Towards Energy-Efficient Mobile Sensing: Architectures and Frameworks for Heterogeneous Sensing and Computing

by

Songchun Fan

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Benjamin C. Lee, Supervisor

_____
Landon P. Cox

_____
Alvin R. Lebeck

_____
Bruce M. Maggs

Dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the Department of Computer Science
in the Graduate School of Duke University
2016

Abstract

Towards Energy-Efficient Mobile Sensing: Architectures and Frameworks for Heterogeneous Sensing and Computing

by

Songchun Fan

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Benjamin C. Lee, Supervisor

_____
Landon P. Cox

_____
Alvin R. Lebeck

_____
Bruce M. Maggs

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University
2016

Abstract

Modern sensing apps require continuous and intense computation on data streams. Unfortunately, mobile devices are failing to keep pace despite advances in hardware capability. In contrast to powerful system-on-chips that rapidly evolve, battery capacities merely grow. This hinders the potential of long-running, compute-intensive sensing services such as image/audio processing, motion tracking and health monitoring, especially on small, wearable devices.

In this thesis, we present three pieces of work that target at improving the energy efficiency for mobile sensing.

1. In the first work, we study heterogeneous mobile processors that dynamically switch between high-performance and low-power cores according to tasks' performance requirements. We benchmark interactive mobile workloads and quantify the energy improvement of different microarchitectures.

2. Realizing that today's users often carry more than one mobile devices, in the second work, we extend the resource boundary of individual devices by prototyping a distributed framework that coordinates multiple devices. When devices share common sensing goals, the framework schedules sensing and computing tasks according to devices' heterogeneity, improving the performance and latency for compute-intensive sensing apps.

3. In the third work, we study the power breakdown of motion sensing apps on wearable devices and show that traditional offloading schemes cannot mitigate sensing's high energy

costs. We design a framework that allows the phone to take over sensing and computation by predicting the wearable's sensory data, when motions of the two devices are highly correlated. This allows the wearable to offload without communicating raw sensing data, resulting in little performance loss but significant energy savings.

To my teachers.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to take this opportunity to express my gratitude to everyone who has helped me reach this point. First of all, I thank my advisor, Dr. Benjamin Lee. During these years we worked together, he has guided me through the mist of system research, with his broad knowledge from computer architecture, to statistics, to game theory, to various areas that I would have never experienced without him. Till today, I am still amazed by his devotion to work, his expertise in research, and most importantly, his inquisitiveness towards new research fields. Especially, I appreciate that Ben truly cares about his students. As an advisor, he concerns about my career development and spares no effort in helping me grow strengths in both system and theory. From him, I learned how to approach problems, to develop ideas and to be professional. I very much hope I can continue receiving his advise in the future.

I would also like to thank professors at Duke, especially Dr. Landon Cox, Dr. Alvin Lebeck and Dr. Bruce Maggs. It is my great honor to have them as my committee members. They were there for each of my milestones — RIP proposal, RIP final, prelim, and finally defense. Throughout my time in grad school, they and the classes they taught have given me countless inspirations in mobile systems, networking and energy efficiency. I've also benefited greatly from conversations with Dr. Daniel Sorin and Dr. Andrew Hilton in the weekly computer-architecture reading group.

I want to thank my undergraduate supervisor Dr. Guihai Chen, who lead me into system research, master supervisor Dr. Romit Roy Choudhury, who introduced me to mobile

computing, and Dr. Erran Li and Dr. Theodoros Salonidis who mentored me during my internships. I thank my co-authors Dr. Xuan Bao, Mahanth Gowda, Dr. Hyojeong Shin, Seyed Majid Zahedi and Qiuyun Wang — I learned so much from each of them.

Finally, I would like to thank all my friends who have accompanied me on this five-year long journey. Friends from the lab, from the compter science department, from Duke and from my family — the list of their names will be too long. But I know from the bottom of my heart that without their love and support I would not be able to come this far.

1

Introduction

The mobile ecosystem is blooming. First, hardware manufacturers such as Apple, Samsung and Qualcomm regularly release new generations of devices and system-on-chips. Then, thousands of software developers explore these devices to create services and entertainments. Meanwhile, the hardware manufactures improve the performance of popular services, such as graphics and video streaming, in the next generation of their hardware. This cycle of development stimulates the users to upgrade their devices, generating revenue in economics and advancements in technology.

The mobile workloads have diverse performance requirements. Some involve frequent user inputs and are computationally intensive, while others are background services that minimally uses system resources. Some involve short, bursty interactions, while some others require long-running, continuous sensing. Mobile hardware advances as mobile workloads evolve. For example, to enable computational intensive tasks, mobile devices are supplied with an increasing number of CPU cores, GPU, DSP and various accelerators. For background services and non-intensive tasks, mobile devices integrate asymmetric multi-processing (AMP), which achieves both performance and energy efficiency by dynamically switching between high-performance and low-power cores. For ubiquitous sensing, mobile

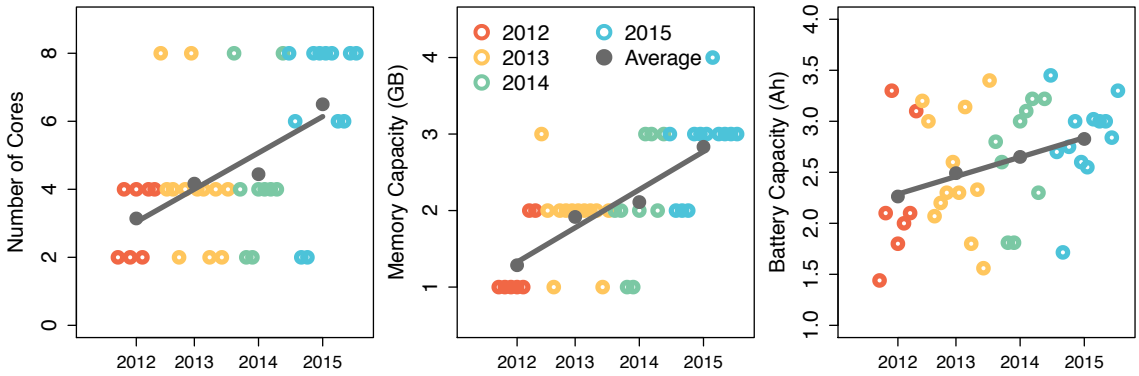

FIGURE 1.1: A survey of the number of cores, memory capacity and battery capacity in mobile phones released in recent years. From 2012 to 2015, the average number of cores in mobile devices increases by 107%, the memory capacity increases by 120%, while the battery capacity increases by only 25%.

devices install sensors such as accelerometer, gyroscope, compass, heart rate monitor and many more, giving rise to prevalent sensing and context learning algorithms.

Performance is not the only demand from users. An extended battery life is a prerequisite for the execution of the diverse workloads. Sadly, mobile devices' battery capacities do not grow as fast as their processing capabilities. In Figure 1.1, we survey 40 mobile phones released between 2012 and 2015 (the full list is in Appendix A). These devices have increasing numbers of cores, memory capacity in gigabytes and battery capacity in Amp-hour. From 2012 to 2015, the number of cores increased by 107%. This, together with the implementation of 14nm technology (was 32nm in 2015), the increasing clock rate, and the big.Little microarchitecture, demonstrates the large improvements in mobile processors. Similarly, the memory has doubled capacities and data rates (upgraded from LPDDR2 to LPDDR4). However, the battery capacity has only improved by 25% in three years. Consequently, many works from all system stacks have been done, in an attempt to achieve better performance for various workloads without consuming more energy.

In this thesis, we present three pieces of works that target at improving the energy efficiency of mobile devices. They investigate different aspects of mobile systems, from the heterogeneous architectures in mobile processors, to distributed networking frameworks on

mobile phones, to efficient sensing on wearable devices, and discover the sources of energy inefficiency. Although these problems are in different system domains, the key solution is always to manage hardware resources such that energy goes to services that use it most efficiently.

In the first work, we find that existing AMP architectures can exploit inter-application diversity but fail to exploit intra-application diversity—an opportunity for significant energy savings. Exploiting the latter requires emerging AMP architectures that share the cache hierarchy and reduce switch latency by orders of magnitude. To explore the AMP design space, we propose a set of realistic mobile benchmarks on Android that consider user actions and inputs. We simulate the benchmarks and show that mobile apps benefit substantially from responsive AMPs that switch quickly to exploit fine-grained microphases. We find that adaptive AMPs are responsive and use the little core for up to 47% of the instructions in mobile apps. Fine-grained switching produces 19-42% energy savings. The largest savings are seen for user actions, such as scrolling and reading, and for background services, such as a music service. (Chapter 2)

In the second work, we consider a particular class of apps, in which the computation is overly intensive for a single device, but multiple devies can collaborate toward aligned sensing and computing objectives. The sharing of the sensing and computing resources can largely improve the throughput and latency, while extending individual devices' battery lives. We present Swing, a framework that aggregates a swarm of mobile devices for the new generation of mobile sensing apps. We design and implement Swing to manage (i) parallelism in stream processing, (ii) dynamism from mobile users, and (iii) heterogeneity from the swarm devices. We build an Android-based prototype and deploy sensing apps – face recognition and language translation – on a wireless testbed. We show that Swing manages multiple devices to meet the real-time performance goals of modern sensing apps, with negligible overhead on each device. (Chapter 3)

Given that users often carry a wearable device and a phone simultaneously, in the third

work, we manage the ensemble of the two devices to improve the energy efficiency for continuous sensing apps on wearables. We design Telepath, a framework that supports communication-free offloading for wearable devices. With offline training, computational tasks can be offloaded from the wearable to the user's phone, without transferring raw sensing data. The key observation is that when the user is carrying both devices, the sensing streams on the two devices are highly correlated. By exploiting the correlation, the phone can estimate the watch's sensing data and emulate the watch. Our evaluations shows that with Telepath, the phone performs accurately on activity recognition tasks that are designed for watches, achieving on average 85% of the watch's accuracy and 2.5x improvement relative to using the phone's raw sensing data, while extending the watch's battery life by 2.1x. (Chapter 4)

In summary, this thesis presents the following contributions.

- Analysis on the development and trend of mobile processors, their microarchitectures and detailed power breakdown.
- Realistic mobile benchmarks that consider user interactions and background services.
- Analysis on continuous sensing apps and their power and computational bottlenecks.
- Prototyping of a distributed dataflow system that schedules multiple mobile devices to assist compute-intensive sensing services.
- Analysis on the energy constraints in wearable devices and power consumption of activity tracking apps.
- Prototyping a framework that minimizes sensing and computing using sensory offloading with a systematic signal processing approach.

2

Evaluating Asymmetric Multiprocessing for Mobile Applications

Today's mobile processors attempt to achieve two competing goals at the same time: high performance and low energy consumption. Traditional symmetric multiprocessing (SMP) might attain the first goal, but fails the second one – identical cores are not efficient for the diverse performance requirements of mobile applications. As the number of cores grows, new organizations of multiprocessing systems are needed.

Table 2.1 details mobile processor evolution in three major mobile system-on-chip families. Observe that in addition to increasing the number of cores and the clock frequency, the industry has taken two approaches to address SMP limitations. The first approach, proposed by Qualcomm, provides independent power rails to each of the four cores, so that each core can adapt its voltage and frequency according to its workload. The second approach, taken by Samsung (ARM Big.LITTLE) and Nvidia, uses additional low-power core(s) to handle less intensive tasks and allow the big cores to be turned off.

This second approach, namely, asymmetric multiprocessing (AMP), provides high performance with a "big" core and energy efficiency with a "little" one. The system

dynamically switches between big and little execution modes according to workload levels. However, AMP performance and efficiency greatly depends on the physical organization of its asymmetric functionalities. Today's designs loosely couple big and little cores via shared memory with private L2 caches. When switching, the L2 of the current core must first flush its dirty cache lines to DRAM. As a result, the responsiveness of switching is inherently limited by data movement and DRAM latencies. Although they are sufficiently responsive to coarse-grained system phases, they are incapable for exploring finer-grained software dynamism.

Mobile app diversity provides AMPs many opportunities to switch. The first type of diversity is *inter-application diversity*. Some apps, such as video and web browsers, are compute-intensive and requires big cores; some apps, such as weather and calendar, require minimal computation and can be performed on little cores. Loosely coupled big and little cores that share memory are sufficiently responsive to inter-application diversity.

The second type of diversity exists inside individual apps, *intra-application diversity*. A mobile app normally contains more than one "activity window," allowing users to perform distinct types of actions such as scrolling, reading and typing. Different actions correspond to different patterns of computation. We observe that user input often triggers a short burst of processor activity, executing a few billion instructions in a few seconds. Input-triggered computation exhibits irregular control flow for which speculative and out-of-order execution perform poorly. For this type of computation, big cores perform no better, and sometimes worse, than little cores. Therefore, intra-app diversity provides opportunities for tightly coupled AMPs that share a cache hierarchy and switch quickly between execution modes.

Designing new microarchitectures for mobile AMP requires a comprehensive analysis of mobile workloads. Existing mobile benchmarks contain inter-app diversity but neglect intra-app diversity that arise from user actions and inputs. In this chapter, we propose a set of mobile benchmarks that specifically include typical user activities and a systematic method to evaluate performance under different AMP designs. Specifically, we make three

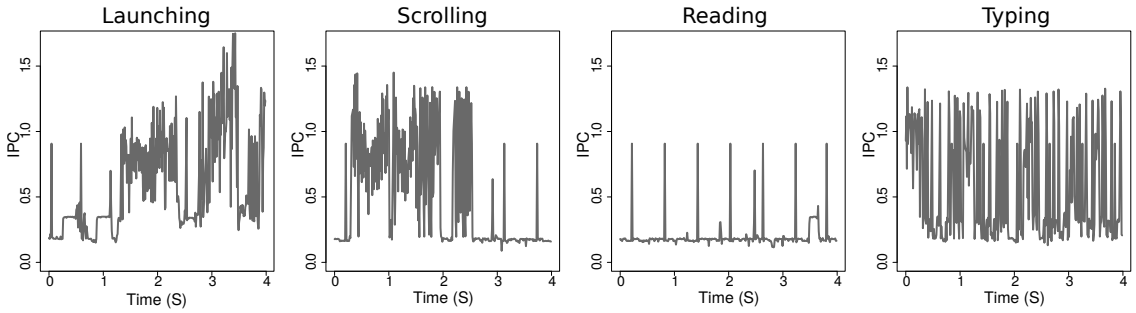

FIGURE 2.1: Types of actions in one Twitter session

contributions:

Realistic Mobile Apps. In this chapter, we design and implement a set of realistic mobile benchmarks that are composed of Android apps. They include typical user actions, such as launching apps, scrolling down lists, reading contents and typing words. They also include foreground and background apps, such as games and music players. We integrate our benchmark apps with gem5 Binkert et al. (2011), a cycle-accurate simulator. To assist future work on mobile architecture design, we open-source the Android code and the simulator's disk images, checkpoints and configurations Fan (2015).

Responsive Asymmetric Multiprocessing. With our mobile benchmarks, we evaluate three AMP designs: 1) loosely coupled big and little cores with shared memory, 2) tightly integrated physical cores with a shared last-level cache, and 3) a single physical core with an adaptive datapath and shared cache hierarchy.

Efficient Big-Little Computation. We find that adaptive AMPs are responsive and use the little core for up to 47% of the instructions in mobile apps. Fine-grained switching produces 19-42% energy savings. The largest savings are seen for user actions, such as scrolling and reading, and for background services, such as a music service.

Collectively, we study realistic benchmarks that reflect intra-application diversity to show that emerging AMP architectures can save substantial energy without harming the performance of mobile applications.

Table 2.1: Specs of Mobile Processors

SoC Model	CPU	Freq	L1 Cache	L2 Cache	Semi-conductor	Launch
Nvidia Tegra 2	Dual-Core ARM Cortex-A9	1.2GHz	32KB/32KB	1MB	40nm	2011
Nvidia Tegra 3	Quad-Core ARM Cortex-A9 (4-Plus-1)	1.6GHz	32KB/32KB	1MB	40nm	2012
Nvidia Tegra 4	Quad-Core ARM Cortex-A15 (4-Plus-1)	1.9GHz	32KB/32KB	2MB	28nm	2013
Nvidia Tegra K1	Quad-Core ARM Cortex-A15 (4-Plus-1)	2.3GHz	32KB/32KB	2MB	28nm	2014
Qualcomm Snapdragon S3	Dual-Core Scorpion	1.7GHz	32K/32KB	512KB	45nm	2010
Qualcomm Snapdragon S4	Quad-Core Krait	1.7GHz	4KB/4KB L0 + 16KB/16KB L1	2MB	28nm	2012
Qualcomm Snapdragon 800	Quad-Core Krait	2.26GHz	4KB/4KB L0 + 16KB/16KB L1	2MB	28nm	2013
Qualcomm Snapdragon 805	Quad-Core Krait	2.7GHz	4KB/4KB L0 + 16KB/16KB L1	2MB	28nm	2014
Samsung Exynos 4	Dual-Core ARM Cortex-A9	1.4GHz	32KB/32KB	1MB	45nm	2011
Samsung Exynos 4	Quad-Core ARM Cortex-A9	1.6GHz	32KB/32KB	2MB	32nm	2012
Samsung Exynos 5	Qual-Core ARM Cortex-A15 + Quad-Core ARM Cortex-A7	1.9GHz/ 1.3GHz	32KB/32KB	2MB/ 512KB	28nm	2013
Samsung Exynos 5	Qual-Core ARM Cortex-A15 + Quad-Core ARM Cortex-A7	2.1GHz/ 1.5GHz	32KB/32KB	2MB/ 512KB	28nm	2013

2.1 Mobile Benchmarking

Mobile apps are generally less computationally intensive; bursts of processor activity are often triggered by user inputs. Moreover, mobile apps often contain more than one activity window in order to provide multiple functionalities and let users perform distinct types of actions. Thus, an ideal benchmark set should capture not only the different performance requirements between apps, but also various user actions inside each app.

Existing mobile benchmarks neglect intra-app diversity, considering only end-to-end app execution. For example, BBench Gutierrez et al. (2011) provides a set of webpages, automatically loaded to test mobile web browsing behaviors. It also includes an interactive game which runs only on testbeds and requires manual inputs to execute, preventing the usage of cycle-accurate simulators.

MobileBench Pandiyan et al. (2013) includes photo viewing and video playback which are not interactive. Moby Huang et al. (2014) provides popular mobile apps such as email, word processing, maps and social network. Typical operations in such apps, such as loading webpages and opening files, were simulated and do not include user inputs (click, scroll, type) that interact with apps. In contrast, we create a set of benchmarks for mobile apps that

focus on typical user actions.

2.1.1 Application and User Actions

Consider the apps inside our phones. Some are simple (e.g., weather, photo viewer, or reader) and require only one type of user action, if any. Other apps, however, provide more functionality and allow users to perform multiple types of actions frequently. For example, users of a social app, such as Twitter, may frequently switch between multiple actions. These actions may include scrolling through a list of tweets, reading the details of a tweet, and typing a tweet. Reading does not require the mobile processor to do anything other than display content, but scrolling or launching new activities may trigger a sequence of computation. We expect different performance requirements for different actions.

To demonstrate this intuition, we conduct an empirical study. We use a hardware performance monitoring unit to record, in real time, the number of instructions committed per cycle (IPC) when a human user launches and uses Twitter (version 5.8.1 on Android JellyBean 4.3) on an ARM Versatile Express development board with one Cortex-A15 core activated ARM (2012). During a 60-second session, the user performs several actions – launching new activity windows to open tabs or settings, scrolling to see more tweets, reading, and typing. We record user inputs and corresponding timestamps by reading the kernel input driver file and tracing click events.

Figure 2.1 presents selected user actions and the corresponding IPC time series. IPC shows distinct patterns for each action. Launching an activity window introduces a burst with a maximum IPC of 1.7. Scrolling produces a smaller burst with a maximum IPC of 1.5. Reading exhibits no computation except for the activity due to background sync tasks, which produce an average IPC of 0.2. Typing causes sustained IPC fluctuation. Differentiating such actions is important. If we were to view the Twitter session as a monolithic benchmark, end-to-end measurement would lose the rich information contained in various user actions.

In our setting, the mobile device is running the full system, with regular background

tasks. For example, the IPC spikes when reading are caused by background network threads from the stock email client. This setting represents realistic mobile device usage and we will reproduce the same setting in our simulations.

2.1.2 Microbenchmarks

We create a set of microbenchmarks to represent typical user actions in a social app. In particular, the microbenchmarks correspond to the four actions in Figure 2.1.

- **Launching.** We create a set of activities and inject touch screen events that switch between them. A touch event is injected every 1 second to mimic a user launching different activity windows (e.g., clicking to view tweet details).
- **Scrolling.** We create a listview that automatically extends itself. Automatic swipes are injected to scroll down the list. A swipe is injected every 500ms to mimic a user quickly scrolling down and browsing a list of content.
- **Reading.** We display an activity with text and pictures (e.g., reading a tweet). No specific inputs are injected.
- **Typing.** We create a textview and inject keyboard events to type (e.g., writing a tweet). Keyboard events are injected at a frequency of two letters per second.

To implement these benchmarks, we create "activities" within Android, loading activity text and pictures locally. We inject touches, swipes, and keyboard events using Android Instrumentation, an API that allows app developers to test their activity windows with emulated user behavior. This method requires access to application source code; our benchmarks are open-sourced.

Other methods use Android MonkeyRunner or write I/O events to the Linux input driver file. However, these methods require a time-stamp for each injected event and precisely specifying the time-stamp to trigger the right event during cycle-accurate, microarchitectural

simulation is difficult. Alternatively, AutoGUI supports record-replay through VNC and may be useful once it becomes public Sunwoo et al. (2013).

2.1.3 Macrobenchmarks

In addition to microbenchmarks for input-triggered computation, we include two foreground tasks, a game and a mobile web browser. We benchmark a game in which the user controls a submarine by touching the screen. The game uses 2D graphics, which exercises the processor and neglects the GPU. We benchmark the web browser and use a script, BBench Gutierrez et al. (2011), that automatically loads local webpages.

Table 2.2: Choices of Benchmarks

Name	**Scope**	**IPC(B)**	**IPC(L)**	**Inputs**
Launching	MicroBenchmarks	1.01	0.80	Periodic
Scrolling		0.90	0.70	Periodic
Reading		0.84	0.70	None
Typing		1.05	0.76	Frequent
BBench	Web Browsing	0.98	0.76	Periodic
Music	Background Task	0.76	0.63	None
Submarine	Gaming	1.12	0.80	Periodic
SunSpider	Javascript	1.09	0.79	
Linpack	CPU	1.26	0.95	

To compare mobile workloads against compute-intensive ones, we further deploy 0xbench Chang (2013) for testing smartphone performance. This open-source benchmark suite includes a Java implementation of Linpack and a Javascript benchmark called SunSpider.

Table 2.2 lists our benchmark and application suite. We classify benchmarks by the frequency of user inputs. We deploy the benchmarks on the gem5 simulator – see Section 2.3.2 for detail. We provide deployment checkpoints, disk images, and methods in our open-source project Fan (2015).

2.2 Asymmetric Mobile Processors

Based on the level of sharing in the cache hierarchy, AMP organizations can be classified into three categories: shared memory, shared last-level cache, and shared first-level cache. In this section, we first explain existing (and emerging) AMP systems, and abstract them into our evaluation models by quantifying their switching costs.

2.2.1 Shared Memory

Both Samsung and Nvidia have adopted AMP in their SoCs. However, in addition to the number of cores and their specifications, these two processors have some major differences.

Differences. The key of AMP is the performance/power gap between the big core and the little core. To achieve this gap, Samsung and Nvidia chose different methods. In Nvidia Tegra, the four big cores and one little core are identical ARM Cortex-A9 CPUs, but the cores are fabricated in different process technologies. The big core uses a general process (G) and a little core uses a low power one (LP). G transistors are conventional transistors, switching quickly but leaking more static power. LP transistors leak less but switch slowly nvi (2011, 2013). On the other hand, Samsung Exynos uses different microarchitectures for big and little cores. The big ones are out-of-order ARM Cortex-A15 CPUs and the little ones are in-order ARM Cortex-A7 CPUs.

The two SoCs also differ in their switching management. Nvidia Tegra implements a hardware switch controller, and thus the switching between big and little is OS transparent. In contrast, Samsung Exynos rely on the Linux kernel to schedule transitions.

Similarities. Despite all the differences, both systems share some common characteristics. First, big cores are identical and share an L2 cache; little core(s) are identical and share another L2 cache. Therefore, both systems can be viewed as one big cluster with one little cluster, each with a private last-level cache (LLC). Second, either the big or little cluster is active at any given time. Cluster have separate power domains that allow big cores and their

caches to power down when small cores are active, and vice versa.

Based on these characteristics, we abstract these two AMP processors into a model with one out-of-order core and one in-order core, which have private LLCs and shared DRAM. A big/little switch transfers a thread from one core to another. Because the LLCs of the big and little clusters are not shared, in order to ensure coherence, the dirty cache lines of the currently active LLC are flushed to DRAM before switching. This delay dominates the switching latency.

Switching Cost. The switching delay can be computed by the size of dirty LLC lines divided by the DRAM bandwidth. That is, if the LLC capacity is 512KB and memory bandwidth is 12.8GB/s, assuming on average 25% of cache lines are dirty at any given time,[1] a 1GHz core must wait 10K cycles (= 25% $\times$ 1GHz $\times$ 0.5MB $/$12.8GBps) for the switching to complete. Although it optimistically assumes transfers at peak memory bandwidth, 10K cycles is a useful, order-of-magnitude estimate of switching latency for asymmetric cores that share main memory. In our evaluations later, we use this number to evaluate this AMP category.

Summary. Existing systems-on-chips have proved that AMP with shared memory is cost-efficient – tuning operating and fabrication parameters for existing core designs would be sufficient to build such processors. Switching between big and little is responsive to *coarse-grained system phases* (e.g., standby/active mode of the phone), but further flexibility is constrained by switching overheads.

2.2.2 Shared Last-Level Cache

Although current asymmetric mobile processors share memory, we envision next-generation AMP with shared last-level cache (LLC). Compared with shared memory, shared LLCs improve performance in several ways. First, the active core benefits from more cache lines since the aggregate size of the shared LLC is larger than that of a statically partitioned one.

[1] This number is measured by our mobile workloads

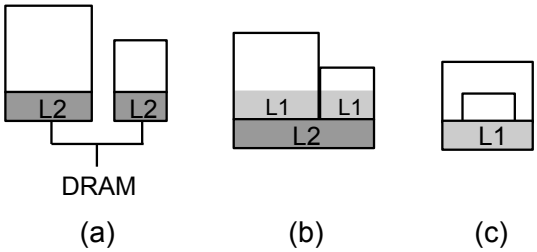

FIGURE 2.2: Asymmetric multiprocessor organizations, each with different big/little transition latency: (a) shared memory requires 10,000 cycles, (b) shared last-level cache requires 500 cycles, (c) shared first-level cache with adaptive datapath requires 30 cycles.

For example, the ARM Big/Little allots 1MB and 0.5MB of L2 for its big and little cores, whereas a shared cache would provide 1.5MB to be used by active cores ARM (2012).

Switching Cost. Second, a shared LLC reduces switching latency, because compared with flushing dirty lines of a large LLC into DRAM, flushing smaller private caches into the LLC is much faster. If 25% of lines in a 32KB L1 cache are dirty, flushing the L1 into the L2 requires 500 cycles (= 25% $\times$ 32KB $/$ 16B/cy). By allowing switchings to avoid writing to main memory, the shared LLC reduces switching latency by two orders of magnitude.

Summary. Although one might argue that implementing shared LLC for AMP requires much design effort, previous works have demonstrated that such an architecture requires only modest design costs, unless it needs to be tailored to a specific workload mix Lee and Brooks (2007a, 2008b); Guevara et al. (2014a). With reduced switching latencies, such an AMP processor can respond to *coarse-grained application phases*, such as periodic behaviors with distinct resource demands Kumar et al. (2003).

2.2.3 Shared First-Level Cache

If sharing the last-level cache is fast, sharing the entire cache hierarchy is faster. Recently proposed microarchitectures adapt the datapath between out-of-order (OOO) and in-order (IO) execution, thus providing big and little virtual cores in a single physical core. The cache hierarchy and its contents remain useful and valid across datapath transitions. In effect, big and little cores share the cache hierarchy, beginning with the L1.

To design an adaptive core, the architect begins with a high-performance datapath that implements dynamic instruction scheduling. OOO execution requires register renaming, issue queues with wake-up and selection logic, and a reorder buffer. For adaptivity, the architect then adds logic and multiplexors to bypass OOO structures and execute IO. Physical registers might power off, or the issue queue might switch between CAM and FIFO blocks. Two recently proposed adaptive cores for general-purpose computing are noteworthy. MorphCore adapts from an OOO core into a highly multi-threaded IO one Khubaib et al. (2012). Composite Cores uses a shared front-end that feeds instructions into one of two backends, one IO and the other OOO Lukefahr et al. (2012a).

Switching Cost. Transition latency is negligible. Datapath transitions affect only architected state in registers, not the cache hierarchy. Two implementations are possible. In the first, a transition flushes the pipeline, powers off OOO structures such as the physical register file, and resumes instruction fetch in IO mode. Transition latency is proportional to instruction window size. If the OOO datapath commits two instructions per cycle, flushing a pipeline with a half-occupied, 192-entry reorder buffer requires approximately 50 cycles. In a second implementation, the datapath explicitly spills and fills architected state to and from the L1 cache, which requires 30 cycles in MorphCore.

