven-story building (Figure 5.1) at our university campus. Our deployed system monitors a street and parking lot in front of the building and analyzes traffic patterns continuously. The system has waterproof capabilities.
The goal of the system energy management is to: 1) stay above 8 V by 8 AM each morning when the energy harvesting resumes; and 2) provide the best stable QoS throughout the rest of the day. Accordingly, our power budget model calculates an appropriate power load based on the current supercapacitor voltage and the predicted solar power supply.

Compared to our laboratory experiments in the previous subsections, our real system deployment faces two additional challenges: First, we need a camera to capture live data. Un-
fortunately, commodity cameras (including the one we use) do not exhibit strong power proportionality features that CPUs possess [56]. Their significant static power (about 1 W in our camera) renders system QoS adaptation ineffective in adjusting power. Therefore we use duty-cycle based management (Sec. 5.4.4) in our deployment. During the active (non-suspension) period, we set the CPU configuration to 4 core/640 MHz and fix the application processing speed at 7 frames/second. The system then adjusts its active duration over 5 minute intervals to achieve the average target power load.

Second, our deployed system supports 24-hour operations and it uses solar panels to harvest power during the day. Therefore our power management requires a model of the solar power supply. The solar power supply prediction has been investigated in the literature [2,51]. While a precise night time energy model is crucial for system sustainability, daytime energy management does not have to be accurate (i.e., energy is not about to be drained). Our system predicts the solar power supply based on a simplified algorithm from [51]. Specifically, solar power in different time slots of the day is estimated based on a weighted average of observed solar power levels of the same slots in previous days.

Figure 5.15 shows the deployment result under various weather conditions. In all cases, we maintain a stable night-time QoS by precise supercapacitor budgeting and power management. The day-time QoS is less stable due to the imperfect solar supply model. Yet, our system shows a strong adaptability during the day. It maintains a high QoS in a sunny day, reduces the duty-cycle ratio under overcast weather and adapts accordingly under mixed weather condition. The sudden QoS shift in Figure 5.15 (C) is due to an unexpected burst of sunshine in the late afternoon after a gloomy day. As the buffered energy increases rapidly, so does the system duty-cycle ratio. Figure 5.16 further illustrates the changes on the supercapacitor voltage and the solar power over time on one of the sunny days. While there are frequent disruptions in solar power input (due to the occasional cloud), the application data processing frame is able to stay at the highest rate most of the time. This is due to the adaptive mechanism of our system—short term variation has very minimum impact on the system energy budget on the long run (i.e. for the rest of the day and night). Thus drastic application quality-of-service adjustments are avoided.
<table>
<thead>
<tr>
<th>Time of day</th>
<th>Supercap voltage</th>
<th>Duty cycle ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>8am</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12pm</td>
<td>4</td>
<td>0.5</td>
</tr>
<tr>
<td>4pm</td>
<td>8</td>
<td>1.0</td>
</tr>
<tr>
<td>8pm</td>
<td>12</td>
<td>0.5</td>
</tr>
<tr>
<td>12am</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>4am</td>
<td>20</td>
<td>0.5</td>
</tr>
<tr>
<td>8am</td>
<td>24</td>
<td>1.0</td>
</tr>
</tbody>
</table>

(A) Sunny weather

(B) Overcast weather

(C) Mixed weather

Figure 5.15: System duty-cycle ratio (indicating data capture quality-of-service) and supercapacitor voltage changes in 24-hour periods of our outdoor system deployment under sunny (A), overcast (B) and mixed (C) weather.
5.7 Related Work

Field sensor systems have been studied extensively for many years [24, 44, 58]. Energy self-sustainability is a critical issue for these systems. Two particular factors distinguish our work from prior efforts: First, existing systems typically support only small amount of data capture using temperature, motion, light, and humidity sensors among others, but fall short of incorporating high data rate sensors (e.g., high-speed, high resolution camera). High-data-rate sensing and processing in the field is important for emerging applications such as intelligent transportation [101] and wildlife camera traps [72, 102]. Second, our use of supercapacitors as the sole energy buffering mechanism provides critical benefits of high reliability, low maintenance cost and environmental friendliness.

