RICE UNIVERSITY

System Support for Loosely Coupled Resources in Mobile Computing

by

Xiaozhu Lin

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Ian Zhong, Chair
Associate Professor of Electrical and Computer Engineering and Computer Science

Alan L. Cox
Professor of Computer Science and Electrical and Computer Engineering

Peter J. Varman
Professor of Electrical and Computer Engineering and Computer Science

Houston, Texas
July, 2014
ABSTRACT

System Support for Loosely Coupled Resources in Mobile Computing

by

Xiaozhu Lin

Modern mobile platforms are embracing not only heterogeneous but also loosely coupled computational resources. For instance, a smartphone usually incorporates multiple processor cores that have no hardware cache coherence. Loosely coupled resources allow a high degree of resource heterogeneity that can greatly improve system energy efficiency for a wide range of mobile workloads.

However, loosely coupled resources create application programming difficulty: both resources and program state are distributed, which call for explicit communication for consistency. This difficulty is further exacerbated by the large numbers of mobile developers and mobile applications.

In order to ease application programming over loosely coupled resources, in this thesis work we explore system support – at both user level and OS level – that bridges desirable programming abstractions with the underlying hardware. We study three loosely coupled architectures widely seen in mobile computing: i) a smartphone accompanied by wearable sensors, ii) a mobile device encompassing multiple processors that share no memory, and iii) a mobile System-on-Chip (SoC) with multiple cores sharing incoherent memory.

In order to address the three architectures, this thesis contributes three closely related research projects. In project Dandelion, we propose a Remote Method Invo-
cation scheme to hide communication details from application components that are synchronizing over wireless links. In project Reflex, we design an energy-efficient software Distributed Shared Memory (DSM) to automatically keep user state consistent; the DSM always employs a low-power processor to host shared memory objects in order to maximize sleep periods of high-power processors. In project K2, we identify and apply a shared-most OS model to construct a single OS image over loosely coupled processor cores. Following the shared-most model, high-level OS services (e.g., device drivers and file systems) are mostly unmodified with their state transparently kept consistent; low-level OS services (e.g., page allocator) are implemented as separate instances with independent state for the sake of minimum communication overhead.

We report the research prototypes, our experiences in building them, and the experimental measurements. We discuss future directions, in particular how our principles in treating loosely coupled resources can be used for improving other key system aspects, such as scalability.
Acknowledgements

I would most like to thank my advisor Prof. Lin Zhong for his advice and guidance. He provided great vision, encouragement and resources for me to complete the PhD program. The experiences working with him will be a great fortune in my future career as a research professional.

I would like to thank the rest of my thesis committee, Prof. Alan L. Cox and Prof. Peter J. Varman, for their insightful comments and useful suggestions on interesting research directions to explore.

My gratitude extends to the fellow graduate students at RECG, including Zhen Wang, Ardalan Amiri Sani, Robert LiKamWa, Clayton Shepard, and Mian Dong for many hours of mutual support and insightful discussions.

Finally, I would like to especially thank my family for giving me continuous love, care, and support. I could not have done it without them.
# Contents

Abstract ii  
List of Illustrations x  
List of Tables xiii  

1 Introduction 1  
1.1 Benefits of Loosely Coupled Resources 1  
1.2 Challenges to Software 2  
1.3 Thesis Overview 3  

2 Background 7  
2.1 Light Tasks in Mobile Systems 7  
2.2 Trends in Mobile Hardware 8  
2.2.1 Central, Powerful Processor Is Inadequate 8  
2.2.2 Heterogeneous Mobile Architecture 9  
2.3 Major Programming Paradigms 14  

3 Related Work 17  
3.1 User-level Support 17  
3.1.1 Direct Programming 18  
3.1.2 Library-based Approaches 18  
3.1.3 Virtual Machine 18  
3.1.4 Domain-specific Abstractions 19  
3.1.5 Software Distributed Shared Memory 20  
3.2 OS-level support 21
3.2.1 Single OS Image ............................................... 21
3.2.2 Sharing Reduction in OS ..................................... 23
3.2.3 Refactoring Mature OS ...................................... 24

4 Remote Method Invocation Across Mobile Devices 25
4.1 Motivation: Tension and Gap ................................. 25
4.2 Overview of Dandelion .......................................... 27
4.3 Programming Support ........................................... 29
  4.3.1 Template for Senselet ...................................... 29
  4.3.2 Platform Services .......................................... 31
  4.3.3 Remote Method Invocations ............................... 33
  4.3.4 Senselet Executable Generation ......................... 36
4.4 Execution Support .............................................. 36
  4.4.1 Dandelion Runtime System ................................. 37
  4.4.2 Senselet Resource Exception .............................. 38
4.5 Prototype Realization .......................................... 39
  4.5.1 Smartphone-side Software ................................ 40
  4.5.2 Sensor-side Software ...................................... 40
4.6 Evaluation ....................................................... 41
  4.6.1 Benchmark Health Applications .......................... 42
  4.6.2 Source Code Examination ................................. 42
  4.6.3 Overhead Measurements .................................... 45
4.7 Summary ........................................................ 48

5 Energy-efficient Distributed Shared Memory Over Mobile Processors 49
5.1 Overview of Reflex ............................................. 49
  5.1.1 Programming Constraints ................................. 49
5.1.2 How Reflex Works ................................................. 50
5.1.3 Hardware Requirements ................................. 51

5.2 Distributed Runtime Design ....................................... 51
  5.2.1 Module Execution Model .................................. 52
  5.2.2 Message Transport ......................................... 53

5.3 Software DSM Design .............................................. 54
  5.3.1 General Design Choices ................................... 54
  5.3.2 Protocol Invariant ......................................... 56
  5.3.3 Asymmetric Module Roles ................................ 56
  5.3.4 Finite-state Machine Specification ...................... 57
  5.3.5 Support for Synchronization ............................. 60
  5.3.6 Notable Design Aspects ................................... 61

5.4 Compiler Support for DSM ......................................... 62
  5.4.1 Code Instrumentation .................................... 62
  5.4.2 Optimization for Communication ......................... 63

5.5 Prototype Realization ............................................. 64
  5.5.1 Reflex Runtime ........................................... 64
  5.5.2 Reflex Compiler Toolkit ................................ 65
  5.5.3 Hardware Platform ....................................... 66

5.6 Evaluation .......................................................... 69
  5.6.1 Benchmark Applications ................................. 69
  5.6.2 Overall Performance of Reflex .......................... 70
  5.6.3 Source Code Examination ................................. 71
  5.6.4 Programmer Study ......................................... 73
  5.6.5 DSM Performance .......................................... 73
  5.6.6 Asymmetric Module Role Assignment .................... 73
  5.6.7 Storage and Memory Overhead ........................... 74
  5.6.8 DSM Execution Overhead ................................. 74
6 Single OS Image Over Non-coherent Asymmetric Processors

6.1 Motivation ................................................................. 78
6.2 The Case for Replicating OS Services ................................. 79
6.3 System Design ............................................................. 81
  6.3.1 Design Goals ......................................................... 81
  6.3.2 Architectural Assumptions ......................................... 82
  6.3.3 The Shared-most OS Model ....................................... 83
6.4 The K2 OS ................................................................. 85
  6.4.1 Hardware Platform ............................................... 85
  6.4.2 OS Structure ....................................................... 86
  6.4.3 Refactoring Linux ................................................ 88
  6.4.4 Build K2 from Source .......................................... 89
6.5 Memory Management ................................................... 91
  6.5.1 Unified Kernel Address Space ................................... 91
  6.5.2 Physical Memory Management .................................. 94
  6.5.3 Software Cache Coherence ....................................... 97
6.6 Interrupt Management ................................................ 100
6.7 Nightwatch Threads .................................................... 101
6.8 Experimental Results ................................................ 102
  6.8.1 Efforts in Refactoring ........................................... 103
  6.8.2 Independent Service ............................................. 104
  6.8.3 Shadowed Service ............................................... 105
  6.8.4 Energy Efficiency Benefits .................................... 113
6.9 Observation: Need for Locality-aware Memory Allocation ....... 116
6.10 Summary ....................................................... 118

7 Concluding Remarks ........................................... 119
  7.1 Design Hints .................................................. 119
  7.2 Scalability .................................................... 120
  7.3 Recommendation to SoC architects ....................... 121
  7.4 Conclusion .................................................. 122

Bibliography ..................................................... 124
Illustrations

1.1 An overview of the thesis work .................................................. 3

2.1 Trend in mobile architectures. Both axes are logarithmic. .............. 10
2.2 A smartphone (Nexus 5) accompanied by a wearable sensor (Fitbit Flex). Nexus 5 features full-fledged, quad-core Cortex-A15 ARM processors and Fitbit Flex incorporates low-power Cortex-M3 processor. The two devices have separate physical memory and are connected over Bluetooth low energy link .......................... 11
2.3 TI OMAP4 mobile Application Processor includes two microcontroller cores based on ARM Cortex-M3 in addition to the usual, dual-core ARM Cortex-A9. The M3 cores run their own real-time OS and do not have a coherent memory view with the A9 cores. ................................................................. 12
2.4 Example code based on message passing. The code is written for TI OMAP4 based on the Syslink inter-processor message passing API. To make the list concise, we hide low-level code such as message queue management ......................................................... 15
2.5 Example code based on shared memory. A lock is used to prevent data race ................................................................. 16

4.1 The tension in developing sensor data processing for smartphone body sensor applications ................................................. 26
4.2 Skeleton code for data processing on smartphone. This style for sensor data processing shown here is widely adopted by smartphone programming frameworks, including Android, iPhone OS, Symbian S60, and Maemo .................................................. 27

4.3 The skeleton code of a Senselet class ..................................... 28

4.4 Pseudo code of a simplified SenseletBase, as a template to wrap an event loop skeleton for Senselet class ........................................ 30

4.5 The internals of an example RMI PassData(). (a) The stub compiler generates both caller and callee stubs that translate the RMI into messages. Both main body and senselet can be caller or callee. (b) During the RMI, the caller transfers the data and its control flow to the callee ................................................................. 32

4.6 The overview of Dandelion runtime architecture .......................... 35

4.7 The Dandelion prototype hardware, based on Nokia N900 smartphone (left) and Rice Orbit body sensor (right) ....................... 39

4.8 An implementation of FallDetector with senselet ......................... 43

5.1 The structure of Reflex. This example asymmetric, heterogeneous hardware architecture is a TI OMAP4-based smartphone with an added MSP430 processor .................................................... 52

5.2 State transitions in the Reflex DSM coherence protocol. All transitions are triggered by requests from a requester. ..................... 59

5.3 The tri-processor prototype (Top) and its architecture diagram (Bottom). ................................................................. 67

5.4 The average system power consumption of benchmark applications .. 71
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.5</td>
<td>The source lines of code of benchmark applications. Comment or blank lines are excluded. For a column that consists of two parts, the bottom part shows the source lines of the central module; the top part shows that of the peripheral module(s)</td>
<td>72</td>
</tr>
<tr>
<td>5.6</td>
<td>The average requester latency in fetching an object, without and with batch prefetching</td>
<td>75</td>
</tr>
<tr>
<td>6.1</td>
<td>K2 in the spectrum of OS structures</td>
<td>83</td>
</tr>
<tr>
<td>6.2</td>
<td>The structure of K2</td>
<td>87</td>
</tr>
<tr>
<td>6.3</td>
<td>Refactoring Linux OS services for K2</td>
<td>88</td>
</tr>
<tr>
<td>6.4</td>
<td>The unified kernel virtual address space of K2</td>
<td>92</td>
</tr>
<tr>
<td>6.5</td>
<td>The physical memory management of K2</td>
<td>95</td>
</tr>
<tr>
<td>6.6</td>
<td>OS benchmarks for testing energy efficiency</td>
<td>114</td>
</tr>
</tbody>
</table>
## Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>The source lines of code in developing in-sensor data processing, with different programming styles</td>
<td>45</td>
</tr>
<tr>
<td>4.2</td>
<td>Memory breakdown of sensor runtime, in byte</td>
<td>45</td>
</tr>
<tr>
<td>4.3</td>
<td>Memory overhead of senselet executables, in byte and the percentage of the whole sensor memory. Assume whole memory of 10KB RAM and 48KB ROM</td>
<td>46</td>
</tr>
<tr>
<td>4.4</td>
<td>Execution overhead in common sensor runtime operations, in processor cycles</td>
<td>47</td>
</tr>
<tr>
<td>4.5</td>
<td>Execution overhead in the most common data processing path, in processor cycles and the percentage of processor time in each sampling period. FallDetector and Pedometer acquire sensor data at 50Hz, while EKG does at 512Hz. Assume a clock rate of 4MHz</td>
<td>47</td>
</tr>
<tr>
<td>5.1</td>
<td>The processors used in the tri-processor prototype. At run time, unused processors are turned off to reduce the system idle power</td>
<td>68</td>
</tr>
<tr>
<td>6.1</td>
<td>Heterogeneous cores in the two coherence domains of OMAP4</td>
<td>86</td>
</tr>
<tr>
<td>6.2</td>
<td>Source code changes and additions to the Linux kernel 3.4, which contains around 10 million SLoC. Much of the OMAP4-specific Cortex-M3 code is adopted from the linux-panda project [65]</td>
<td>103</td>
</tr>
</tbody>
</table>
6.3 Latencies of physical memory allocations in K2, in $\mu s$. The performance of K2’s main kernel allocator has no noticeable difference from the Linux page allocator. 

6.4 A breakdown of the latency in a software coherence page fault, in $\mu s$. 

6.5 The major operations of the DMA controller driver, as well as the key data structures they access. The worst-case overhead due to software cache coherence is also shown. Delay in $\mu s$. 

6.6 DMA throughputs of K2 when the DMA driver is invoked in both kernels concurrently, in MB/Sec. The throughput differences as compared to the original Linux are also shown. 

6.7 The major operations of the Linux ramdisk driver, as well as the key data structures they access. The worst-case overhead due to software cache coherence is also shown. Read or write touches one page of data. Delay in $\mu s$. 

6.8 The major operations of the Linux VFS, as well as the key data structures they access. The worst-case overhead due to software cache coherence is also shown. The depth of file path is three. Delay in $\mu s$. 

6.9 Power consumptions of the heterogeneous OMAP4 cores, in mW. Both cores consumes less than 0.1 mW when inactive. The frequencies with asterisks are used in the benchmarks for testing energy efficiency.
Chapter 1

Introduction

Today, mobile platforms have extensively embraced loosely coupled computational resources. In such a platform, resources such as processors, memory, and I/O are physically distributed and operate independently. As there is no hardware cache coherence, such resources are essentially in multiple coherence domains, and the domains communicate with message passing primitives provided by the hardware.