Summary. This emerging class of adaptive microarchitectures motivates a fresh perspective on asymmetric processors and their potential for improving efficiency in mobile computing. Since a big/little transition can occur every few hundred cycles when each transition requires only tens of cycles, the adaptive datapath gives an asymmetric processor agility with which it can respond to an application's *fine-grained microphases*, e.g., instruction throughput fluctuationsLukefahr et al. (2012a).

Figure 2.2 summaries the three AMP organizations above. The lower the switching cost, the more responsive the AMP can be to dynamism in mobile workloads. With greater little core utilization, energy savings are expected to be larger. In the rest of this paper, we evaluate the benefit of these three AMP designs according to their distinct switching

latencies.

2.3 Methodology

We simulate, with gem5, varied AMP architectures to understand the impact of intra-app diversity on energy efficiency. We assume that only one core, either big or little, is active at any given time. We quantify the maximum percentage of instructions that can be executed on the little core that safeguards performance (i.e., instructions per cycle) yet improves energy efficiency. We consider an oracle that collects processor activity during app execution and, offline, evaluates big-little transitions with varying specifications and costs, as well as with perfect knowledge of the future.

First, we divide time into intervals. At the beginning of every interval, the oracle determines whether the other core would offer better performance or efficiency. For example, if the big core is currently active, the oracle switches to the little core in the next interval, if IPC(little) is similar to IPC(big). Three parameters affect the calculation of IPC.

Switching interval, measured in instructions, determines how frequently the oracle makes a switching decision. For example, suppose the interval is 1000 instructions and the oracle knows that, for the next 1000 instructions, IPC(little) = IPC(big) = 1. The little core executes instructions more efficiently with no performance penalty and the oracle considers a switch. Yet this switching decision is not final because, even if the little core seems capable, switching requires data migration and an additional delay that may cause performance to be lower than expected.

Switching cost, measured in cycles, quantifies the additional delays. Following the previous example, suppose that a big-little switch requires 500 cycles. To complete the next 1000 instructions, the little core requires 1500 cycles after accounting for switching cost whereas the big core requires only 1000. We define

$$\text{IPC}_{\text{new}} = \frac{[\text{Interval Insns}]}{[\text{Interval Insns}]/\text{IPC}_{\text{old}} + \text{Cost}},$$

which is the IPC after accounting for switching cost. In the previous example, performance clearly suffers since $\text{IPC}_{\text{new}} = 0.7$ and $\text{IPC}_{\text{old}} = 1.0$. Therefore, the oracle might not consider a switch. However, this is still not the final decision because a switching scheme might trade performance losses for efficiency gains.

Performance penalty specifies the performance loss that can be tolerated after switching. The oracle switches from big to little as long as IPC(little) is greater than or equal to penalized performance IPC(big)/Penalty. In our recurring example, the oracle would switch only if the penalty tolerance is at least $1.5\times$. Altogether, the switching interval, switching cost, and performance penalty define a space of control parameters for switching between big and little cores.

2.3.1 Oracular Switching

With the parameters defined, the oracle identifies all switch points within the app with the following procedure. It first compares big and little IPC for each instruction interval to find those that satisfy conditions for a switch with respect to the penalty tolerance. For the original IPC time series in Figure 2.3(1), the oracle marks the time serials with "big" and "little" flags which suggest the most beneficial executing mode for each interval. If adjacent intervals are marked differently, they define an initial switch point. See points *A* and *B* in Figure 2.3(2).

The oracle then examine each big-to-little switch point based on the switching cost. Consider switch point *A*. Suppose that immediately after the switch, interval i_k reports IPC(big) $= 1.0$ and IPC(little) $= 0.8$. After applying a 500-cycle switching cost, IPC(little) $= 0.57$, which exceeds the tolerated performance penalty such that the little core is no longer good for interval i_k. The oracle revises its decision, marks interval i_k big, and moves on to consider whether i_{k+1} should use the little core.

Similarly, the oracle examines little-to-big switch points. Consider point *B*. After applying a 500-cycle switching cost to interval i_j, IPC(big) $<$ IPC(little) $\times 1.25$ and the big

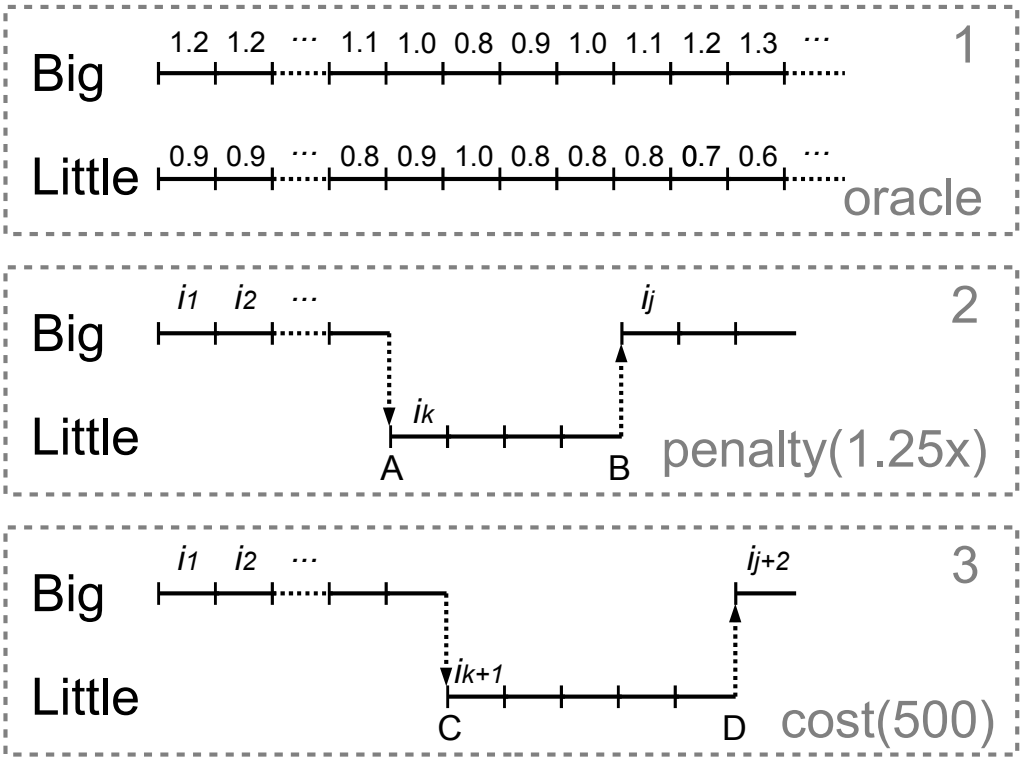

FIGURE 2.3: Switching based on oracle knowledge

core is no longer suitable for interval i_j. The oracle revises its decision, marks interval i_j little, and moves on to subsequent intervals. After applying switching costs, the oracle obtains final switch points that differ from the initial ones – see points *C* and *D* in Figure 2.3(3). The final marks and switch points dictate total performance loss and energy savings.

The final analysis marks more intervals as little because switching to big is costly; the big core's performance advantage must be large enough to offset switching delays. Often, an initial little-to-big switch is discarded in the final analysis and the little core is used instead. Even though the little core's performance is low, the big core's performance is even lower after accounting for switching costs (e.g., interval i_{j+1} in Figure 2.3(3)). As costs increase, an AMP might use the little core more often because once a little core is used, high costs make a transition back to the big core hard to justify.

Table 2.3: Big/Little Specs

	Freq.	Width	#ALU	ROB	PRF	L1 I/D	L2 Cache
Big	1GHz	3	2	192	256	32KB	512KB
Small	1GHz	3	1	32	96	32KB	512KB

2.3.2 Simulation

Each mobile benchmark is installed in an Android (version 4.0.2) system image that is checkpointed immediately before the launching of a benchmark app. To study opportunities for fine-grained transitions, we collect performance statistics for intervals of 1000 instructions. The oracle uses IPC reported from big and little executions to determine the optimal execution mode for each interval.

Because the hardware register counters on the ARM development board cannot support nanosecond-level measurements, we use gem5 Binkert et al. (2011), a cycle-accurate simulator for evaluation. Gem5 supports full-system Android simulation for our mobile benchmarks. In Table 2.3, we specify out-of-order and in-order cores for big-little asymmetry.

We model an in-order core with issue logic that enforces first-in, first-out order. Without dynamic instruction scheduling, the datapath has fewer instructions in flight. Our datapath shrinks the physical register file and the reorder buffer to reflect less demand for these structures, but a more efficient in-order design would bypass them entirely. Our big and little configurations capture the out-of-order performance advantage and the in-order efficiency advantage.

Power. We use McPAT Li et al. (2009) to estimate dynamic power for big and little cores. Figure 2.4 shows the power breakdown for big and little cores when running BBench. Our in-order core saves much of its power in the rename because it reduces the size of the physical register file and the reorder buffer. However, because the rename was not completely removed, a small amount of power continues to be dissipated by this unit. Thus,

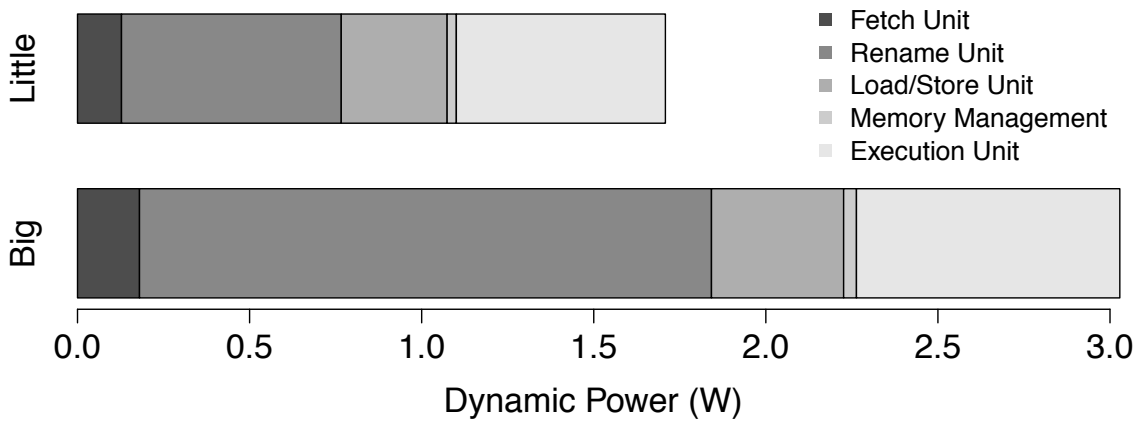

FIGURE 2.4: Power Breakdown

power and energy savings that we report are conservative; an in-order core designed from first principles forgoes a rename unit and would dissipate even less power.

We compare power numbers from McPAT against power measurements collected, using on-board power meters, from an asymmetric ARM processor comprised of Cortex A15's and A7's ARM (2012). When one big core is active and running BBench, voltage varies from 0.9 to 1.05V and average power is 1.10W. When one little core is active, voltage is 0.9V and average power is 0.36W. The big-little power ratio is approximately 2.5×.

McPAT reports a lower ratio of 1.7×. The simulated little core differs from the ARM Cortex A7 in two regards – rename and superscalar width. If the simulated little core were to bypass rename, McPAT would report a big-little power ratio of 2.6×, which is more consistent with our ARM measurements. Without rename, our little core dissipates more power due to its wider datapath; it issues up to three instructions per cycle whereas the A7 issues only one. We evaluate sensitivity to superscalar width in later sections.

Finally, static power plays a large role in asymmetric processor efficiency. When the big core is active, the little core is idle and dissipates static power (and vice versa). The processor cannot use C-states to eliminate static power since that would require separate power supplies and milliseconds for wake-up. Instead, an asymmetric processor reduces dynamic power with idleness and reduces static power by power-gating with low-leakage

PMOS headers, which wake in a few cycles.

We estimate static power as 30% of dynamic power or, equivalently, as 23% of the total. This estimate is consistent with recent processor implementations: IBM Power 7 static power is 25% of the total in a 45nm technology Zyuban et al. (2011). Circuit designers and CAD optimizers tune supply and threshold voltages to balance static and dynamic power. If static power were too large a fraction of the total, designers would increase Vth at the expense of circuit speed Horowitz et al. (2005); Nose and Skurai (2000).

2.4 Evaluation

We evaluate three asymmetric processor organizations: shared memory, shared last-level cache, and shared first-level cache with an adaptive datapath. By abstracting these designs into switching costs, we compare their energy savings for our suite of mobile benchmarks. We also assess sensitivity to other parameters such as interval length, IPC prediction accuracy, superscalar width, and voltage/frequency.

Parameters. Initially, we set the interval length to be 1000 instructions, which illustrates microphase behavior Lukefahr et al. (2012a) and exercises the adaptive processor with a shared first-level cache. Later, we explore little core utilization and its sensitivity to this parameter.

Switching costs are dictated by data movement costs in each asymmetric processor design strategy: (a) 10K cycles for shared memory, (b) 500 cycles for shared last-level cache, and (c) 30 cycles for shared first-level cache. Although switching cost normally includes delays from data movement and wake-up, we assume data movement delay hides wake-up delay as is the case when power-gating.

We consider a range of tolerable performance penalties, measured in terms of the performance ratio between big and little cores. If the performance penalty were to exceed this ratio, little core utilization would reach 100%. Our evaluation sets the penalty below

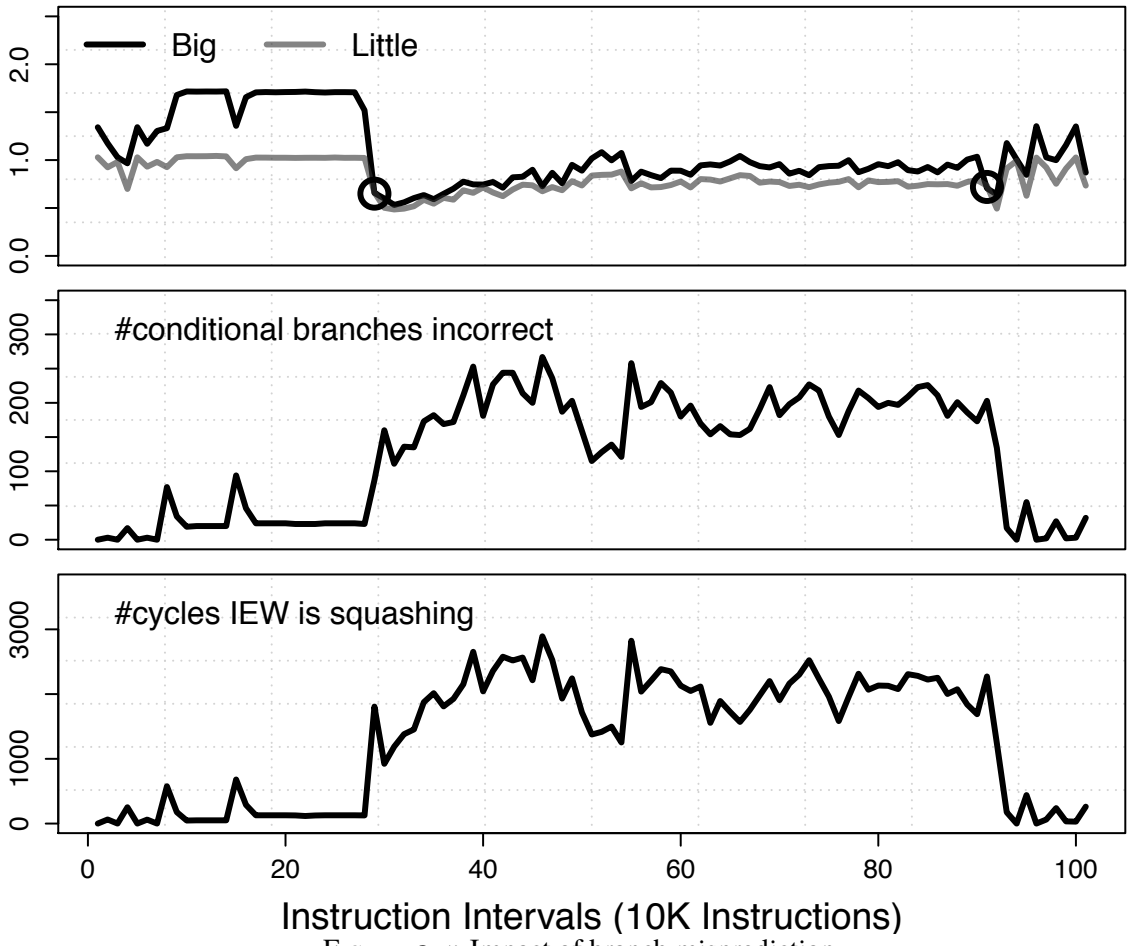

FIGURE 2.5: Impact of branch misprediction

this ratio, ranging from 1.00 to 1.45.

Questions and Metrics. Our evaluation answers three questions about asymmetric multiprocessors for mobile workloads. First, is it necessary to have a little core? To quantify little core utilization, we use the method described in the previous section to find all the intervals that are marked little and compute the percentage. Second, is the little core too slow? We calculate total slowdown to quantify the end-to-end app delay when permitting little core execution. Third, what can we gain from the little core? We calculate total energy savings, which measures the ratio of energy consumed by big-little cores to the ratio of energy consumed by a big core alone.

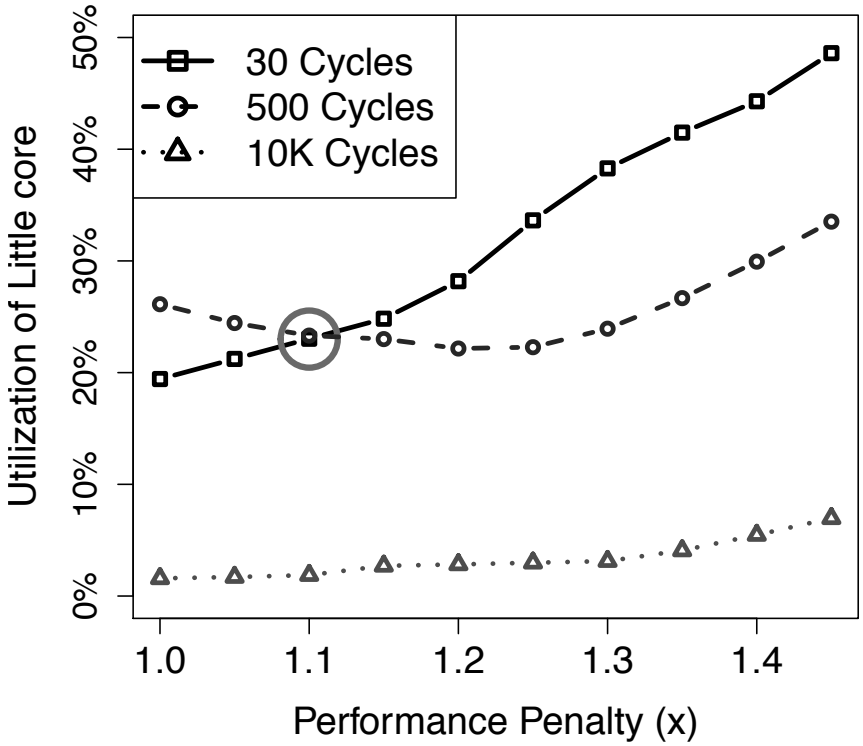

FIGURE 2.6: Utilization of the little core for scrolling

2.4.1 Case Study with Scrolling

Scrolling is a fundamental microbenchmark for mobile phones. It is characterized by periodic user inputs that trigger bursts of computation. Once the user touches the screen, the processor is active for only two to three seconds to execute a few billion instructions. In many instruction intervals, the little core performs as well as, if not better than, the big core.

For example, Figure 2.5 illustrates the impact of branch mis-prediction. Bursty computation when scrolling means that branch predictor training is less effective and speculative execution is often wasted in the big core. Such short and bursty computation, also observed in game and web browser benchmarks, has a direct impact on how an asymmetric multiprocessor is exploited.

Lower switching costs increase little core utilization, as illustrated by Figure 2.6. The adaptive core (30-cy) and shared-LLC (500-cy) strategies use the little core for 20-25% of

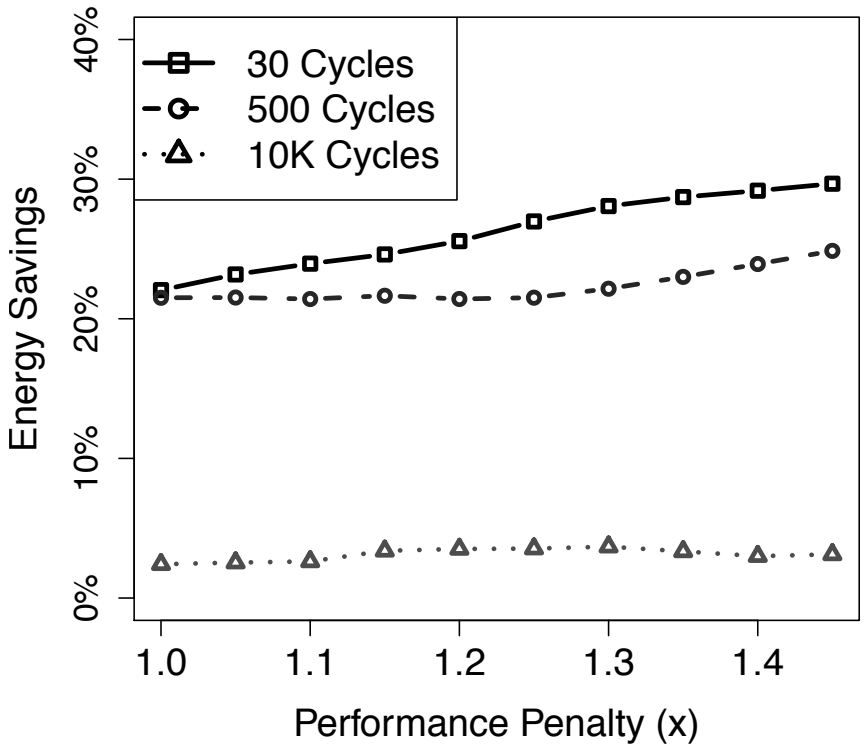

FIGURE 2.7: Total energy saved, relative to big core energy

instruction intervals, even under stringent constraints on performance penalties. For short intervals, today's shared-memory (10K-cy) strategy is ineffective and utilizes the little core for only 5% of intervals.

The cross-over point between 30-cy and 500-cy strategies highlights a counter-intuitive observation. Sometimes, the 500-cy strategy uses the little core more often. This is because switching back to the big core is difficult when switching costs are high and performance penalties cannot be tolerated – see Section 2.3.1.

Figure 2.7 shows energy savings of up to 30% from responsive AMPs. Recall that our power analysis for the little core is conservative, which makes the reported energy savings conservative as well. Big and little energy is calculated by multiplying core power and the number of active cycles in each mode. In addition, switching energy is calculated by multiplying big core power and the number of cycles spent switching.

For short intervals, there is no case in which today's AMPs are effective. The 10K-cy

strategy performs worst for all three metrics, showing low utilization and low energy saving. Containing only 1,000 instructions, each interval is too short when compared to the switch cost and opportunities to switch are limited.

2.4.2 Generalizations with Benchmark Suite

For scrolling, the little core can be well utilized and save energy without harming performance. To generalize this conclusion, we present results for little core utilization and energy savings across all the benchmark apps. To simplify the illustration, we fix the performance penalty to be 1.15x. In practice, the value of the performance penalty can be tuned to achieve certain requirements (*e.g.*, an energy budget).

Figure 2.8 shows utilization of the little core. Mobile apps are more little-core friendly than computationally intensive benchmarks such as Javascript and Linpack. Apps that do not process user input, such as Read and Music, require little processor activity and utilize the little core most. Other apps receive periodic user input that demands computation, such as Launch and Scrolling, but still use the little core 15-24% of the time. Typing is a notable exception and uses the big core to process frequent user inputs. The 30- and 500-cy strategies use the little core often (24% and 22%) while the 10K-cy strategy uses it less (9%).

Figure 2.9 shows total slowdown when the tolerable performance penalty is set to 1.15x. Fast, fine-grained switches between execution modes can produce negative slowdowns (*i.e.*, speedups) because the little core can out-perform the big one (*e.g.*, due to branch mis-prediction penalties). Slow, coarse-grained switches may harm performance but slowdowns are capped by the penalty tolerance of 1.15x.

Figure 2.10 shows energy savings. Both the 30- and 500-cy strategies reduce energy by as much as 38%. Reading and Music are much less computationally intensive than other tasks, providing the most opportunities for the little core. The three strategies offer energy savings of 24%, 22% and 6%, on average, when the penalty tolerance is stringent.

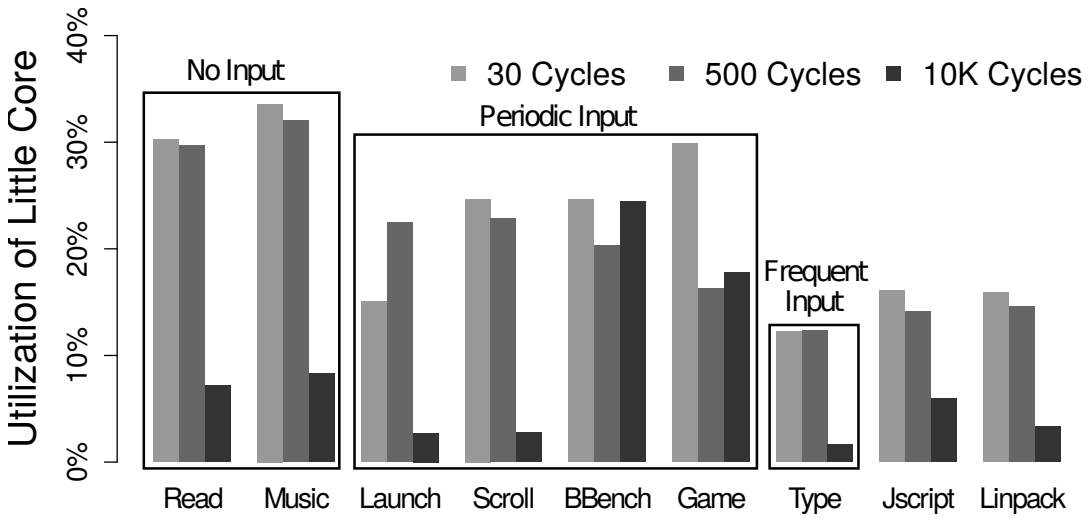

FIGURE 2.8: Little core utilization across apps

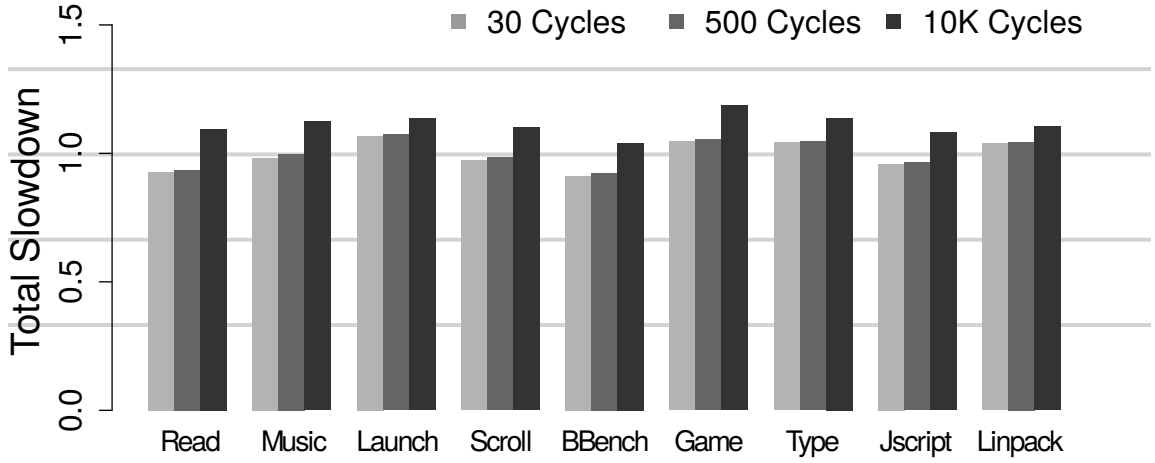

FIGURE 2.9: Total slowdown across apps

When performance targets are relaxed and permit up to a 1.45x penalty, the 30- and 500-cy strategies offer energy savings of 31% and 25% (average), and 42% and 39% (best case).