Energy modeling of rechargeable battery-based systems have been investigated in the past. Zhang et al. [110] calibrated a battery discharge curve to produce the mapping between observed voltage and battery energy capacity. However, they cautioned that the mapping requires per-battery calibration and may lose accuracy due to battery aging. Dong and Zhong [19] leveraged the availability of current sensors on some battery interfaces and expanded the modeling rate through software-level event-driven power models. In comparison, the state of a supercapacitor
can be more easily modeled from its observed terminal voltage, which enables precise energy management and optimization at the system level.

Although supercapacitors have been widely used in industry (e.g., automotive [50,103], elevators [84]), their uses in computer systems have been limited so far. Wang et al. [99] evaluated the effectiveness of supercapacitors in data centers as secondary, short-duration energy buffers when the power load drops temporarily. Work by Jiang et al. [49], Dutta et al. (Trio) [21], Zhu et al. [115], and Renner et al. [81] utilized supercapacitors to support low-power sensor motes (typically 10s of milliWatts or less power). We also recognize that hybrid energy storage (containing both batteries and supercapacitors) offers stronger energy density and is particularly attractive for high-energy-use systems like electric vehicles [103]. Our work shows that supercapacitors alone are able to sustain off-the-shelf tablet computers and high-data-rate sensing applications for practical field operations. This chapter presents a hardware prototype and software support to enable such systems.

The modeling and management of supercapacitors have been addressed in the past. Zhu et al. [115] and Renner et al. [81] proposed predictive energy management with supercapacitors for sensor motes. Weddell et al. [100] explained that supercapacitor self-discharge (leakage) behavior is dependent on its internal state. In the new context of supporting continuous, high-data-rate sensing and processing in the field, our work studies the practical impact of supercapacitor leakage and effective capacitance issues, as well as the interplay between precise energy modeling and the energy-constrained adaptation of continuous data processing systems.

There is substantial prior work on managing low-power or energy-constrained systems. TinyOS [44] is an operating system construction that was tuned for very small system images on low-power sensor motes. Pathak et al. [74] studied energy profiling and modeling of smartphones. K2 [57] enables coherence domains in low-power, heterogeneous platforms. Challen and Hempstead [13] argued that heterogeneous, low-power systems can achieve power proportionality by agile adjustments of hardware components. Min et al. [66] devised a prediction framework that estimates the power impact of continuous sensing applications at installation time. The unique contribution of this work is that we make the first demonstration that self-sustainable systems for continuous, high-data-rate sensing can rely solely on supercapacitor energy buffering.
5.8 Conclusion

This chapter demonstrates the feasibility of using supercapacitors as the sole energy buffer in self-sustainable, high-data-rate field sensing systems. We constructed a prototype system with a block of eight 3000 F supercapacitors. This energy buffering mechanism, with typical solar power input, can continuously support a Nexus 7 based tablet computer running high-speed, high-resolution camera-driven data collection and processing.

Although leakage (self-discharge) is generally a concern for supercapacitor-based systems, our measurements show that this is a minor problem (0.07 W leakage when fully charged and quickly approaching zero at lower voltages). Such leakage power is not significant for high-data-rate sensing and processing systems. Our result is in direct contrast with low-power sensor systems [115] that reported much larger impact of supercapacitor leakage.

On the other hand, the simple supercapacitor energy model of $E(V) = \frac{1}{2} CV^2$ incorrectly assumes that the capacitance is a constant. In practice, the capacitance decreases gradually as the supercapacitor voltage decreases, necessitating a corrected energy model of $E(V) = \frac{1}{2}C_0V^2 + \frac{a}{3}V^3$. The corrected model implies an energy over-estimation using the simple model as the stored energy is near depletion (i.e., at low voltages). We addressed the voltage-dependent effective capacitance using a discrete-time-simulation-based supercapacitor energy model. With this corrected model, we observed a reduction in the time-to-depletion prediction error from 7–9% to 2–3%.