1.1 Benefits of Loosely Coupled Resources

Loosely coupled resources offer two key benefits that are unprecedented. First, they enable a high degree of resource heterogeneity and thus can greatly improve system energy efficiency. Today’s mobile devices face a wide range of workloads, from demanding interactive tasks with rich graphics and high-speed networking to light tasks running in the background such as cloud synchronization and context awareness. A recognized, effective approach to achieve high energy efficiency for such diverse workloads is to exploit hardware heterogeneity, i.e., executing a workload with hardware component that offers the best power-performance tradeoff. Driven by the pursuit of heterogeneity, mobile architectures with multiple processors that can operate independently are popular in both industry [67, 88] and academia [72, 84, 48, 2]. The resulting hardware architecture is asymmetric with multiple loosely coupled cores that differ in processing power by orders of magnitudes.
Second, (even more) loosely coupled resources open the door to diverse I/O devices. Because mobile users usually keep their phones within the arm’s reach [70], smartphone has already served as the personal hub for wearable devices, bridging the latter with human users and the Internet [98]. With providing very rich I/O devices, e.g., a secondary screen, speakers, camera, or various sensors, wireless wearable devices naturally extend the “sense” of a smartphone and open doors to new applications, such as healthcare. Numerous research prototypes have been reported in literature, and there has been a handful of commercial applications in recent years, e.g. Google Glass [35] and the Nike+iPod Sport Kit [6].

1.2 Challenges to Software

In tapping into loosely coupled resources, the biggest challenge is programmability. Processors from multiple coherence domains essentially constitute a distributed system. Programming them is known to be difficult to massive third-party programmers, who must maintain consistency for partitioned program components with messages. Not surprisingly, there are very few third-party mobile applications that actually distribute user workloads over multiple coherence domains. Compared to user-level programming, OS engineering is even more challenging. We observe that mobile programs exercise an extensive set of OS components, which requires OS execution to be efficiently replicated over multiple coherence domains. However, all previous work implement a small set of OS services from scratch for a subset of coherence domains, e.g., sensor reading, and assume all other OS services are provided by other domains. The resulting OS services are limited in functionality and hard for application developers to use due to a fragmented system image. What is even worse, constructing those OS services themselves incur high development burden.
1.3 Thesis Overview

The theme of this thesis is to address the programmability challenge with system software design, or system support. Our goal is to relieve application developers from directly dealing with loosely coupled resources so that they can easily port legacy applications and develop new ones. While this goal is reminiscent of that of programming other heterogeneous, distributed systems, we face a set of unique challenges when dealing with mobile platforms: extreme architectural asymmetry, energy efficiency, and entrenched programming models.

An overview of the thesis work is shown in Figure 1.1. In exploring the design space of system support, our work is focused on three representative types of integration in loosely coupled mobile architectures, as shown along the horizontal axis of Figure 1.1: i) chip-level integration, namely a mobile System-on-Chip (SoC) with multiple cores that share incoherent memory, ii) board-level integration, namely a mobile device
with multiple processors that share no memory, and iii) multi-device integration, namely a smartphone accompanied by wearable sensors. For these architectures, we further decompose the respective system support into user level and OS level. With user-level support we seek to simplify user program compilation, deployment, and state consistency; with OS-level support we seek to provide programming abstractions for OS development such as writing driver and file systems. We will present the entire work as three closely related projects, as labelled in Figure 1.1.

Chapter 4 presents project Dandelion [47], a user-level framework towards hiding communication details from distributed user program components. Combining both programming and execution support for in-sensor data processing, Dandelion provides transparency on two key aspects: programming style transparency, which makes writing code for wireless sensors very close to developing traditional smartphone applications; target platform transparency, which makes the development, distribution, and deployment of sensor code independent of any particular sensor platform. We experimentally evaluate our prototype with real-world applications, showing that the Dandelion framework reduces the application developers’ burden significantly. Furthermore, it does so with an overhead of less than 5% of the memory capacity and less than 3% of the processor time of a typical low-power body sensor.

Chapter 5 presents Reflex [48], a suite of compiler and runtime techniques towards offering consistent program state over loosely coupled processors that are of high-power and low-power. The heart of Reflex is a novel software Distributed Shared Memory (DSM) design. The DSM allows code on multiple heterogeneous processors to easily exchange data, even without hardware-coherent memory. The Reflex DSM contributes multiple technical innovations. First, it maintains architectural energy efficiency by using a peripheral processor to serve coherence requests in order to keep
more power-hungry ones in sleep. Second, it creates a shared memory abstraction across processors that differ in processing power by orders of magnitude, and minimizes disruptions to frequent data processing on peripheral processors. Finally, it employs code instrumentation and aggressive compile-time optimization to greatly improve the DSM performance. Our prototype of Reflex works for a TI OMAP4430 development kit and a tri-processor mobile system built with a commercial smartphone, 32-bit and 16-bit microcontrollers. Reflex reduces the average power consumption by up to 81%. As compared to the message passing paradigm, the DSM reduces the source lines of application code by up to 38%, with less than 5% source code difference from the legacy counterpart.

Chapter 6 presents K2 [50], an experimental OS for mobile SoC with heterogeneous coherence domains. Through the design of K2, we seek to answer the following question: *is it possible to span an OS across multiple coherence domains?* In particular, (i) the resulting OS should preserve the established programming models for mobile app development, (ii) the OS itself shall leverage mature operating systems without reinventing wheels, and (iii) the system should deliver the same performance for high-performance applications. Our study shows that one can span an OS across multiple coherence domains by properly refactoring, but without overhauling, existing OS. we identify a *shared-most* OS model where most OS services are replicated with transparent state coherence for all domains under question and a small, selective set of services operate independently in each domain. Applying the shared-most model, we build the K2 OS, which spans a single system image over multiple coherence domains and thus improves energy efficiency by up to 10x. K2 has met three key design goals. First, it presents a coherent, single Linux image to programmers and therefore preserves the widely used programming model; it also provides a familiar programming
abstraction for distributing user workloads among heterogeneous domains. Second, it replicates existing high-level OS services, e.g., device drivers, across multiple domains and transparently maintains coherence for them. As a result, K2 is able to reuse most kernel source. Third, it preserves the current level of OS performance by alleviating inter-domain contention: it creates independent instances of low-level services and properly coordinates them; it avoids multi-domain parallelism within a process through scheduling.

The thesis work is a rethinking of the fundamental structure of mobile software stack in face of emerging architectures. Our experiences show that with system software designed under novel principles, it is feasible for mass mobile programs to exploit loosely coupled resources with existing programming paradigms. Although our designs and implementations are based on mobile platforms, as loosely coupled resources become increasingly pervasive in larger-scale systems like manycore servers [10], We hope many of our design principles can benefit them as well.
Chapter 2

Background

To provide the background for the thesis work, we first discuss the characteristics of light tasks in mobile systems, and next review the mobile architectural trend that is moving towards heterogeneity and lacking of hardware cache coherence. We then discuss two major paradigms used for programming loosely coupled and heterogeneous architecture.

2.1 Light Tasks in Mobile Systems

Serving as the personal information hub, today’s mobile devices execute a rich set of ‘background’ tasks, or light tasks. One major category of light tasks is sensing. As mobile systems are embracing a variety of sensors (e.g., either inside a smartphone or connected to the smartphone wirelessly), many emerging applications often continuously use these sensors without user engagement, such as inferring user physical activities and monitoring surrounding environment [74, 54]. Other important light tasks include keeping users connected with social networks and the cloud [96].

Light tasks usually have the following characteristics:

- *Not performance demanding:* Without an awaiting user, light tasks do not directly affect the user experience and thus do not require high processor MIPS.

- *Mostly IO-bound:* Light tasks perform extensive IO operations for exchanging information with the external world. During IO operations, core idle periods...
are many, however not long enough for a core to become inactive.

- **Requiring both user and OS execution**: Light tasks not only run user code for application logic, but also invoke diverse OS services such as device drivers [49] and page allocator.

- **Scheduled to execute throughout daily usage**: Light tasks are executed not only when a user is interacting with the device, but also during user’s think periods between interactions and when a user pays no attention. Since user interaction periods are sparse and usually short, most executions of light tasks happen when the entire system is lightly loaded.

- **High impact on battery life**: The nature of frequent executions has a high impact on battery life. For instance, a recent study [96] shows that each run of background email downloading reduces device standby time by 10 minutes.

### 2.2 Trends in Mobile Hardware

#### 2.2.1 Central, Powerful Processor Is Inadequate

To meet the compute demand of modern applications, mobile devices are equipped with powerful cores that have gigahertz frequency and rich architectural features. These powerful cores, however, offer poor energy efficiency for light tasks, due to three sources of inefficiency:

- **High penalty in entering/exiting active power state**: Since strong cores are inactive for most of the time, periodic executions of light tasks will inevitably wake them up from time to time.
• **High idle power.** In executing IO-bound light tasks, powerful cores will spend a large number of short periods (of milliseconds) in idle, which are known to consume a large portion of energy in mobile systems [97].

• **Over-provisioned performance.** Although a strong core may increase its energy efficiency by lowering its frequency, shown as ‘DVFS’ in Figure 2.1, the lowest possible frequency and active power are limited by its architecture and fabrication process, which still over-provision performance to light tasks with a low performance demand [37].

• **Far from data source.** In a very loosely coupled mobile system, e.g., a smartphone accompanied by wearable sensors, the strong cores are usually far away from the source of the data to be processed, such as the sensors. Compared to processing raw data in-place on the sensors, moving them to the central powerful cores over wireless links can be expensive.

### 2.2.2 Heterogeneous Mobile Architecture

Many have proposed to add simple, low-power processors for mobile devices [2, 72, 84], and for wireless sensor nodes [56, 64, 37], to execute light tasks. Mobile application processors are actually embracing low-power cores [67, 73, 78, 89, 88]. Moreover, sensors intended for mobile devices are also embracing low-power microcontrollers for in-sensor data processing [69, 86]. All result in extremely asymmetric hardware architecture with peripheral processors orders of magnitudes weaker and lower-power than the central processor.

A low-power processor is likely to have a different microarchitecture than the central, full-fledged processor. This makes software relying on the single ISA inadequate.
For example, a ultra-low power microcontroller core like TI MSP430 is necessary in achieving mWatt power consumption for continuous sensing on smartphones [72]. TI OMAP4, as shown in Figure 2.3, employs ARM Cortex-A9 as the central processor but ARM Cortex-M3 as the newly added microcontroller cores. Compared to the Cortex-A9 cores, the Cortex-M3 cores only support a small subset of instructions while lacking many advanced features such as the SIMD extension.

Next we briefly discuss two typical mobile architectures that embrace heterogeneous, loosely coupled processors.

**Smartphone Accompanied with Wearable Sensors**

One typical case of heterogeneous mobile system is a smartphone teaming up with wearable sensors. In this case, the wearable sensors usually embrace efficient compute
Figure 2.2: A smartphone (Nexus 5) accompanied by a wearable sensor (Fitbit Flex). Nexus 5 features full-fledged, quad-core Cortex-A15 ARM processors and Fitbit Flex incorporates low-power Cortex-M3 processor. The two devices have separate physical memory and are connected over Bluetooth low energy link.

resources, e.g., low-power processors and small physical memory. The processors of smartphone and that of the wearable sensors are connected over wireless link, and do not share any physical memory. Although the efficiency of the wireless link has been significant improved, it is still fairly energy-hungry to move data over the link, compared to local computation. Therefore, the sensors sample raw data in real time, process data at early stage and discard most of the data accordingly.

Figure 2.2 shows a real-world example of such a mobile system, consisting of a smartphone (Nexus 5) and a wearable sensor (Fitbit Flex), which collaborate to serve one mobile user. While Nexus 5 is equipped with gigahertz, powerful processors for cutting-edge user experience, it does not involve in sensor data processing for most of the time; on the other hand, sensor data have been processed on the Fitbit Flex with its low-power Cortex-M3 processor. Only at rare occasions do these two devices
Figure 2.3: TI OMAP4 mobile Application Processor includes two microcontroller cores based on ARM Cortex-M3 in addition to the usual, dual-core ARM Cortex-A9. The M3 cores run their own real-time OS and do not have a coherent memory view with the A9 cores.

communicate over the Bluetooth low energy link, for instances, when user queries data or interesting events are detected. In such a loosely coupled mobile system, most raw data have been processed and discarded on Fitbit Flex, thus resulting in high energy efficiency overall.

Heterogeneous System-on-Chips (SoCs)

Today’s mobile SoCs incorporate heterogeneous cores with different tradeoffs between performance and power, by placing them in a single or multiple coherence domains.
A handful of heterogeneous SoCs host all heterogeneous cores with a single coherence domain [36, 68]. As shown as ‘coherent heterogeneity’ in Figure 2.1, although the introduced heterogeneity indeed widens dynamic power range, the global cache coherence becomes the bottleneck for more aggressive energy efficiency, for a few reasons: i) a unified hardware cache coherence mechanism restricts architectural asymmetry, ii) the coherent interconnect itself consumes significant power, and iii) cores co-located in the same coherence domain are likely to suffer from similar thermal constraints.

**Incoherent Heterogeneous Processors.** Unlike PCs with a symmetric multiprocessing (SMP) processor, hardware cache coherence is difficulty for smartphones with extremely asymmetric processors. Peripheral processors can barely realize a hardware coherence protocol under the tight energy constraint. The micro-architectural difference between strong and weak cores further makes hardware coherency less attractive. For instance, TI OMAP4 Cortex-A9 and Cortex-M3 cores physically share non-coherent memory as shown in Figure 2.3. To exchange data, cores of different types need to copy data to and from the memory, and notify each other with the hardware mailbox.

For aggressive energy efficiency, several newer SoCs remove chip-level hardware cache coherence. They embrace multiple coherence domains, each of which can host multiple cores; hardware cache coherence exists within a domain but not across domains. Examples include OMAP4 [88], OMAP5 [89], and Samsung Exynos [78]. As shown in Figure 2.1, the absence of hardware cache coherence enables high asymmetry among cores belonging to different domains. For instance, while the lowest power of different cores in the same domain can differ by 6x [82], that of different domains can differ by up to 20x. This allows using weak cores to greatly reduce the three inefficiencies mentioned above.
2.3 Major Programming Paradigms

Due to extreme architectural symmetry, the resulting system is heterogeneous and lacks of coherence. To leverage low-power processors, a developer not only needs to learn a new programming environment but also needs to carefully manage inter-processor data exchange. This is because light tasks often need to exchange data with the rest of the application. Examples of such data include sensing parameters (static or adaptive), intermediate or final results, and control/status variables. Such data can be read or written by multiple code parts. Furthermore, light tasks access the shared data more frequently than its complicated, infrequent counterpart, often by orders of magnitude, due to the cascaded design mentioned in Section 2.1.

Without hardware coherent memory, all existing programming solutions support program state consistency with certain forms of message passing. With systems reported in [84, 9], the developer has to explicitly maintain state consistency between central and peripheral processors. To so between the A9 cores and Cortex-M cores on TI OMAP4, the developer must use an inter-processor message passing API called Syslink. Not surprisingly, there are very few third-party smartphone applications that actually use a peripheral processor to save energy.

Programming with Message Passing

Message passing is a programming paradigm where distributed code parts communicate by explicitly copying messages, being data or program objects. Programming with message passing for smartphone sensing is a nontrivial task. We next use an example based on OMAP4 to illustrate the reasons. Figure 2.4 shows two code snippets excerpted from a simple sensing application, which uses messages to exchange data with the Syslink API provided by TI. The two code snippets are executed on
handling on the peripheral processor should not be executed so frequent processing task and waiting. Similarly, message architectural asymmetry. Message-based communication should complete in time, because the peripheral processor is stalling its processor message passing API called Syslink. Not surprisingly, in Figure 2. Due to the discrepancy of ISAs, developers may need to convert samples among different data formats. For shared objects that contain pointers, e.g., a linked list, the efforts will be more tedious.