2.4.3 Sensitivity to Management Parameters

Interval Length. The interval length determines the switching granularity. Figure 2.11 shows the energy savings for varied interval lengths. The brighter the color, the greater the energy savings. The 30- and 500-cy strategies enable the shortest intervals and provide the

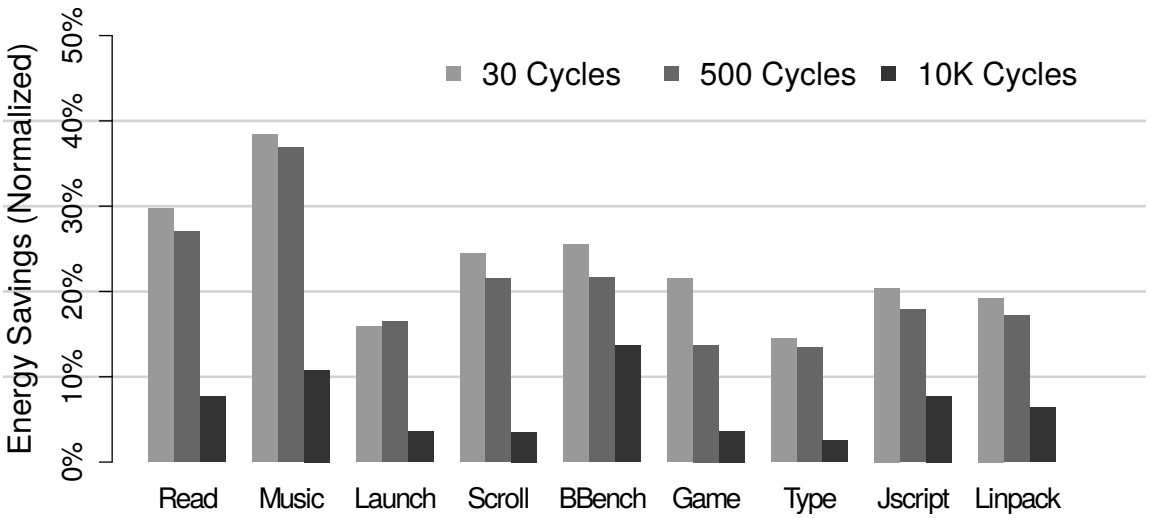

FIGURE 2.10: Energy savings across apps

greatest savings; the top-left corner is brighter. The longest intervals do not reduce energy costs; the right-most columns are darker. The 30- and 500-cy strategies are insensitive to interval length, but the 10K-cy strategy saves energy only when interval length is greater than 10K instructions.

Apps that process periodic user input, such as Launching, Scrolling, and Submarine, are sensitive to switching interval and cost. As the interval length increases, the performance ratio between big and little cores in each interval approaches the average ratio across all intervals (1.0 to 0.7) and reduces switching opportunities. When the interval includes 10M or 100M instructions, the little core is rarely used and energy savings are low as indicated by the dark columns on the right. Finally, compute-intensive apps, such as Linpack, Javascript and Typing, are insensitive to interval and cost – they rarely use the little core and save little energy.

Performance Prediction Error. Thus far, our oracle-based evaluations have demonstrated little core utilization and its energy efficiency. In reality, perfectly knowing processor performance ahead of execution is impossible. It is possible, however, to build a model that predicts the performance of the big/little core with historical statistics obtained from hardware performance counters. The predicted IPCs can then be used to determine if a switch

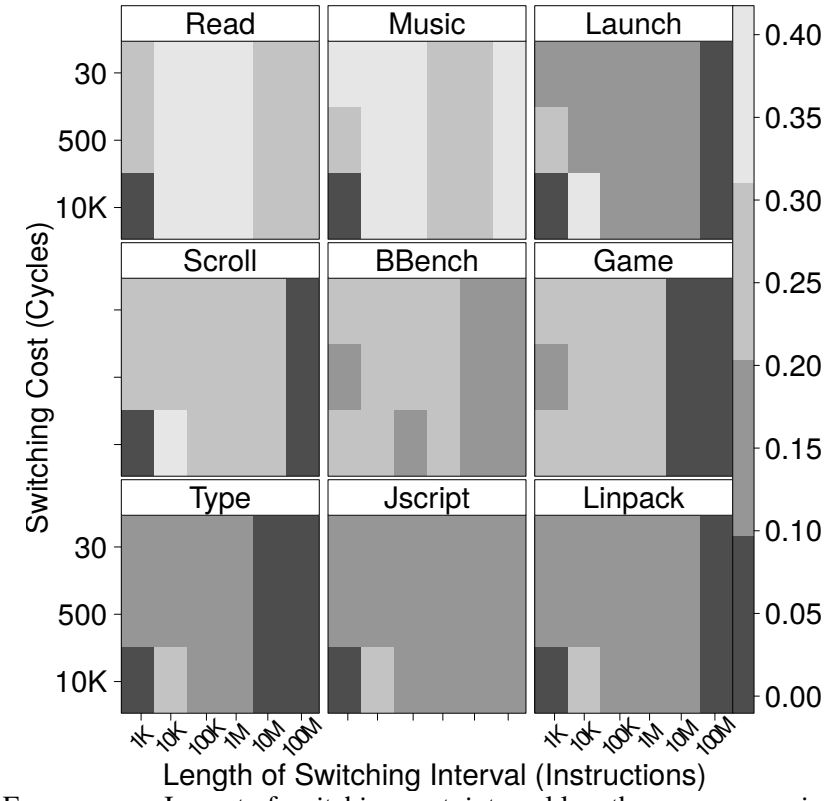

FIGURE 2.11: Impact of switching cost, interval length on energy savings

should happen for the next instruction interval. Such prediction models introduce error, and in this subsection, we discuss the impact of misprediction to the little core utilization.

To model the relationship between inputs from various hardware performance counters, linear regression models are often used. In such models, error terms are independent and identically distributed, following a normal distribution with mean of zero and variance of σ^2, To capture this effect in our oracle, we add Gaussian white noise (mean = 0, sd = 0.06) to the oracle's IPCs, mimicking predicted IPCs with errors with a distribution that is carefully tuned to correspond with related work Lukefahr et al. (2012a). We repeat this process ten times and, each time, recalculate little core utilization.

Figure 2.12 shows the impact of prediction error on little core utilization. Each bar shows oracle-based utilization while each circle shows prediction-based utilization. Most of the time, the impact of misprediction is negligible. An outlier exists in the 10K-cycle

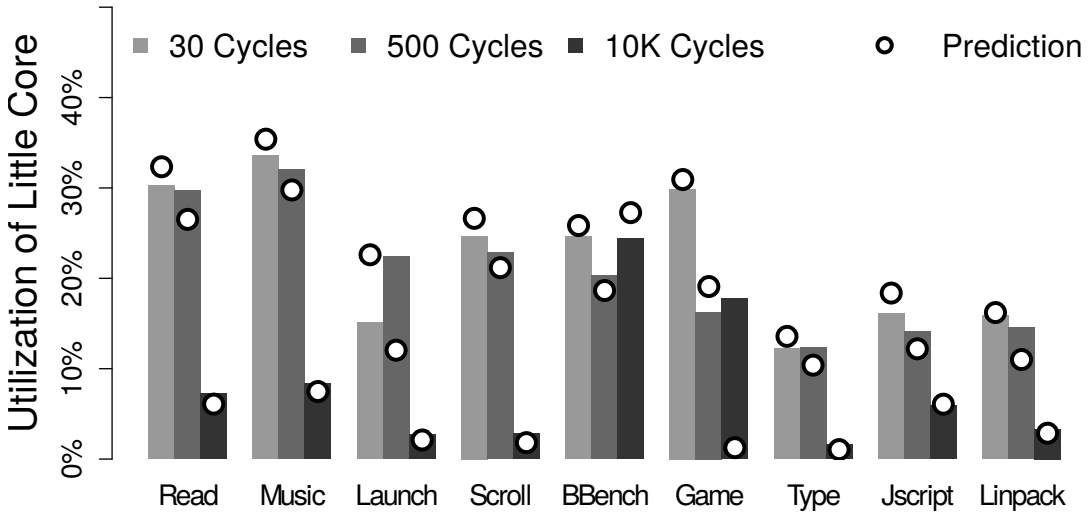

FIGURE 2.12: Impact of prediction errors on little core utilization

strategy running Submarine Game. Utilization with predicted IPCs tend to be much lower than that of the oracle-based result.

We explain the outlier as follows. The 10K-cy strategy with 1K switching interval normally leads to modest utilization of the little core. Utilization that is larger than expected (e.g., nearly 20% for Game) is explained by rare cases in which the oracle switches from big to little and then stays in little mode for long periods due to high switching overheads. Prediction errors avoid becoming stuck on the little core by removing the rare big-to-little switches.

Other than this outlier, the predicted little core utilizations for 30- and 500-cycle strategies deviate from the oracle by a maximum of 10.4%. This means that our oracle-based evaluation represents a realistic estimation of little core utilization. Based on existing prediction methods, the energy savings in our results are achievable.

2.4.4 Sensitivity to Design Parameters

Superscalar Width. By reducing the width of the little core, the performance gap between big and little will grow and little core's utilization will decrease. On the other hand, energy savings might increase because the little core now dissipates much less power. Figure 2.13

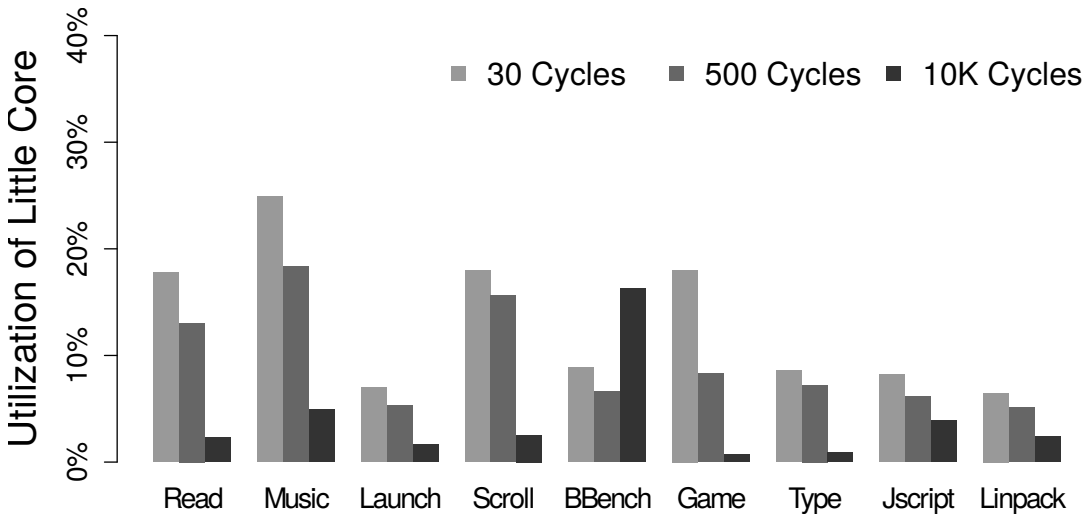

FIGURE 2.13: Little core utilization with 1-wide little core

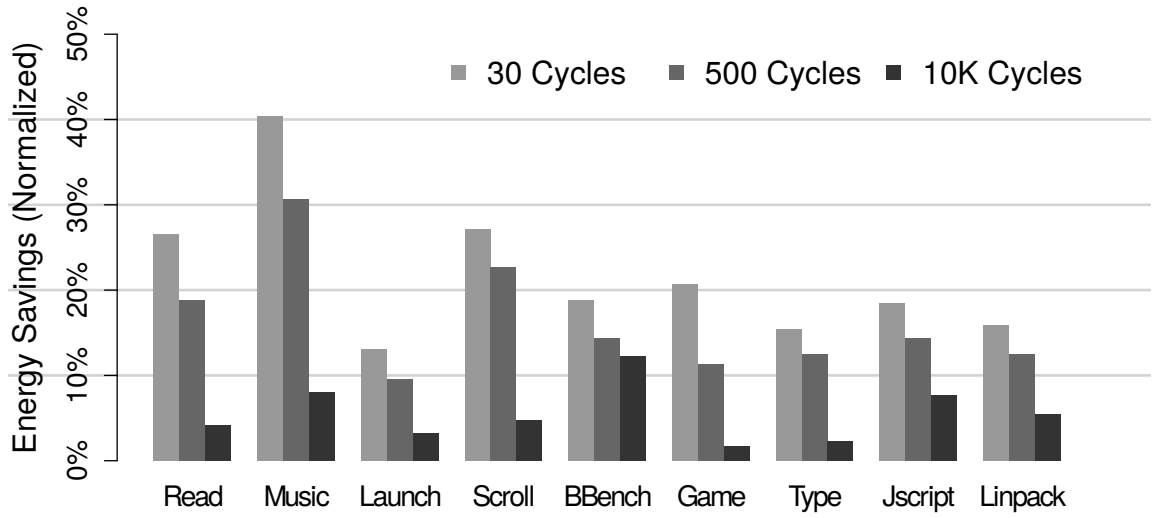

FIGURE 2.14: Energy savings with 1-wide little core

shows utilization of a 1-wide little core when the tolerable performance penalty is 1.15x. Compared with Figure 2.8, little core utilization decreases across all AMP design strategies and mobile benchmarks due to the larger performance gap. Figure 2.14 shows that, although reducing the width reduces the little core's dynamic power from 1.85W to 0.96W, the little core is rarely used and power savings are modest. Thus, the little core must be designed to balance performance and energy savings.

Voltage and Frequency. Existing AMPs provide individual power supplies for big

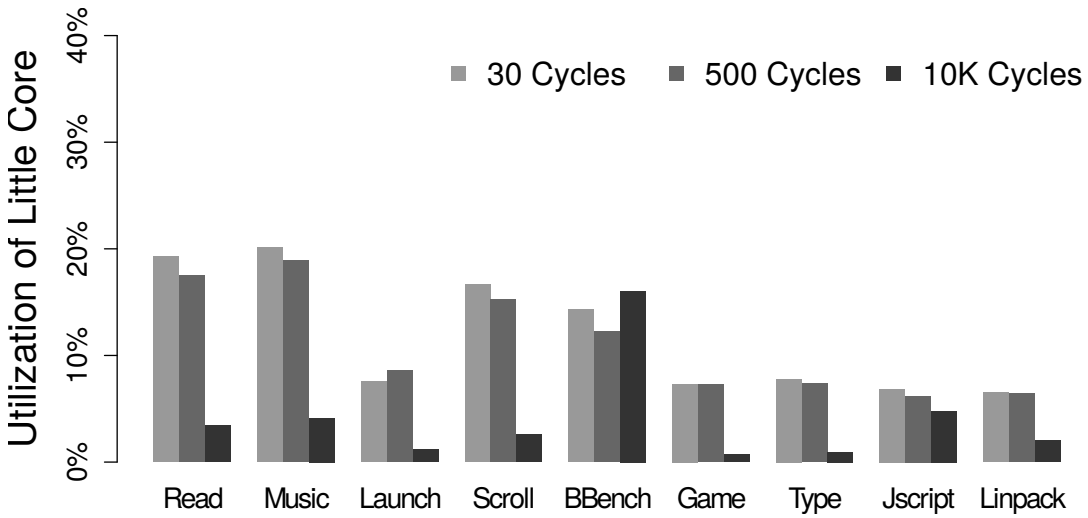

FIGURE 2.15: Little core utilization with 800MHz little core

and little cores, which are organized into clusters, allowing the little and big cores to use different clock frequencies. We evaluate the case when the little core's frequency is 800MHz instead of the big's 1GHz.[2] Since frequency differs, we no longer use IPC to determine switch points. Rather, we scale IPC by the frequency difference and measure instructions per second (IPS) such that IPS(big) = IPC(big)/1 and IPS(little) = IPC(little)/1.25, where 1.25 is the slowdown at 800 MHz.

Figure 2.15 shows little core utilization if 1.15x performance penalty can be tolerated. Reducing little core performance harms its utilization. Although dynamic power falls by half, since voltage can be reduced linearly with frequency, Figure 2.16 shows that most mobile apps do not benefit from reduced frequency.

In summary, reducing little core performance in return for power savings, either through smaller superscalar width or slower clock frequency, would harm the little core's energy efficiency provided. To maximize energy savings, the little core should provide performance within some competitive range of the big core's so that a switching mechanism can exploit program dynamism at fine granularities.

[2] This difference corresponds to big-little configuration in the ARM Versatile Express development board ARM (2012).

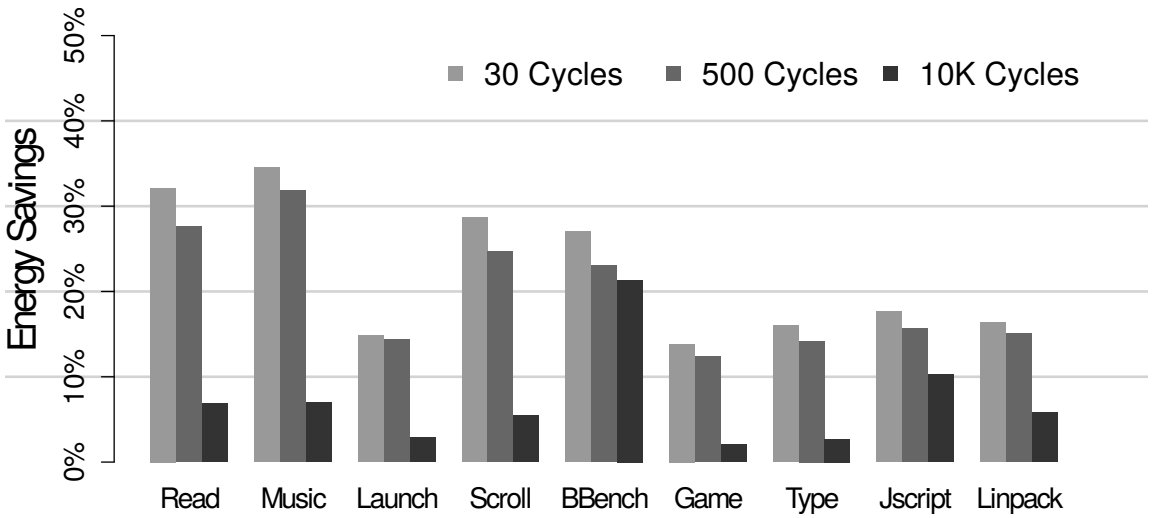

FIGURE 2.16: Energy savings with 800MHz little core

2.5 Related Work

Heterogeneous multicore architectures, due to their exceptional energy efficiency, have been studied extensively. Previous works have focused on workload assignment Annavaram et al. (2005); Kumar et al. (2004); Guevara et al. (2013), scheduling Saez et al. (2010); Li et al. (2007); Koufaty et al. (2010); Fan et al. (2016), thread performance prediction Shelepov and Fedorova (2008); Van Craeynest et al. (2012), and design space exploration Van Craeynest and Eeckhout (2013); Guevara et al. (2014b); Lee and Brooks (2007b). The rise of core fusion Ipek et al. (2007); Kim et al. (2007) and adaptive out-of-order cores Lee and Brooks (2008a); Lukefahr et al. (2012a); Khubaib et al. (2012) requires computer architects to rethink heterogeneity and management.

The scheduler for heterogeneous cores must bypass the operating system and make decisions within short time windows. Predicting the next interval's performance is challenging as performance variation increases when the interval length shortens Padmanabha et al. (2013). Researchers have proposed dynamically estimating app performance on another core type Lukefahr et al. (2012a); Shelepov and Fedorova (2008); Van Craeynest et al. (2012). Briefly, the IPC on the other core is estimated with a profiled linear model that

takes memory-level parallelism, instruction-level parallelism, and hardware performance counters (IPC, cache hits and misses, branch mispredictions, etc.) as inputs.

Our work is inspired by previous works in heterogeneous processing for mobile platforms. Little Rock Priyantha et al. (2011) prototypes a sensing platform in which a small processor is dedicated to always-on sensor stream processing. GreenDroid Goulding-Hotta et al. (2011) uses specialized, low-power cores to accelerate the Android software stack. Zhu and Reddi Zhu and Reddi (2013) analyze the loading of webpages on big and little mobile processors, and they propose to predict and schedule the loading of webpages onto heterogeneous cores according to webpage complexity.

2.6 Conclusion

We propose a set of interactive mobile benchmarks that captures intra-app diversity, which arises from frequent user inputs. Deploying these benchmarks on Android systems and the gem5 simulator, we explore three asymmetric multiprocessing (AMP) strategies. Strategies with tightly-coupled big and little cores, which share the cache hierarchy, can efficiently utilize little cores and reduce energy by up to 42% for interactive apps. Our benchmarks are open-sourced to help future mobile architecture research Fan (2015). Our evaluation illustrates the rich design space underlying asymmetric multiprocessing for mobile phones.

Acknowledgements

This work is supported by National Science Foundation grants CCF-1149252 (CAREER), CCF-1337215 (XPS-CLCCA), SHF-1527610, and AF-1408784. This work is also supported by STARnet, a Semiconductor Research Corporation program, sponsored by MARCO and DARPA. Any findings or conclusions expressed in this material are those of the author(s) and do not necessarily reflect those of sponsors.

3

Swarm Computing for Mobile Sensing

In this chapter, we explore the potential of distributed computing on multiple mobile devices and pursue the performance demanded by next generation apps. We consider a particular class of apps in which co-located users collaborate toward aligned sensing and computing objectives.

Specifically, consider the following usage scenarios that illustrate our application domain. Suppose a group of app users travel to a foreign country in which a different language is spoken. The users wish to better understand native speakers they encounter on the road, by performing real-time voice recognition and translation on their mobile devices. Because the travel companions share these objectives, they collaboratively sense and analyze the audio streams — one user senses audio signals and distributes them for processing, while some users compute for recognition and some others for translation. In another example, suppose multiple users comprise a security team that patrols a route. The users wish to perform face detection and recognition on their mobile devices. The team collaborates — one user senses and distributes video frames to others for detection and recognition.

In these usage scenarios and many others, Internet connectivity is sparse and users prefer to rely on local assets rather than the cloudChun et al. (2011); Cuervo et al. (2010).

Moreover, the mobility of the users forbids the usage of stationary infrastructures such as cloudletSatyanarayanan et al. (2009); Ra et al. (2011). Such circumstances leave users with no choice but either run the app entirely on their own devices, or collaboratively using the surrounding devices.

We present Swing (SWarm computing for mobile sensING), a framework that aggregates a set of mobile devices from multiple users to collaboratively compute a shared answer. We view such a collection of nearby mobile devices as a *swarm*, and by utilizing it dynamically, the framework improves performance, efficiency, and scalability for mobile sensing applications that require intense computation on sensed data streams. Specifically, we make the following contributions:

1. **Design and implement the system.** We build a distributed computing framework for compute-intensive mobile sensing applications. We detail Swing's programming model and workflows, which implement stream processing for the swarm devices. Our design provides efficient APIs for programmability, allowing sophisticated sensing apps to be ported easily.

2. **Design swarm management policies.** Swing's mechanisms support a wide spectrum of management policies for various performance objectives. We illustrate several strategies that distribute app computation across mobile devices, and manage data flow according to heterogeneous device capability. Further, the framework dynamically handles mobility of the devices, as well as their joining and leaving.

3. **Evaluate the system with sensing apps.** On our Android-based Swing prototype, we deploy two sensing apps – face recognition and language translation – on a wireless testbed with up to nine heterogeneous, mobile phones and tablets. Collectively, our results show that Swing efficiently manages multiple devices to meet the real-time performance goals of modern sensing apps with negligible processing overhead.

3.1 The Case for Swing

Aligned Incentives – Why should I participate? We target at application scenarios where each user wants results from the computation. The only question is whether she leverages the swarm resources or relies on her own device. Our preliminary study shows that an individual mobile device is insufficient for apps that require continuous, intense computation on sensed data streams. Figure 3.1 illustrates this observation as various phones process video frames for a face recognition app. Each device can process 4∼10 frames per second (FPS), which is far below the minimal 24 required for smooth video playback. Over time, mismatched arrival and processing rates cause new frames to queue and end-to-end delays to increase. Observe that even the fastest device (H, a quad-core LG Nexus 5) fails to keep up — its end-to-end frame delay increases to 1.2s after only 5s of computation.

Even if a user could perform the computation by herself, the energy burden would be unbearable. We observe that the camera-based face recognition app exhausts a fully charged phone battery in about two hours, with 40% of the energy consumed by computation. If several proximate users demand results from the same application, collaborative computing can reduce the energy costs for any one user. Moreover, it mitigates the limitations of individual devices and improves service quality for all users.

Thus, Swing leverages endogenous incentives, which arise naturally from self-interested participants in our application scenarios. Each user shares resources now in return for reciprocal treatment in future. Although simple, such incentive models are wildly popular in practice. Examples include P2P file sharing, FON's WiFi sharing and Open Garden's Internet sharing. In contrast, other crowd-based systems such as Amazon Mechanical Turk and Medusa Ra et al. (2012) rely on exogenous incentives such as physical payments, implicitly assuming that users will participate in the market.

Privacy & Security – Why should I trust other devices? Swing employs existing and trusted app distribution models. Users download Swing from an app store, or download

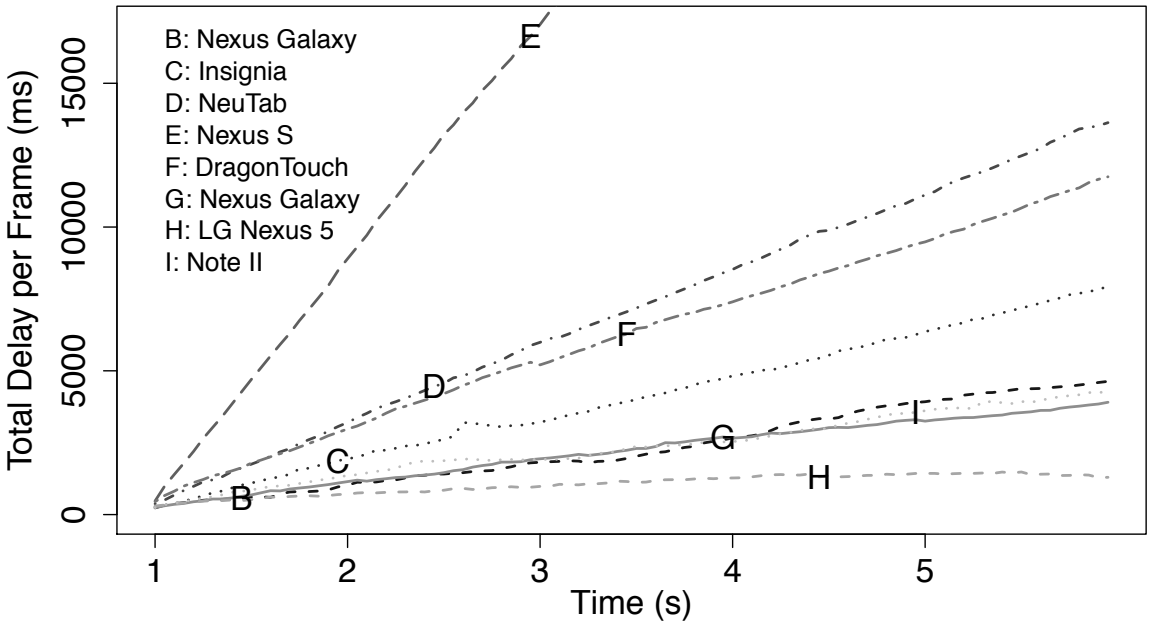

FIGURE 3.1: Delay per frame when processed on different phones at 24 FPS load. Delays build up rapidly, and different phones have different reactions to the same load.

it from a master device who distributes the app signed by its developer. Based on aligned incentives, users desire the results from the collaborative computation. Thus, no device would benefit by deviating from the normal behavior or generating false data.

Swing does not reveal private data, because Swing users only communicate data that is already freely available from the environment. Neither the sensed data nor its analysis is private and sensitive – anyone in the vicinity could have sensed or analyzed the data on her own. It is possible that in some scenarios, sensed data should be shared with restrictions. For example, users might wish to restrict audio sensing and voice recognition data to devices inside a conference room. In such scenarios, Swing can benefit from existing WiFi security protocols, such as WPA, that are external to the system.

3.2 Related Work

Swing inherits the design principles of distributed computing systems. In addition, it is inspired by many other works on mobile-cloud computing, stream processing and mobile

distributed computing. To the best of our knowledge, no prior work has been done in building a distributed computing framework for continuous mobile sensing applications.

Mobile computation offloading. The concept of offloading mobile computation to more powerful servers was first articulated as cyber foraging in Satyanarayanan (2001). Since then, it has been applied in the context of mobile cloud computing. Existing mobile cloud computing frameworks such as CloneCloud and MAUI partition and offload mobile code to the cloud Chun et al. (2011); Cuervo et al. (2010). Cloud offloading techniques cannot easily support real-time applications due to high delays between mobile and cloud. Cloudlets Satyanarayanan et al. (2009) reduce delay by bringing server infrastructure closer to the mobile devices (e.g. LAN or WLAN level) and have been shown to support real time applications Verbelen et al. (2012); Ra et al. (2011). Cloudlets do not enjoy the economies of scale of centralized cloud computing. Thus they may require relatively costly investment in bringing compute infrastructure close to all mobile devices. They may exist in locations where a specific real-time application is needed but may not be as widely available as proximal mobile phones, especially in collaborative ad hoc applications targeted by Swing. Nevertheless, Swing does support "cloudlet mode" through Android virtual machines if a cloudlet infrastructure is available.

Odessa Ra et al. (2011) is a cloudlet system that uses a dataflow graph processing model. Odessa is executed between a mobile and a server and provides mechanisms for adapting parallelism at server and migrating execution stages between mobile and server. Our system poses unique challenges because it requires to distribute load and computations among multiple mobile devices. In our system, we do not do migration of computation stages at runtime (which incurs additional delays an overhead) but instead pre-install the functional units and use dynamic data routing for adaptation.

All the above approaches require constant communication with cloud or cloudlet servers, which may not be easily accessible (sparse Internet, mobile) or required in the collaborative application scenarios targeted by Swing. In contrast, we pursue to enhance mobile

application performance by aggregating resources from multiple, proximate mobile devices.