We also pointed out the particular importance of power proportionality to self-sustainable, continuous-sensing systems. Such systems rarely enter idle states or peak load levels where most modern computer systems are optimized for. Thus poor power proportionality would lead to low energy efficiency. In addition, power proportionality indicates the amount of the leverage a system has in controlling its power consumption through load adjustment. Therefore, a poor power-proportional system is less adaptive and susceptible to QoS fluctuation. We improved the power proportionality of our prototype system by exploiting features in dynamic power systems such as Tegra3 in the Nexus 7 and typical work patterns of continuous sensing applications. Specifically, we determined that, while the sensing capture operation has to continue at a steady rate, the processing of the frames can be delayed and computed in burst mode, and
therefore widen the dynamic power range of the system, resulting in improved power proportionality. This approach, we call *delayed bursts*, enhanced the application frames-per-second rate substantially compared to the optimal static CPU configuration (by over 25\% at a wide range of power budgets).
6 Conclusions and Future Directions

6.1 Conclusion

As computing systems going off-grid, energy is one of the primary bottleneck resources that limit the system performance and usability, necessitating careful management and optimization. Yet, new development in hardware, application and energy buffering mechanism have rendered the existing techniques inadequate. In this dissertation, we present several system-level approaches to capitalize this new technology trend and make new contributions to efficiently managing energy for resource constrained systems.

We recognize the unique energy-discounted computing opportunities for the new smartphones. The power disproportionality of multicore processors and limited parallelism of typical mobile applications make it possible to process background tasks in a more energy efficient way. We show that, for optimal co-run energy discount, the background task processing must not elevate the overall system power state —specifically, no reduction of the multicore CPU idle state, no increase of the core frequency, and no impact on the system suspension period. We realize this through unconventional non-work-conserving scheduling. In addition, we use available performance counters to identify co-run resource contention on the multicore processor and throttle background task when it interferes with interactivity.

By taking a user-centric approach, we develop an application-transparent execution context for mobile operating systems that reflects the criticality of current execution on user interactivity. We track various system-level events that signal the initiation and propagation of
interactivity-related executions. This interactive context enables new optimizations in CPU scheduling and power state management. Our system consciously recognizes and tolerates the inherit OS-level tracking inaccuracies. Dependency events are treated differently in propagating execution contexts based on their trustworthiness in reflecting the innate application semantics. Uncertainties in execution context propagation are carefully preserved and exposed to the resource scheduler to minimize priority inversion. We further recognize the importance of prompt propagation of the interactivity context at the earliest possible moment to facilitate immediate prioritization of interactive execution. Our system utilizes the interactivity context to enable background workload consolidation and differential per-core frequency control to achieve high energy efficiency without impacting user experience.

Advancements in the supercapacitor energy density and low-power processors have reached an inflection point, where a data-intensive field-deployed system can rely solely on supercapacitors for energy buffering. This brings critical benefits of high reliability, low maintenance cost and environmental friendliness. We address the challenges of maintaining quality-of-service on a limited energy buffer and demonstrate the first working prototype of such a system. We leverage the voltage-to-stored-energy relationship in capacitors to enable precise energy buffer modeling. To achieve high precision, we find that it is necessary to account for the variation of effective capacitance, particularly lower capacitance at lower voltages nearing energy depletion. Modern mobile processors operate most efficiently at very high load (when most cycles are effectively utilized) or very low load (when fewer cores are active at a lower frequency). We propose delayed bursts—continuous low-power data capture and bursts of data processing at a higher CPU configuration—to improve the power proportionality and realize high quality-of-service at varying energy budget.

6.2 Future Directions

As the battery technology struggles to make breakthrough in energy buffering density, energy will continue to be a critical bottleneck resource for off-the-grid computer systems in the foreseeable future. Thus energy management and optimization will continue to play an important
role and require continuous innovation. We list some potential future directions that are related to the topics in this dissertation.