In the code, message passing exhibits its well-known drawbacks. First, developers need to synchronize distributed copies of the same object with a customized communication protocol, (1) (3) (4) (6) in Figure 2.4. Due to ISA discrepancy, developers may need to convert samples among different data formats. For shared objects that contain pointers, e.g., a linked list, the efforts will be more tedious.

The use of message passing is further complicated by the extreme architectural asymmetry. Message-based communication should not interfere with the sensor data processing on the weak, peripheral processors. For example, the computation in (2) should complete in time, because the peripheral processor is stalling its frequent processing task (5) and waiting. Similarly, message handling (6) on the peripheral

Figure 2.4: Example code based on message passing. The code is written for TI OMAP4 based on the Syslink inter-processor message passing API. To make the list concise, we hide low-level code such as message queue management.

Two different processors, and both use the samples array. The peripheral processor code, running on the Cortex-M3 cores, fills samples as it acquires sensor readings; the central processor code, running on the Cortex-A9 cores, reads samples and clears historical values when necessary.
Handling (6) on the peripheral processor should not be executed so frequently as processing task (5) and waiting. Similarly, message passing API called Syslink. Not surprisingly, in Figure 2. Due to the discrepancy of ISAs, developers may need to convert or unnecessary data transfer.

In the code, message passing exhibits its well-known drawbacks. Objects that contain pointers, e.g., a linked list, the efforts will be more tedious.

For example, the computation in (2) should be close to what can be achieved by hand-optimized message passing. Finally, for energy efficiency, the developer must arrange the communication so that no message should wake up the central processor from a sleep state.

**Programming with Software Distributed Shared Memory (DSM)**

The archrival to message passing in programming distributed systems without hardware-coherent memory is software DSM. With a software protocol that coordinates multiple distributed memory components to collaboratively serve memory access, software DSM creates the key illusion of a shared address space. Compared to messages, shared memory is a much more familiar abstraction to smartphone developers. The resulting application code is concise and similar to legacy smartphone code, as shown in Figure 2.5. However, DSM may introduce communication overhead due to thrashing or unnecessary data transfer.

---

**Figure 2.5**: Example code based on shared memory. A lock is used to prevent data race.

![Code Snippet](attachment:code_snippet.png)

(a) The code for the ARM A9 central processor

(b) The code for the ARM Cortex-M3 peripheral processor
Chapter 3

Related Work

Hardware heterogeneity is a recognized way to improve energy efficiency. Driven by the pursuit of heterogeneity, mobile architecture with loosely coupled resources, that is, multiple processors operating independently without hardware cache coherence, is popular in both industry [67, 88] and academia [72, 84, 48, 2]. Although the goal of abstracting and managing loosely coupled resources has been extensively studied in prior work, the goal brings new twists into the context of mobile computing. Since programming loosely coupled resources share many challenges and solutions with programming heterogeneous resources, the following discussions will also include resource heterogeneity. We categorize the related work into that of user level and OS level, and discuss them respectively.

3.1 User-level Support

Project Dandelion and Reflex, as presented in this dissertation, are among the first to ease user-level programming for low-power cores in mobile devices. However, the study of user-level abstractions for loosely coupled resources is not new. We next discuss the major categories of abstractions that have been proposed previously. It is important to keep in mind that we face a set of unique challenges when dealing with mobile devices: mature programming languages, energy efficiency and extreme architectural asymmetry.
3.1.1 Direct Programming

In face of loosely coupled resources, the most straightforward approach is to shift the burden to application developers, i.e., directly exposing the facts of discrete program components and fragmented software state to applications, requiring application developers to handle them explicitly. In most heterogeneous systems [2, 84, 94], the developers have to program each processor in the system directly without any transparency. Many systems support inter-processor communication with message-like primitives. For example, Hydra [94] supports distributed “offcodes” to communicate with channels. Lime [39] supports code distributed on CPU and FPGA to exchange data through “buffer”, a message-passing abstraction. Compared to programming support presented in this dissertation, programming with message passing can be tedious and error-prone, in particular for mobile applications.

3.1.2 Library-based Approaches

Taking one step further, some systems facilitate application development by abstracting resources with a pre-defined set of functions, an approach that is commonly seen in programming wearable devices. Examples include MobiCare [16], CodeBlue [60], and SPINE [41]. In particular, SPINE provides a portable library for in-sensor signal processing. While this approach gains certain programming ease for particular applications that use such pre-defined functions, compared to Dandelion discussed in Chapter 4, it restricted expressiveness in user-level programming.

3.1.3 Virtual Machine

Virtual machines can be used to abstract away heterogeneity. Some systems support programming transparency with a unified OS abstraction or distributed runtime
system, mostly based on a virtual-machine approach [66] to hide ISA variances. Additionally, type safety, as featured by most language VMs, provides vital information for translating memory objects among address spaces, a central task of distributed shared memory. The VM approach, however, proves to be difficult on low-power processors [45]. In contrast, Dandelion and Reflex hide ISA heterogeneity with an extra compilation phase that happen on the target device, and explicitly require type-safe programming at user level.

3.1.4 Domain-specific Abstractions

Many programming systems have been proposed for loosely coupled and/or heterogeneous systems. Most of them define custom programming interfaces to bridge the developer’s code with the underlying architecture. In the context of wireless sensor networks (WSN), solutions such as nesC [31] and protothreads [26] provide node-level programming abstractions. Macro-programming [20, 40, 57] enables developing WSN applications by describing the network-level behaviors.

Qilin [55] provides API for parallelizable computation so that the computation can be dynamically mapped to heterogeneous resources. Merge [51] defines new language constructs to support the map-reduce pattern that fits in parallel execution. Cell-Ss [11] requires annotations of source-code to exploit the task-level parallelism. These custom programming models are specific to their targeted applications, including WSN, scientific, or analytic workloads, making it is difficult for smartphone developers to adopt them. In contrast, our work aims at supporting the mature paradigms of smartphone app development.
3.1.5 Software Distributed Shared Memory

Generally, the idea of creating a shared memory illusion by software with messages, or software DSM, is well-known. There is a rich body of software DSM designs for conventional loosely-coupled distributed systems like workstation clusters. Aiming at minimizing the performance overhead of shared memory, these DSM designs have employed OS kernel support [32], compiler techniques [79] and runtime systems [46, 15, 4]. Also, heterogeneity has been addressed [99]. Most of them rely on hardware MMU support. While they inspire the design of the Reflex DSM, none of them is designed for energy efficiency or the extreme architectural asymmetry. More recently, Hera-JVM [63] realizes a software cache among heterogeneous cores of Cell processor. Unlike the Reflex DSM, it implements the cache coherence in a symmetric fashion and with a coarse granularity, by simply flushing the entire cache of a peripheral processor (i.e. SPE) to the main memory during synchronization. This makes Hera-JVM energy-inefficient and improper for smartphone sensing. Partitioned Global Address Space (PGAS) allows multiple processors (many homogeneous ones [10] or a few heterogeneous ones [77]) that share no coherent memory to logically share a window of shared addresses. Unlike Reflex, such a shared window is implemented with symmetric coherence protocols. ADSM [32] provides a software DSM between CPU and GPU with fully implementing DSM on the CPU, a strategy opposite to the Reflex DSM. Although the strategy is asymmetric, it targets performance instead of energy efficiency and therefore will not work for smartphone sensing applications.

Hardware-coherent memory is also related. While most symmetric multiprocessing (SMP) systems support hardware-coherent memory as exemplified by commercial multicore processors, only a few heterogeneous systems [91, 93] rely on it. As we analyzed in Chapter 5, hardware-coherent memory is difficult, if possible at all, for
low-power peripheral processors concerned by mobile devices.

3.2 OS-level support

While some existing systems have focused on user programming models [48] and policies [72], none offers OS support as K2 does. The lack of OS support will defeat the goal of energy efficiency: as shown by our study of mobile workloads [49], demanding and non-demanding tasks share an extensive set of OS services, which must be executed on each coherence domain.

3.2.1 Single OS Image

Programming multiple OSes or kernels can be made a lot easier with a single system image. While the accurate definition of a single OS image may vary, we see two aspects are fundamental and related to our work: unified namespaces and unified resource management.

Unified OS Namespace

Many previous work is focused on maintaining unified namespaces without unified resource management. Targeted a cluster of workstations, an OS often consists of a set of distributed services, each of which manages some resources that are local to the corresponding workstation (i.e., a coherence domain). Thus, a single OS image is often created by forwarding OS requests to the corresponding coherence domain. A service may have multiple instances on difference domains; however, such instances manage disjoint sets of resources and still have to communicate with each other frequently.

Examples abundant. For scalability, the V distributed system [19] distributes microkernels over workstations and forwards requests as network messages to cor-
responding OS servers via network. Project OpenSSI [13] refactors and replicates Linux, a monolithic kernel, over multiple workstations of a cluster. Multicellular OSes such as Hive [17] and Cellular IRIX [81] provide single system images over multiple cells by running independent kernels which cooperate using explicit communication, e.g., RPC. Since kernels are independent, an IO request must be served in the only cell where the IO device is located. Similar software structure has been employed for a single machine. fos [95], Helios [66], and NIX [8] dedicate a set of cores or hardware accelerators to a specific OS service, and pass corresponding service requests as explicit messages. For resource isolation, Tessellation [21] allows certain OS components (called a partition) run on top of a dedicated set of resources. Disco [14], which runs different unmodified OSes on a single machine, does coordination at user level with standard network protocol, thus providing a partial single system image.

Unlike them, K2 cannot rely on request forwarding for a single OS image, which will frequently involve multiple heterogeneous coherence domains and thus undermines either performance or energy efficiency. In creating a single OS image, K2 calls for not only unified namespaces but also unified resource management: all OS replicas must manage all resources collaboratively.

**Unified Resource Management**

Unified resource management has been proposed for distributed OSes towards better resource utilization, as statically partitioning resources is wasteful. Many systems feature redesigned local resource managers that cooperate at fine grains [10, 95, 28, 53], which are effective however difficult to be applied to mature mobile OSes. Similarly, Libra [3] enables a general-purpose OS and multiple specialized library OSes to share physical memory dynamically; it also employs redesigned memory managers
in the specialized OSes. Unlike K2, most of them are focused on memory resources while leaving out other important resources, e.g., IO and interrupts. Some virtual machines [92] enables resource sharing (in particular physical memory) among multiple guest OSes while minimizing modifications to them. K2 retrofits the idea of balloon driver from virtual machines; in particular, K2 transfers the ownership of physically contiguous memory regions among individual kernels, without assuming any virtualization hardware.

As an extreme design point, vNUMA [18] places a distributed shared memory layer at the hypervisor level, enabling one virtual machine to span over networked physical workstations. This shared-everything approach, although minimizing modifications to OS and apps, introduces relatively high communication latency (e.g., that of ethernet) into performance-sensitive OS components, such as page allocator, thus undermining performance. In comparison, with K2 we argue for replicating the state of performance-sensitive OS components for the sake of reduced communication and higher performance.

Maintaining cache coherence with software is a classic approach to hide distributed program state from programmers [46], which has been applied to new incoherence architectures like CPU-GPU system [32] and smartphones [48]. K2 borrows mature designs of software cache coherence, and explores its use on supporting OS services over multi-domain SoCs.

3.2.2 Sharing Reduction in OS

Reducing sharing in OS has been extensively studied for shared memory machines, mostly for scalability. Many argue for structural overhauls. Hurricane [90] organizes OS in a hierarchical way to improve scalability. Barreelfish [10] takes an extreme design
point to make all OS state non-shared by default. FOS [95] argues to replicate given OS services to a subset of processors and coordinate them with messages. New programming abstractions have been proposed, too. Tornado [29] and K42 [5] argue for object-oriented OS designs, enabling each OS service to have its internal mechanism to distribute over SMP. Inspired by them, K2 treats OS services differently in how to replicate them over coherence domains; unlike them, K2 targets reusing legacy OS code rather than a clean-slate implementation. For manycore machines, Corey [12] enables applications to control sharing within processes. Inspired by it, K2 employs NightWatch thread as a mechanism for expressing performance expectation.

3.2.3 Refactoring Mature OS

In reusing mature OSes for new architectures, virtualization and refactoring are two major approaches. While K2 borrows idea from virtual machines for memory management, the latter usually enforces isolation among OS instances, rather than providing a single system image over them. Refactoring, as done in DrawBridge [71] and Unikernels [58] recently, restructures OS while keeping individual OS components. Their approaches are inspiring, but solve a different problem – rearranging OS services vertically across layers. In comparison, for K2 we refactor Linux by replicating OS services horizontally across coherence domains.
Chapter 4

Remote Method Invocation Across Mobile Devices

4.1 Motivation: Tension and Gap

Figure 4.1 shows the tension between system energy efficiency and ease of body sensor application development. At one extreme (ease of development), the developer can treat the sensors as dumb data suppliers and processes the raw data on the smartphone (“Dumb supplier” in the figure). Figure 4.2 uses a skeleton code to illustrate the smartphone style for processing sensor data, which uses a clear interface in the object-oriented fashion. While very friendly to smartphone developers, this paradigm is extremely energy inefficient because moving excessive raw data over the wireless link consumes a lot of energy, and consequently reduces the battery lifetime of the phone and sensors.

To improve the efficiency, many research prototypes [41, 33] implement a fixed set of sensor routines to support a pre-defined set of data processing in the sensor (“Pre-defined routines” in Figure 4.1). This is similar to the Application Profiles provided by Bluetooth. While this approach improves energy efficiency for some applications, its lack of sensor programmability inevitably restricts developers from creating applications that require novel, application-specific data processing. At the other extreme, the application developers can carefully partition the data processing between the phone and the sensors, and then program both directly. While this paradigm can be very efficient, it usually requires the developer to master sensor node-level program-
Figure 4.1: The tension in developing sensor data processing for smartphone body sensor applications

programming, with low-level abstractions that are tightly coupled with various underlying platforms [30, 38, 52, 26], and probably involves a different programming language. For example, TinyOS requires developers using nesC language [31] to write sensor code as a set of software components. In addition, TinyOS requires developers to define events and commands as the interface for each component, and properly wire interfaces together. Although very effective at programming nodes in wireless sensor networks, such programming requirements are foreign to many smartphone developers and are significantly different from existing smartphone programming styles. Macro-programming models at the network-level [20, 40, 57] target relatively large-scale sensor networks. The introduction of new language constructs such as new statements, predicates, rules, or task graphs, makes them even more difficult for smartphone programmers to manage. In short, the gap between the smartphone programming style and that of various sensor platforms constitutes a practical barrier for the majority
class MySensorListener : SensorListener {

public:

// called when the processing starts
OnCreate() { ... };

// called when the processing stops
OnDestroy() { ... };

// called when new sensor data is acquired
OnNewData(sensor_id, data) { ... };

private:

// private states as variables
}

Figure 4.2: Skeleton code for data processing on smartphone. This style for sensor data processing shown here is widely adopted by smartphone programming frameworks, including Android, iPhone OS, Symbian S60, and Maemo

of smartphone developers to leverage body sensors and create applications.