Medusa Ra et al. (2012) is a crowd computing framework for mobile phones. Crowd computing tasks are defined using a task dataflow graph approach similar to ours. These tasks are high level specifications of actions that cater to human users and not real-time computations. Furthermore, Medusa does not allow direct collaborative interactions between workers as our system. However, our system could use a framework like Medusa for app distribution.

Data stream processing systems. Swing uses a dataflow graph computation model similar to data stream processing systems Castro Fernandez et al. (2013); Zaharia et al. (2012). Such systems typically process a large number of data streams inside compute clusters or data centers. Sonora Yang et al. (2011) is a stream processing system that can support processing on mobile phones. However, it is still based on a client-server model and does not support adaptive offloading of stream computations between mobile and cloud. In contrast, we use this model to program, decompose and distribute collaborative mobile sensing applications on multiple mobile devices. Unlike Sonora which assumes that a computation task can enjoy infinite compute capacity once it is offloaded to the server, our framework utilizes surrounding mobile devices which incur challenges in the resource management.

Distributed mobile computing. Pocket Switched Networks Hui et al. (2006), Throwboxes Banerjee et al. (2007), and ferry-based networks Guo et al. (2006) are distributed mobile frameworks that focus on efficient communication in mobile networks with intermittent connectivity as opposed to efficient collaborative mobile computation on sensed data streams. Misco Dou et al. (2010), Hyrax Marinelli (2009), CWCArslan et al. (2012) implement MapReduce-like frameworks for parallel task execution on mobile phones. MapReduce caters to a batch processing model rather than a real-time computation model targeted by our work. MobiStreams Wang and Peh (2014) provides a distributed stream processing runtime for mobile phones and focuses on the orthogonal issue of fault tolerance. It does

not provide any mechanisms for efficient sharing of execution load among the devices and requires the assistance of a cellular network and a centralized server in the cloud for coordination. In contrast, our framework relies purely on mobile devices and can utilize mobile hotspot APs, Wi-Fi Direct, WLAN or cellular, as networking technologies. REPC Dong et al. (2014), which studies task assignment on mobile devices to achieve performance objectives. However, it solves a static task assignment problem for a set of tasks, whereas in our framework tasks arrive online, and the assignment has to cope with dynamic changes in processing delay, network delay and dynamically handle device joining and leaving.

Distributed Mobile Sensing. When a user carries multiple devices, or when multiple users face the same ambiance, the devices and sensors can collaborate. CoMon+ Lee et al. (2016) is a general-purpose sensing framework that focuses on ambience monitoring, such as dust monitoring, CO2 sensing and GPS localization. The authors demonstrate the opportunities for collaboration in devices' daily usages and propose management policies that maximize energy savings, based on individual device's sensing qualities at different times. The work is based on a distributed framework called Orchestrator Kang et al. (2010), which computes static task-distributing plans for devices to adopt at real time, with respect to concurrent apps and various sensors. These works improve the energy efficiency of continuous sensing and ambiance monitoring on mobile devices. In contrast, our works is a framework of distributed computing designed for compute-intensive sensing apps, using dataflow graphs to assign sensing and computing tasks dynamically.

3.3 Challenges

User mobility and device heterogeneity exacerbate the challenges in managing parallelism in a distributed system. In this section, we conduct preliminary experiments to discover the sources of dynamism and heterogeneity, as well as the performance objectives of the system.

Table 3.1: Performance Heterogeneity

Phone ID	B	C	D	E	F	G	H	I
Processing Delay (ms)	92	121	167	463	166	82	71	78
Throughput (FPS)	10	8	6	2	5	12	13	12

Experimental setup. Today's mobile devices include phones and tablets that deploy a broad spectrum of hardware. First, we characterize this inherent performance heterogeneity with a simple experiment. We use nine phones – A: Galaxy S3, B: Galaxy Nexus, C: Insignia7 tablet, D: NeuTab7 tablet, E: Galaxy S, F: DragonTouch tablet, G: Galaxy Nexus, H: LG Nexus4, I: Galaxy Note2. All devices are connected to a wireless router (Linksys E1200 802.11n 2.4GHz channel 1) and located in the same office. We let phone A send video frames containing faces, at a rate of 24 FPS, to another phone $i \in \{B, C, \cdots, I\}$ to conduct face recognition. The processing delay is measured on each i, and after each i finishes processing, it sends back an ACK so that A can record the end-to-end delay for each frame. Experiments are conducted during the night to reduce chances of interference from other wireless communications. Each experiment runs for 10 minutes (14400 video frames).

Throughput and performance heterogeneity. Table 3.1 shows the average processing delay per frame for each phone (and the numbers also correspond to Figure 3.1). Each *throughput* number here is the inverse of the processing delay (the largest previous integer), representing the capability of each device. Intuitively, if the processing delay is 92 milliseconds for each frame on device B, it means that device B can process at most 10 frames per second (assuming frames are being processed one after another). Observe that differences between phones are significant – the fastest phone H reports throughput that is 6 times higher than that of the slowest phone E. However, even H cannot provide a throughput that is as high as the input rate of 24 FPS. To achieve the desired overall throughput, Swing must utilize multiple devices collaboratively, with respect to their performance heterogeneity.

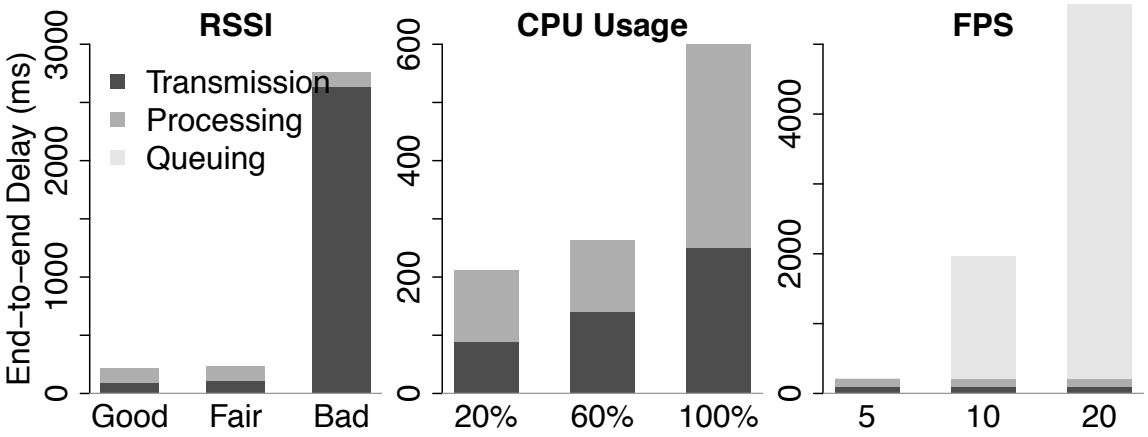

FIGURE 3.2: Decomposition of delays in remote face-recognition processing. Transmission delay changes with WiFi signal strength. Processing delay changes with CPU usage. Queuing delay changes with input data rate.

End-to-end delay and dynamism. Swing must minimize end-to-end delay, because for real-time sensing applications, results that arrive late will have no use. Processing delay is only part of the end-to-end delay. Network transmission delay and queuing delay also contributes largely to the end-to-end delay. To understand each of the three parts, we let A send video frames to B for processing, under three different scenarios: (1) B is placed in regions of different signal strength; (2) B's processor simultaneously runs another compute intensive benchmark. (3) A sends frames to B at different rates. As shown in Figure 3.2, WiFi signal strength, processor utilization, and input data rate affect delays in transmission, processing and queuing. Signal strength varies due to user mobility and thus, the system should divert more tasks to easily reachable devices. Processor utilization varies as the user's private tasks (email, social network, gaming, etc.) demand computation and the system should steer tasks to accommodate the reduced computing capability when such changes occur. Lastly, the system should control queuing delay on each device by matching its input data rates (e.g., FPS) to its capability. Above all, to adapt to all the dynamism, the scheduling decisions should be computed online.

Efficiency and overhead. Figure 3.3 indicates that a same data input rate would cause different levels of overhead on different devices – 5 FPS incurs 7% CPU utilization on phone

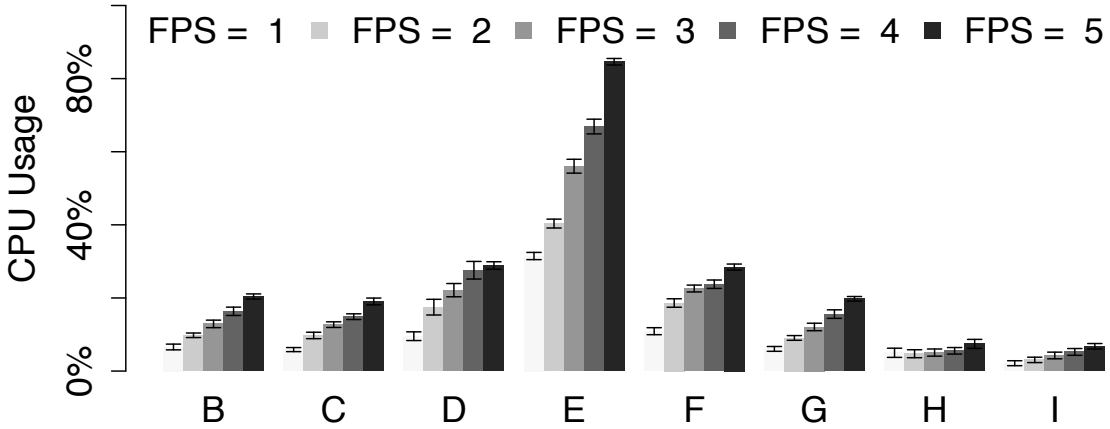

FIGURE 3.3: CPU usage on each phone increases, by varying degrees, as frame rate increases in a face recognition app.

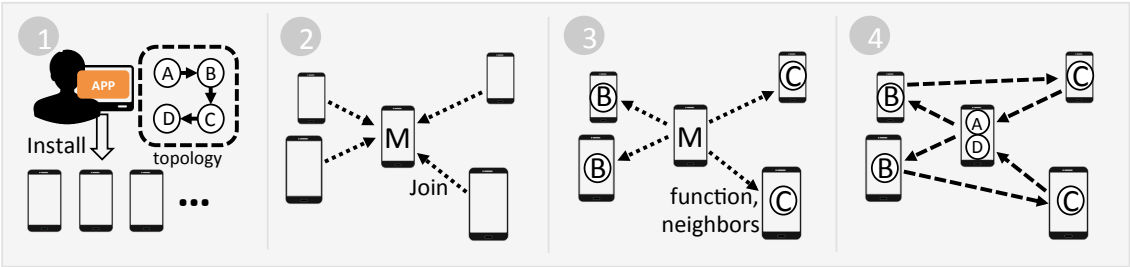

FIGURE 3.4: Workflow of Swing: Installing, Joining, Deploying and Running

I, but 85% on phone E. A high CPU utilization directly affects users' experience during their regular interaction with their phones. Moreover, the CPU utilization is proportionally correlated to processor energy consumption. Therefore, Swing must carefully and efficiently control the input data rate to each device in order to minimize the CPU utilization and energy cost.

3.4 System Overview

With the challenges in mind, we design and implement Swing, a general-purpose framework that supports the collaboration between multiple, mobile devices. In this section, we first present the overview of the system at a high level, including its programming model and workflow, and then detail its implementation.

Each Swing instance can be viewed as a multi-threaded program that runs across multiple devices. A *master* thread controls multiple *worker* threads, assigns software functions to them, and manages data flows between them. Since masters and workers correspond to software threads, a single device could contribute multiple workers, or it could act as both master and worker simultaneously.

3.4.1 Programming Model

Swing uses a stream processing model, which represents a mobile sensing app as a directed dataflow graph. Graph vertices correspond to computational parts of the app, which we refer to as *function units*. In this programming model, app functionality is divided into several interconnected function units. For example, the face recognition app consists of four function units that (A) use the camera to capture video frames, (B) detect faces inside video frames, (C) match faces with names, and (D) display the results.

Graph edges represent data flow between function units. During app execution, each function unit receives a *data tuple* from a previous unit via an edge in the graph. The data tuple contains a list of serializable data structures, such as a bitmap image, a matrix of floating-point values, and a text string. The function unit processes the incoming tuple, computes an intermediate result, and passes it to the next unit in the graph — also in the form of a tuple. From the perspective of a given function unit, a unit from which data arrives is called an *upstream* unit and a unit for which data leaves is called a *downstream* unit. Each unit may interface with multiple upstream or downstream units. A unit without an upstream is a *source* and a unit without a downstream is a *sink*. Function units A and D are the source and sink in our example.

To use the Swing framework, the programmer defines apps as function units with Swing APIs. Specifically, the programmer constructs the app graph by defining function units, including a source and sink, and defining edges that create a topology. The code below shows an example of defining an application graph.

```
// The code below defines the application graph
public AppGraph compose(){
 /* Define tuple structure */
 ArrayList<String> tuple = new ArrayList<String>;
 tuple.add("value1"); //first part: a byte array
 tuple.add("value2"); //second part: a string
 /* Define function units */
 FunctionUnit src = FUBuilder(new Source(),srcId,tuple);
 FunctionUnit f1 = FUBuilder(new FunctionA(),aId,tuple);
 FunctionUnit snk = FUBuilder(new Sink(), snkId, tuple);
 /* Define topology */
 src.connectTo(f1);
 f1.connectTo(snk);
}
```

Each function unit is programmed to first receive data, and then perform certain tasks. For example, the code below defines a function unit that transforms a received data tuple into a graphical object, processes it, and sends the result to the next tuple.

```
// The code below defines a function unit
public class FunctionA implements FunctionUnitAPI {
  @Override
  public void processData(Tuple data) {
    /* get a byte array from the data tuple received */
    // use "value1" as the key
    byte[] bytes = (byte[])data.getValue("value1");
    /* transform array to an image object */
    Mat mGray = new Mat();
    mGray.put(0,0,bytes);
    matToBitmap(mGray, mBitmap);
    /* process on the object */
    String name = faceRecognizer.recognize(mBitmap);
    /* pass the result data to the next function unit */
    Tuple output = data.setValues(null, name);
    send(output);
  }
}
```

The programmer can define performance requirements that affect resource allocation and task scheduling. We also suggest programmers create a separate function unit for each computation stage that could be heavy. For example, create separate function units for

`detect()` and `recognize()`. At runtime, Swing determines their deployment with respect to devices' capabilities.

3.4.2 Workflow

Suppose a group of users agree to compute collaboratively for a mobile sensing app. Figure 3.4 illustrates the Swing workflow in which users install the app, join the system, and execute the computation.

Step 1: Installing the App. Each participating user downloads and installs the specific stream processing app in her device. Users could obtain the app from the master device, or from an online app-store in which programmers submit their apps written with Swing APIs.

Step 2: Launching and Joining. After app installation, one user launches a master thread and others launch worker threads. The master initiates the app, broadcasts its IP address (and port number) so that other devices on the local network may discover it, and launches a socket server to receive incoming connections. Simultaneously, workers discover the master's IP by listening to the WLAN and join the system by connecting to the master's socket server. The master holds the IP addresses of all participating devices in order to connect them with each other (see 4.4 for detail).

Step 3: Deploying Functions. The master deploys the application dataflow graph by assigning functions (vertices) and connecting devices (edges). Since each device already holds all of the app's function units, the master simply gives each worker the name of the functions it must run (see 3.5.1 for detail). Second, the master informs each worker about the IP addresses of its upstream and downstream function units' worker threads — the connections between workers could then be formed.

The master thread is responsible only for control, bootstrapping connections and sending start/stop commands. Since it neither relays data nor executes functions during run-time, the master thread can co-locate on the same device with worker threads. By default, the source and sink function units are deployed on the same device that initiates the master

thread (see Figure 3.4③④).

Step 4: Executing the App. After function deployment, the master sends a start command to the source device, instructing it to sense data and generate tuples. Function units are data-triggered and begin computation as soon as data starts to flow from the source. In Figure 3.4④, source A distributes data tuples to the two devices running B, which then pass intermediate results to downstream devices running C. Ideally, when the system is balanced, sink D receives results at the same rate that A distributes data. The rate at which results are received at the sink defines the maximum system throughput.

In summary, developers write sensing apps using Swing APIs, and users download and install these apps on their devices. When a group of users and their devices wish to run the app collaboratively, one of them initiates a master thread and the others initiate worker threads. The master connects workers together according to the app's dataflow graph, deploys the function units, and manages the data flow during run-time. Swing encapsulates run-time dynamics and, therefore, the end user needs only touch a button to signal her participation.

3.4.3 Implementation

We implement Swing atop SEEP Castro Fernandez et al. (2013), a stream processing platform written in Java. SEEP provides a convenient interface for defining graph topologies by abstracting away the details of TCP socket connections and inter-thread communications. In addition to the effort of porting the framework (caused by the change of Java runtime — same code might behave differently on JVM and Dalvik), we enhances SEEP with the following functionalities:

Dynamic Swarm Management. In Swing, instead of sending code to workers threads, the master "activates" function units of the workers by informing them the classname of the function unit. Given an application dataflow graph, deciding which function units to activate on each device in order to maximize performance objectives (latency or throughput)

with subject to resource constraints is an NP-hard optimization problem Cuervo et al. (2010). In our application scenario, assignment decisions must promptly respond to dynamism (e.g., device joining/leaving). Thus, we propose a heuristic greedy algorithm to solve the function-to-device mapping problem.

At runtime, Swing manages device usages to achieve performance objectives and energy efficiency. Each upstream thread has a routing table with downstream threads' IDs and their weights. In order to adapt to real-time changes in network and device usages, we periodically monitor processing-delay at the downstream and per-frame end-to-end delay at the sink and update the routing tables accordingly (see 3.5 for the design of dynamic swarm management strategies).

Discovery. Swing automatically establishes connections between the swarm devices upon users' participation. During initialization, the master broadcasts itself by registering a Network Service on the network, using Android Network Service Discovery (NSD). Each worker device maintains a background service that listens for the master. Upon discovery, the background service connects to the master, also using NSD, so that the master knows the worker device's IP address.

Handling Joining and Leaving In a mobile environment, users join and leave the system during run-time. To quickly involve newly joined devices, the Swing master constantly listens for incoming connections and activates function units on the new devices as they join. The routing tables of other workers are updated accordingly.

When a data link is broken, due to departing users or network error, the affected upstream units automatically delete the corresponding downstream threads from their routing tables and re-route data to other threads.

Serialization. Communicating through socket connections requires the data to be serialized first. SEEP uses Kryo and default Java serialization methods to serialize the data tuples transmitted between function units. However, mobile sensing apps may involve transmitting customized objects, such as an image container, a multi-dimensional sensor

vector, or audio files. Swing extends SEEP's serialization function and transforms such customized objects into a byte array at the sender, which is serializable, and transform the array back to the object at the receiver.

Reorder buffer. Data parallelism and performance heterogeneity cause each tuple's end-to-end delay to differ. Tuples that are dispatched to workers earlier may arrive later, and vice versa. To solve this problem, we buffer results as they arrive at the sink and sort them in-order before playback. A larger size of the buffer ensures better ordering, but postpones the display of the results for longer. In our current implementation, the size of the buffer equals to the data rate generated at the source, i.e., timespan of 1 second. We will evaluate the effectiveness of this length in our evaluation section.

Background Service. Swing extends Android Service to run in the background without interrupting other user activities. Swing acquires a CPU wake lock to prevent service termination due to processor sleep modes. Swing runs on Android version 4.0 or higher, which includes the majority of Android devices.

3.5 Managing Swarm

The biggest difference between Swing's swarm computing and traditional distributed computing is that mobile devices are heterogeneous. To guide our swarm management design, we first formalize three objectives that our system desires to achieve.

• **Throughput**: the rate at which results are produced by the sink. Swing must maximize throughput, though the upper bound is implicitly determined by the rate at which data is produced by the source (and that rate is hereafter noted as Λ).

• **Latency**: the end-to-end delay from the time a data frame leaves the source till its result is produced by the sink (hereafter noted as latency). Swing must minimize not only the average latency per frame, but also the variance — a large variance of latency would require a larger reorder buffer which postpones the display of the results.

• **Efficiency**: throughput divided by the sum of energy consumption on each participating device. Swing must consider not only the performance but also the power consumption of the whole system.

Given a task graph, finding a task assignment (e.g., determine which device to run which function and how much data to process) that minimizes the latency with throughput constraints is proven to be NP-hard Dong et al. (2014). In our system, we also need to consider the energy efficiency of such assignment. Moreover, the task assignment must react fast to dynamism in the system, thus the time it takes to compute the assignment should be minimized. In order to solve this problem, we propose a heuristic algorithm comprising three procedures: (1) **Function Activation**, (2) **Worker Selection**, and (3) **Data Routing**. Each procedure is repeated periodically to handle all the dynamism: joining/leaving, transmission delay changes, processing capacity changes, etc.

At a high level, Function Activation connects newly joined devices with existing devices in the execution pool and activates the function units on them. Worker Selection periodically decides which downstream threads to send data to, for each upstream thread. Data Routing decides which exact thread, among the selected downstream threads, to send each data tuple to. In this section, we explain the details and the choices we made in designing each procedure.

3.5.1 Function Activation

At the bootstrapping stage, the master activates function units on each device. The master first commands every participating device to launch m worker threads, where m is the number of function units (except source and sink) in the app graph. As the pseudocode below shows, each thread in the worker device then activates one function unit. Consequently, every device can execute any function unit at run-time.

Since the actual execution of worker threads is data triggered, by controlling data flow at the upstreams, the Data Routing procedure can manage the usage of the threads and thus

Algorithm 1 Function Activation Procedure

```
1: procedure FUNCTION ACTIVATION
2:     while listen for new connections do
3:         receive new device s and add it to device pool S
4:         command device s to launch m worker threads
5:         T_s = the thread pool of s
6:         for j = {1, ··· , m} do
7:             activate function j on worker thread t_{s_j}
8:         end for
9:         connect thread t_{s_1} to source
10:        connect thread t_{s_m} to sink
11:        for j = {1, ··· , m − 1} do
12:            connect t_{s_j} with t_{s_{j+1}}
13:            for each other device s' in S do
14:                connect t_{s_j} with t_{s'_{j+1}}
15:                connect t_{s'_j} with t_{s_{j+1}}
16:            end for
17:        end for
18:    end while
19: end procedure
```

the resource consumption of devices. Function Activation procedure only needs to ensure that every thread is connected to each other, providing maximum flexibility (see Figure 3.5). In addition to the bootstrapping stage, this procedure is also called when new devices join.

3.5.2 Worker Selection

After Function Activation, each upstream thread has multiple downstream threads connecting to it. When data tuples are ready to enter the next function unit, the upstream thread needs to decide which downstream threads should receive them. The simplest policy for centralized data scheduling is round robin (RR) – the upstream sends data tuples to each downstream in turns. Apparently, RR is unlikely to work well in mobile crowds since the devices rarely have equal or sufficient capability. The existence of "stragglers", i.e., devices of low computational capabilities, increases the average end-to-end delay per frame and also degrades the overall throughput.

In addition to avoid "stragglers", Swing seeks to maximize energy efficiency. Because computation in stream processing is data triggered, a downstream thread that receives more data performs more processing and thus, worker selection determines each thread's

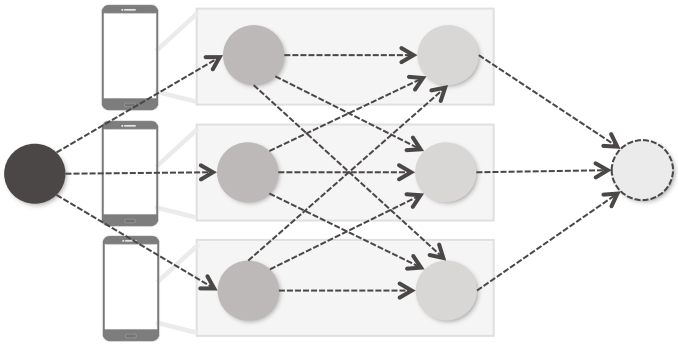

FIGURE 3.5: Topology after Function Activation: An Example

utilization and the corresponding device's energy consumption. The Worker Selection procedure needs to minimize overall energy consumption while maximizing performance.

A solution exists for both performance and efficiency. Recall that the throughput of the system has an input rate Λ that is determined by the source. Efficiently achieving performance targets means that mobile sensing apps should meet but not exceed goals for service quality. For example, the face recognition app should process and replay video at 24 FPS. Throughput that is lower than this target fails to meet requirements for real-time analysis, but throughput that is higher consumes additional resources without perceptible benefit. Therefore, Worker Selection procedure can meet performance targets with only a few of the most capable devices since mobile devices with the newest processors analyze data streams more quickly and consume less energy doing so.

Identifying the most efficient subset of threads is a variation of the Knapsack problem – given many downstream threads, each capable of analyzing tuples at rate λ_i, choose a subset of threads that minimize the sum of their rates while meeting the target rate Λ. For tractability, we relax the optimization objective and minimize the number of selected downstream threads rather than the sum of their processing rates. Specifically, we solve the following:

$$\text{minimize} \sum_{i}^{N} x_i, \text{ such that} \sum_{i}^{N} \lambda_i x_i \geqslant \Lambda,$$

where x is a binary vector that denotes whether downstream i is selected ($x_i = 1$) or not ($x_i = 0$), and N is the number of downstream threads. $\lambda = 1/W$ where W is the processing delay of a downstream thread. Our solution sorts devices in descending order by λ_i and chooses the top devices such that their throughput sums to at least Λ. Algorithm 2 details each step in Worker Selection.

Algorithm 2 Worker Selection Algorithm

```
1:  procedure WORKER SELECTION
2:      for each downstream function unit j ∈ M do
3:          S_sorted = sort (i) based on W_{i_j}, where i ∈ S
4:          while Σ_{i∈S_j} 1/W_{i_j} ⩽ Λ do
5:              add t_{i_j} to T_j
6:              remove i from S_sorted
7:          end while
8:          select T_j for downstream function unit j
9:      end for
10: end procedure
```

Profiling of Processing Delay: The worker selection procedure requires to the upstream threads to know the processing delay of each downstream thread on each device (W_{i_j}). Therefore, to adapt to changes of W, the upstream threads periodically operates with round robin (i.e., sending data to all the downstream threads in turns), so that each downstream thread processes the data and reports its W_{i_j} to the upstream thread. The upstream thread then repeats the selection procedure.

3.5.3 Data Routing

Given a set of selected downstream threads, the upstream thread needs to distribute data tuples to them. A natural solution is to route different portions to different threads. We adopt *probabilistic routing* — each upstream thread maintains a routing probability table that holds the probability p_i of selecting downstream i. Every time the upstream routes a data tuple, it generates a weighted random number according to the probability table and sends the tuple to the specified downstream ID.

One way of computing the probabilities is to ensure that the faster a thread processes data,

the more data is sent to it. In other words, this policy computes probabilities in proportion to the inverse of thread's processing delay:

$$\text{for all } i, p_i = \frac{i/W_i}{\sum_j (1/W_j)},$$

The processing delay may change: the user may launch another computationally intensive app even as Swing runs on the same device; in addition, different frames may have different processing costs. For this reason, each downstream thread reports its W_i periodically (e.g., every second) to the upstream thread which then updates its routing table. Notice that we directly measure the processing delay, instead of indirectly estimating it from CPU and memory usage. Although the latter has been widely used in other systems, it does not provide accurate information about how fast a thread processes data tuples. For instance, a thread on a fast device with a high CPU usage might still process data faster than a thread of a slow device with a low CPU usage.

However, as we have stated earlier, processing delay is merely part of the delay. Routing purely based on processing delay might end up using devices that are located in regions of weak WiFi signals, resulting in performance degradation. Therefore, another way of routing is to use probabilities that are inversely proportional to threads' latencies, so that more data routes to threads with lower latencies:

$$p_i = \frac{1/L_i}{\sum_j (1/L_j)},$$

where L_i is measured as follows: the upstream attaches a time-stamp to each tuple before sending. Each downstream, after finishing processing the tuple, sends back an ACK to the upstream with its own ID and the tuple's time-stamp. Upon receiving the ACK, the upstream calculates the overall latency of the tuple by subtracting the time-stamp from the current time.

The measured latency L_i therefore includes the network delay (dominated by transmission delay) from the upstream thread's device to the downstream thread's device, the

processing delay and the queuing delay in the downstream thread (and the return trip's transmission delay which is negligible due to the small size of an ACK). Notice that since the subtraction is done on the upstream device only, no clock synchronization is needed. The upstream then registers the latency to the corresponding worker thread ID for the probability calculation.

Compared with a purely processing delay based routing, latency based routing aims to minimize overall latency and might rely heavily upon threads that are computationally less capable.

Putting everything together — For latency based routing to work better, the Worker Selection procedure needs to ensure that the selected devices perform better than unselected ones with respect to latency. Therefore, in Algorithm 2 line 3, instead of sorting threads based on the processing delay W, Swing sorts downstreams based on latency. The algorithm then selects a set of devices that are capable to provide a throughput of Λ. The upstream then routes data with probabilities that are inversely proportional to the latency numbers of the selected downstreams. Collectively, this routing method considers both throughput and latency, while the efficiency is achieved by the worker selection procedure. In the next section, we will evaluate this routing scheme by comparing it against different strategies.

Notice that Swing requires the downstream threads to upload either its processing delay (integer, 4 bytes) or a time-stamp (long integer, 8 bytes) back to the upstream periodically (e.g. every second). Since the packet sizes are small, the extra network traffic is negligible.