We are already witnessing new trends in processor development. Embedded CPUs are increasingly heterogeneous. The improved agility in power adjustment is a big blessing for energy efficiency. Yet, the corresponding software support is far from adequate. Understanding the quality-of-service requirements of different workloads and mapping these workloads to the most appropriate hardware for efficient processing is a big challenge for the current operating system. Fine-grained system abstractions that can differentiate and encapsulate different types of application executions are very much needed. Other than the application-transparent OS-level techniques, developers with deep understanding of the application semantics and users with the knowledge of the specific usage scenarios should also get involved in the energy management. The OS should provide them with the right interfaces, for example, to give hints to the OS about application threads importance or expected quality-of-service of a background application.

In addition to heterogeneous processors, customized application-specific integrated circuits (ASIC) or co-processors are also emerging. ASICs provide superb performance and energy efficiency for specific types of workload. The key to wider adoption is to identify common application patterns and provide the right set of programming abstractions for the right tasks.

Most of the work in this dissertation centers around processor energy management and optimization. Non-CPU components typically does not posses sophisticated power states. Yet some of them are now consuming significant energy that simply cannot be ignored. Some of the techniques presented in this dissertation are equally applicable to them. For example, radio transmission hardware is known to have long tail states where the module would continue to consume nontrivial energy after each transmission. Similar to energy discounted computing, this would be a great opportunity to do energy discounted transmission—make best-effort transmissions for noncritical packages such as backup tasks. In addition, awareness of interactivity could help to distinguish critical transmissions from noncritical ones, improving energy efficiency with minimal user-perceived network slowdown.

For certain critical hardware components, it is worthwhile to develop more sophisticated energy-saving states. For example, camera sensors are critical components for sensing systems.
They usually draw static power with few control options. This makes self-sustained camera-based sensing systems inefficient and less adaptive to energy source variations. Making them more power-proportional would greatly enhance the efficiency and adaptability of the entire system.

Thanks to the rise of a wave of new applications including portable computer, drone and electric vehicle, energy buffering has become one of the research focal points. Various solutions have been proposed to greatly enhance the energy buffering density.

Lithium-air based battery can theoretically hold more than 40 times the charge as a lithium-ion battery at the same weight [55]. It draws in oxygen from the air, which causes a reaction in the lithium that discharges energy. Since the oxygen does not need to be stored within the battery, the weight of the battery is minimized, leading to greater energy density. However, the life cycle of lithium-air batteries need significant improvements for practical use.

Common supercapacitors contain two metal plates coated with a porous material known as activated carbon to increase the surface area for higher capacitance. Recent supercapacitor research replaces the activated carbon with graphene or carbon nanotubes, further increasing the relative surface area and energy density while reducing the overall weight.

These emerging energy buffering mechanisms can sustain high power platforms with more peripherals for longer period, enabling new usage scenarios. A hybrid approach can be taken to overcome some of the issues with the new technologies. For example, lithium-air battery can be used as a secondary energy reserve in the field system, minimizing its discharge cycle while preparing the system against extended inclement weather conditions.
7 Bibliography


[55] Lithiumair battery.  


[59] LTC1624CS8 High-Efficiency SO-8 N-Channel Switching Regulator Controller.  


[Online; accessed November 2017].


[67] Monsoon power meter. [https://www.msoon.com/LabEquipment/PowerMonitor/](https://www.msoon.com/LabEquipment/PowerMonitor/)


[73] OpenCV. [Online; accessed November 2017].


[76] Processes and application life cycle. [Online; accessed November 2017].

[77] The power of snapdragon processors: Asynchronous processing. [Online; accessed November 2017].


[92] Number of apps available in leading app stores as of March 2017. [Online; accessed November 2017].

[93] Number of smartphone users worldwide. [Online; accessed November 2017].

[94] Measure the world visually, through your users eyes. [Online; accessed November 2017].


[96] Report: Smartphone owners are using 9 apps per day, 30 per month. [Online; accessed November 2017].


[116] ZoneMinder. [Online; accessed November 2017].