4.2 Overview of Dandelion

Our goal is to achieve both high efficiency and ease of development by making sensor programming transparent to the smartphone developers (“Dandelion” in Figure 4.1). By “transparent”, we mean that smartphone developers do not have to deal with the native programming abstraction or hardware/software specifications of the sensor. Our solution toward this goal is Dandelion, a framework that allows smartphone application developers to “program” body sensors by simply writing the code as a smartphone software module. Dandelion achieves such design goal with senselet, a smartphone-style, platform-agnostic programming abstraction for in-sensor data
class MySenselet : public SenseletBase {

    public:
        // App-specific initialization
    void OnCreate() {...};
    // App-specific finalization
    void OnDestroy() {...};
    // Receive and process new sensor data
    void OnData(data) {...};

    private:
        // All senselet states are private variables
    }

    // ... in the main body code ...
    LoadSenseletInstance(MySenselet);

Figure 4.3 : The skeleton code of a Senselet class

... in the main body code ...

Dandelion supports this abstraction from both the programming aspect (Section 4.3) and the execution aspect (Section 4.4). For programming support, Dandelion defines the degree of transparency with three key design decisions: a compact set of platform services for senselet to access platform resource, the Remote Method Invocation (RMI) mechanism for integrating senselets with the application code running on the smartphone (called main body in this work), and the two-phase compilation technique for generating senselet executable.

For execution support, Dandelion manages and executes senselets with its distributed middle layer or runtime system. The runtime system consists of a smartphone runtime component and one or more sensor runtime components. Runtime
components communicate with messages. The smartphone runtime acts as the coordinator to command all sensor runtimes, and a sensor runtime fulfills such commands and executes the senselet code.

4.3 Programming Support

Toward making sensor programming transparent, Dandelion targets practical transparency. A complete transparency is ideal to the developer: the development and execution of a senselet will be indistinguishable from a smartphone software module. However, such complete transparency is very expensive, if not impossible, given the huge discrepancy between the hardware and software environments on the smartphone and on body sensors. For example, implementing coherent shared memory across the smartphone and sensors incurs prohibitive overhead.

Dandelion defines practical transparency for senselet with four design decisions: 1) hide platform-dependent programming styles inside the template (Section 4.3.1), 2) choose a compact set of platform services as the unified interface to platform resources (Section 4.3.2), 3) use the RMI mechanism to transparently integrate senselets into the smartphone application (Section 4.3.3), and 4) compile and distribute senselets as platform-independent intermediate representations (IRs) and translate the IRs into platform-specific binaries during application installation (Section 4.3.4). We will discuss the four decisions in details below.

4.3.1 Template for Senselet

Senselet is the core of Dandelion that abstracts in-sensor data processing. As shown in the object-oriented skeleton code in Figure 4.3, a senselet is a subclass that inherits the virtual base SenseletBase. SenseletBase declares a concise interface as a set of virtual
Figure 4.4: Pseudo code of a simplified SenseletBase, as a template to wrap an event loop skeleton for Senselet class

methods for concrete senselets. For example, the senselet class overrides OnCreate() to initialize its state, and overrides OnData() to receive new sensor data for processing. Compared to various native sensor programming abstractions, the senselet abstraction has three distinct features: 1) a just-fit interface for data processing, 2) smartphone style programming (very close to the skeleton code shown in Figure 4.2), and 3) platform-independence.
SenseletBase contains the template code written with sensor native programming abstraction, while leaving the implementation of the actual data processing to senselets. Essentially, a SenseletBase class plays two important roles. First, it acts as an adaptor between the senselet abstraction and the sensor native programming abstraction. Second, it hides low-level platform configurations, e.g. hardware parameters from the developer.

Since the implementation of a SenseletBase is platform-specific, it is the sensor vendor’s responsibility to provide SenseletBase as a basic library. When a Dandelion application is being installed on a smartphone, the developer-provided senselet code is linked with vendor-provided SenseletBase to generate the final senselet executable, as will be further addressed in Section 4.3.3.

Example: We use a simple example in Figure 4.4 to illustrate how SenseletBase works as a template. Suppose that the sensor platform requires any of its software modules to be coded as an event loop. SenseletBase populates such an event loop in the function _EventLoop(), with dispatching events to the proper methods implemented by a concrete Senselet class (e.g. OnStart and OnData).

4.3.2 Platform Services

Dandelion abstracts platform resource as a set of platform services, which resemble system calls in traditional OS. In designing platform services, we seek to strike a balance between their portability and the flexibility of programming; more platform services allow the developer to program senselet in a more flexible way, but they may be supported by fewer platforms.

By examining data processing in a wide range of body sensor applications, both in the market and reported in literature, we identify three core platform services for
Since the implementation of a SenseletBase is platform-specific, it is the sensor vendor's responsibility to provide specific senselet code, linked with vendor-provided SenseletBase to generate the final senselet executable, as will be further addressed in Section 3.3.

To make SenseletBase as a basic library, when a Dandelion application, both in the market and reported in literature, is being installed on a smartphone, the developer-event loop in the function _EventLoop(), with dispatching low-level platform configurations, e.g., hardware parameters, to the proper methods implemented by a concrete Senselet class (e.g., OnStart and OnData).

SenseletBase contains the template code written with sensor native programming abstractions, while leaving the implementation of any software module to the desired interval. The senselet can also use methods at the desired interval. The senselet may need to process data with timing, and use dynamic memory for storing intermediate processing results, and use "malloc()" to request dynamic memory for storing intermediate processing results, and use "free()" to release the memory after use.

A senselet executes on a body sensor, while the main application, as follows.

**Acquire sensor readings.** Get raw sensor readings, using either pull mode, where the senselet actively polls for readings:

```java
data = PollSensorData(sensor_id);
```

or push mode, where the senselet requests runtime to periodically deliver readings, with:

```java
RegisterSensorData(sensor_id, rate);
```

Figure 4.5: The internals of an example RMI PassData(). (a) The stub compiler generates both caller and callee stubs that translate the RMI into messages. Both main body and senselet can be caller or callee. (b) During the RMI, the caller transfers the data and its control flow to the callee senselets, as follows.
Sensor runtime delivers new reading to the senselet by invoking its OnData() method. A senselet may need to process data with timing, periodically or aperiodically. A senselet can register a timer with:

```c
timer_id = RegisterTimer(interval);
```

Consequently, sensor runtime will invoke its OnTimer() method at the desired interval. The senselet can also use UnRegisterTimer(timer_id) to cancel a registered timer.

Memory management. A senselet can use Malloc() to request dynamic memory for storing intermediate processing results, and use Free() to release the memory after use. Finally, we note that new services can be added as they become critical to data processing and widely supported by sensor platforms, e.g. storage.

### 4.3.3 Remote Method Invocations

A senselet executes on a body sensor, while the main application body still runs on the smartphone. However, to logically work as an integral application, they need 1) transfer data to each other, e.g. a senselet passes preliminary results to the main body for further processing, and 2) transfer control flow to each other, e.g. the main body commands to change the sampling rate used by a senselet and waits for a confirmation from it. Rather than require developers to hand-code integration with messages, Dandelion employs remote method invocation as the solution to address both requirements.

RMI is a cross-platform communication mechanism widely used in distributed systems such as Network File System. From the perspective of a developer, an RMI that happens between a senselet and the main body in Dandelion, appears to be very similar to a local call. All a developer needs to do is to specify the name and
types of parameters used by the remote method, using Interface Description Language (IDL) [87]. All RMIs in Dandelion are synchronous, i.e. the caller returns after the remote method finishes execution.

We provide a stub compiler that automatically generates the implementation of two stub procedures for a RMI, based on its IDL description. One stub is used by the caller to issue the RMI, with the data objects to transfer as method parameters. When the caller stub is invoked, it creates a message and marshals method parameters into the message payload. The other stub procedure is used by the callee to un-marshal the parameters from the received message and actually calls the remote method. Figure 4.5(a) illustrates the roles of the programmer, stub compiler, and runtime in implementing a RMI.

We choose RMI for integration because of two reasons. First, it is a widely used way for separated execution contexts to interact with each other. Second, IDL is already an essential part in smartphone development, e.g. AIDL in Android [34], XML for dbus in Maemo [59]. Recent proposal to offload execution from mobile devices also leverage IDL-based methods to serialize passed parameters [22].

Shared memory is an alternative solution for data transfer: smartphone and sensor runtimes collaboratively implement a memory coherence protocol by exchanging messages. With this scheme, senselets and the main body can access shared variables. While it may be desirable to support shared memory for smartphone developers who may be unfamiliar with distributed programming, its overhead is significant for body sensors, both in terms of implementation complexity and communication (i.e. frequently sending small memory updates over wireless link).

Example: We use a simple but general example to show how RMI transfers data and control flow. In this example, the senselet calls a remote method defined in the
Figure 4.6: The overview of Dandelion runtime architecture

main body, named PassData, to transfer processed data. The programmer describes the following method interface using XML-style IDL [59]:

```xml
<method name="PassData">
  <arg type="ai" name="Data" direction="in" />
</method>
```

The IDL description above indicates that the method PassData carries one parameter Data, which is an integer array (‘ai’). At compile time, our stub compiler generates the caller stub for the senselet:

```c
void PassData (int *Data, int actual_len) {...};
```

The senselet code simply calls PassData() provided by the caller stub to send the integer array, and the main body accordingly implements the method PassData() to receive the array. It is worth noting that during the RMI, the senselet code in fact
transfers its control flow to the main body (as shown in Figure 4.5(b)). Therefore, upon returning from PassData(), the senselet is ensured that the execution of remote method in the main body is finished. This guarantee is especially important when the main body uses RMI to configure the senselet, e.g. change the data sampling rate.

4.3.4 Senselet Executable Generation

The challenge of generating senselet executable comes from our design goal of platform transparency: platforms may have various Instruction Set Architectures (ISAs) that developers may not know about. One possible solution to bridge the ISA gap is Virtual Machine (VM). With the same VM installed on all target platforms, a senselet can be compiled into VM byte code and then interpreted by a VM during execution. Although light-weight VMs [45] have been developed for low-power processors to perform simple control tasks, the overhead of VM interpretation is still high for intensive data processing (10 times or more compared to native binary [25]). Therefore, Dandelion adopts the two-phase compilation technique that has been used for other heterogeneous distributed systems such as [66]. A developer compiles a senselet into intermediate representation (IR) before distribution (first phase). When a smartphone user installs the distribution (second phase) on her phone, the corresponding cross-compiler for her sensor translates the IR into sensor native binary. This technique relieves the application developer from worrying about various sensor ISAs while achieving the high performance of native execution.

4.4 Execution Support

The architecture of a running Dandelion system is shown in Figure 4.6. Dandelion employs a distributed middle layer or runtime to manage the execution and communi-
cation of senselets. It also provides resource exception to support application-specific handling of resource depletion during senselet execution.

4.4.1 Dandelion Runtime System

The Dandelion runtime, as the core of execution support, consists of multiple component runtimes: one smartphone runtime and one or more sensor runtimes. Component runtimes communicate with each other through messages. They employ a common message format to serve both RMIs in application code and runtime management functions. While messaging is the building block of Dandelion communication, it is only visible inside the runtime system and totally transparent to the application.

Smartphone Runtime

As the coordinator of the runtime system, the smartphone runtime serves three key management functions. First, when a body sensor application starts, the smartphone runtime discovers sensors by querying sensor runtimes. Second, the smartphone runtime manages the network connections between the smartphone and sensors according to the application requirement. It does so with existing networking capability, e.g. Bluetooth Serial Port Profile. Finally, the smartphone runtime sends senselet native binaries to the corresponding sensor runtimes for execution and stops senselet execution under the application’s request. Again, the communication between the smartphone runtime and the sensor runtimes is based on messages described above.

Sensor Runtime

To minimize the complexity and execution overhead of the sensor software stack, we opt a minimalist design for sensor runtime. A sensor runtime provides two core
functions: 1) it implements the very small set of platform services for senselets, as described in Section 4.3.2 and 2) it fulfills management commands from the smartphone runtime. For function 2, to further reduce the complexity of sensor runtime, we carefully leave most functions to smartphone runtime. For instance, to load a new senselet for execution, we perform the dynamic linking of the executable on the smartphone instead of within the sensor runtime [25], so that the sensor runtime only needs to receive and copy the linked executable into sensor memory. With this minimalist design, the sensor runtime is very lightweight in terms of memory and processor usage. Furthermore, the sensor runtime only requires a minimal set of primitives, namely interrupt handling, digitalized sensor output, and the configurable timer, from the underlying platform. As a result, the Dandelion sensor runtime can be built on top of various embedded OSes, e.g. uC/OSII, Mantis, TinyOS, SOS, Contiki, and with diverse sensor hardware, e.g. MSP430, ARM7, AVR. We will evaluate the sensor runtime design in Section 4.6.

4.4.2 Senselet Resource Exception

Due to the highly limited computing resources on body sensors, a senselet is likely to run out of resources. Unfortunately, it can be hard for developers to match senselet resource demand with available platform resources. On one hand, developers may have limited knowledge about target platforms; on the other hand, senselet resource demands may vary significantly during its execution (e.g. changing sampling rate with human activity). Dandelion supports resource exception to let developers gracefully deal with such resource mismatch during execution, in an application-specific manner. We define two types of resource exceptions for senselet:

- TimeException is raised when running out of processor time: e.g. senselet fails
MemException is raised when running out of memory: e.g. the stack is going to overflow.

The sensor runtime monitors platform resource utilization in a platform-specific way, e.g. read processor utilization from the schedule and raise exceptions by invoking corresponding exception handlers in senselet. Resource exception provides enough flexibility for application code to adapt to constrained sensor resource. For instance, when TimeException is raised, senselet code may lower its sampling rate and notify the main body with a RMI.

### 4.5 Prototype Realization

We have built a Dandelion prototype using Nokia N900, a smartphone based on Maemo Linux, and Rice Orbit, a multi-purpose sensor platform with a TI MSP430 microcontroller and a Bluetooth interface, as shown in Figure 4.7. Rice Orbit also has a tri-axis accelerometer that can be utilized for human activity monitoring, and an
amplified analog input that can be used to sample Electrocardiogram (EKG) signals. Rice Orbit works with a variety of embedded OSes that support TI MSP430, including \( \mu \)C/OS-II, SOS, Contiki, and TinyOS.

4.5.1 Smartphone-side Software

We implement the smartphone runtime using around 1000 lines of C++ code and link it as a static library with the main body. The smartphone runtime only requires network capability from the smartphone platform, therefore it is highly portable and can therefore be realized with all current smartphone platforms we are aware of, including Maemo, iPhone, Android, and S60.