3.6 Evaluation

We evaluate Swing, assessing its ability to address key challenges in mobile swarm-based computing. We first explain the experiment applications, followed by the evaluations of comparing swarm management strategies. We then show how the system adapts to user mobility. Finally, we compare the performance of a swarm based solution with a cloudlet-like

Table 3.2: Lines of code changes

	Same	Modified	Added	Removed
Face Recognition	355	4	604	525
Voice Translation	466	47	375	430

solution.

3.6.1 Experiment Setup

Programmability. We use two open-source sensing applications to evaluate Swing. Table 3.2 shows the lines of code changes for porting the two applications onto Swing.

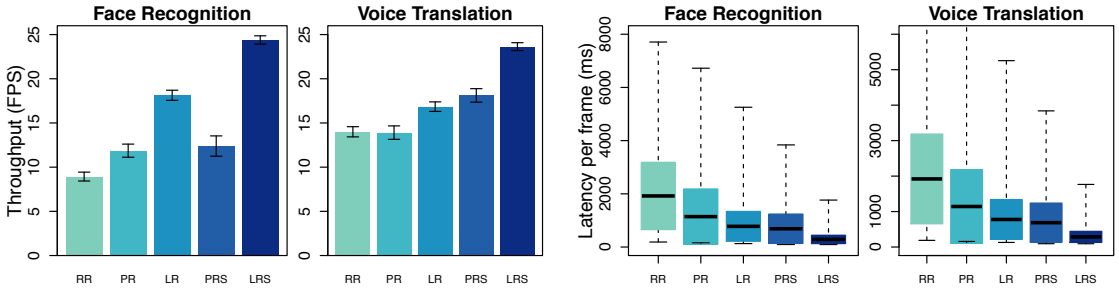

FIGURE 3.6: Throughput and delay of four routing schemes in two applications.

The first app is face recognition Robotic Apps (2013), which uses the OpenCV CascadeClassifier to detect faces in a video frame and the FaceRecognizer class to recognize faces. We modified its source code with Swing API to create four function units: reading video frames from files (source); detecting faces from frames (detector); matching faces with databases and return results (recognizer); and displaying results (sink). The size of

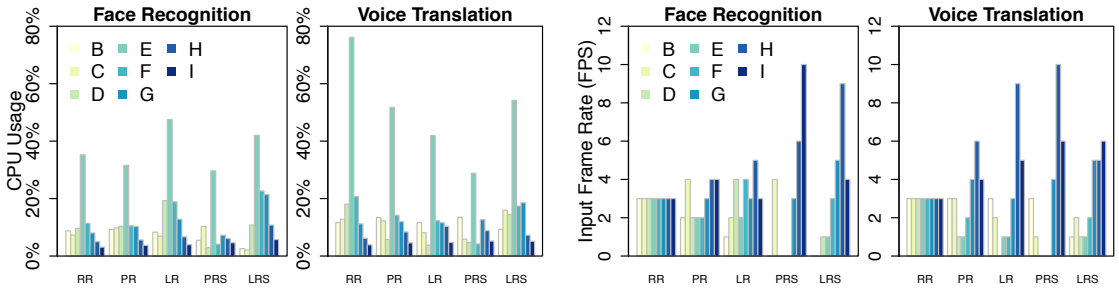

FIGURE 3.7: Resource usage and data rate of each device in two applications.

each image frame is 400×226 pixels (6.0kB).

The second app is voice translation. It contains four function units: reading audio frames from files (source); recognizing audio streams into English words (based on CMU Pocketsphinx Huggins-Daines et al. (2006)); translating those words into Spanish (based on Apertium Forcada et al. (2011)); and displaying results (sink). The size of each audio frame is 72.0kB.

3.6.2 Comparison of Data Routing Methods

To evaluate the swarm management strategy, we compare it with four different strategies:

(1) round robin (RR) — the baseline of baseline;

(2) processing-delay based routing w/o selection (PR);

(3) latency based routing w/o selection (LR);

(4) processing-delay based routing with selection (PRS);

(5) latency based routing with selection (LRS);

We deploy eight phones ($A \cdots I$ in 3.3) as worker phones and one phone as the source and sink. To account for network heterogeneity, we placed devices B, C, D at locations of poor WiFi signals, which forces them to user lower data rates and doubles their transmission delay. We compare the performance of the five data routing methods.

Performance

Figure 3.6 shows the average throughput of the system and the min, max, average and variance of end-to-end delay per frame. Observe that latency based routing methods provide better delay per frame (smaller mean and variance). Processing-delay based methods fail to provide the desired performance, because they schedule data purely based on the capabilities of downstream threads, regardless of their locations in the network. Specifically in this experiment, PR and PRS often route data to threads on device B and C which are located in regions of weak signals. As a result, the TCP protocol requires the sender to lower data

transmission rates for these devices, which directly reduces the number of tuples transmitted per unit time.

The results also demonstrate that the worker selection procedure provides a smaller variance in delay compared with routing without selection. This is intuitive because with worker selection, Swing essentially utilizes fewer devices and avoid "stragglers". LRS indeed performs best, providing **throughput** that is 2.7x higher and **end-to-end delay** per frame that is 6.7x lower than RR's.

Overheads

We measure overheads in terms of processor and network utilization. Furthermore, we estimate energy, which is correlated with activity.

Processor Usage: Figure 3.7's left two graphs show processor utilization for each device from running `top`, which reports usage as a percentage of total processor time across all cores. In addition to data tuple assignment, the processor usage depends greatly on hardware capability. Swing computation consumes a larger share of processor time when the processor is weak (e.g., E) and a smaller share when the processor is strong (e.g., I). In general, for a range of routing methods, `Top` reports average CPU usage below 20% across devices. This level of overhead has no impact on user experience. Furthermore, Swing runs as a non-blocking, background thread, which can be swapped to give priority to front-end apps that receive user interaction.

Data Transmission: Figure 3.7's right two graphs show the amount of data transmitted from the source to each worker devices. The base-line RR sends an equal amount of data to each device; PR and LR send data to most of the devices. In contrast, PRS and LRS send data only to certain devices, such as H and I, and avoid "straggler" devices such as E and F.

Energy: We estimate the processor power consumption of each worker device using a linear model Xu et al. (2013); Mittal et al. (2012); Cuervo et al. (2010). First, we measure

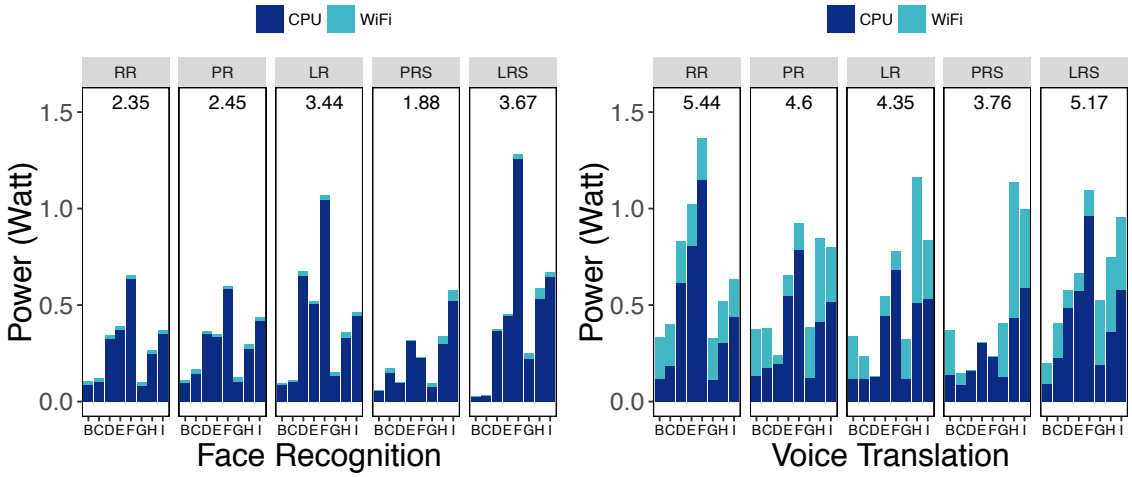

FIGURE 3.8: Energy consumption each device

idle power. Then, we measure peak power by stressing the processor with 100% utilization for 30 minutes, recording the change in battery level during this time, and estimating the corresponding change in energy given the device's battery capacity. Finally, we estimate processor power as a percentage of peak based on the measured processor usage. We estimate the WiFi power consumption in a similar way. We first measure idle power. We then measure peak WiFi power by sending data at maximum bandwidth with iperf for 30 minutes and recording the change in battery level. We then estimate WiFi power as a percentage of peak based on the measured data rate.

Figure 3.8 shows the estimated power consumption (processor + WiFi) on each device. Observe that latency-based routing methods consume more energy as they provide better performance. Also, PR and LR achieve better balance in power consumption compared with methods with worker selection. On the other hand, worker selection greatly improves **efficiency** in terms of throughput-per-watt. For both applications, PRS is 40%∼48% more efficient than PR, and LRS is 14%∼35% more efficient than LR.

Tuple Order: Figure 3.9 illustrates this loss of order by showing the time at which the result for each tuple arrives at the sink (see the gray dots) for the face recognition app.

The solid lines in Figure 3.9 illustrate the playback times for re-ordered tuples when

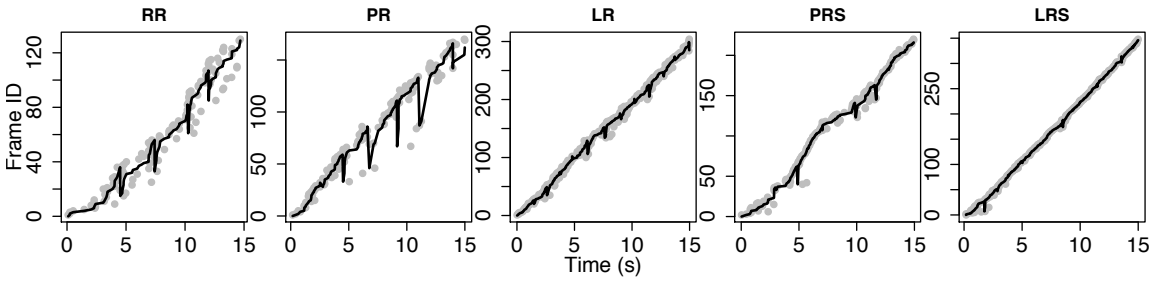

FIGURE 3.9: Ordering of frames: gray dots represent frames' arrival timings; solid line represents the reordering using a buffer.

the reorder buffer length is 24 (i.e., one second). Routing methods with worker selection have smoother curves than others because they produce smaller variances in end-to-end delay, which corresponds to our previous observation. In contrast, methods without worker selection would require larger buffers for smooth playback.

3.6.3 Handling Mobility

In this subsection, we demonstrate Swing's ability to adapt to mobile users, who join the system during computation, move in ways that affect network connectivity, and leave the system abruptly.

Joining: Initially, we use phones B, D for computation. As these phones compute, we launch a new Swing worker thread on G, which then joins the computation automatically. Upon detecting an incoming connection, the upstream adds the corresponding downstream worker IDs to the routing table and re-calculate all the probabilities using LRS. Figure 3.10 (left) shows that, within a second of G's arrival, throughput rises to its maximum level of 24 FPS in the face recognition app. The system preserves links during the transition and no data tuple is lost.

Moving: The system deploys phones B, G, H with the LRS routing method, which copes with variations in network delay. Initially, phones are placed at a location with good WiFi signals (RSSI¿-30dBm). After one minute, G's user walks to a location with slightly weaker signals (between -70dBm and -60dBm), stays there for one minute, and then walks

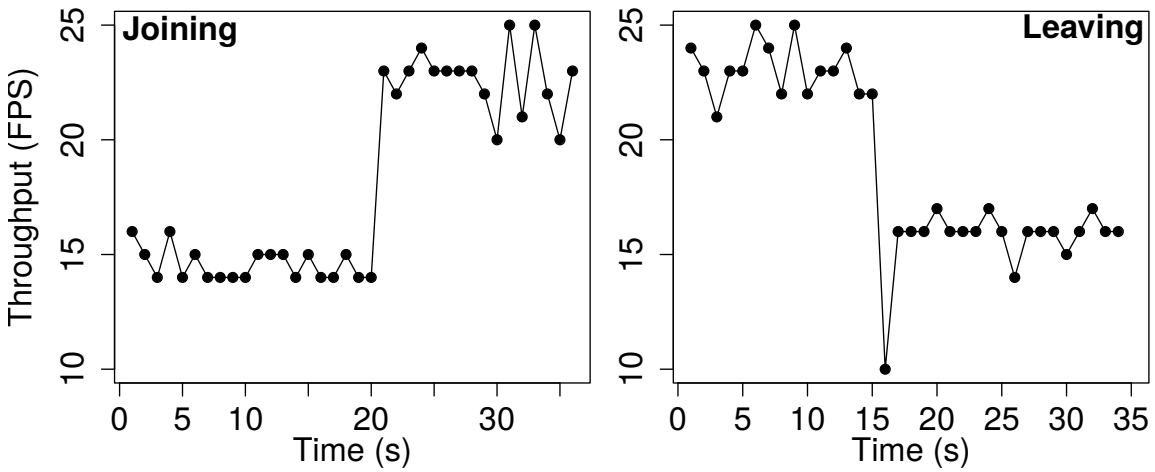

FIGURE 3.10: Throughput changes when device joins, leaves

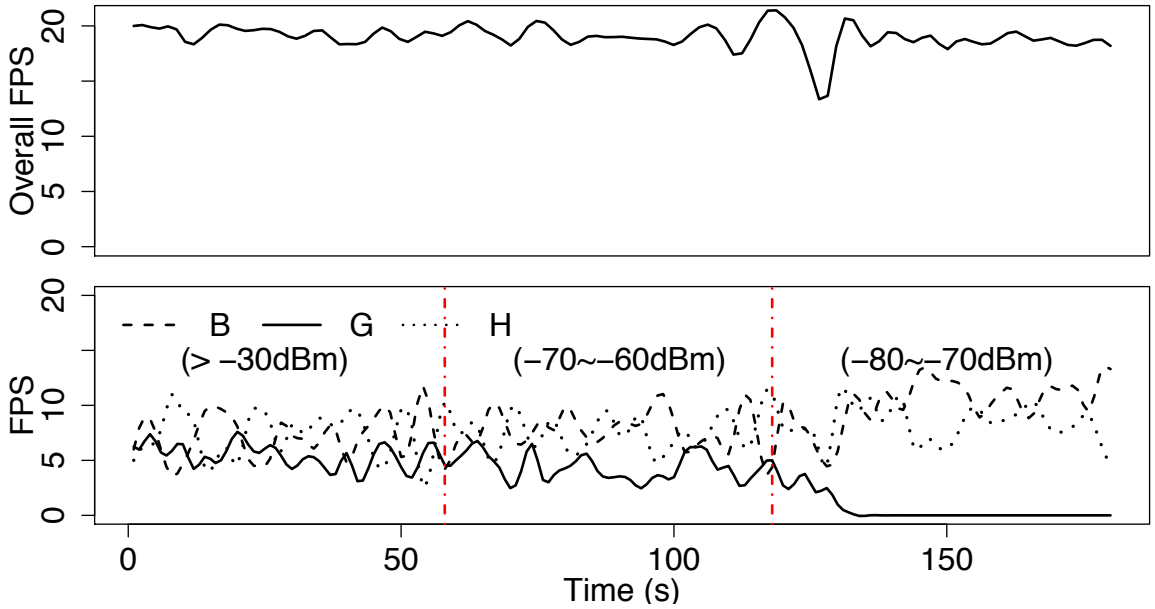

FIGURE 3.11: Throughput, load changes when device moves

to a third location with poor signals (-80 -70dBm). Figure 3.11 shows the effect of mobility on signal strength, which affects overall throughput (top graph) and per-device throughput (bottom graph). Overall throughput recovers quickly after G moves to a region with weak signals as Swing re-routes data to the other two phones.

Leaving: Finally, consider the scenario in which three phones participate in Swing and one phone leaves abruptly. The system deploys phones B, G, H with the LRS data routing

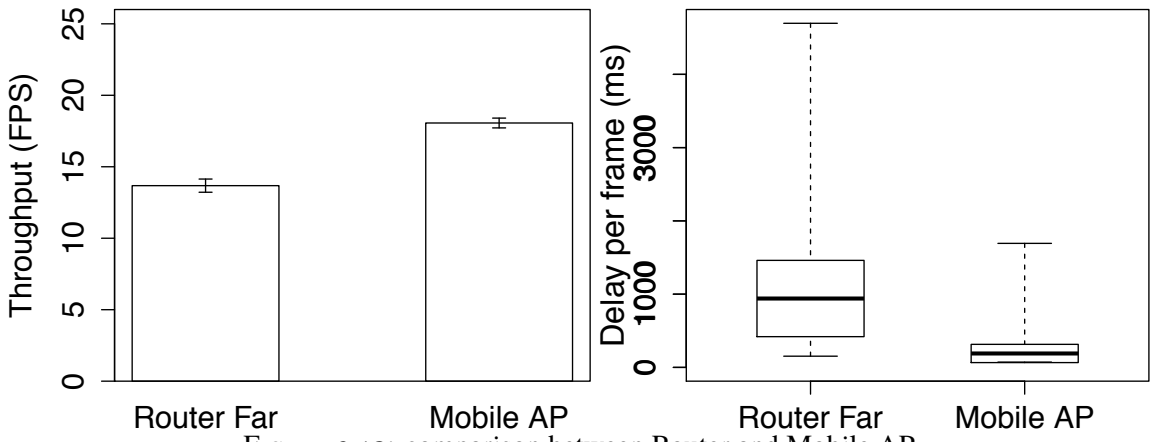

FIGURE 3.12: comparison between Router and Mobile AP

scheme. In the midst of computation, we manually terminate the Swing thread on G. Upon detecting the lost connection, the upstream traces the connection's downstream ID, removes the ID from the routing table, and re-calculates data routing probabilities for the remaining downstreams. Figure 3.10 indicates that real-time throughput drops drastically after the device leaves and before the system updates the routing table. The upstream attempts to route data to the disconnected device and, during the recovery phase, 13 frames are lost. Yet, within one second, throughput recovers to 16 FPS, which is the best that the remaining devices can achieve.

3.6.4 Mobile Hotspot

Suppose all users move and increase their distance from the WiFI access point. The transmission rate would be low even if the devices could communicate with each other. In this setting, a mobile hotspot would perform better. We measure its performance advantage by placing a master, which serves as a mobile hotspot, and three worker devices in a location with weak WiFi signal. Figure 3.12 indicates that devices communicating via the distant router perform much worse than when they communicate via the hotspot. Thus, Swing can improve performance with a hotspot when public WiFi is insufficient.

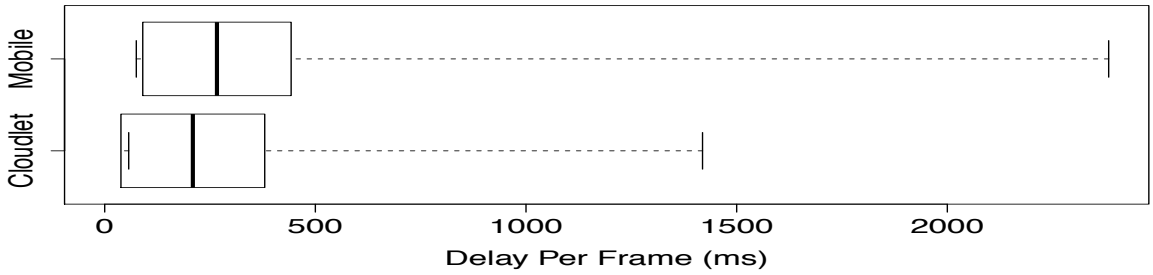

FIGURE 3.13: Comparison between cloud and mobile

3.6.5 Cloudlet Mode

Although Swing targets at application scenarios where the cloud or cloudlet is not accessible, Swing also supports offloading computation from mobile devices to server machines, with the help of an Android virtual machine, Here we compare the performance of mobile distributed computing against a cloudlet-like solution. We use Genymotion Android-x86 emulator as a virtual machine running on a desktop located in the campus network. The virtual machine is configured with two cores and 512MB of memory and is connected to the Internet through bridged Ethernet. The master phone connects to this virtual phone through a campus WiFi router and then the campus LAN. The virtual phone can easily process the input data sent from the master phone at 24 FPS. The processing delay, 30ms, is several times smaller than the processing delay on the phone. However, Figure 3.13 shows that the average end-to-end delays of processing on the desktop and the phones (eight devices, LRS) are comparable. For this application, even though the processing delay on the desktop is much smaller, the transmission delay fills up the gap, and the overall difference is negligible and less than 100ms on average.

3.7 Conclusion

We propose Swing, a framework that enables compute-intensive and delay-sensitive mobile sensing applications, by collaboratively using multiple mobile devices. The framework uses a stream computing model, allowing devices to perform computations based on a dataflow

graph. We identified the major challenges for achieving the performance potential and propose resource management and routing techniques for coping with device heterogeneity and dynamics. We built a system prototype using Android devices and a Java-based stream processing platform. We demonstrated the programmability of the framework API by porting two sensing applications onto Swing. By evaluating with multiple mobile devices on a wireless testbed, we show that performance requirements and efficiency can be satisfied through device management with minimal computational overhead.

4

Sensory Offloading for Wearable Devices

New generations of wearable devices enable comprehensive sensing and computing, with capabilities from simple step detection to complex gesture recognition. However, limitations in the devices' battery capacities constrain these applications' future. We find that continuously executing activity detection on a smart watch (*e.g.*, Samsung Gear Live) drains the battery in less than six hours. This leads to *"range anxiety"* — users are reluctant to use continuous sensing apps for fear that the watch runs out of battery during the day. These apps include long-running activity recognition services such as fitness tracking, sports training, and health monitoring.

An opportunity lies in typical mobile device usage patterns — a user often carries both a wearable device and a phone simultaneously (*e.g.*, during commute). For example, a watch's activity tracking app continuously records users' daily activities (*e.g.,* walking, cycling and running) and displays the number of calories burned. For this app, the phone can act as a *"range extender"* by taking over app execution and preserving the watch's energy. Our key observation is that when the user is carrying both devices, their sensory data are highly correlated and permits *sensory offloading, the idea of offloading not only computation but also sensing*.

Traditionally, offloading architectures shift computation from less capable devices (e.g., mobile) to more capable ones (e.g., server) that clone code and receive input data Cuervo et al. (2010). Unfortunately, continuous sensing apps cannot offload in this manner as they process high volumes of sensory data that would require prohibitively high transmission costs. To reduce data movement, wearable devices could pre-process or compress data but doing so would consume power and offset gains from offloading.

In this chapter, we present Telepath, a framework for sensory offloading that transfers app processing from a wearable device to a phone without communicating raw data. Telepath allows the phone to take over sensing and computation by *predicting the wearable's sensory data*. The phone deploys an app clone that reads predicted sensory data, computes activity recognition results, and sends them to the wearable. The wearable uses them as if they had been sensed and computed locally. Telepath decides whether to offload dynamically based on real-time correlation between device data. From the user's perspective, an app employs both wearable and phone's resources transparently.

Telepath estimates data with transfer function models that statistically capture the relationship between two devices' motions. With accurate models, the wearable can offload sensing and computing to the phone with little performance loss but significant energy savings. Yet Telepath encounters challenges. One model cannot capture all the dynamics between wearable devices and phones. Raw sensory data may vary, even for the same activity, due to device placement. We address these challenges with a modeling pipeline that segments sensing data and tailors models for different user activities and device placements. The following summarizes our contributions:

- **Motivation (Section 4.1).** We study the power breakdown of continuous sensing apps on wearable devices and show that traditional offloading schemes cannot mitigate sensing's high energy costs.

- **Abstractions (Section 4.1).** Telepath defines an application-layer shim that automatically

offloads sensing and computing tasks from wearables to a more capable device. We provide programmable abstractions that support cross-device apps without explicit management.

- **Design and Implementation (Section 4.3 - Section 4.4).** Telepath contains offline and online components. Offline, we build hierarchical transfer functions with sensing traces collected from both the phone and the wearable. Online, the phone uses these transfer functions to predict sensory data and classify activities.

- **Evaluation (Section 4.5 - Section 4.6).** We prototype Telepath in Android Wear and evaluate it on a Samsung Gear Live watch, with popular activities and a broad spectrum of recognition algorithms. We show Telepath achieves real-time performance, with 85% of the watch's accuracy on average while extending the watch's battery life by $2.1\times$.

4.1 Motivation

4.1.1 Activity Recognition on Wearables

Activity recognition applications ("apps" for short) capture human behavior with motion sensors such as accelerometers, gyroscopes, magnetic position sensors. Before phones were equipped with these sensors, researchers strapped sensors to specific body parts (e.g., arm and legs) to track motions online and recognize activities offline Mantyjarvi et al. (2001); Lukowicz et al. (2004); Laerhoven and Cakmakci (2000); Karantonis et al. (2006); Ravi et al. (2005); Intille et al. (2005). These "wearable" sensors were uncomfortable, which prevented broader adoption in practice Zhang and Sawchuk (2012).

The recent proliferation of smart phones led researchers to activity recognition algorithms that used phones' sensors Kwapisz et al. (2011); Brezmes et al. (2009); Khan et al. (2013); Fan et al. (2014); Shoaib et al. (2014); Brush et al. (2010). A typical approach extracted features from accelerometer or gyroscope signals and trained classifiers, such as support vector machines, for a set of activities Anguita et al. (2012b, 2013, 2012a). With continuous sensing and computing, phones classified activities and supplied feedback on

Table 4.1: Hardware specifications for typical watch, phone.

Specs	**Samsung Gear Live Watch**	**Samsung Galaxy Nexus Phone**
Processor	Quad-core 1.2 GHz (Snapdragon 400)	Dual-core 1.2GHz (ARM Cortex-A9)
Battery	300 mAh	1750 mAh
DRAM	512MB	1GB
Flash	4GB	16GB
Sensors	Accelerometer Gyroscope Compass Heart-rate Monitor	Accelerometer Gyroscope Compass

sporting progress or health status.

Today, a new generation of technology brings activity recognition apps back to wearable devices such as sports trackers, watches, and armbands Weiss et al. (2016); Bieber et al. (2013); Bhattacharya and Lane (2016); Shoaib et al. (2015). Unlike phones that might be placed or oriented in varied and obscure locations, such as pockets or purses, wearables are typically placed in a fixed location on the user, such as the left wrist. This unique property of wearables makes them better candidates for activity recognition, especially for special-purpose activities involving hands Ghasemzadeh et al. (2009); Asselin et al. (2005); Bächlin and Tröster (2012); Mortazavi et al. (2014). Recent smart watches provide attractive platforms for app developers, providing rich sets of sensors, capable systems-on-chip, and (most importantly) APIs that are compatible with those on phones.

4.1.2 Energy Consumption

Today's wearable devices are equipped with capabilities like those in phones. Indeed, Table 4.1 shows that a watch may have more processor cores than a phone. However, the watch has limited battery capacity due to its size. Figure 4.1 lists 34 mobile and wearable devices from the past three years and shows a 5-17$\times$ difference in watches and phones' battery capacities. Watches' short battery lives hinder long-running activity recognition

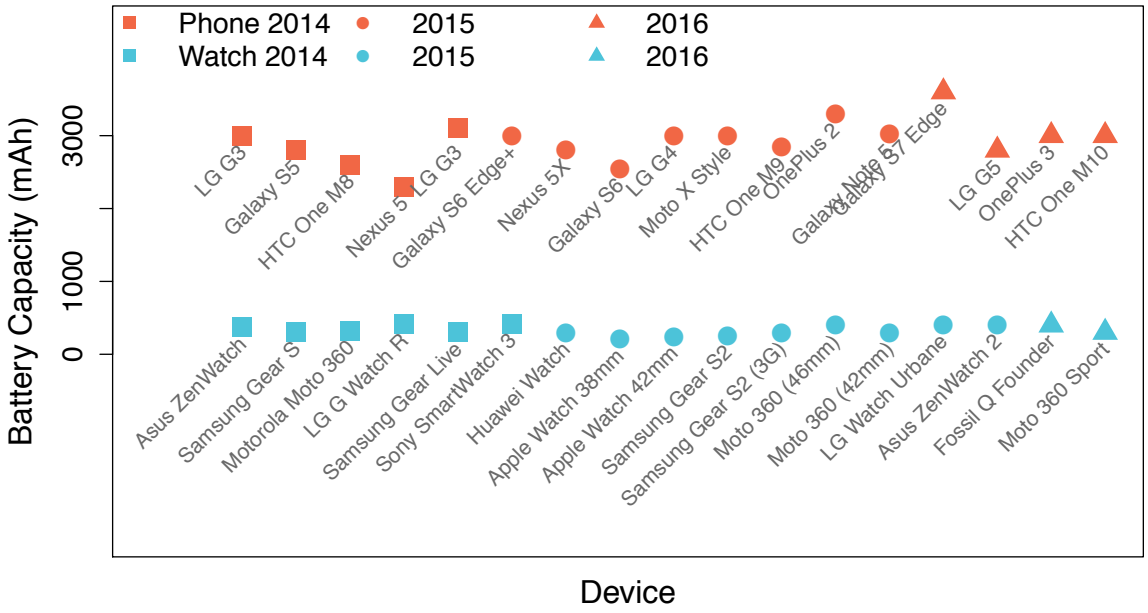

FIGURE 4.1: Battery capacity comparison. The average watch battery holds 330mAh, only 11% of a phone's 2924mAh.[1]

apps. Empirically, we show that sensing and computation rapidly consume battery charge in wearable devices. Moreover, we find that traditional offloading strategies are as expensive as the original computation.