We have built a stub compiler that takes IDL description to generate RMI stub source code for both the main body and senselets. We implement such stub compiler with around 500 lines of Python code, by referring to dbus implementation. We have validated the feasibility of two-phase compilation by using LLVM-msp430, a high-efficient compiler infrastructure with an experimental MSP430 backend, to successfully build simple senselets with two phases. However, since the MSP430 backend is under heavy development, in the evaluation we still choose MSPGCC, the widely-used MSP430 port of gcc, as the C/C++ compiler kit to produce senselet binaries. As LLVM-msp430 matures, we expect it to produce production-quality MSP430 binary with two-phase compilation.

4.5.2 Sensor-side Software

We implement the Dandelion sensor runtime on top of \( \mu \)C/OS-II [42], a popular, lean OS for resource-constrained embedded systems. Since \( \mu \)C/OS-II provides no direct support for any of three platform service, we only rely on it for multitasking and
implement all platform services inside sensor runtime. We use a µC/OS-II task to implement sensor runtime management functions, and use another task to execute senselet code (with a fixed 64 byte stack).

The sensor runtime is highly portable, thanks to its minimalist design. Its two major components, platform services and management functions can be similarly realized on top of most embedded OSes. In many cases, the platform services are just thin wrappers over existing OS support. It is worth noting that protothreads [26] can be used to implement blocking in synchronous RMI with a negligible overhead of 2 bytes, which is useful for purely event-driven OSes that lack direct support of blocking. What’s more, even without an underlying OS, it is possible to build sensor runtime on bare-metal with the following four hardware primitives: 1) hardware interrupts, 2) access to digitalized sensor data, e.g. through a built-in ADC or a digital interface with the sensing apparatus, 3) a configurable timer, and 4) a bi-directional, interrupt-enabled communication port that enables network capability. Such hardware primitives are available on most 16-bit and even some 8-bit microcontrollers.

4.6 Evaluation

To evaluate if Dandelion achieves its design goal, we develop in-sensor data processing from three real-world smartphone health applications and in three alternative styles: 1) the bare-bone style, without embedded OS or Dandelion; 2) the embedded-OS style, implemented with an event loop; 3) the Dandelion style, implemented as senselets. We compare the developers’ burdens in coding in these three styles by examining the resulted source code. To understand the cost in supporting transparency, we also experimentally quantify the memory and execution overhead of senselet and Dandelion runtime.
4.6.1 Benchmark Health Applications

We use the following three real-world applications in the benchmark:

FallDetector uses a wearable accelerometer to detect fall accidents occurred to the user and raises an alert to the main body accordingly. The main body reacts by sending out an SMS to registered phone numbers.

The EKG application [75] monitors the user’s heart rate by calculating the average RR interval in the real-time EKG trace. The main body can retrieve the calculated heart rate, or raw EKG traces for further analysis.

Pedometer [80] uses a shoe-mounted accelerometer to count user steps and recognize walking distance, and further calculates walking speed and consumed calorie. The main body regulates the thresholds for steps, speed, or consumed calorie, all calculated in the senselet. The senselet in turn notifies the main body when any quantity exceeds its threshold. The main body can also poll these quantities.

4.6.2 Source Code Examination

We use FallDetector as the simplest example to compare developer’s burdens in three programming styles. The data processing is simple: measure the acceleration energy (calculated as the magnitude of X, Y, Z readings) and apply a simple low-pass filter to the energy trace. If the filtered energy exceeds a pre-defined threshold, an alert is raised to the main body.

The FallDetector senselet is shown in Figure 4.8, which consists of only 17 lines of C++ code. The OnCreate() method requests periodic accelerometer readings at 50 Hz. Later on, OnData() is invoked by sensor runtime when new reading is acquired. OnData() detects fall accident and raises the alert by invoking the remote method FallAlert() of the main body. Senselet abstraction relieves the developer from dealing
```cpp
class SenseletFall : public SenseletBase {
public:

    SenseletFall () {_avg_energy = 0;};

    void OnCreate() {RegisterSensorData(ACCEL, 50);};

    void OnData(uint8_t *readings, uint16_t len) {
        uint16_t energy = readings[0]*readings[0] + \\
                         readings[1]*readings[1] + \\
                         readings[2]*readings[2];
        //do a simple low-pass filtering
        _avg_energy = _avg_energy / 2 + energy / 2;
        // detect fall accident with the filtered energy
        if (_avg_energy > THRESHOLD) {
            theMainBody.FallAlert();  //RMI
        }
    }

    void OnDestroy() {UnRegisterSensorData(ACCEL);};

private:

    uint16_t _avg_energy;
};
```

Figure 4.8: An implementation of FallDetector with senselet

with hardware and communication details.

For comparison, we also write FallDetector in the form of a main event loop, which is used for tasks with many embedded OSes, e.g. SOS, Contiki, and µC/OS-II. In developing the code, we favorably assume that the OS provides all peripheral drivers, event queues, and supports blocking, which may not be true for all. Even with this favorable assumption, the same data processing takes three times as many
lines of code. The bare-metal implementation is even more verbose. While it also implements a main loop driven by several hardware interrupts to perform processing and communication, it has to deal all low-level, hardware-related operations, e.g. manually triggering the power state transitions.

The same observations also apply to other two benchmark applications (source statistics in Table 4.1). As reflected in the numbers of source lines, without Dandelion support, complex data processing and communication requires more programming efforts; what is even worse, the resulting code is non-portable and error-prone.

By further comparing the source code, we can see Dandelion saves considerable development efforts in three aspects.

First, senselet only exposes to developers a very concise interface necessary for in-sensor data processing. The other two direct sensor programming styles put extra burden on developers: populate a main loop woven with the data processing and drive the loop with (sometimes low-level) events.

Second, the use of RMI facilitates integration and relieves the developer from dealing with smartphone-sensor communication. Compared to issuing a RMI with a single line in senselet, the other two styles need to spend 30-50 lines of code to manually construct a message and send through the wireless link, when accounting for link reliability. Additionally, the bare-bone code has to manage the power state of Bluetooth interface.

Third, platform independence makes senselet code portable. For example, to periodically acquire sensor readings; embedded OS style requires using timer event to drive the acquisition, while the bare-bone style additionally requires manually configure ADC. In contrast, Dandelion sensor reading service covers these differences.
<table>
<thead>
<tr>
<th>Programming style</th>
<th>FallDetector</th>
<th>EKG</th>
<th>Pedometer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dandelion</td>
<td>17</td>
<td>96</td>
<td>258</td>
</tr>
<tr>
<td>Embedded OS</td>
<td>72</td>
<td>146</td>
<td>248</td>
</tr>
<tr>
<td>Bare-bone</td>
<td>84</td>
<td>194</td>
<td>296</td>
</tr>
</tbody>
</table>

Table 4.1: The source lines of code in developing in-sensor data processing, with different programming styles

<table>
<thead>
<tr>
<th>Module</th>
<th>RAM</th>
<th>ROM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensor reading service</td>
<td>12</td>
<td>144</td>
</tr>
<tr>
<td>Timer service</td>
<td>36</td>
<td>250</td>
</tr>
<tr>
<td>Memory management service</td>
<td>4</td>
<td>248</td>
</tr>
<tr>
<td>Message communication</td>
<td>64</td>
<td>286</td>
</tr>
<tr>
<td>Management functions</td>
<td>64</td>
<td>818</td>
</tr>
<tr>
<td>Total</td>
<td>180</td>
<td>1746</td>
</tr>
</tbody>
</table>

Table 4.2: Memory breakdown of sensor runtime, in byte

4.6.3 Overhead Measurements

We measure the overhead of Dandelion framework in terms of both memory consumption and processor cycle consumption. In the measurement, we compiled all code using MSPGCC version 3.2.3, with optimization level -O2. We are aware of that the overhead is affected by the choice of the underlying OS: complete support for runtime functionalities from OS can largely simplify sensor runtime implementation and therefore reduces its overhead. In the evaluation, we estimate the upper limit
Table 4.3: Memory overhead of senselet executables, in byte and the percentage of the whole sensor memory. Assume whole memory of 10KB RAM and 48KB ROM of runtime overhead by using µC/OS-II merely for multitasking, and implement most sensor runtime functionalities from scratch, instead of choosing many other embedded OSes with existing support for peripheral drivers, timer, or communication.

**Memory Overhead**

We break down the memory overhead of Dandelion into two parts: 1) that of sensor runtime, and 2) that of senselet executable, including template and RMI stubs, as shown in Table 4.5 and Table 4.4.

The runtime memory overhead, due to the minimalist design, is a small constant: 180 bytes of RAM and 1746 bytes of ROM. For the second part, in a single senselet executable, template takes a small, fixed amount memory. Additionally, each RMI stub only uses 50-100 bytes ROM. For a typical sensor controller, the TI MSP430F161 (10KB RAM, 48KB ROM) that is widely used in various sensors such as Rice Orbit and Shimmer [1], the fixed overhead is less than 5% of the whole memory.

The measurements have two implications. First, the small overall memory overhead allows Dandelion be implemented with many resource-constrained sensors. Second, the compact template and RMI stubs incur small energy overhead when a sense-
To gain insight into the execution overhead of a senselet, we first measure execution overhead in common operations of sensor runtime. Note that we do not include cycles spent on sending and receiving message bytes into message RX/TX overhead. Further, the overhead of an RMI varies with the number and types of its parameters, and 300 cycles are typical for passing an integer array. Results in Table 4.4 show that

Table 4.4: Execution overhead in common sensor runtime operations, in processor cycles

<table>
<thead>
<tr>
<th>Operation</th>
<th>#Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer notification</td>
<td>53</td>
</tr>
<tr>
<td>RMI</td>
<td>50-300</td>
</tr>
<tr>
<td>Message RX</td>
<td>924</td>
</tr>
<tr>
<td>Malloc</td>
<td>239</td>
</tr>
<tr>
<td>Sensor reading</td>
<td>130-168</td>
</tr>
<tr>
<td>Load senselet</td>
<td>716</td>
</tr>
<tr>
<td>Message TX</td>
<td>720</td>
</tr>
<tr>
<td>Free</td>
<td>120</td>
</tr>
</tbody>
</table>

Table 4.5: Execution overhead in the most common data processing path, in processor cycles and the percentage of processor time in each sampling period. FallDetector and Pedometer acquire sensor data at 50Hz, while EKG does at 512Hz. Assume a clock rate of 4MHz

<table>
<thead>
<tr>
<th></th>
<th>FallDetector</th>
<th>EKG</th>
<th>Pedometer</th>
</tr>
</thead>
<tbody>
<tr>
<td># Cycles</td>
<td>251</td>
<td>221</td>
<td>251</td>
</tr>
<tr>
<td>% per sampling period</td>
<td>0.3%</td>
<td>3%</td>
<td>0.3%</td>
</tr>
</tbody>
</table>

let executable is transmitted over wireless link.

**Execution Overhead**

To gain insight into the execution overhead of a senselet, we first measure execution overhead in common operations of sensor runtime. Note that we do not include cycles spent on sending and receiving message bytes into message RX/TX overhead. Further, the overhead of an RMI varies with the number and types of its parameters, and 300 cycles are typical for passing an integer array. Results in Table 4.4 show that
the sensor runtime incurs small execution overhead; with a typical processor clock rate of 4MHz, even the most costly operation, message RX, takes less than 250 us.

To estimate the execution overhead in long-run, we further measure the performance of the most common execution path in senselets. Since in all three senselets data processing is driven by periodic sampling, we define such path as data processing performed in each sampling period (without RMI involved), e.g. EKG senselet reads in new data and computes to determine if it is in R wave. The results in Table 4.5 show that, even as the data processing becomes intensive, Dandelion execution overhead remains small (around 250 cycles). Even with the most challenging EKG benchmark, where data processing consumes around 3900 cycles (∼50% of processor time) in every 2 ms sampling period, the execution overhead only takes ∼3% processor time. We have observed that the execution overhead comes from timer notification, sensor reading, and the template code.

4.7 Summary

We present Dandelion, a novel programming framework for phone-centered wireless body sensor applications. Dandelion allows developers to easily write data processing code to be executed on sensors, in a programming style similar to traditional smartphone development. With the minimalist design of the runtime system, Dandelion incurs very small overhead and therefore can be easily ported to various resource-constrained sensor platforms. We believe that by enabling smartphone developers to easily write body sensor code, Dandelion supports a healthy ecosystem that promotes the interests of all involved parties: sensor vendors, smartphone developers, and users.
Chapter 5

Energy-efficient Distributed Shared Memory Over Mobile Processors

5.1 Overview of Reflex

Our goal in developing Reflex is to relieve developers from directly dealing with the heterogeneous, distributed platform so that they can easily port legacy applications and develop new ones to leverage peripheral processors for energy efficiency.

5.1.1 Programming Constraints

In order to make the system design practical, we require application developers to be aware of the peripheral processor in the following two ways. First, developers need to properly encapsulate the code for a peripheral processor as a peripheral module. Accordingly, we call the code designated for the central processor the central module. To ease the development of peripheral modules, Reflex provides a virtual base class called ModuleBase. Developers write a peripheral module as a subclass of ModuleBase.

Second, Reflex restricts system services that a peripheral module can use, since a peripheral processor does not have all the resources available on the central processor, e.g., Internet connectivity. Moreover, sensor data processing, according to our observation, only requires limited system services: dynamic memory, timer, and sensor data acquisition.
5.1.2 How Reflex Works

With an application properly developed as discussed above, the Reflex compiler and distributed runtime collectively create an illusion that the application is running on a single processor. The key to this illusion is to support data exchange among modules running on different processors. Reflex recognizes two forms of data exchange: procedure call and shared memory. Reflex automatically translates the procedure call between distributed modules into a synchronous remote procedure call (RPC) and supports the shared memory with software DSM. While the realization of RPC is standard and straightforward, the DSM poses significant technical challenges due to the extreme architectural asymmetry.

Reflex resorts to DSM in order to conceal the underpinning messages, with the following key rationales. First, the compiler and runtime can automate the architecture-dependent mechanics of message passing. Second, the overhead of DSM diminishes as the access contention decreases [46]. Fortunately, sensing applications tend to have light access contention, since only the data processing task frequently accesses the shared objects. Finally, with compiler techniques, the overhead can be reduced close to what can be achieved by hand-optimized message passing.

When the application is launched, Reflex treats the central module as a normal smartphone application and executes it on the central processor. Then, the runtime ships the peripheral module binaries to proper peripheral processors for execution. When the central module terminates, the Reflex runtime will terminate all running peripheral modules of the application.
5.1.3 Hardware Requirements

We design Reflex to have a minimalist hardware requirement for peripheral processors and their integration with the central processor: at least 8KB of ROM, several KB of RAM, the capability to handle interrupts, and the capability to acquire sensor data without involving the central processor. Reflex only requires a data link capable of interrupt-driven and bi-directional data exchange between peripheral and central processors. The data link can be a low-speed hardware bus, e.g., I2C; it can also be realized with a physically shared memory and hardware support for inter-processor interrupts, as in TI OMAP4.

Reflex does not require a hardware Memory Management Unit (MMU). Many existing distributed systems leverage MMU to create a shared address space or to trap local memory operations. Some peripheral processors indeed have dedicated MMU, e.g., Cortex-M3 in OMAP4. However, it is very difficult to implement MMU for even weaker peripheral processors that are necessary for energy efficient sensing applications.

5.2 Distributed Runtime Design

Figure 5.1 shows the structure of Reflex. Reflex currently is implemented as a distributed runtime and a library. This user-level implementation makes Reflex compatible with existing smartphone operating systems and platforms, although more disruptive kernel-level implementation is possible.

As shown in Figure 5.1, the Reflex runtime has one component on each processor: the central runtime on the central processor as the coordinator and peripheral runtimes running on peripheral processors. Locally, a component runtime provides
modules with a unified abstraction of the resources. We next describe in detail two important services provided by the distributed runtime.

5.2.1 Module Execution Model

Unlike many other heterogeneous systems that execute offloaded code synchronously, Reflex executes modules in parallel. To accommodate the DSM and the RPC, the Reflex runtime is enhanced to execute modules with an event-driven model, which is implemented with native execution units (e.g., thread or process). A Reflex event is a formatted data unit sent to a module from either the other modules or the local component runtime. These data units can be inter-processor messages (i.e., the primitive used to implement the DSM and the RPC) or can be used in accessing local system services. A module has a private event queue to which the local component
runtime asynchronously delivers events.

A module must poll its event queue to retrieve an event. Polling is non-blocking and lightweight: when the event queue is empty, it only involves checking a flag variable. Such a polling-based model provides the module with the control over exactly when to handle asynchronous events. Polling occurs 1) when the control flow returns from an event handler into the event loop and 2) according to the instrumentation by the Reflex compiler. In addition, when a module stalls its execution of the application code waiting to receive certain DSM or RPC messages, it also polls its event queue to process DSM and RPC messages that are received but not yet handled. Compared with preemptive event handling, the polling-based model prevents data processing on a weak processor from being overwhelmed by coherence requests from stronger processors.

With the event-driven execution model, the code of a module consists of three parts: the event loop as the skeleton of the module, which keeps retrieving and dispatching various events; the application-specific handling of the events, as provided by the developer; and functions that support DSM and RPC and wrap the system services.

5.2.2 Message Transport

Globally, component runtimes collectively implement a lightweight message transport as shown in Figure 5.1. The message transport serves as the only interface for communication among runtime components as well as the lowest-level primitive of communication among distributed application modules, a design also adopted by many other heterogeneous systems, e.g., Helios [66] and Barrelfish [10]. A message will be delivered as an event to the destination module.
The message transport is best-effort and does not guarantee reliable delivery. This is because higher-level protocols on top of the message transport, such as the DSM and the RPC, all use messages in request/response pairs between the two communicating modules. Thus, the response message implicitly indicates the delivery of the earlier request message. The higher-level protocols employ message sequence numbers to identify and therefore discard duplicate messages due to timeout and retransmission.

5.3 Software DSM Design

We now present the design of the Reflex software DSM. The key design objectives are to leverage the architectural asymmetry for energy efficiency while minimizing performance overhead.

5.3.1 General Design Choices

Overall, the Reflex DSM realizes release consistency and maintains memory coherence at object-level, as will be discussed below.

Memory Consistency Model: To developers, the memory consistency model defines the expected outcome of memory accesses. The Reflex DSM implements release consistency [85, 15]. Release consistency introduces two synchronization operations (acquire and release) and guarantees that a correctly synchronized application will exhibit sequential memory consistency, just as if the application were running on a single processor. By correctly, we mean using synchronization operations (release and then acquire) to construct critical sections to prevent concurrent accesses to the same object, if at least one of these accesses is write.

We choose release consistency for two reasons. First, it is easy for smartphone developers to use. We notice that writing synchronized applications is already a
common task in contemporary smartphone development, e.g., Android application
development in Java greatly encourages synchronized applications [61]. Second, re-
lease consistency is lightweight to implement: it can greatly reduce the communication 
overhead by allowing most communication to happen at synchronization points, a sig-
nificant benefit to our targeted architecture where the inter-processor communication 
latency is relatively high.

Coherence Granularity: A DSM design must determine the basic memory unit 
for which coherence is maintained, i.e., the coherence granularity. A larger unit may 
better leverage the spatial locality to amortize the communication overhead, but also 
increases unnecessary data transfer due to false sharing.

Many DSM solutions support block-level granularity, using a fixed-size memory 
block (or a fixed-size page defined by MMU) as the basic memory unit for coherence. 
In contrast, the Reflex DSM uses software objects as the basic memory unit for two 
reasons. First, Reflex does not assume MMU. Second, a pre-defined block size leads 
to a fixed tradeoff between the communication overhead and the false sharing of the 
resulting DSM. In contrast, object-level granularity allows the Reflex DSM to employ 
the compiler-supplied program information to better leverage locality.

The Reflex DSM supports sharing two types of objects: global objects and heap 
objects. In facing the extremely limited local memory on peripheral processors, a 
module only caches heap objects it accesses, rather than all heap objects in the 
application.

The Reflex DSM associates each shared object with an application-wide integer ID that is valid across all modules. The IDs are allocated in ascending order as objects are created, either statically by the compiler (for global objects) or dynamically by the DSM (for heap objects). Each module has its own range to allocate new IDs so
it can do so independently. The DSM maintains a table of shared objects in every module; each entry in the table records the metadata of one shared object, including the object ID, type, its local address range, and information about other modules that also share this object.

We next present the coherence protocol based on the design choices made above. The protocol specifies how modules sharing an object should behave in accessing the object.

5.3.2 Protocol Invariant

At the core of our coherence protocol is the following invariant:

*Any coherence communication between two processors must be initiated by the stronger one.*

This invariant guarantees the energy efficiency of the Reflex DSM: in accessing shared objects, a weak processor will never wake up stronger processors with coherence communication. Such a unique invariant clearly distinguishes our coherence protocol from existing ones, as will be described in detail below.

5.3.3 Asymmetric Module Roles

Given a shared memory object in an application, we define its sharing group as all modules in the application that access it. Each module in a sharing group has a local copy of the shared object. All write/read operations by a module on a shared object are actually performed on the local copy of the object.

In a sharing group, all modules have asymmetric roles based on the relative processing power of the processors that these modules are executed with. To facilitate the discussion, we mention a module as strong or weak to refer to the relative process-
ing power of its associated processor. The weakest module in the group is the home of the object, while the other modules are requesters of the object. Since a module may access multiple shared objects, it can participate in multiple sharing groups and play different roles in them.

While the home of a global object is assigned by the compiler statically, the home of a heap object is determined by the DSM on-the-fly and automatically. Initially, a heap object O allocated by a module M has M as its home; later, if a pointer to O is passed to another module N that is weaker than M, N notifies M that itself will be the new home of O and retrieves the metadata of O from M. M therefore redirects any subsequent requester of O to the new home N, a one-time effort for each requester. The role asymmetry employed by the Reflex DSM is different from existing heterogeneous DSM solutions in an important way. With performance as the primary goal, most existing heterogeneous DSM solutions employ the most resource-rich (and power-hungry) processor to serve requesters, e.g., [37, 93]. In contrast, the Reflex DSM greatly favors energy efficiency and, therefore, uses the weakest processor to host the home role. Only if the weakest processor handles memory requests will the protocol invariant (Section 5.3.2) hold. Therefore, other stronger processors are able to remain in sleep mode as much as possible.

5.3.4 Finite-state Machine Specification

We now provide details regarding the behaviors of the home and requesters in terms of finite-state machines. A module has a state with regard to any sharing group it participates in; if participating in multiple sharing groups, it will have multiple independent states with regard to each of those groups. A module state can be one of the following two:
The home can be either Owned or Invalid.

A requester can be Owned, Invalid, or Shared.

The states have the same semantics as in most coherence protocols. In a sharing group, conceptually one and only one module is in the Owned state. A module in the Owned state can read and write to its local copy; a module in the Invalid state can neither read nor write to its local copy; a requester module in the Shared state can read but not write to its local copy. A module in the Owned or Shared state has the most updated value of the shared object in its local copy.

Modules in a sharing group change states only by requests sent by a requester. The home module never initiates a change itself. A requester module uses four requests: write_miss, write_back, read_miss, and revalidate.

Figure 5.2 (a) and (b) illustrate the interactions between a requester and the home. In the two cases, the requester needs to read and write to the object, respectively. Before reading the object (Figure 5.2 (a)), the requester in the Invalid state sends a read_miss request to the home. The home responds by sending back the most updated value of the shared object. The read_miss request will change the state of the requester to Shared. Later, when the requester sends a revalidate request to the home, the home responds to indicate that the object has not been modified since the last read_miss request. Therefore, the requester module remains in the Shared state. If the home responds that the object has been modified (not shown in the figure), the requester will transit to the Invalid state, and it will need to send read_miss again prior to any further read of the object. Note that revalidate is only necessary during synchronization, as we will explain in details in Section 5.3.6.
Before writing to the object (Figure 5.2 (b)), the requester in the Shared or Invalid state sends a write_miss request to the home in order to transit into the Owned state; it sends a write_back request (along with the latest value from the local copy) to the home in order to transit back into the Shared state. Accordingly, the home will transit to Invalid and Owned after responding to write_miss and write_back requests, respectively.

The protocol described above introduces two types of latency in applications, namely the home latency and the requester latency. The home latency is introduced when a home in the Invalid state stalls its execution and passively waits until it is in the Owned state. The requester latency is introduced when a requester stalls its
execution to wait for responses to read_miss or write_miss from the home. We will experimentally evaluate the impact of the above latency to applications in Section 5.6.

5.3.5 Support for Synchronization

Release consistency defines two synchronization operations for applications: acquire and release. The Reflex DSM provides lock to support both operations, implementing a lock as a shared variable using the coherence protocol. For example, to acquire or release a lock, a requester sends a write_miss or write_back request for the lock object to the home of the lock, respectively.

According to release consistency, if a module performs release and another module subsequently performs acquire, the first module must make its updates visible to the second one. In propagating updates during synchronization, homes are lazy and requesters are eager. When the home releases a lock, it performs no coherence communication; it will send an updated value to a requester only when the requester asks for it. In contrast, when a requester releases a lock, it firstly examines the sharing group of the lock, looking for modules weaker than itself, and then writes back all its updated objects that have been requested from those modules.

Such asymmetric module behaviors are critical to the design goals of the Reflex DSM. The home’s laziness guarantees high energy efficiency, as it maintains the protocol invariant stated in Section 5.3.2. At the same time, the requester’s eagerness keeps the home’s stalling period small in order to minimize the resulting disruption to frequent processing.
5.3.6 Notable Design Aspects

We next highlight three notable design aspects that are essential to the objectives of the Reflex DSM. First, a home resolves its access of an unavailable object by passive waiting. This is to maintain the protocol invariant: running on the weakest processor, the home cannot initiate communication to other modules.

Second, to keep an object updated, a requester in the Shared state actively revalidates the object with the home. This is unlike most other DSMs where a read-only object is passively invalidated or updated by other hosts. The reason is that the home cannot send out messages to invalidate or update requester’s copies. Instead, the home tracks whether a Shared requester’s copy is still up-to-date, by observing whether the object has been written since the requester’s last read_miss or write_back request. Accordingly, the home responds to revalidate requests, as described in Section 5.3.4.

One may think that the requester-initiated revalidation incurs excessive communication, but actually it does not. In a nutshell, on acquire of a lock, requester will revalidate all its objects that have the same home as the lock has, rather than sending a revalidation request before each read access. This is because a module is guaranteed to see the updated value only after each acquire operation. In addition, the Reflex DSM piggybacks the communication of revalidation to that of acquire, adding no more than one byte per message.

Third, the Reflex DSM aggressively reduces the home latency at the cost of increased requester latency. This is achieved by keeping the home in the Owned state as much as possible, with the following designs. 1) In propagating updates during synchronization, requesters are eager and the home is lazy, as described in Section 5.3.5. 2) The home handles coherence requests only when its frequent processing is idle or stalled, supported by the execution model of modules as discussed in Section 5.2.1.
This is because of the extreme resource asymmetry: requesters, running on stronger processors, can produce requests at a much higher rate than the home, running on the weakest processor, can handle. Furthermore, we choose to mitigate the requester latency by compiler optimization described in Section 5.4 below.

5.4 Compiler Support for DSM

Without assuming MMU support, the Reflex DSM cannot perform coherence operations within trap handlers, as many other DSM systems do. Instead, the Reflex compiler statically automates invocations to coherence operations in developer’s code. In addition, the compiler aggregates nearby coherence operations to greatly reduce the overhead of DSM.

5.4.1 Code Instrumentation

In order to realize the coherence protocol specified above, the Reflex compiler analyzes the application code and instruments two types of code: pre-access and post-access. By design, for each access of a shared object the Reflex compiler inserts necessary pre-access immediately before the access. The pre-access code determines the target object of the access and checks the corresponding module state. It does nothing if the state is Owned (before read or write access) or Shared (before read access). Otherwise, it performs coherence operations that are specific to the nature of the access (read or write) and the role of the module (home or requester). In addition, after a write access in a requester module the Reflex compiler inserts post-access code to send out a write_back request and sets the requester back to Shared.
5.4.2 Optimization for Communication

The instrumented code introduces overhead in both checking the module state and inter-module communication. The Reflex compiler reduces the former overhead by employing the batching technique similar to that of Shasta [79]: if a module state has been checked by earlier pre-access and no related coherence communication has occurred since then, a later pre-access code can be eliminated. In the following discussion, we will focus on optimizations for reducing the communication overhead.

**Batch Prefetching.** The Reflex DSM maintains coherence for each shared object. However, fetching each object with an individual message incurs high communication overhead. The Reflex DSM leverages spatial locality to amortize the communication overhead by speculatively fetching multiple objects in a batch.

Once pre-access code finds it is necessary to send read\_miss or write\_miss to fetch an object from the home, the DSM also examines objects that have the same home and with adjacent IDs. Since object IDs are allocated in ascending order, having adjacent IDs implies potential spatial locality in accessing such objects: they are either global objects defined close to each other in the source code or heap objects allocated consecutively during execution. The DSM adds multiple objects to a batch, fetches them and receives responses with two single messages, respectively. We will experimentally show the impact of batch parameters in Section 5.6.

**Deferred Write-back.** Due to temporal locality, it is common that a shared object is repeatedly accessed in a short period of time, for example, in a loop. By only inserting one post-access after all these accesses, the Reflex compiler essentially merges multiple write\_back requests of the same object, greatly reducing the number of messages. A requester should defer write\_back only moderately, in order to keep
the home latency low. Given that requesters are executed much faster than the home, DSM leverages release consistency to defer all write_backs until the next release. The compiler does so by inserting post-access right before each release to send out all deferred write_backs.

To summarize, by design, before each access of shared objects the compiler will instrument pre-access; however, after optimization only a small, necessary portion of such code is actually emitted. If pre-access in requesters finds out that the current module state disallows the memory access, it immediately sends a message of read_miss or write_miss (with batch-prefetching). In addition, the compiler instruments post-access before each call site of release. Such code in requesters immediately sends a message of all deferred write_backs.

5.5 Prototype Realization

We have implemented the complete Reflex design in a modular way for portability. In particular, DSM is fully implemented as a library that is linked into the module code, as shown in Figure 5.1. The DSM internally uses the message transport supplied by the Reflex runtime. The Reflex compiler instruments the developer’s code with invocations to coherence operations defined in the module library. Our Reflex implementation, with small hardware-specific revisions, runs on two hardware platforms, TI OMAP4 and a custom tri-processor platform.