Costs of Activity Sensing. Activity recognition apps have three components. First, the app reads data from sensors. Then, the app performs signal processing on raw sensory data and computes features that combine multiple sensor measurements. Finally, the app recognizes activities from extracted features using classifiers.

We profile three tasks to understand the power breakdown in an activity recognition app. In the first, the activity recognition app senses but does not process sensory data.

In the second, the app senses and extracts features such as the standard deviation of acceleration in each of three dimensions (see Table 3 for complete list). Features are computed once per second based on a 128-element array of doubles that hold raw sensory data. Finally, in the third task, the app performs all the computation needed for activity

[1] Ampere-Hour is often used to describe battery capacity. Actual energy is the product of ampere-hour and voltage, which is 3.8V in standard mobile li-ion batteries.

Table 4.2: Power consumption and battery life time comparison between watch and phone.

Task	**Samsung Gear Live Watch**	**Samsung Galaxy Nexus Phone**
Accelerometer Sensing	200.5 mW (5.7 hrs)	229.7 mW (29.8 hrs)
Sensing+Feature	217.7 mW (5.2 hrs)	231.8 mW (29.6 hrs)
Sensing+Feature+Classifier	239.2 mW (4.8 hrs)	267.0 mW (25.0 hrs)
Offloading Strategies		
Sensing + Bluetooth	261.6 mW (4.4 hrs)	
Sensing + Feature + Bluetooth	224.3 mW (5.1 hrs)	

recognition—sensing, feature extraction, and classification with a support vector machine.

We measure the power consumption of two mobile devices — a Samsung Gear Live smart watch and a Samsung Galaxy Nexus phone — for a variety of tasks. We turn off the display and run the task for an hour. For tasks that do not require wireless communication, we disable Bluetooth and WiFi. We measure average power during task computation using Android Batterystats.

Table 4.2 shows power consumption and estimated battery life when the device runs only its assigned task. For example, the watch dissipates 200.5mW when using the accelerometer. At this discharge rate, a fully charged watch battery is estimated to last 5.7 hours. Although the accelerometer only consumes 1.6mW InvenSense (2014), transporting data from sensor to system-on-chip consumes additional power. Moreover, sensing apps often request a wake lock that prevents the CPU from entering the idle state. Consequently, sensing is power-intensive.

The 1st-3rd rows present tasks associated with activity recognition. For a given task, the watch benefits from more energy-efficient hardware and consumes less power than the phone. However, the large gap in battery capacities – the watch's 1.1Wh versus the phone's 6.8Wh – means the watch battery lasts only 4.8 hours during activity recognition (sense + feature + classifier) whereas the phone battery lasts $5\times$ longer. This analysis is optimistic

because, in daily use, the watch's battery is often partially charged and is shared by other apps. Practical battery scenarios would further constrain long-running apps.

Costs of Offloading. For activity recognition tasks, traditional offloading reduces modest computational costs but incurs significant communication overheads. To illustrate this trade-off, we measure power when offloading computation from watch to phone, via Bluetooth, in one of two ways. First, the watch collects and sends raw sensory data to the phone. The phone computes features, classifies them, and returns recognition results to the watch. Alternatively, the watch collects sensory data, computes features, and sends features to the phone. The phone classifies features and returns results.

According to Table 4.2, the watch cannot benefit from either offloading strategy. Transmitting raw sensory data to the phone increases watch power due to high communication cost. Transmitting processed features to the phone only marginally reduces watch power due to high computational costs.

Traditional approaches that offload computation are unhelpful because wearables dissipate most of their power in sensing, not computation. To address this challenge, Telepath exploits correlated sensory data across multiple devices to "offload" both sensing and computation, eliminating communication between one device's sensors and another's system-on-chip.

4.2 Telepath Overview

The key observation behind Telepath is the correlation between the wearable and the phone's sensing streams. Figure 4.2 presents sensory data (with noise filtered) from a walking user who wears a Samsung Gear Live watch on her left wrist and places a Samsung Galaxy Nexus phone in her right pocket. When walking, her feet step while her hands swing. Thus, leg and arm motions are coordinated. In the top figure, a valley in the watch's x-axis acceleration always matches a valley in the phone's, albeit with slight time shifts. The

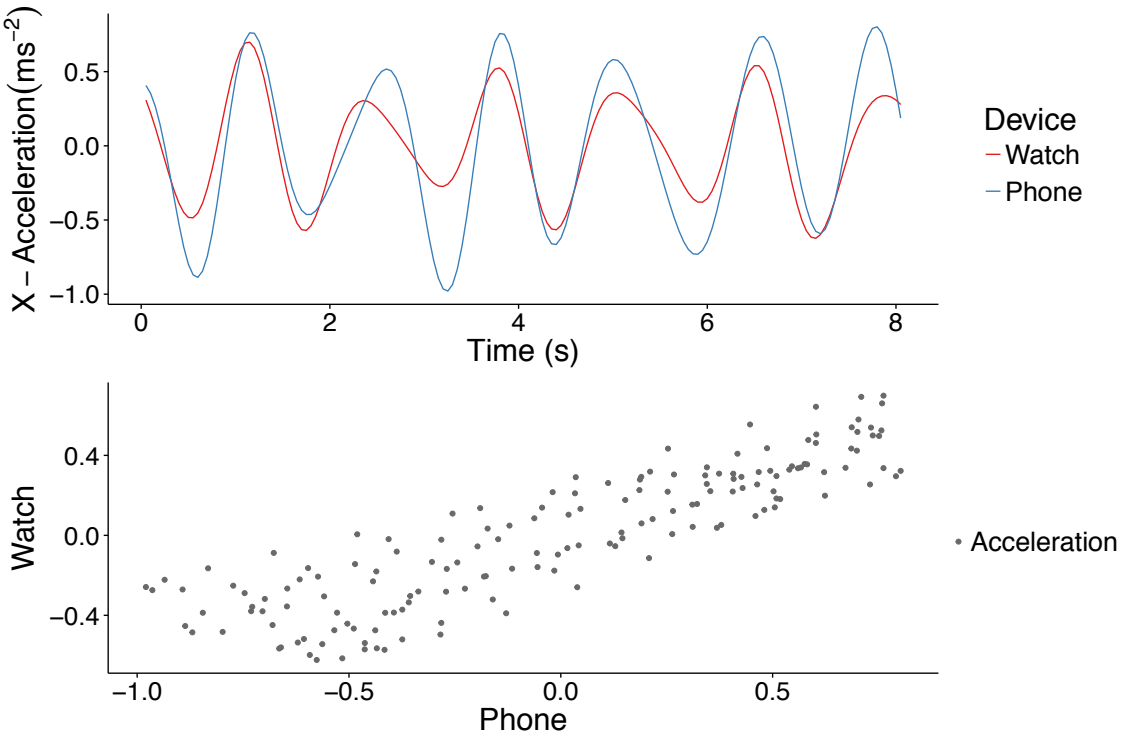

FIGURE 4.2: Sensory correlation between two devices

scatterplot indicates the two streams are highly correlated.

Telepath exploits this correlation to leverage the phone's sensors and avoid the expensive communication of the watch's sensory data. In this section, we outline the the framework's workflow at runtime. We then show how programmers can use Telepath in their apps.

Workflow. Figure 4.3 shows the workflow of Telepath. In local mode, TelepathWear interfaces to the wearable's OS, reading sensory data from local drivers and classifying features for activity recognition. In remote mode, TelepathWear neither senses nor computes. Instead, TelepathPhone reads sensory data from the phone, predicts sensory data for the watch, and recognizes activity using cloned computation. The result is sent back to TelepathWear for the app UI. Periodically, Telepath verifies predictor accuracy and halts remote execution when accuracy is poor.

Shim. Telepath is an application-layer shim that lies between the app and Android OS. A Telepath instance has two modules – TelepathWear and TelepathPhone – running on the

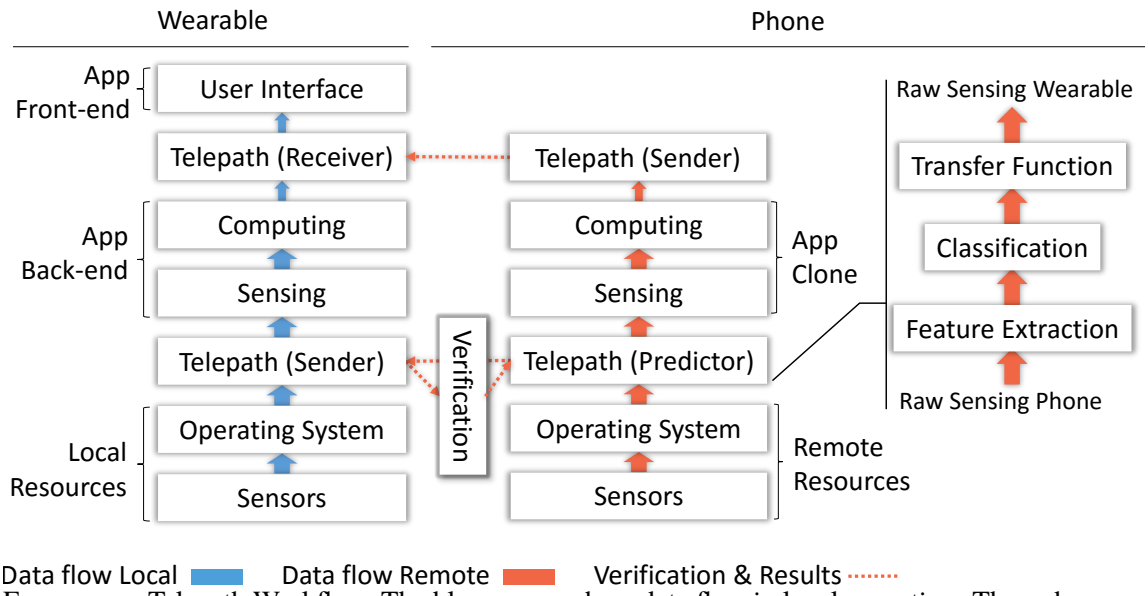

FIGURE 4.3: Telepath Workflow. The blue arrows show data flow in local execution. The red arrows show data flow in remote execution.

respective devices.

On the wearable's side, the app interfaces to Telepath Receiver, which receives results computed either locally on the wearable or remotely on the phone. The local app back-end includes computing and sensing. It receives data from Telepath Sender, which resides atop Android and requests offloading from the phone. When an offloading request is accepted, the Sender deactivates the local sensors. Otherwise, it receives sensing data from the local sensors and passes it to the upper layer.

On the phone's side, a Telepath Sender sends computed results to the wearable's Telepath Receiver. Results are produced by a clone of wearable app's back-end. A Telepath Predictor receives sensing data from the phone's local sensors, predicts corresponding sensor readings on the watch, and sends them to the cloned app for computation. The Predictor has three sub-modules — Feature-Extraction, Classification, and Transfer-Function — which we detail in Section 4.3.

A Verification module communicates with the phone's Telepath Predictor and the wearable's Telepath Sender. It compares predicted data and groundtruth data received from

the wearable to determine whether an offloading request should be accepted. The phone declines requests when Verification finds that the phone's predictions of the wearable's sensing data is inaccurate.

API. Telepath APIs extend Android's for interfacing with sensors. Developers can easily port their apps without changing the original code flow. We demonstrate an example below.

Suppose a developer designs a watch app that recognizes activities in a user's exercise routine. The app includes a trained support vector machine (SVM) that classifies activities based on standard deviation in the accelerometer's three axis' readings. The developer specifies the flow of data from sensors to classifier. First, the app registers the accelerometer's listener to receive sensor events (*i.e.*, readings) at the highest frequency. Upon receiving an event, the app computes the standard deviation σ of the current accelerations. SVM then uses σ to label activities.

To port her app to Telepath, the developer modifies the wearable app to include TelepathWear and creates a cloned app for TelepathPhone. The clone shares the same back-end code, including trained classifiers' binaries, as the wearable app. An app registers Telepath sensors like it would invoke the Android SensorManager API. The process is illustrated by Figure 4.4. Instead of computing on each sensing event, the app now listens to TelepathEvents, which could be local sensing events or remote result events.

The developer need not worry about offloading decisions or resource management. Because the phone predicts the watch's sensory data, recognition classifiers trained for wearables need not be re-trained for the phone.

4.3 Predictor Design

To predict the wearable's raw sensing data from the phone's, the predictor must first be trained offline. Offline, raw data from both devices are collected, processed, and clustered to develop phone-to-wearable transfer function models. Online, the phone's sensing data is

w/o Telepath

```
public class MainActivity extends WearableActivity
implements SensorEventListener {
    SensorManager sensorManager = (SensorManager)
getContext().getSystemService(SENSOR_SERVICE);
    Sensor mSensor = sensorManager.getDefaultSensor(
Sensor.TYPE_ACCELEROMETER);
    sensorManager.registerListener(this, mSensor,
SensorManager.SENSOR_DELAY_FASTEST);
    public void onSensorChanged(SensorEvent event){
        float[] accel = event.values;
        double feature = getStandardDeviation(accel );
        int label = (int) svm.predict(svmModel, feature);
        display(label);
    }
}
```

w/ Telepath

```
public class MainActivity extends WearableActivity
implements TelepathSensorListener{
    TelepathWear tp = new TelepathWear(context, this);
    tp.getTelepathSensor(Sensor.TYPE_ACCELEROMETER,
SensorManager.SENSOR_DELAY_FASTEST);
    public void onTelepathEvent(TelepathEvent event){
        int label = tp.getResults(event); //dynamic
        display(label);
    }
    @override
    public int compute(TelepathEvent event){
        float[] accel = event.values;
        double feature = getStandardDeviation(accel );
        int label = (int) svm.predict(svmModel, feature);
        return label;
    }
}
```

FIGURE 4.4: Code snippet for example wearable app, before and after integrating with Telepath.

fetched, processed, and classified to invoke the models that predict the wearable's sensing data. In this section, we introduce transfer function models. We then describe signal processing techniques that improve prediction accuracy.

4.3.1 Offline Training

Periodically, the phone and the wearable reads and uploads its sensing data to the cloud for offline training. The key training technique, transfer function modeling, is an autoregressive model that captures the relationship between two time series (a.k.a. ARMAX model).

Transfer Function Models

Transfer function modeling describes the relationship between two strongly correlated time series Helmer and Johansson (1977). A classic example of such time series is the relationship between advertising costs and sales. Let A_t and S_t be random variables that denote advertising and sales at time t. The general form of a transfer function is:

$$S_t = v_0 A_t + v_1 A_{t-1} + v_2 A_{t-2} + \cdots + N_t.$$

A_{t-1} is advertising cost at time $t-1$ and v_1 is the advertisement's effect after one time period. N_t is the sum of effects from all factors other than advertising and should be independent

of A_t. The function is often rewritten as

$$S_t = (v_0 + v_1 B + v_2 B^2 + \cdots) A_t + N_t, \quad \text{or}$$
$$S_t = v(B) A_t + N_t,$$

in which B is a backshift operator defined as

$$B A_t = A_{t-1} \quad \text{or} \quad B^m A_t = A_{t-m}.$$

and $v(B)$ is a transfer function. The target time series S_t (*i.e.*, the output) can be predicted from the current observation of A_t as well as the history of A_t (*i.e.*, the inputs). Given the nature of human motion, we can assume that our input and output time series are bounded, leading to stable transfer functions.

The output can be affected by not only present and past but also future values of the input such that $v_j \neq 0$ for $j < 0$. This effect arises because two devices can experience a time lag when sensing the same motion (e.g., hand moves first than the leg). It could also be caused by drifting clocks. To accommodate these data artifacts, prediction is performed periodically on buffered data to leverage multiple observations from an extended time period.

Training Transfer Function Models. Offline training identifies polynomial $v(B)$ and noise N_t. First, based on training data A_t and S_t, the impulse response weights of the transfer function, $v_0, v_1, \cdots$, are initialized from the correlation coefficients of the two time series. Then, N_t is checked to determine whether it is white noise.

If N_t is not noise and exhibits some relationship with the time series, there is still information left to be captured by the transfer function. In this case, the coefficients of $v(B)$ are revised with maximum likelihood estimation. Training proceeds iteratively until the transfer function captures much of the variance in the time series and N_t resembles white noise. Online prediction estimates S_t from A_t given $v(B)$ and the forecast of N_t.

Segmentation for Stationarity. To train valid transfer function models, the time series must be stationary, which means their statistical properties (*e.g.*, mean, variance, autocor-

relation) do not change over time.[2] Stationarity is essential because it makes time series easier to predict.

Unfortunately, we observe that when users perform different motions, the sensing data presents distinct statistical properties. Consider leg motions when jumping and walking. These two activities are so different that data transformations, such as finite-order differencing or logarithmic operations, cannot produce a stationary time series that includes both jumping and walking.

On the other hand, each independent activity consists of regular and repetitive motions that produce stationary statistical properties. This finding implies that we must segment our data such that stationarity can be guaranteed within each segment of the time series.

Training Pipeline

In addition to segmenting the data and identifying transfer function models for each segment, Telepath's offline training contains additional steps to process the data and improve prediction accuracy. Fortunately, these steps can be parallelized to reduce the training time.

Interpolation. The phone and the wearable device could be equipped with sensors of different sampling rates. To match the timestamps from a pair of sensing streams, linear interpolation is the first data processing step. After interpolating the time series for every 20 millisecond window, the output is two aligned time series, one for the phone and one for the wearable. This process assumes that clocks in two devices are perfectly synchronized. This assumption is rather strong, but slightly unsynchronized clocks have little practical impact.

De-noising. Data streams from motion sensors are inherently noisy. In addition, the devices' unstable placements (e.g., in the pocket) generate noise. To extract meaningful motion from noisy data, we use a Butterworth filter on the Fourier domain to remove high frequency noises. Our heuristic cutoff frequency is based on the observation that meaningful motions cannot have a frequency higher than 5Hz.

[2] Strictly speaking, this is the definition of weak stationarity but it suffices for our purposes.

Pre-whitening. Telepath uses the phone's data as inputs A_t into transfer functions and the wearable's data as output S_t. Each device has multiple data streams from various sensors and each stream is multi-dimensional (*e.g.*, three-axis acceleration). Instead of predicting each axis separately, Telepath employs multivariate transfer functions that span dimensions and sensors.

For instance, to predict the x-axis accelerometer reading in the wearable, all x-, y- and z-axis accelerometer readings from the phone are used. In addition, all three-axis gyroscope readings from the phone are used. Since these six variables are highly correlated by nature, we remove redundant information by calculating the autoregressive function for one variable and filtering it out from the other variables. The remaining series are then used for the transfer function modeling. This process is referred to as pre-whitening.

Segmentation - Feature Extraction. We extract features that commonly represent motion in time and frequency domains (see Table 4.3). Features are extracted on sampling windows. Each window has a width of 128 data points (i.e., a duration of 2.56 seconds), and they overlap with each other by 50%. This configuration matches the speed of human movement Anguita et al. (2013).

Segmentation - Clustering. To segment sensing streams collected from two devices, we classify data points into clusters. Each cluster presents a set of motions, during which the two devices share a unique correlation that differs from those in other clusters. However, the clusters need not have semantic meaning. We use unsupervised clustering, *k-means*, to cluster values for extracted features into $k = 10$ number of clusters. The number of clusters represents not only different motions, but the same motion with different device placements. Each cluster produces a stationary time series.

Transfer Function Modeling. Finally, we identify one transfer function for each cluster using the algorithm explained earlier. The transfer function is re-trained if clusters change. The training dataset should be large, diverse, and adaptive to capture the most recent relationships between sensors on two devices. When their batteries permit (e.g., charged

Table 4.3: Selected features for clustering.

ID	Description
1	Standard deviation of x-axis acceleration
2	Standard deviation of y-axis acceleration
3	Standard deviation of z-axis acceleration
4-6	Median of absolute values of i-axis, $i \in \{x, y, z\}$
7-9	Interquartile Range of i-axis
10-12	Coefficients of auto-regression of i-axis
13-15	Largest frequency component in the spectrum of i
16-18	Weighted frequency average of i
19-21	Spectrum skewness of i

to more than 80% of capacity), wearables and the phone record paired time series that contain raw sensory data. When charging, the phone sends this data to servers that train transfer functions. Periodically, the phone downloads updated functions and organizes them according to clusters' centroids.

4.3.2 Online Prediction

Verification of Model Accuracy. The online process predicts the raw sensing data on the wearable from the raw sensing data on the phone. In practice, the user may not be carrying both devices at the same time, precluding the use of Telepath. Therefore, the online process must include a verification stage, assuring that the current placement of the devices are suitable for data prediction.

Prior to, and in the midst of offloading, a small set of sensing data is periodically collected and sent from the wearable to the phone. If the data does not correlate with the data on the phone, Telepath informs the wearable that offloading is canceled and the wearable proceeds with local execution mode.

To determine if the offloading is suitable, Telepath estimates the correlation between the pair of sensing series collected from both devices. Traditionally, Pearson's correlation coefficient is used to measure the correlation of two variables. However, this metric requires

the variables to be approximately normal. Our sensing data series are generated from human motion and normal distributions on the data cannot be assumed.

Instead, we use Kendall's rank correlation coefficient to measure the extent that ranks of the two series match each other. This metric does not have requirements on the distributions of the two series. To quantify the correlation, the two series are filtered to remove noise and transformed into rankings. Then, the Kendall Tau-b coefficient is calculated, which accounts for concordant, discordant, and tied pairs in two series McLeod and McLeod (2011). A corresponding p-value is computed on a two-sided test of the null hypothesis, *H0: the two time series are not correlated*, and is compared against a 0.01 significance level.

If $p \geqslant 0.01$, we do not reject the null hypothesis and do not have sufficient confidence that the phone's data can predict the wearable's. Perhaps the phone and the wearable are not carried by the user simultaneously or the phone is vibrating for an incoming phone call. In such cases, the phone informs the wearable of its limitations and the wearable cancels the request for offloading.

Prediction Pipeline. When the phone accepts a request for offloading, it predicts the wearable's sensing data stream in several steps.

During feature extraction, the phone's real-time sensing data enters a buffer of width 2.56s and overlap 1.28s. The buffer is configured to match the sampling window during training. When the buffer is full, its data is processed to produce features (see Table 4.3). Notice that buffering introduces a delay in predicting the data, which does not affect activity tracking apps. Moreover, the buffer size can be tuned to match with app requirements.

During classification, because multiple transfer functions are generated for different clusters of motion, the prediction process must identify the cluster corresponding to the measured motion. Extracted features are classified and assigned to a cluster based on their distance to cluster's centroids. Then, the cluster's corresponding transfer function model is fetched.

During prediction, the transfer function estimates the wearable's raw sensing data from

the phone's. Although models are trained with filtered data, in which random noise is removed, models are evaluated with raw sensing data that includes noise. Noise in model inputs produce noise in model's outputs. In practice, because motion recognition apps remove noise in their first processing step, apps can accommodate noisy sensor data and noisy predicted data in the same manner. Ultimately, the apps' results are unaffected.

4.4 Implementation

4.4.1 Predictor Implementation

Offline. We implement the offline training pipeline in R. In particular, we used `approx()` in the stats packages for linear interpolation, `butter()` and `filtefilt()` in the `signal` package for de-noising, `kmeans()` in `stats` and `cl_predict()` in `clue` for clustering and classification, and `auto.arima()` and `forecast()` in the `forecast` package for transfer function learning and prediction.

On a 48-core AMD Opteron processor, training on a pair of time series with 65k data points and 21 extracted features completes in 6.6 minutes. The predictor module exports trained transfer functions as model coefficients written in json files. The module exports k-means clusters as their centroids in text files.

Online. On the phone, the Telepath instance downloads from the cloud the files containing transfer functions' coefficients and cluster centroids and stores them locally, on a daily basis. When sensing is offloaded, Telepath loads the files with the `org.json` library at runtime. Given sensing data features and a set of centroids, the closest cluster is identified and the appropriate transfer function model is selected. The prediction function is implemented in Android based on the R `forecast` library.

4.4.2 Runtime Implementation

A Telepath app has two instances, TelepathWear and TelepathPhone, which run on the respective devices. They implement separate modules to manage interfaces and offloading

decisions.

TelepathWear. For sensing apps, TelepathWear sends a 1-second buffer of data to the phone every 15 minutes to determine if offloading is suitable. If the phone rejects the request for offloading, TelepathWear executes locally. Otherwise, it starts (or continues) remote execution.

During local execution, TelepathWear registers local sensors, listens to sensor events, and receives sensory data. TelepathWear calls the compute() function defined by the developer and classifies activity before returning results to the UI. When local execution halts, TelepathWear unregisters sensors and stops sensing.

During remote execution, when an offloading request is accepted, TelepathWear sends a "start" command, along with sensor IDs, their sampling frequencies and the frequency that results are sent from the phone. Instead of listening for sensor events, TelepathWear listens to incoming data events over Bluetooth, receives classification results from the phone, and returns results to the app UI. When a remote execution halts, TelepathWear sends a "stop" command, with sensor IDs, instructing the phone to stop sensing and computing.

TelepathPhone. TelepathPhone is a background service that runs continuously, monitoring the data link and listening for commands from TelepathWear. If the command is "bootstrapping," it receives and stores app configurations and models. If the command is "start," it receives sensor IDs and frequencies that are used to begin sensing and computing. It sends results to TelepathWear periodically based on the configured result frequency. If the command is "stop," TelepathPhone unregisters the sensors and stops both sensing and computing.

TelepathPhone guards against inaccurate prediction and classification. Upon receiving an offload request, the phone obtains buffered sensing data from the watch and compares it against Telepath's prediction. Kendall's rank correlation is computed and the p-value is checked (see Section 4.3). Offloading requests are rejected when the p-value exceeds 0.01, which indicates that the phone is used in ways that preclude accurate sensing and prediction.

Table 4.4: Popular tracking apps on Android Wear.

App	Running	Walking	Cycling
C25K	✓		
Endomondo	✓	✓	✓
Google Fit	✓	✓	✓
Loseit		✓	
Map My Fitness	✓	✓	✓
Map My Hike	✓	✓	
Map My Ride			✓
May My Run	✓		
Map My Walk		✓	
MevoLife	✓		
Misfit		✓	
Runkeeper	✓		
Runtastic	✓	✓	✓
Under Armour Record	✓	✓	✓

Table 4.5: Benchmark classifiers

Classifier	Description	Category
C5.0	Decision tree boosting	Decision Tree
glm	Logistic regression	Linear Classifiers
knn	k-Nearest neighbor	Kernel estimation
lda	Linear discriminant analysis	Linear Classifiers
lssvmRadial	Least squares SVM	SVM
mlpWeightDecay	Multi-layer perceptron	Neural Networks
nb	Naive Bayes	Linear Classifiers
nnet	Single-layer perceptron	Neural Networks
parRF	Parallel randomforest	Decision Tree
svmRadial	SVM with radial basis kernel	SVM

4.5 Experimental Methods

Activities and Classifiers. Our evaluation studies the performance and efficiency of Telepath for sensing applications. To benchmark typical wearable device usage, we survey the characteristics of the top fourteen, free activity tracking apps on the wearable app market (see Table 4.4). They constantly check motion sensors, acquiring wake locks and motion

sensors in their permissions, to record running, walking, or cycling activities during the day.

We evaluate Telepath's accuracy in recognizing these three activities. Because we cannot know the algorithms used in these commercial apps, we test Telepath on ten popular machine learning classifiers (see Table 4.5). The classifiers are diverse and vary in complexity, from the simple decision tree to more complex neural networks.

Data. During data collection, a user wears a Samsung Gear Live watch on her left wrist and places a Samsung Galaxy Nexus phone in her right pocket. A user performs each activity—running, walking, cycling— for 15 minutes to produce approximately 270K raw sensing data points in total. Both devices record data at the highest sampling rate. The user labels data with corresponding activity to supply groundtruth. We sample 70% of the data for training and 30% for testing. We use an equal amount of data from each activity.

4.6 Evaluation

We compare Telepath against four alternatives with less sophisticated data and analysis: (1) Telepath without using gyroscope, (2) Telepath without using multivariate transfer function models, (3) Telepath without de-noising and (4) directly using the phone's raw sensing data without Telepath's prediction. Groundtruth is sensing data collected from the watch.

4.6.1 Prediction Accuracy

An error metric is needed to measure the accuracy of prediction. Traditional error metrics, such as root-mean-square error, measure the distance between the predicted multivariate time series and the groundtruth, which cannot accurately represent the effectiveness of Telepath. Human motion is not always coordinated. Take walking as an example, during which arms and legs have corresponding movements most of the time, but sometimes the leg moves faster than the arm and vice versa. If leg and arm motions do not match perfectly during online prediction, estimated watch data can lag groundtruth. If estimates differ from groundtruth only due to a small time shift, the distance can be large even when the model

Table 4.6: Confusion matrix of using knn to classify Telepath's prediction.

	Predicted Activities				
	biking	running	walking	total	recall
biking	**149**	0	22	171	0.871
running	0	**162**	6	168	0.964
walking	52	0	**116**	168	0.690
total	201	162	144	507	
precision	0.741	1.000	0.806		

can be considered accurate and sufficient for motion recognition.