5.5.1 Reflex Runtime

We realize the Reflex runtime support with the software structure shown in Figure 5.1. On the bottom of Reflex is a Platform Abstraction Layer, which wraps platform-specific functions, such as kernel API and hardware message passing. On top of the
Platform Abstraction Layer, we build the Reflex runtime and the module library that implements DSM and RPC.

The execution model of a module is implemented with the module library, which resides with the runtime and is dynamically linked with the developer’s code during module execution. The module library is implemented with around 1500 lines of commented C/C++ code. Corresponding to the module code organization described in Section 4.1, the library consists of routines that are for the event loop, for DSM and RPC, and for wrapping system services. The Reflex runtime is implemented in around 4000 lines of commented C/C++ code. The central runtime manages the execution of peripheral modules. The message transport dynamically allocates memory for messages in order to reuse the limited memory.

5.5.2 Reflex Compiler Toolkit

We build the Reflex compiler as a transformation pass on top of the LLVM compiler infrastructure [44]. LLVM compiles the source code into LLVM-IR and the Reflex compiler instruments the IR for DSM. Before the module is deployed as part of the application, the module IR is transformed into to a native binary depending on the target processor. The object-level coherence of the Reflex DSM requires object types to be determined at compile time. To meet such a need, currently Reflex supports application development with a strong-typed subset of C++, by applying rules such as using typed malloc, disallowing pointer arithmetic and disallowing type-casting, a strategy used in many embedded systems [24]. After compiling the application source code into the LLVM IR, the Reflex compiler certifies that the IR is strong-typed; it then outputs descriptions of all types in Interface Description Language (IDL).

Reflex includes a small stub compiler that produces code for (un)marshaling ob-
jects across different ISAs. Implemented with around 700 lines of Python code, the stub compiler takes the compiler-generated type descriptions and produces (un)marshaling code used by DSM and RPC. The major task of the (un)marshaling code is to convert the formats of typed objects. Most data conversion is straightforward except for the pointers. Since processors have separate address spaces, one pointer in a module needs to be converted to be used by another module. To do so, the (un)marshaling code uses a portable pointer as the intermediate format among local pointers in separate address spaces.

A portable pointer is 32-bit (the same as the longest local pointer in the system), encoded with the following information: object ID (13 bit), the byte offset within the object (10 bit), type id (6 bit, as used in the compiler-generated IDL), the home id (3 bit). Before sending out a local pointer via message, the marshaling code replaces the local pointer with a portable pointer in place. On receiving a portable pointer via message, the unmarshaling code retrieves the corresponding local pointer based on the encoded object ID and offset. If the unmarshaling code finds the object ID new to this module (e.g., a heap object newly created by another module), the code allocates an object on the local heap with the type specified in the portable pointer, and adds a corresponding entry in the local object table.

5.5.3 Hardware Platform

Reflex is highly portable, thanks to its layered structure. After replacing the message passing implementation and changing hundreds of source lines of the Platform Abstraction Layer, we have ported Reflex to two platforms: TI OMAP4 and a custom, tri-processor hardware prototype.
TI OMAP4

We have realized Reflex on TI OMAP4. We build the Platform Abstraction Layer to wrap the Syslink inter-processor message passing API, the Linux kernel, and the SYS/BIOS kernel. In particular, the message passing abstraction offered by Syslink enables us to implement the Reflex DSM on OMAP4, despite the fact that message passing is actually using a non-coherent shared memory. Due to page limit, in the rest of the paper we will focus on the implementation and evaluation using the tri-processor hardware prototype, which is introduced for its first time by this paper.

Tri-processor system prototype

We developed the tri-processor system prototype by extending a Nokia N900 smartphone. As shown in Figure 5.3, the prototype employs N900’s powerful OMAP3630
Table 5.1: The processors used in the tri-processor prototype. At run time, unused processors are turned off to reduce the system idle power.

<table>
<thead>
<tr>
<th></th>
<th>OMAP3</th>
<th>LPC</th>
<th>MSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Freq.</td>
<td>600MHz</td>
<td>72MHz</td>
<td>3MHz</td>
</tr>
<tr>
<td>Local Memory</td>
<td>256KB</td>
<td>8KB</td>
<td>10KB</td>
</tr>
<tr>
<td>Local Storage</td>
<td>32GB</td>
<td>32KB</td>
<td>55KB</td>
</tr>
<tr>
<td>Active Power</td>
<td>200mW</td>
<td>42.9mW</td>
<td>7.5mW</td>
</tr>
<tr>
<td>Idle Power</td>
<td>13.4mW</td>
<td>7mW</td>
<td>3.2mW</td>
</tr>
</tbody>
</table>

CPU (OMAP3) as the central processor and employs two ultra-low power microcontrollers, LPC1343 (LPC) and MSP430F1611 (MSP), as the peripheral processors. Table 5.1 summarizes the characteristics of these three processors. The prototype system uses Maemo Linux shipped with N900 as the central kernel and runs two separated µC/OS-II [42] on two peripheral processors. Built on top of the Platform Abstraction Layer, the peripheral runtime is a set of procedures linked with the vanilla µC/OS-II kernel. Peripheral modules are executed as µC/OS-II tasks.

**I2C-based interconnect.** The three processors are integrated via an I2C bus at 100KHz and physically share no memory. In order to physically access the I2C interface and interrupt line of OMAP3, we remove the N900’s camera module and hijack its connector with OMAP3. In moving data on the I2C bus, a byte chunk with a size of typical Reflex messages usually takes tens of milliseconds (6ms for a 48 byte chunk and 18ms for a 176 byte chunk). Built on top of the I2C bus, the message transport of Reflex runtime itself incurs little computing overhead, which takes 1500 cycles (MSP), 891 cycles (LPC), and 1800 cycles (OMAP3) in sending a
message, and 1560 cycles (MSP), 1612 cycles (LPC), and 1800 cycles (OMAP3) in receiving a message, excluding actual byte sending/receiving. Compared to the tens of milliseconds interconnect baseline latency, the computing overhead is negligible.

**Sensors.** N900, like all commercial mobile devices, tightly integrates its built-in sensors with the central processor (OMAP3). Therefore, it is very difficult for a peripheral processor to access the built-in sensors without waking up the central processor, violating the hardware requirement of Reflex as outlined in Section 5.1.3. As illustrated by Figure 5.3, we add a KXM52 tri-axis accelerometer to MSP through ADC; we add an analog microphone and an MN5010HS GPS receiver to LPC through ADC and UART, respectively.

### 5.6 Evaluation

We evaluate Reflex and its DSM with the tri-processor prototype reported in Section 5.6 and four real-world smartphone sensing applications.

#### 5.6.1 Benchmark Applications

Since there is no standard sensing application benchmark, we employ a combination of classic and recently reported applications as benchmarks as detailed in Appendix. We implement each benchmark in three ways: N900’s programming framework (Legacy), on the tri-processor prototype with message passing (Message) and with the Reflex DSM (Reflex DSM). A simple GUI is implemented for central modules using the Qt framework of the N900. Legacy implementations use the N900’s built-in sensors while the other two use sensors connected to the peripheral processors. Like most smartphone sensing applications, these benchmark applications spend most time in
common scenarios of simple sensor data processing, which determine the overall energy characteristics of the application.

5.6.2 Overall Performance of Reflex

We first examine how well Reflex as a whole achieves its goal in saving energy, by measuring the power of the entire system. To do so, we sample current and voltage using USB-2533 (from Measurement Computing) at a measurement rate of 100 HZ. For Legacy implementations, we measure the power of N900 from its phone-battery hardware interface directly: we physically tap into the interface and simultaneously sample the current (using a 0.1ohm current sense resistor) and the voltage. For Reflex-based implementations on the tri-processor prototype, we additionally include the power of all added hardware. In both cases, we power off unused hardware such as baseband.

Sensing applications in benchmarks perform periodic processing. We benchmark an application scenario by running it for a given period of time (typically 10 min) with Legacy and Reflex, respectively. During the benchmark period, Legacy and Reflex finish the same amount of sensor data processing. Therefore, the average power over the benchmark period reflects the system energy efficiency.

As expected from the addition of low-power cores, our measurement shows that Reflex reduces the system power consumption by up to 83% for the most common scenarios of each benchmark. Only in scenarios when the central processor is involved do Reflex-based implementations incur slight power overhead (around 3%) from the added peripheral processors. Because such scenarios are relatively rare in real life usage, Reflex-based implementations will be significantly more efficient than legacy implementations. With the per scenario actual power measurement, we emulate the
scenario transitions (based on the application usage in Table 2 in Appendix) to compute the application average power. As shown in Figure 5.4, compared to Legacy implementations, Reflex reduces the average system power consumption by up to 81% (65% on average).

5.6.3 Source Code Examination

We investigate how Reflex facilitates app development by examining the source code of the benchmarks. First, how is Reflex different from the legacy development style? As shown in Figure 5.5, the Reflex-based and Legacy implementations of the same benchmark have very similar numbers of source lines of code; more importantly, they share most of the source code with 95% identical source lines. Their source code only differs in the way of gluing the programming framework and invoking the system services described in Section 5.1. This observation validates that Reflex has achieved its goal of maintaining the contemporary programming style and facilitating porting
steps. (UI) to query the step information. The two modules share a few
counts steps and a central module that provides a user interface
and recently reported smartphone sensing applications. Table 2
employ the following four benchmarks, a combination of classic
Since there is no standard sensing application benchmark, we
8.1 Benchmark Applications

Figure 7. compared to Legacy implementation s, Reflex reduces
the overall energy characteristics of the applications.

Second, how does DSM ease the application development compared to message passing? To make the source code comparison fair, we hand-optimized the message-based code with best efforts, including implementing the common-used message manipulations as subroutines and managing objects with a table and iterating over the table to operate objects. DSM greatly eases the application development by enabling data exchange in benchmarks to be concisely coded as in the source code of Figure 2.5. Compared to the message-passing implementation, DSM reduces the lines of code by 30% on average as shown in Figure 5.5. Compared with message passing, the synchronization operations required by DSM impose small development burden: they take at most 10 lines per application.
5.6.4 Programmer Study

Even the 30% more lines of code do not fully capture the development burden with message passing. Programming with message passing challenges the design of sensing applications. We conduct an informal study with seventeen programmers drawn through the authors’ social networks. We ask participants to design a small subset of data exchange in RAPS: exchanging two values among three modules with messages in pseudo code. The participants are required to follow programming rules that resemble the message passing constraints in sensing applications. We ask them to finish the same task with shared memory in pseudo code as well. Out of 1-5, 5 being the most difficult, the participants rate the difficulty of using messages as 3.5 on average and that of shared memory as 1.7. All prefer or strongly prefer shared memory to messages in this task with average rating being 1.5, 1 as strongly preferring shared memory and 5 as strongly preferring messages. Although all think they clearly understand the goal of the task, five indicate that the rules in using messages are difficult to understand and use.

5.6.5 DSM Performance

We next zoom into the behavior of the Reflex DSM and examine the effectiveness of its important design choices.

5.6.6 Asymmetric Module Role Assignment

The Reflex DSM exploits the weakest processor to host the home module of a shared object and, therefore, keeps more powerful processors in sleep, a strategy that is critical to the energy efficiency goal. We compare this strategy with the following two alternatives that are widely used in other DSM designs: 1) using the strongest
processor for the home module and 2) using the module that lastly writes to the object as the home module. Based on the per scenario actual power measurement, we show the emulated overall power consumption under both alternative strategies in Figure 5.4 (‘Strongest Module as Home’ and ‘Last Writer as Home’). With the alternative strategies, the system power consumption increases by up to five times as compared to the Reflex DSM, because the strongest processor is waked up to serve coherence requests and therefore unable to remain in the sleep mode for a long time.

5.6.7 Storage and Memory Overhead

For a peripheral processor, Reflex introduces a small overhead in storage and memory, i.e., mainly the code size and memory usage (around 8KB ROM and 0.2KB RAM), through the runtime and the module library. Of all ROM and RAM equipped by one peripheral processor, the storage (ROM) overhead is less than 25% and the memory (RAM) overhead is less than 2.5%. The Reflex DSM further introduces little storage/memory overhead through the compiler-instrumented code and the object table. All pre-access or post-access code only increases the ROM usage of a peripheral module by at most 200 bytes. In each central module, the instrumented code ranges from 200 to 700 bytes, less than 1.2% of the module storage usage. Each entry in the object table requires 6 8 bytes of extra RAM per module. The increased RAM is up to 119 bytes per module in our benchmarks.

5.6.8 DSM Execution Overhead

The DSM execution overhead comes from three major sources: module state checking as done by pre-access, the requester latency, and the home latency. In the common application scenarios where no inter-module data exchange occurs, the DSM execution
overhead only comes from pre-access. With three ISAs in the tri-processor prototype, each pre-access consists of one bit-shift operation, two to three load instructions and one compare instruction, resulting only 5~10 extra processor cycles. Furthermore, the Reflex compiler aggressively eliminates pre-access for most accesses, with Mod/Ref analysis and the object type information. We estimate that the state checking incurs negligible overhead (less than 3%) in the execution time of all benchmarks.

DSM introduces home latency and requester latency. The measured home latency is 20ms on average, dominated by the message passing delay. The requester latency is effectively amortized over multiple objects by batch prefetching. To demonstrate its benefit, we compare the average requester latency per object without and with the batch prefetching. In the experiment, we vary batch parameters that specify the maximum size and the maximum count of objects that can be fetched with a single message. For example, batch parameters (4 objects, 128 bytes) specify that DSM

Figure 5.6: The average requester latency in fetching an object, without and with batch prefetching
will flush a request message to the transport if objects requested by the message are more than 4 or their total size exceeds 128 bytes. As shown in Figure 5.6, the batch prefetching can reduce the average requester latency by up to 75% as compared to the case without batch prefetching. In addition, we experimentally verify that without deferring write backs, DSM will slow down the execution of benchmark applications by up to 1000 times due to excessive messages, which is prohibitively expensive.

5.7 Discussion

Type-safe Languages. The Reflex DSM leverages type information to aggressively reduce runtime overhead with static optimization. In the current prototype, we ensure that the type information is statically available by subsetting C++. A more ambitious implementation should directly support modern type-safe languages that are widely used in smartphone development, e.g., Java and C#. We notice that several existing systems already enabled type-safe languages for resource-constrained processors, e.g., Darjeeling [6], .NET micro framework, and Hera-JVM [23]. We are working towards supporting type-safe languages under extreme resource scarcity.