Instead, we use dynamic time warping (DTW), a distance metric, to evaluate prediction accuracy. DTW pairs two time series, shifts and distorts them slightly in time, and finds the minimum distance between them across all possible alignments Giorgino et al. (2009). We allow a maximum shift of one second and calculate the distance between the predicted time series and groundtruth for the watch's sensing data. Both time series are de-noised.

The DTW distance is defined as

$$D(A_t, F_t) = \min_{\phi} d_{\phi}(A_t, F_t), \tag{4.1}$$

where d_ϕ is the accumulated distortion between time series A_t and F_t when they are applied warping functions $\phi(k) = \big(\phi_a(k), \phi_f(k)\big)$, for each time step k Giorgino et al. (2009).

Figure 4.5 shows the distance between sensing data predicted by Telepath and groundtruth. The y-axis is the DTW distance normalized to the length of the time series, and a smaller value means higher accuracy. Across activities, using Telepath significantly outperforms not using it. Not surprisingly, raw sensing data from the phone looks very different than that from the watch. Interestingly, Telepath without multivariate models and Telepath without de-noising have smaller distances to the groundtruth. However, a smaller distance is only one measure of fit and does not evaluate activity classification.

4.6.2 Classification Accuracy

We define "app accuracy" to measure the accuracy of activity classification when using predicted sensing data. Our metric is the F1 score. Given a classifier, for each activity i,

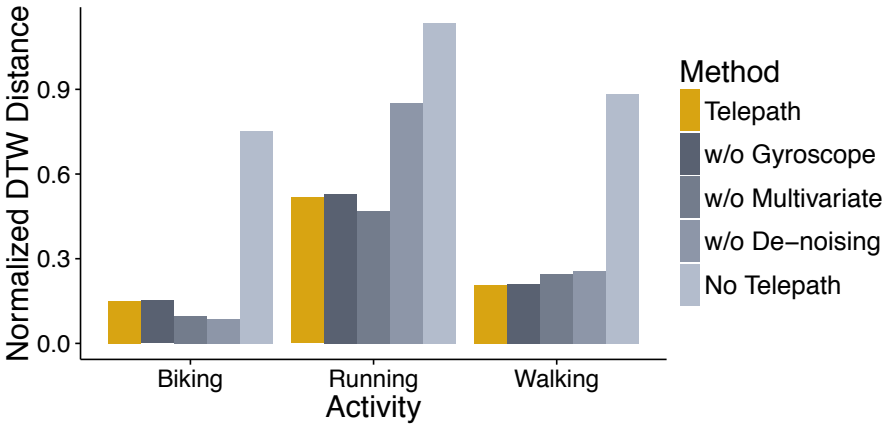

FIGURE 4.5: Data prediction accuracy, measured by DTW distance between predicted time series and groundtruth (smaller is better). Telepath has smaller distance than alternatives.

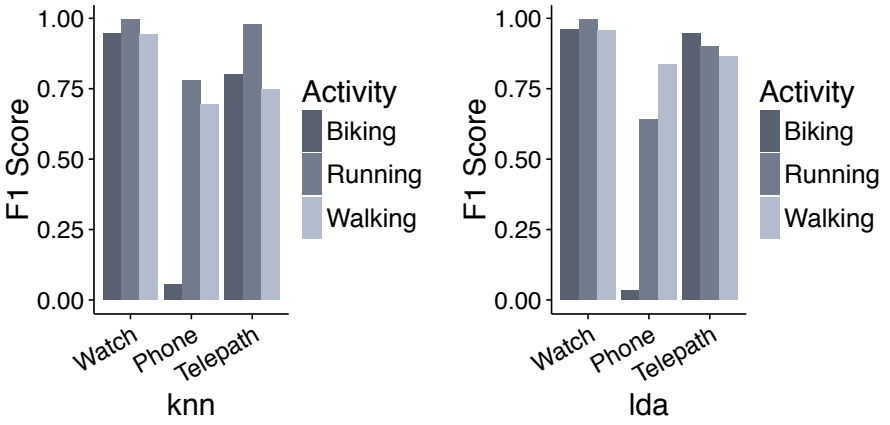

FIGURE 4.6: Activity classification accuracy, measured by F1 scores, for representative activities and classifiers.

Precision$_i$ is the fraction of instances classified as activity i that indeed correspond to i. Recall$_i$ is the fraction of instances of activity i that are recognized as such. A good classifier has both recall and precision, identifying more instances of an activity with fewer false positives. The F1 score accounts for recall and precision on a scale of zero (worst) to one (best):

$$\text{F1\_Score}_i = 2 \times \frac{\text{Precision}_i \times \text{Recall}_i}{\text{Precision}_i + \text{Recall}_i}.$$

For insight, Table 4.6 presents the confusion matrix for KNN classification for three activities when using Telepath's predictions. The bold numbers are correctly recognized

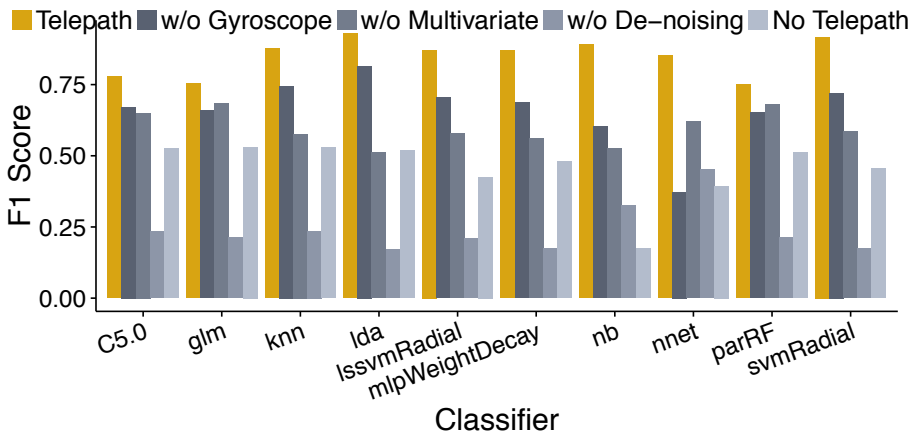

FIGURE 4.7: Activity classification accuracy, measured by F1 scores normalized to those when using groundtruth sensing data. Telepath classifies more accurately than alternatives.

activities. The first column shows 201 data points recognized as biking, but groundtruth says that 52 of them are actually walking and precision for biking is 0.74. Similarly, we calculate precision and recall for each activity.

In Figure 4.7, we evaluate app accuracy with the motion recognition classifiers in Table 4.5. The F1 Scores are averaged across activities and normalized to those when using groundtruth data, the watch's sensing data for activity recognition. Telepath outperforms other variants and achieves, on average, 85% of groundtruth accuracy. Observe that gyroscope data has a moderate impact on improving accuracy, while multivariate modeling is crucial for accurate data prediction and activity classification. Low scores without de-noising mean that Telepath is sensitive to noise and data pre-processing is required.

4.6.3 Step Counting Accuracy

Some motion tracking apps offer step counting as a service. We evaluate Telepath using an open source pedometer app Bagi (2011) and test data sampled from walking. The app detects and computes the lags between peaks and valleys in the sensed signals. If the lag exceeds a threshold, a step is detected. The threshold determines detection sensitivity—a low threshold detects more steps.

Figure 4.8 shows that the service produces similar step counts whether using the phone's

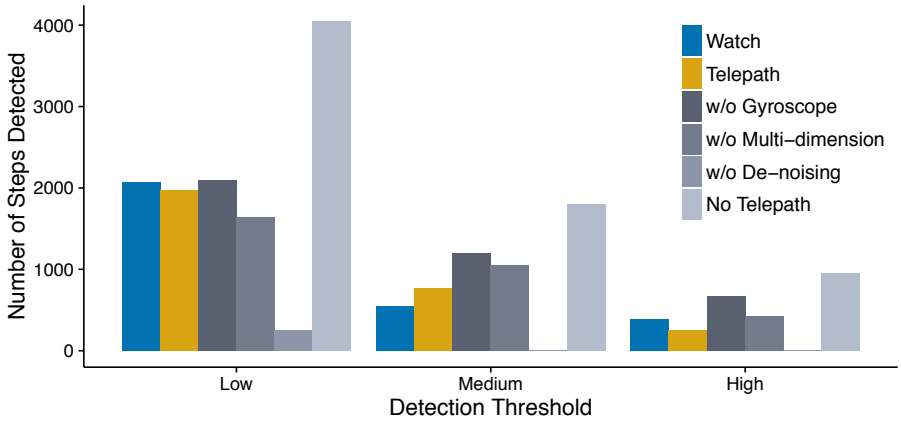

FIGURE 4.8: Step counting accuracy, measured by the number of steps, which decreases as the detection threshold increases. Blue bar shows groundtruth with wearable data. Gold bar shows Telepath estimates, which are closest to the watch's.

predictions of watch data or using the watch's groundtruth data. Telepath without de-noising has small step counts as it generates predictions with a much smaller variance. Noise obscures the correlation between the two devices, hindering the construction of accurate transfer functions.

On the other hand, not using Telepath but directly using the phone's sensing data results in high step counts. The readings on the phone have a much higher variance than those on the watch. A step counting model tailored for the watch cannot be directly applied to the phone.

4.6.4 Verification Accuracy

Verifying the feasibility of offloaded sensing and computing is essential to device management. Telepath permits offloading only when sensing streams on the phone and watch exhibit strong correlation. Otherwise, Telepath's phone will decline requests for offloading and Telepath's watch will rely on its local sensors. To assess correlation in sensing streams, Telepath computes Kendall's rank correlation on two de-noised time series. The conservative p-value is 0.01.

To evaluate verification, we collect three sets of sensing data. The first includes the

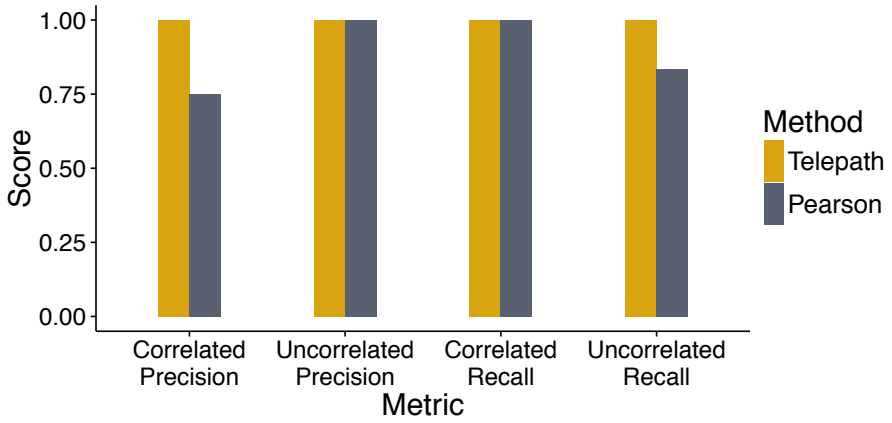

FIGURE 4.9: Verification accuracy, measured by the correlation between devices' data streams. A high score on recall for independence means Telepath abandons offloading when two devices' data are uncorrelated.

phone and watch's sensing streams, W_1 and P_1, collected when the user is walking. The second includes streams, W_2 and P_2, collected when the user is jumping. In these two datasets, the user wears both devices simultaneously. A third, W_3 and P_3, is collected when the devices are placed on a table.

We verify that W_1 is correlated with P_1, but neither P_2 nor P_3. Assessing correlation between W_1 and P_2 simulates a user walking with the watch on her wrist and the phone bouncing inside her backpack. Similarly, assessing correlation between W_1 and P_3 represents a user who carries the watch but leaves the phone in the office. W_i should be correlated with P_i if $i = j$ and uncorrelated otherwise.

Figure 4.9 shows that Telepath accurately recognizes correlated data streams. Compared with Pearson, Telepath's Kendall achieves high scores on recall for lack of correlation and on precision for correlation. Good recall for lack of correlation means that, when two devices' data are uncorrelated, Telepath rejects the watch's offload request with high confidence. Good precision for correlation means that, when Telepath accepts requests for offloaded sensing, the two devices' data are very likely correlated.

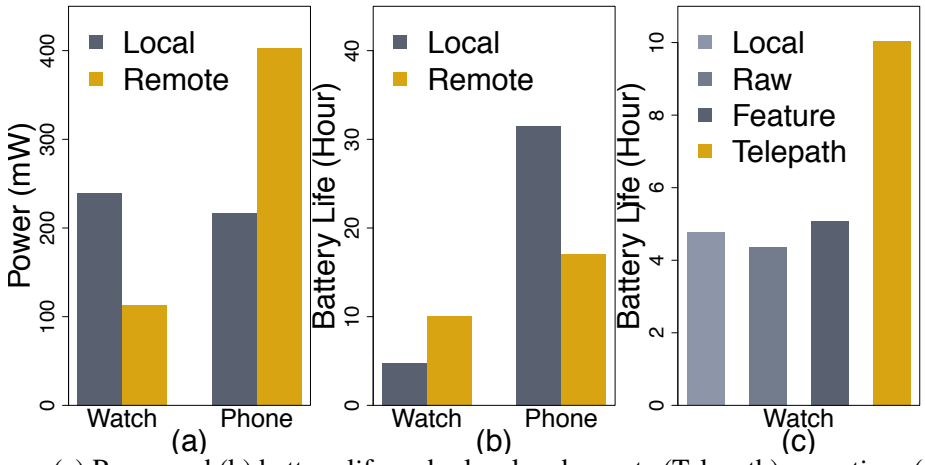

FIGURE 4.10: (a) Power and (b) battery life under local and remote (Telepath) execution. (c) Battery life under variants of remote execution that transmit raw data, transmit extracted features, or transmit nothing by relying on Telepath prediction.

4.6.5 Energy Efficiency

We compare system power in local and remote execution modes, profiling TelepathWear and TelepathPhone with Android Batterystats. We profile a Telepath app that recognizes biking, running and walking activities from accelerometer readings.

Figure 4.10(a) shows how watch power decreases by $2\times$ while phone power increases when switching from local to remote execution. The watch dissipates power during remote execution because it receives classification results from the phone every second (configured by app). Similarly, the phone dissipates power during local execution because it listens continuously for Telepath command.

Figure 4.10(b) shows that remote execution increases the watch's battery life from 4.8 hours to 10.0 hours, but reduces the phone's batter life to 17 hours. Note that Telepath could further increase the watch's battery life with asynchronous recognition. When activity logging apps are neither real-time nor interactive, the phone can buffer and send classification results less frequently to reduce transmission costs.

Figure 4.10(c) shows that Telepath's offloading strategy benefits battery life more than traditional strategies. Invoking the phone for both sensing and computing extends the

wearable's battery life to 10 hours. In contrast, relying on the watch's sensors but transmitting raw data or extracted features to the phone for computation does little for battery life. Telepath outperforms these strategies by 2.3x and 2.0x, respectively. Indeed, these strategies perform no better than using the watch's local resources.

Transmitting data or features to the phone for computation is ineffective as increased communication costs outweigh reduced computation costs. However, invoking the phone's sensors and transfer function models dramatically extends the wearable's battery life. Thus, Telepath addresses range anxiety by using the phone as a range-extender for wearables.

4.6.6 Costs and Overheads

Training. Telepath models must train quickly to permit frequent updates. However, a tradeoff exists between training time and prediction accuracy. Training time increases with the dataset size (Figure 4.11A). When the dataset has 6.5k paired data points, Telepath takes roughly 7 minutes to train models for 10 clusters. But increasing the dataset size significantly increases app accuracy (Figure 4.11B).

Training time can be reduced by increasing the number of clusters (Figure 4.11C). Each cluster trains in parallel with a small dataset per cluster. Increasing the number of clusters improves app accuracy as more transfer functions are generated (Figure 4.11D).

Prediction. During remote execution, the transfer function predictor on the Samsung Galaxy Nexus phone requires 3.42 milliseconds, on average, to compute a 21-dimensional feature vector, classify, and predict. This prediction latency is shorter than the sensors' 10ms sampling period, which means the predictor may be invoked for every sensor event. In practice, a buffer holds sensory data and only periodically (e.g., once per second) performs computation to process and predict features. Prediction costs do not affect app performance.

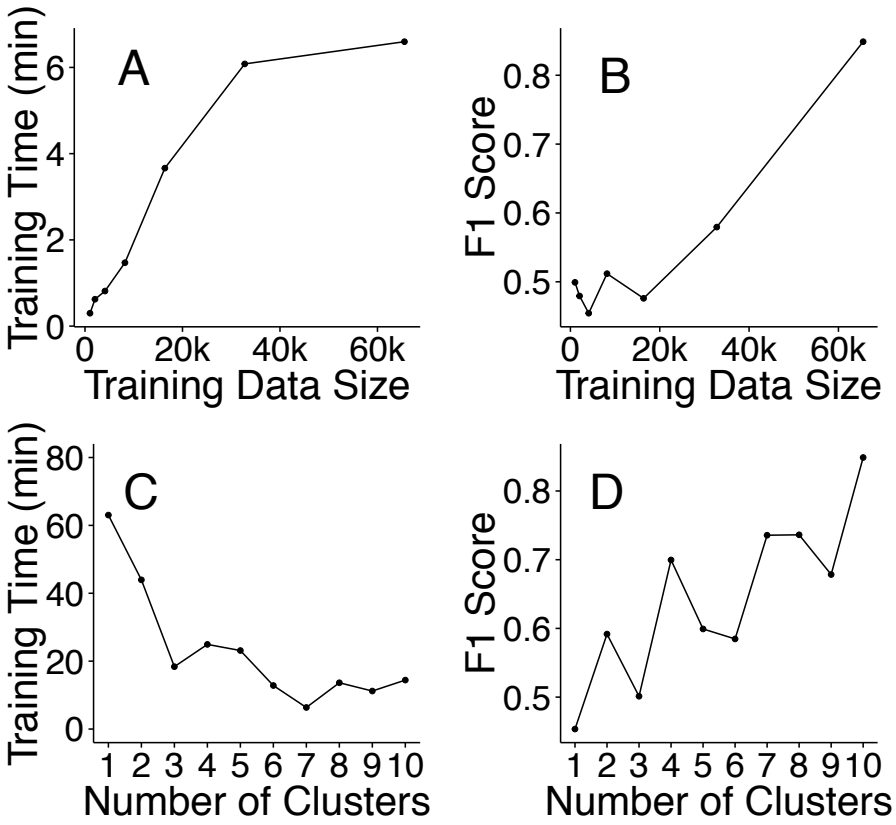

FIGURE 4.11: Impact of training data size and the number of clusters on the training time and app accuracy.

4.6.7 Sensitivity to Device Placement

A transfer function learned from one device placement may be inaccurate when placement changes. We test two device placements—phone in right and left pocket—while the watch is worn on the left wrist. We compare app accuracy when model is (1) trained with right pocket and tested with left pocket; (2) trained and tested with left pocket; (3) trained with left and right pockets but tested with left pocket.

Figure 4.12 shows, unsurprisingly, that transfer functions trained from one placement cannot accurately model another. Testing and training with different placements achieves only 71% of the accuracy when testing and training with same placements. However, models can be made resilient by enriching the dataset. When training includes both left- and right-pocket placements, accuracy increases by 44%, on average. Telepath should periodically

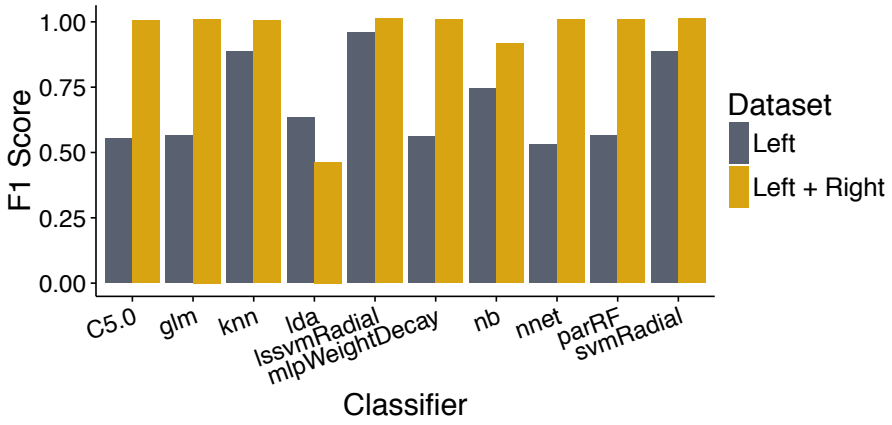

FIGURE 4.12: Activity classification accuracy when using datasets that differ in device placement.

update datasets and re-train predictors for new placements.

4.6.8 Sensitivity to Users

Heterogeneous users might share transfer function models, which would accelerate model training. Exploring this possibility, we compare two training datasets. A homogeneous dataset contains sensing data from user 1's walking, running, and biking. A heterogeneous dataset mixes user 1's walking and biking with user 2's running. Both users wear the watch on the left wrist and place the phone in the right pocket.

Figure 4.13 shows that transfer functions learned from the heterogeneous dataset, which has incomplete training data from user 1, cannot accurately predict user 1's watch data. App accuracy drops by 83% compared to apps that use models of user 1's complete dataset. Differences in body motion and device usage may require different transfer functions. When the user population is small, Telepath may require tailored models for individual users. As the population grows, however, users may naturally fall into clusters characterized by similar motions. Crowd-trained models are interesting avenue for future work.

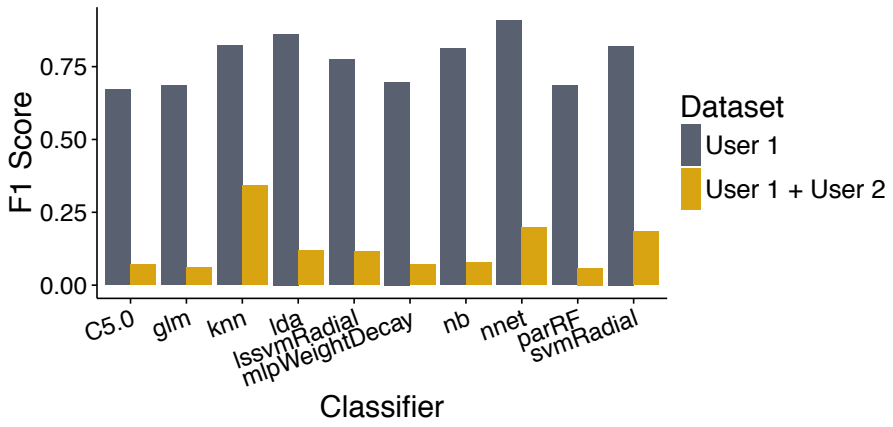

FIGURE 4.13: Activity classification accuracy using a homogeneous dataset with user 1 and a heterogeneous dataset with user 1 and 2. Shared training data does little to improve prediction accuracy for any one user.

4.7 Related Work & Discussion

Offloading has been extensively studied Shiraz et al. (2013); Dinh et al. (2013); Fernando et al. (2013) for mobile-cloud computing (MCC). Mobile devices offload compute-intensive apps—image, video, audio processing, gaming—to the cloud to reduce execution time, reduce energy, or improve service quality Hauswald et al. (2015); Balan et al. (2007); Flinn et al. (2002); Su and Flinn (2005); Chun and Maniatis (2009). Some MCC frameworks migrate virtual machines Satyanarayanan et al. (2009); Chun et al. (2011) while others partition applications Cuervo et al. (2010); Kristensen (2010). Offloading decisions are based on run-time analyses of network latency, transmission data size, devices' energy models, and code execution time Cuervo et al. (2010); Chun et al. (2011); Kristensen (2010); Zhang et al. (2011); Kumar and Lu (2010); Wang and Li (2004).

Telepath adapts offloading to extending battery life for wearable devices. For sensing apps, prohibitively high communication costs preclude traditional offloading. In response, we leverage cross-device correlation to "offload" sensing and minimize data transmission. In future, as Telepath aggregates sensing and computing from multiple devices, coordinated sensing with lower and higher measurement frequencies on wearables and phones, respec-

tively, could permit fine-grained device management. Policies for duty-cycling sensors could consider Telepath's impact on aggregate battery life of both devices.

Advances in hardware greatly improve the energy efficiency of mobile devices Halpern et al. (2016). With asymmetric multiprocessing, mobile devices can equip low-power cores to handle long-running sensing tasks that are not compute-intensive, leaving high-power cores asleep Lukefahr et al. (2012b); Gaudette et al. (2016). Sensing apps can benefit from accelerators that reduce the power and latency in signal processing and activity classification Mahajan et al. (2016); Arnau et al. (2012); Manatunga et al. (2015); Rajovic et al. (2013). Telepath is orthogonal to these approaches as it allows wearable devices to preserve energy and utilize idle resources on phones, however they are implemented.

Admittedly, Telepath is constrained to motion sensors such as accelerometers and gyroscopes. Wearables are equipped with other sensors such as GPS and heart rate monitors, which do not yet benefit from Telepath. GPS sensing can be offloaded without Telepath's sophisticated signal processing Liu et al. (2012). Heart monitoring requires more comprehensive learning and training, as there is no correlated sensor on the phone.

4.8 Conclusions

We design and implement Telepath to improve the lifetime of wearable devices for sensing applications. When users carry a phone and watch together, Telepath uses only the phone's sensory data to accurately predict the watch's sensory data by exploiting correlation between devices' motions. Telepath is triggered when the correlations between two devices are significant, and the phone's predictions extend its battery life by reducing the watch's sensing demands. Telepath accurately predicts the watch's sensing data with 85% app accuracy while extending the watch's battery life by 2.1x.

5

Conclusion and Lessons Learned

This thesis presents three pieces of works that improves the energy efficiency of mobile sensing and computing. There is room for improvement for each work. The first work focuses on individual devices and explores the design space of mobile processors, leaving the scheduling policies as future work. The second work focuses on a distributed system that utilizes multiple devices. There exists a rich set policies to be applied to the system to manage the devices for performance, energy and fairness targets. The third work focuses on the signal processing approach to enable sensory offloading. A scheduler can be designed to provide trade-offs between performance and energy according to user preferences. Overall, the works presented in this thesis emphasize more on developing platforms and less on managing them.

Throughout the research process, many lessons have been learned, from problem finding, to problem formation, to problem solutions. System research should not be limited to, or constrained by specific hardware platforms. Indeed, research works that build on the invention of new, "trendy" hardware platforms are easy to publish papers on. New platforms often bring new problems, accelerating the problem finding phase. However, the approaches taken to solve these problems should not be constrained to the very platform. Often, we

are attempted to make assumptions about hardware, their computing bottlenecks and their power limitations, and format our problem statements on top of such assumptions. And very often, we are told that these assumptions do not hold in the real world, or do not hold any more. For example, mobile devices evolve so fast, that an assumption on the hardware might not hold for longer than six months. My entire thesis, for instance, might turn out to be completely useless, if someone on the other end of the earth makes a breakthrough in the li-ion technology that improves today's battery capacities by two-hundred fold (I will have to revisit this number after doing some research). Or better, by harvesting energy from the ambient, your phones will never run out of battery. The energy constraint will be gone, together with the meanings of all the papers that rely on it. This might not happen (yes, I am still holding on to my assumptions), but in an extreme case it does, I will leave the judgment for my readers.

A good system research work should able to separate from the platform. Similarly, an approach that seems overly complicated for a certain problem might serve a great purpose somewhere else. My own inspirations come from software — applications, workloads, essentially users. A software is always fairly bounded to the hardware, but also relatively independent. But, when using workloads as benchmarks, one should again, always be aware of the mind trap: our brains seem to be fond of making assumptions. One should, at least, be aware that an assumption is being made, and it might not hold.

Appendix A

Mobile Trend Survey

Table A.1: Hardware specifications of mobile devices from 2012 to 2016. Data collected from Wikipedia. Observe series like Samsung Note*, LG G* and Galaxy S* that upgrade every year.

Device	Release Year	Big Cores	Frequency (GHz)	Little Cores	Frequency (GHz)	64-bit	Memory (GB)	Battery (mAh)	System-on-chip	Manufacturer
iPhone 5	2012	2	1.3				1	1440	Apple A6	Apple
Galaxy S3	2012	4	1.4				1	2100	Samsung Exynos 4 Quad	Samsung
Droid Razr Maxx HD	2012	2	1.5				1	3300	Snapdragon S4	Motorola
One X	2012	4	1.5				1	1800	Nvidia Tegra 3	HTC
Lumia 920	2012	2	1.5				1	2000	Qualcomm Snapdragon S4	Nokia
Nexus 4	2012	4	1.5				2	2100	Qualcomm Snapdragon S4	LG
Galaxy Note2	2012	4	1.6				2	3100	Samsung Exynos 4412 Quad	Samsung
Note 3	2013	4	1.9	4	1.3		3	3200	Samsung Exynos 5 Octa 5420	Samsung
LG G2	2013	4	2.26				2	3000	Qualcomm Snapdragon 800	LG
Moto G	2013	4	1.2				1	2070	Qualcomm Snapdragon 400	Motorola
Moto X 1	2013	2	1.7				2	2200	QualcommSnapdragon S4	Motorola
Nexus 5	2013	4	2.26				2	2300	Qualcomm Snapdragon 800	LG
Galaxy S4	2013	4	1.6	4	1.2		2	2600	Exynos 5 Octa	Samsung
HTC One M7	2013	4	1.7				2	2300	Qualcomm Snapdragon 600	HTC
Optimus G Pro	2013	4	1.7				2	3140	Qualcomm Snapdragon 600	LG
BlackBerry Z10	2013	2	1.5				2	1800	BlackBerry	BlackBerry
Xperia Z	2013	4	1.5				2	2330	Qualcomm Snapdragon S4 Pro	Sony
iPhone 5s	2013	2	1.3			✓	1	1560	Apple A7	Apple
Lumia 1520	2013	4	2.2				2	3400	Qualcomm Snapdragon 800	Nokia
Galaxy S5	2014	4	1.9	4	1.3		2	2800	Samsung Exynos 5 Octa	Samsung
HTC One M8	2014	4	2.26				2	2600	Qualcomm Snapdragon 801	HTC
iPhone 6	2014	2	1.4			✓	1	1810	Apple A8	Apple
iPhone 6 Plus	2014	2	1.4			✓	1	1810	Apple A8	Apple
LG G3	2014	4	2.5				2	3000	Qualcomm Snapdragon 801	LG
OnePlus 1	2014	4	2.5				3	3100	Qualcomm Snapdragon 801	OnePlus
Nexus 6	2014	4	2.7				3	3220	Qualcomm Snapdragon 805	Motorola
Moto X 2	2014	4	2.5				2	2300	Qualcomm Snapdragon 801	Motorola
Note 4	2014	4	1.9	4	1.3	✓	3	3220	Exynos 7 Octa 543364-bit	Samsung
Nexus 6P	2015	8	2			✓	3	3450	Qualcomm Snapdragon 810 v2.1	Huawei
Nexus 5X	2015	6	1.8				2	2700	Qualcomm Snapdragon 808	LG
iPhone 6s	2015	2	1.85			✓	2	1715	Apple A9	Apple
iPhone 6s Plus	2015	2	1.85			✓	2	2750	Apple A9	Apple
Galaxy S6 Edge+	2015	4	2.1	4	1.5	✓	3	3000	Samsung Exynos 7 Octa 7420	Samsung
Galaxy S6 Edge	2015	4	2.1	4	1.5	✓	3	2600	Samsung Exynos 7 Octa 7420	Samsung
Galaxy S6	2015	4	2.1	4	1.5	✓	3	2550	Samsung Exynos 7 Octa 7420	Samsung
Note 5	2015	4	2.1	4	1.5	✓	4	3020	Samsung Exynos 7 Octa 7420	Samsung
LG G4	2015	2	1.82	4	1.44	✓	3	3000	Qualcomm Snapdragon 808	LG
Moto X Style	2015	6	1.8			✓	3	3000	Qualcomm Snapdragon 808	Motorola
HTC One M9	2015	4	1.555	4	1.958	✓	3	2840	Qualcomm Snapdragon 810	HTC
OnePlus 2	2015	4	1.77	4	1.56	✓	3	3300	Qualcomm Snapdragon 810	OnePlus
Galaxy S7	2016	4	2.3	4	1.6	✓	4	3000	Samsung Exynos 8890	Samsung
Galaxy S7 Edge	2016	2	2.15	2	1.59	✓	4	3600	QualcommSnapdragon 820	Samsung
LG G5	2016	2	2.15	2	1.59	✓	4	2800	Qualcomm Snapdragon 820	LG
OnePlus 3	2016	2	2.15	2	1.59	✓	6	3000	Qualcomm Snapdragon 820	OnePlus
HTC One M10	2016	2	2.15	2	1.59	✓	4	3000	Qualcomm Snapdragon 820	HTC
Note 7	2016	2	2.15	2	1.59	✓	4	3500	QualcommSnapdragon 820	Samsung

Bibliography

(2011), "Variable SMP – A multi-core CPU architecture for low power and high performance," in *NVIDIA White Paper*.

(2013), "NVIDIA Tegra 4 family CPU architecture: 4-PLUS-1 quad core," in *NVIDIA White Paper*.

Anguita, D., Ghio, A., Oneto, L., Parra, X., and Reyes-Ortiz, J. L. (2012a), *Ambient Assisted Living and Home Care: 4th International Workshop, IWAAL 2012, Vitoria-Gasteiz, Spain, December 3-5, 2012. Proceedings*, chap. Human Activity Recognition on Smartphones Using a Multiclass Hardware-Friendly Support Vector Machine, pp. 216–223, Springer Berlin Heidelberg, Berlin, Heidelberg.

Anguita, D., Ghio, A., Oneto, L., Parra, X., and Reyes-Ortiz, J. L. (2012b), "Human Activity Recognition on Smartphones Using a Multiclass Hardware-friendly Support Vector Machine," in *Proceedings of the 4th International Conference on Ambient Assisted Living and Home Care*, IWAAL'12, pp. 216–223, Berlin, Heidelberg, Springer-Verlag.

Anguita, D., Ghio, A., Oneto, L., Parra, X., and Reyes-Ortiz, J. L. (2013), "A Public Domain Dataset for Human Activity Recognition Using Smartphones," in *ESANN 2013 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*.

Annavaram, M., Grochowski, E., and Shen, J. (2005), "Mitigating Amdahl's Law Through EPI Throttling," *SIGARCH Comput. Archit. News*.

ARM (2012), "CoreTile Express A15x2 A7x3," in *ARM Technical Reference Manual*.

Arnau, J.-M., Parcerisa, J.-M., and Xekalakis, P. (2012), "Boosting mobile GPU performance with a decoupled access/execute fragment processor," in *ACM SIGARCH Computer Architecture News*, vol. 40, pp. 84–93, IEEE Computer Society.

Arslan, M. Y., Singh, I., Singh, S., Madhyastha, H. V., Sundaresan, K., and Krishnamurthy, S. V. (2012), "Computing While Charging: Building a Distributed Computing Infrastructure Using Smartphones," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pp. 193–204, New York, NY, USA, ACM.

Asselin, R., Ortiz, G., Pui, J., Smailagic, A., and Kissling, C. (2005), "Implementation and evaluation of the personal wellness coach," in *Distributed Computing Systems Workshops, 2005. 25th IEEE International Conference on*, pp. 529–535, IEEE.

Bächlin, M. and Tröster, G. (2012), "Swimming performance and technique evaluation with wearable acceleration sensors," *Pervasive and Mobile Computing*, 8, 68–81.

Bagi, L. (2011), "Open source Android Pedometer," `https://github.com/bagilevi/android-pedometer`.

Balan, R. K., Gergle, D., Satyanarayanan, M., and Herbsleb, J. (2007), "Simplifying cyber foraging for mobile devices," in *Proceedings of the 5th international conference on Mobile systems, applications and services*, pp. 272–285, ACM.

Banerjee, N., Corner, M., and Levine, B. (2007), "An Energy-Efficient Architecture for DTN Throwboxes," in *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, pp. 776–784.

Bao, X., Fan, S., Varshavsky, A., Li, K., and Roy Choudhury, R. (2013), "Your Reactions Suggest You Liked the Movie: Automatic Content Rating via Reaction Sensing," in *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '13, pp. 197–206, New York, NY, USA, ACM.

Bhattacharya, S. and Lane, N. D. (2016), "From Smart to Deep: Robust Activity Recognition on Smartwatches using Deep Learning," .

Bieber, G., Haescher, M., and Vahl, M. (2013), "Sensor requirements for activity recognition on smart watches," in *Proceedings of the 6th International Conference on PErvasive Technologies Related to Assistive Environments*, p. 67, ACM.

Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D., and Wood, D. A. (2011), "The Gem5 Simulator," *SIGARCH Comput. Archit. News*.

Brezmes, T., Gorricho, J.-L., and Cotrina, J. (2009), *Distributed Computing, Artificial Intelligence, Bioinformatics, Soft Computing, and Ambient Assisted Living: 10th International Work-Conference on Artificial Neural Networks, IWANN 2009 Workshops, Salamanca, Spain, June 10-12, 2009. Proceedings, Part II*, chap. Activity Recognition from Accelerometer Data on a Mobile Phone, pp. 796–799, Springer Berlin Heidelberg, Berlin, Heidelberg.

Brush, A. J. B., Krumm, J., and Scott, J. (2010), "Activity Recognition Research: The Good, the Bad, and the Future," .

Castro Fernandez, R., Migliavacca, M., Kalyvianaki, E., and Pietzuch, P. (2013), "Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pp. 725–736, New York, NY, USA, ACM.

Chang, J. (2013), "0xbench, intergrated Android benchmark suite by 0xlab," `http://code.google.com/p/0xbench/`.

Chun, B.-G. and Maniatis, P. (2009), "Augmented Smartphone Applications Through Clone Cloud Execution." in *HotOS*, vol. 9, pp. 8–11.

Chun, B.-G., Ihm, S., Maniatis, P., Naik, M., and Patti, A. (2011), "CloneCloud: Elastic Execution Between Mobile Device and Cloud," in *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pp. 301–314, New York, NY, USA, ACM.

Cuervo, E., Balasubramanian, A., Cho, D.-k., Wolman, A., Saroiu, S., Chandra, R., and Bahl, P. (2010), "MAUI: Making Smartphones Last Longer with Code Offload," in *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pp. 49–62, New York, NY, USA, ACM.

Dinh, H. T., Lee, C., Niyato, D., and Wang, P. (2013), "A survey of mobile cloud computing: architecture, applications, and approaches," *Wireless Communications and Mobile Computing*, 13, 1587–1611.

Dong, Z., Kong, L., Cheng, P., He, L., Gu, Y., Fang, L., Zhu, T., and Liu, C. (2014), "REPC: Reliable and efficient participatory computing for mobile devices," in *Sensing, Communication, and Networking (SECON), 2014 Eleventh Annual IEEE International Conference on*, pp. 257–265.

Dou, A., Kalogeraki, V., Gunopulos, D., Mielikainen, T., and Tuulos, V. H. (2010), "Misco: A MapReduce Framework for Mobile Systems," in *Proceedings of the 3rd International Conference on PErvasive Technologies Related to Assistive Environments*, PETRA '10, pp. 32:1–32:8, New York, NY, USA, ACM.

Fan, S. (2015), "ActionBench Source code," `https://github.com/ispass-anonymous/actionbench-`.

Fan, S. and Lee, B. C. (2016), "Evaluating Asymmetric Multiprocessing for Mobile Applications," in *Proceedings of the 2016 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '16, IEEE.

Fan, S., Gowda, M., and Roy Choudhury, R. (2012), "Poster: saving power for mobile phones with partial Wi-Fi scans," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pp. 509–510, ACM.

Fan, S., Shin, H., and Choudhury, R. R. (2014), "Injecting Life into Toys," in *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*, HotMobile '14, pp. 4:1–4:6, New York, NY, USA, ACM.

Fan, S., Zahedi, S. M., and Lee, B. C. (2016), "The computational sprinting game," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 561–575, ACM.

Fernando, N., Loke, S. W., and Rahayu, W. (2013), "Mobile cloud computing: A survey," *Future Generation Computer Systems*, 29, 84–106.

Flinn, J., Park, S. Y., and Satyanarayanan, M. (2002), "Balancing performance, energy, and quality in pervasive computing," in *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pp. 217–226, IEEE.

Forcada, M., Ginestí-Rosell, M., Nordfalk, J., O'Regan, J., Ortiz-Rojas, S., Pérez-Ortiz, J., Sánchez-Martínez, F., Ramírez-Sánchez, G., and Tyers, F. (2011), "Apertium: a free/open-source platform for rule-based machine translation," *Machine Translation*, 25, 127–144.

Gaudette, B., Wu, C.-J., and Vrudhula, S. (2016), "Improving smartphone user experience by balancing performance and energy with probabilistic QoS guarantee," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 52–63, IEEE.

Ghasemzadeh, H., Loseu, V., and Jafari, R. (2009), "Wearable coach for sport training: A quantitative model to evaluate wrist-rotation in golf," *Journal of Ambient Intelligence and Smart Environments*, 1, 173–184.

Giorgino, T. et al. (2009), "Computing and visualizing dynamic time warping alignments in R: the dtw package," .

Goulding-Hotta, N., Sampson, J., Venkatesh, G., Garcia, S., Auricchio, J., Huang, P., Arora, M., Nath, S., Bhatt, V., Babb, J., Swanson, S., and Taylor, M. (2011), "The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future," *Micro, IEEE*.

Guevara, M., Lubin, B., and Lee, B. (2013), "Navigating heterogeneous processors with market mechanisms," in *HPCA*.

Guevara, M., Lubin, B., and Lee, B. (2014a), "Strategies for anticipating risk in heterogeneous system design," in *HPCA*.

Guevara, M., Lubin, B., and Lee, B. (2014b), "Strategies for anticipating risk in heterogeneous system design," in *HPCA*.

Guo, S., Ghaderi, M., Seth, A., and Keshav, S. (2006), "Opportunistic scheduling in Ferry-Based Networks," in *In WNEPT*.

Gutierrez, A., Dreslinski, R., Wenisch, T., Mudge, T., Saidi, A., Emmons, C., and Paver, N. (2011), "Full-System Analysis and Characterization of Interactive Smartphone Applications," in *IISWC*.

Halpern, M., Zhu, Y., and Reddi, V. J. (2016), "Mobile CPU's rise to power: Quantifying the impact of generational mobile CPU design trends on performance, energy, and user satisfaction," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 64–76, IEEE.

Hauswald, J., Laurenzano, M. A., Zhang, Y., Li, C., Rovinski, A., Khurana, A., Dreslinski, R. G., Mudge, T., Petrucci, V., Tang, L., et al. (2015), "Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers," in *ACM SIGPLAN Notices*, vol. 50, pp. 223–238, ACM.

Helmer, R. M. and Johansson, J. K. (1977), "An exposition of the Box-Jenkins transfer function analysis with an application to the advertising-sales relationship," *Journal of Marketing Research*, pp. 227–239.

Horowitz, M., Alon, E., Patil, D., Naffziger, S., Kumar, R., and Bernstein, K. (2005), "Scaling, power and the future of CMOS," in *IEDM*.

Huang, Y., Zha, Z., Chen, M., and Zhang, L. (2014), "Moby: A mobile benchmark suite for architectural simulators," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pp. 45–54.

Huggins-Daines, D., Kumar, M., Chan, A., Black, A., Ravishankar, M., and Rudnicky, A. (2006), "Pocketsphinx: A Free, Real-Time Continuous Speech Recognition System for Hand-Held Devices," in *Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on*, vol. 1, pp. I–I.

Hui, P., Chaintreau, A., Gass, R., Scott, J., Crowcroft, J., and Diot, C. (2006), "Pocket Switched Networking: Challenges, Feasibility and Implementation Issues," in *Proceedings of the Second International IFIP Conference on Autonomic Communication*, WAC'05, pp. 1–12, Berlin, Heidelberg, Springer-Verlag.

Intille, S. S., Larson, K., Beaudin, J., Nawyn, J., Tapia, E. M., and Kaushik, P. (2005), "A living laboratory for the design and evaluation of ubiquitous computing technologies," in *CHI'05 extended abstracts on Human factors in computing systems*, pp. 1941–1944, ACM.

InvenSense (2014), "MPU-9250 Product Specification Revision 1.0," `http://43zrtwysvxb2gf29r5o0athu.wpengine.netdna-cdn.com/wp-content/uploads/2015/02/MPU-9250-Datasheet.pdf`.

Ipek, E., Kirman, M., Kirman, N., and Martinez, J. F. (2007), "Core Fusion: Accommodating Software Diversity in Chip Multiprocessors," in *ISCA*.

Kang, S., Lee, Y., Min, C., Ju, Y., Park, T., Lee, J., Rhee, Y., and Song, J. (2010), "Orchestrator: An active resource orchestration framework for mobile context monitoring in sensor-rich mobile environments," in *Proceeding of 2010 IEEE International Conference on Pervasive Computing and Communications (PerCom)*.

Karantonis, D. M., Narayanan, M. R., Mathie, M., Lovell, N. H., and Celler, B. G. (2006), "Implementation of a real-time human movement classifier using a triaxial accelerometer for ambulatory monitoring," *IEEE Transactions on Information Technology in Biomedicine*, 10, 156–167.

Khan, W. Z., Xiang, Y., Aalsalem, M. Y., and Arshad, Q. (2013), "Mobile Phone Sensing Systems: A Survey," *IEEE Communications Surveys Tutorials*, 15, 402–427.

Khubaib, Suleman, A., Hashemi, M., Wilkerson, C., and Patt, Y. (2012), "MorphCore: An energy-efficient microarchitecture for high performance ILP and high throughput TLP," in *MICRO*.

Kim, C., Sethumadhavan, S., Govindan, M. S., Ranganathan, N., Gulati, D., Burger, D., and Keckler, S. W. (2007), "Composable Lightweight Processors," in *Micro*.

Koufaty, D., Reddy, D., and Hahn, S. (2010), "Bias Scheduling in Heterogeneous Multi-core Architectures," in *EuroSys*.

Kristensen, M. D. (2010), "Scavenger: Transparent development of efficient cyber foraging applications," in *Pervasive Computing and Communications (PerCom), 2010 IEEE International Conference on*, pp. 217–226, IEEE.

Kumar, K. and Lu, Y.-H. (2010), "Cloud computing for mobile users: Can offloading computation save energy?" *Computer*, pp. 51–56.

Kumar, R., Farkas, K., Jouppi, N., Ranganathan, P., and Tullsen, D. (2003), "Single-ISA heterogeneous multi-core architectures," in *MICRO*.

Kumar, R., Tullsen, D. M., Ranganathan, P., Jouppi, N. P., and Farkas, K. I. (2004), "Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance," *SIGARCH Comput. Archit. News*.

Kwapisz, J. R., Weiss, G. M., and Moore, S. A. (2011), "Activity Recognition Using Cell Phone Accelerometers," *SIGKDD Explor. Newsl.*, 12, 74–82.

Laerhoven, K. V. and Cakmakci, O. (2000), "What shall we teach our pants?" in *Wearable Computers, The Fourth International Symposium on*, pp. 77–83.

Lee, B. and Brooks, D. (2007a), "Illustrative design space studies with microarchitectural regression models," in *HPCA*.

Lee, B. and Brooks, D. (2007b), "Illustrative design space studies with microarchitectural regression models," in *HPCA*.

Lee, B. and Brooks, D. (2008a), "Efficiency trends and limits from comprehensive microarchitectural adaptivity," in *ASPLOS*.

Lee, B. and Brooks, D. (2008b), "Roughness of microarchitectural design topologies and its implications for optimization," in *HPCA*.

Lee, Y., Kang, S., Min, C., Ju, Y., Hwang, I., and Song, J. (2016), "CoMon+: A Cooperative Context Monitoring System for Multi-Device Personal Sensing Environments," *IEEE Transactions on Mobile Computing*, 15, 1908–1924.

Li, K. A., Varshavsky, A., Bao, X., Choudhury, R. R., and Fan, S. (2012), "Method and apparatus for content rating using reaction sensing," US Patent App. 13/523,927.

Li, S., Ahn, J., Strong, R., Brockman, J., Tullsen, D., and Jouppi, N. (2009), "McPAT: An integrated power, area and timing modeling framework for multicore and manycore architectures," in *MICRO*.

Li, T., Baumberger, D., Koufaty, D. A., and Hahn, S. (2007), "Efficient Operating System Scheduling for Performance-asymmetric Multi-core Architectures," in *Supercomputing*.

Liu, J., Priyantha, B., Hart, T., Ramos, H. S., Loureiro, A. A., and Wang, Q. (2012), "Energy efficient GPS sensing with cloud offloading," in *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, pp. 85–98, ACM.

Lukefahr, A., Padmanabha, S., Das, R., Sleiman, F., Dreslinski, R., Wenisch, T., and Mahlke, S. (2012a), "Composite Cores: Pushing heterogeneity into a core," in *MICRO*.

Lukefahr, A., Padmanabha, S., Das, R., Sleiman, F. M., Dreslinski, R., Wenisch, T. F., and Mahlke, S. (2012b), "Composite cores: Pushing heterogeneity into a core," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 317–328, IEEE Computer Society.

Lukowicz, P., Ward, J. A., Junker, H., Stäger, M., Tröster, G., Atrash, A., and Starner, T. (2004), *Pervasive Computing: Second International Conference, PERVASIVE 2004, Linz/Vienna, Austria, April 21-23, 2004. Proceedings*, chap. Recognizing Workshop Activity Using Body Worn Microphones and Accelerometers, pp. 18–32, Springer Berlin Heidelberg, Berlin, Heidelberg.

Mahajan, D., Park, J., Amaro, E., Sharma, H., Yazdanbakhsh, A., Kim, J. K., and Esmaeilzadeh, H. (2016), "TABLA: A unified template-based framework for accelerating statistical machine learning," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 14–26, IEEE.

Manatunga, D., Kim, H., and Mukhopadhyay, S. (2015), "SP-CNN: A Scalable and Programmable CNN-Based Accelerator," *IEEE Micro*, 35, 42–50.

Mantyjarvi, J., Himberg, J., and Seppanen, T. (2001), "Recognizing human motion with multiple acceleration sensors," in *Systems, Man, and Cybernetics, 2001 IEEE International Conference on*, vol. 2, pp. 747–752 vol.2.

Marinelli, E. (2009), "Hyrax: Cloud Computing on Mobile Devices using MapReduce," Master's thesis, Carnegie Mellon University.

McLeod, A. and McLeod, M. A. (2011), "Package 'Kendall'," .

Mittal, R., Kansal, A., and Chandra, R. (2012), "Empowering Developers to Estimate App Energy Consumption," in *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, Mobicom '12, pp. 317–328, New York, NY, USA, ACM.

Mortazavi, B. J., Pourhomayoun, M., Alsheikh, G., Alshurafa, N., Lee, S. I., and Sarrafzadeh, M. (2014), "Determining the single best axis for exercise repetition recognition and counting on smartwatches," in *Wearable and Implantable Body Sensor Networks (BSN), 2014 11th International Conference on*, pp. 33–38, IEEE.

Nose, K. and Skurai, T. (2000), "Optimization of Vdd and Vth for low-power and high-speed applications," in *DAC*.

Padmanabha, S., Lukefahr, A., Das, R., and Mahlke, S. (2013), "Trace Based Phase Prediction for Tightly-coupled Heterogeneous Cores," in *MICRO*.

Pandiyan, D., Lee, S.-Y., and Wu, C.-J. (2013), "Performance, Energy Characterizations and Architectural Implications of An Emerging Mobile Platform Benchmark Suite – MobileBench," in *IISWC*.

Priyantha, B., Lymberopoulos, D., and Liu, J. (2011), "LittleRock: Enabling Energy-Efficient Continuous Sensing on Mobile Phones," *Pervasive Computing, IEEE*.

Ra, M.-R., Sheth, A., Mummert, L., Pillai, P., Wetherall, D., and Govindan, R. (2011), "Odessa: Enabling Interactive Perception Applications on Mobile Devices," in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pp. 43–56, New York, NY, USA, ACM.

Ra, M.-R., Liu, B., La Porta, T. F., and Govindan, R. (2012), "Medusa: A Programming Framework for Crowd-sensing Applications," in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pp. 337–350, New York, NY, USA, ACM.

Rajovic, N., Carpenter, P. M., Gelado, I., Puzovic, N., Ramirez, A., and Valero, M. (2013), "Supercomputing with commodity CPUs: Are mobile SoCs ready for HPC?" in *2013 SC-International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1–12, IEEE.

Ravi, N., Dandekar, N., Mysore, P., and Littman, M. L. (2005), "Activity Recognition from Accelerometer Data," in *Proceedings of the 17th Conference on Innovative Applications of Artificial Intelligence - Volume 3*, IAAI'05, pp. 1541–1546, AAAI Press.

Robotic Apps (2013), "Opensource Face Recognition Application," `https://github.com/ayuso2013/face-recognition`.

Saez, J. C., Prieto, M., Fedorova, A., and Blagodurov, S. (2010), "A Comprehensive Scheduler for Asymmetric Multicore Systems," in *EuroSys*.

Satyanarayanan, M. (2001), "Pervasive Computing: Vision and Challenges," *IEEE Personal Communications*, 8, 10–17.

Satyanarayanan, M., Bahl, P., Caceres, R., and Davies, N. (2009), "The Case for VM-Based Cloudlets in Mobile Computing," *IEEE Pervasive Computing*, 8, 14–23.

Shelepov, D. and Fedorova, A. (2008), "Scheduling on Heterogeneous Multicore Processors Using Architectural Signatures," in *Workshop on the Interaction between Operating Systems and Computer Architecture, in conjunction with ISCA*.

Shiraz, M., Gani, A., Khokhar, R. H., and Buyya, R. (2013), "A Review on Distributed Application Processing Frameworks in Smart Mobile Devices for Mobile Cloud Computing," *IEEE Communications Surveys Tutorials*, 15, 1294–1313.

Shoaib, M., Bosch, S., Incel, O. D., Scholten, H., and Havinga, P. J. (2014), "Fusion of smartphone motion sensors for physical activity recognition," *Sensors*, 14, 10146–10176.

Shoaib, M., Bosch, S., Scholten, H., Havinga, P. J., and Incel, O. D. (2015), "Towards detection of bad habits by fusing smartphone and smartwatch sensors," in *Pervasive Computing and Communication Workshops (PerCom Workshops), 2015 IEEE International Conference on*, pp. 591–596, IEEE.

Su, Y.-Y. and Flinn, J. (2005), "Slingshot: deploying stateful services in wireless hotspots," in *Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pp. 79–92, ACM.

Sunwoo, D., Wang, W., Ghosh, M., Sudanthi, C., Blake, G., Emmons, C., and Paver, N. (2013), "A structured approach to the simulation, analysis and characterization of smartphone applications," in *IISWC*.

Van Craeynest, K. and Eeckhout, L. (2013), "Understanding Fundamental Design Choices in single-ISA Heterogeneous Multicore Architectures," *ACM Trans. Archit. Code Optim.*

Van Craeynest, K., Jaleel, A., Eeckhout, L., Narvaez, P., and Emer, J. (2012), "Scheduling Heterogeneous Multi-cores Through Performance Impact Estimation (PIE)," in *ISCA*.

Verbelen, T., Simoens, P., De Turck, F., and Dhoedt, B. (2012), "Cloudlets: Bringing the Cloud to the Mobile User," in *Proceedings of the Third ACM Workshop on Mobile Cloud Computing and Services*, MCS '12, pp. 29–36, New York, NY, USA, ACM.

Wang, C. and Li, Z. (2004), "Parametric analysis for adaptive computation offloading," in *ACM SIGPLAN Notices*, vol. 39, pp. 119–130, ACM.

Wang, H. and Peh, L.-S. (2014), "MobiStreams: A Reliable Distributed Stream Processing System for Mobile Devices," in *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pp. 51–60, Washington, DC, USA, IEEE Computer Society.

Weiss, G. M., Timko, J. L., Gallagher, C. M., Yoneda, K., and Schreiber, A. J. (2016), "Smartwatch-based Activity Recognition: A Machine Learning Approach," *Proceedings of the 2016 IEEE International Conference on Biomedical and Health Informatics*.

Xu, F., Liu, Y., Li, Q., and Zhang, Y. (2013), "V-edge: Fast Self-constructive Power Modeling of Smartphones Based on Battery Voltage Dynamics," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pp. 43–56, Berkeley, CA, USA, USENIX Association.

Yang, F., Qian, Z., Chen, X., Beschastnikh, I., Zhuang, L., Zhou, L., and Shen, J. (2011), "Sonora: A Platform for Continuous Mobile-Cloud Computing," .

Zaharia, M., Das, T., Li, H., Shenker, S., and Stoica, I. (2012), "Discretized Streams: An Efficient and Fault-tolerant Model for Stream Processing on Large Clusters," in *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pp. 10–10, Berkeley, CA, USA, USENIX Association.

Zhang, M. and Sawchuk, A. A. (2012), "USC-HAD: A Daily Activity Dataset for Ubiquitous Activity Recognition Using Wearable Sensors," in *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, pp. 1036–1043, New York, NY, USA, ACM.

Zhang, X., Kunjithapatham, A., Jeong, S., and Gibbs, S. (2011), "Towards an elastic application model for augmenting the computing capabilities of mobile devices with cloud computing," *Mobile Networks and Applications*, 16, 270–284.

Zhu, Y. and Reddi, V. J. (2013), "High-performance and Energy-efficient Mobile Web Browsing on Big/Little Systems," in *HPCA*.

Zyuban, V., Friedrich, J., Gonzalez, C. J., Rao, R., Brown, M. D., Ziegler, M., Jacobson, H., Islam, S., Chu, S., Kartschoke, P., Fiorenza, G., Boersma, M., and Culp, J. (2011),

"Power optimization methodology for the IBM Power7 microprocessor," *IBM Journal of Research and Development*, 55.

Biography

Songchun Fan was born on March 15, 1989 in Nantong, China. In 2011, she received her Bachelor of Engineering degree from Nanjing University, China, in Software Engineering. She received her Master of Science degree in 2013 and her Doctor of Philosophy degree in 2016 from Duke University in Computer Science. Her research interests include energy-efficient mobile networking Fan et al. (2012), interactive mobile sensing Fan et al. (2014); Bao et al. (2013); Li et al. (2012), interactive mobile app benchmarking Fan and Lee (2016) and deploying heterogeneous mobile processor at large scale Fan et al. (2016). She received Honorable Mention Award in Ubicomp 2013 conference and Best Paper Award in ASPLOS 2016 conference.