Reflex beyond smartphone sensing. While Reflex and its DSM design are motivated for smartphones, their technical innovations have a broader impact. Many consumer electronic devices are embracing sensors, from ebook readers to game consoles to tablets. Reflex and its DSM design will allow developers to write efficient applications for such systems with ease. The core ideas of the Reflex DSM can be further extended to high-performance systems with loosely integrated, heterogeneous resources where energy efficiency has become an increasing practical concern.
5.8 Summary

Sensing applications are considered among the emerging killer applications for smartphones. Reflex is the first endeavor toward making programming heterogeneous, asymmetric smartphones easier. Sensing applications on heterogeneous smartphones pose unique systems challenges that were previously not important. In addressing these challenges, we have not only revised known solutions but also devised novel ones in software DSM that exploit the extreme architectural asymmetry for high energy efficiency. We note that Reflex still imposes a few constraints in programming due to the requirement to encapsulate a peripheral module and the limit on the peripheral system services. However, our experience suggests such relaxation is necessary: eliminating these constraints would be too expensive to be practical, especially with such an asymmetric architecture.
Chapter 6

Single OS Image Over Non-coherent Asymmetric Processors

6.1 Motivation

In Section 2, we have briefly introduced heterogeneous mobile SoCs with multiple cache coherence domain, or coherence domain. Cores in a coherence domain have the same memory view usually via hardware-supported cache coherence. In the case of multiple coherence domains, one coherence domain can host high-performance cores for demanding tasks; another domain can host low-power cores for light tasks. We call them strong domain and weak domain, respectively in this work. We purposefully use strong and weak in order to distinguish them from heterogeneous cores in the same domain that are already known as big and little. The lack of coherence allows cores in the weak domain to be orders of magnitude weaker and lower-power than those in the strong ones.

In tapping into the energy efficiency benefit of multiple coherence domains, the biggest challenge is programmability. Processors from multiple coherence domains essentially constitute a distributed system; programming them is known to be difficult to mass programmers, who must maintain consistency for partitioned program components with messages. Many have recognized this challenge and a handful of systems solutions have been studied [48, 72]. However, prior work mainly focuses on user-space workloads. They implement a small set of OS services from scratch for a
weak domain, e.g., sensor reading, and assume all other OS services are provided by a fixed strong domain. The resulting OS services are limited in functionality, hard for app developers to use because of a fragmented system image, and incurs high burden in system development. More importantly, with this design, only light tasks without a need of OS services beyond those crafted for weak domains can benefit from the efficiency of the weak domains.

In this Chapter, we seek to answer the following question: is it possible to span an OS across multiple coherence domains? In doing so, we would like to satisfy three important goals. (i) The resulting OS should preserve the established programming models for mobile app development, as the programming models have been used in over a million mobile apps by a similar number of developers of which many do not have sophisticated programming skills. (ii) The OS itself shall leverage mature operating systems in wide use without reinventing many wheels. As current mobile OS assumes a coherent memory, this seems to suggest an overhaul of the entire OS, which is increasingly difficult and even undesirable nowadays: the OS is already a huge codebase and continues to receive contributions from numerous parties including device vendors. And (iii) the system should deliver the same performance for high-performance apps.

6.2 The Case for Replicating OS Services

To fully exploit multi-domain SoCs for energy efficiency, the key is to enable OS to serve a request from the domain that is able to deliver the needed performance with the lowest power possible. This requires replicating OS services on all the candidate domains.

As shown by a recent study of mobile workloads [49], the same OS services are
usually invoked with disparate performance and power expectations. For instance, light tasks often access IO regularly, thus sharing the OS services with demanding tasks such as UI rendering. Neither pinning OS services on a single domain nor partitioning them among domains [66] will work here: on one hand, light tasks invoking OS services provided on a strong domain will suffer from all inefficiencies described in §2.1; on the other, demanding tasks invoking OS services provided on a weak domain are likely to fail their performance expectations. Targeting exploiting incoherent heterogeneity for energy efficiency, most prior systems are focused on supporting user workloads and usually adopt ad-hoc solutions for OS services on weak domains [84, 72, 48]. Without systematically replicated OS services, light tasks can only implement a limited set of functionalities.

When running replicated OS services over multiple coherence domains, the OS state is essentially distributed among domains. Inter-domain state synchronization must be done in software explicitly, and is much slower than hardware cache coherence. This challenges application development, OS engineering, and system performance, as discussed below.

**Applications assume a single system image** Mobile applications expect the OS to present a single system image, including a unified OS namespace and a global resource management. The single system image has served as the substrate of one million mobile applications and has been accepted by the similar number of application developers.

**OS Software assume coherent state** All layers of a mainstream mobile OS, from top down, assume hardware cache coherence. Significant changes to core services, e.g., interrupt and memory management, for coping with incoherent state are difficult and
undesirable: their mechanisms and policies have been heavily tuned and proved to work over years. Rewriting extended services, such as device drivers, is costly. For instance, they constitute 70% of the Linux source code [76], and are developed by various software and hardware vendors, such as Google and Samsung.

**Performance impact**  The absence of hardware cache coherence necessitates explicit inter-domain communication in coordinating OS replicas or keeping their state consistent – a much slower mechanism than hardware cache coherence. For example, flushing a L1 cache line takes tens of cycles, while sending an IPI usually takes a few thousand cycles. Such communication overheads will be magnified by inter-domain contention for the same shared state.

### 6.3 System Design

In this section, we sketch our design. We state the design goals, describe the architectural assumptions, and then derive the shared-most OS model.

#### 6.3.1 Design Goals

We seek to design an OS that is capable of replicating its services over multiple coherence domains of a mobile SoC, in order to exploit hardware heterogeneity for energy efficiency. The OS design should meet the following goals:

1. Facilitate application development by relieving app developers from dealing with incoherent program state and fragmented system images.

2. Simplify OS engineering and avoid disruptive changes by maximizing reuse of legacy OS code.
3. Maintain the current performance level of demanding tasks.

### 6.3.2 Architectural Assumptions

We design our system under the assumption of the following architectural features of a mobile SoC:

- **Heterogeneous cores.** The SoC has multiple types of cores that provide disparate performance-power tradeoffs. Cores may have different ISAs. Process migration is difficult and, if possible, requires sophisticated software [23, 83]. Cores have hardware MMUs.

- **Multiple coherence domains.** All cores on the SoC are isolated into a few (2-3) coherence domains, for the sake of high heterogeneity, decoupled power management and reduced thermal constraints. Within a coherence domain, multiple cores (2-8) function as a traditional multicore. There is no hardware cache coherence among domains.

- **Shared platform resources.** Coherence domains are connected to a chip-level interconnect and thus share all platform resources, including RAM and IO peripherals. Interrupts generated by IO peripherals are physically wired to all domains.

- **Aggressive power management.** Cores are taken online (active) and then offline (inactive) from time to time. The system energy efficiency is highly dependent on core power state, i.e., how long cores remain inactive and how often they are woken up.
6.3.3 The Shared-most OS Model

In order to meet the design goals stated in §6.3.1, we identify a shared-most OS model: *i*) to transparently maintain state coherence for extended services, or *shadowed services*, *ii*) to coordinate separated instances of core services, or *independent services*, and *iii*) to avoid multi-domain parallelism within individual process. We view our model as an outcome of considering both the underlying incoherent architecture and the programming challenges mentioned in §6.1. In the spectrum of OS structures shown in Figure 6.1, the shared-most model sits in the middle, with two extremes being ‘shared-everything’ monolithic OS and ‘shared-nothing’ OS. The two extremes, while suiting their respective target architectures well, either incur high performance overheads or heavy programming burdens for multi-domain mobile SoCs. We next discuss the three aspects of the shared-most model in detail, explaining how each of them is connected to the design goals.

**Transparent shared state for extended services** We argue that the OS should transparently maintain state coherence for replicas of extended services, including device drivers and file systems, for the goal of simplified OS engineering. First, as mentioned in §6.2, as extended services constitute the majority of an OS codebase
and evolve rapidly, it is difficult to manually transform them for multiple coherence
domains. Second, extended OS services are less likely to suffer from inter-domain
contention, and are more tolerant of the performance overheads of software coherence,
such as the inter-domain communication latency.

Independent instances of core services  For a core service, the OS creates inde-
pendent, per-domain instances that share no internal state. As mentioned in §6.2, core
services such as memory management are invoked frequently and are performance-
critical. In replicating such services over multiple domains, contention must be care-
fully avoided for performance.

Creating independent instances of core service raises two design problems: (i)
How and when the OS should coordinate independent instances of the same service:
the coordination is key to maintain a single system image, and has to be efficient as
inter-domain communication can be expensive. (ii) How to minimize modifications
to the mature implementations of core services, as a way to meet the goal of simplified
OS engineering. The two problems have to be addressed on a per-service basis. In
§6.5 and §6.6, we will describe our solutions for two important core services, physical
page allocator and interrupt management.

Avoid multi-domain parallelism within individual process  Threads belong-
ing to the same process share an extensive set of OS state, e.g., opened files. Running
them simultaneously on multiple domains may lead to expensive contention. Since
an application would gain little benefit from such parallelism, we argue that such
contention should be prevented by always deferring light task execution if the same
process has more demanding tasks to run. The extra delay introduced to light tasks
is acceptable. First, the delay is unlikely to be long: in principle, mobile demanding
tasks should not saturate strong cores for more than a few hundred milliseconds for
avoiding sluggish GUI. Second, the deferral only applies to light tasks belonging to
the same process.

Multi-domain parallelism, however, should be supported among processes, for
fairness in scheduling and encouraging applications to use the weak domain. If strong
and weak domains were restricted to run alternately, all light tasks in the system,
which likely belong to different processes, will block if any normal task is running,
according to our model discussed above. This essentially makes light task execution
dependent on other applications’ behaviors, discouraging developers from placing code
on the weak domain.

6.4 The K2 OS

In order to test the shared-most model, we build K2, an OS prototype for multi-
domain mobile SoC. In this section, we give an overview of K2: we describe the
hardware platform, sketch the structure of K2, present our heuristics in refactoring
Linux, and discuss how K2 is built from source.

6.4.1 Hardware Platform

We experimentally test our OS model with TI OMAP4, one of the many emerging
multi-domain mobile SoCs [78, 67] that has the best public information. OMAP4
has been powering popular mobile devices of various form factors, including Amazon
Kindle Fire, Samsung Galaxy S2, and Google Glass.

OMAP4 has two coherence domains, each of which hosts one type of 32-bit ARM
Their specifications are listed in Table 6.1. The Cortex-M3 on OMAP4 has a non-
<table>
<thead>
<tr>
<th>Cortex-A9 (strong)</th>
<th>Cortex-M3 (weak)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISA</td>
<td>ARM Thumb-2</td>
</tr>
<tr>
<td>Freq.</td>
<td>350-1200MHz</td>
</tr>
<tr>
<td>Cache</td>
<td>L1 64KB, L2 1MB</td>
</tr>
<tr>
<td>MMU</td>
<td>One ARM v7-A</td>
</tr>
</tbody>
</table>

Table 6.1: Heterogeneous cores in the two coherence domains of OMAP4

standard MMU in which two levels are connected in series, which affects the design of K2 software coherence as will be discussed in §6.5.3. Each domain has dual cores and a private interrupt controller. Both domains are connected to the system interconnect, which further connects the shared RAM and all IO peripherals.

To support multiple coherence domains, OMAP4 provides two facilities. The hardware spinlocks are for inter-domain synchronization, which are a set of memory-mapped bits that support atomic test-and-set. The hardware mailboxes are for inter-domain communication: cores can pass 32-bit messages across domains with interrupting each other. We measured the message round-trip time as around 5 μs.

While K2 is geared towards OMAP4 with many platform-specific optimizations and workarounds, its design is driven by our OS model and based on the architectural assumptions in §6.3.2. We expect the design to be applicable to other multi-domain SoCs.

### 6.4.2 OS Structure

Figure 6.2 depicts the structure of K2. As shown on the top of the figure, K2 carefully exposes hardware heterogeneity to application developers, by requiring them to stat-
Figuratively partition their applications in coarse grains for heterogeneous domains. From an application developer’s perspective, they develop the performance-critical parts of their applications as normal threads for execution on the strong domain; they wrap their light tasks in special NightWatch threads which will be pinned on the weak domain. We will discuss the details of NightWatch threads in §6.7.

Under the single system image, K2 runs two kernels on OMAP4: the full-fledged main kernel in ARM ISA running on the strong Cortex-A9 cores, and the lean shadow kernel\(^*\) in Thumb-2 ISA running on a weak Cortex-M3 core. The two kernels share the same virtual address space (§6.5.1), the same pool of physical memory (§6.5.2), and cooperate to handle IO interrupts (§6.6). As shown in Figure 6.2, most OS services, including device drivers and file systems, are shadowed between both kernels:

\(^*\)“Shadow” reflects the fact that the two kernels are kept coherent in their extended OS services.
they are built from the same source code and K2 transparently keeps their state coherent at run time. A few ‘hotspot’ core services are independent in both kernels and K2 coordinates them on the meta level. Each kernel has its private services.

6.4.3 Refactoring Linux

As a key step in applying the shared-most OS model, we refactor a recent Linux kernel 3.4 into both the main and shadow kernels of K2. In particular, in bringing up Linux on the Cortex-M3 of OMAP4, we adopt code from the linux-panda project [65].

The central job of refactoring is to decide how should individual Linux OS services be adopted by K2. We tackle the problem based on their dependence on processor cores, their functionalities, and the performance impact, with the following steps as shown in Figure 6.3:

1. First, services that are specific to one type of core or that manage domain-local resources are kept private, implemented differently in each kernel, and with
different state. Examples include core power management.

2. In the rest of the services, those performing complicated, rarely-used global operations will be kept private only in the main kernel. Examples include platform-level initializations.

3. In the rest of the services, those having high performance impact are replicated as independent instances in each kernel, with K2 coordinating the instances.

4. The remaining services, which manage platform resources and have low to moderate performance impact, are made as shadowed service in both kernels, with K2 automatically keeping their state coherent. In refactoring, we augment their locks by using the hardware spinlocks for inter-domain synchronization. This is the largest category which includes device drivers, file systems, and network service.

6.4.4 Build K2 from Source

We implement both kernels of K2 in a unified source tree. To build K2, we run two passes of compilations, for the main kernel and for the shadow kernel respectively. In compilation, we apply a suite of techniques to bridge the heterogeneity gap between cores, with the help of automated scripts.

First, the compilation process makes sure each shared memory object has identical load addresses in both kernels. For both kernels, it pads data structures, enforces the same output order of memory objects in each object file, and enforces the same link order of object files.

Second, the compilation process treats function pointers, which are prevalent in the Linux data structures and thus shared among two kernels. According to its
ISA [7], when Cortex-M3 attempts to execute a Cortex-A9 instruction, it falls into an *unrecoverable* core state. To prevent such a situation, the compilation process statically rewrites `blx` instruction – the long jump instruction emitted by GCC for implementing function pointer dereference – with `Undef` instruction. At run time, when Cortex-M3 attempts to dereference a function pointer and thus hits `Undef`, it triggers an exception that is *recoverable*; K2 handles the exception and dispatches the control flow to the Cortex-M3 version of the function. `blx` is sparse in kernel code, constituting 0.1% of all instructions and 6% of all jump instructions.

**Bootstrapping** We design K2 to boot in two stages, for easy deployment and debugging. The main kernel boots just like usual Linux and reserves contiguous memory for shadow kernel’s private region. After the userspace is up, the main kernel loads the shadow kernel and ‘hot-plugs’ the latter onto itself. In loading, it avoids re-initializing shared data structures since the corresponding addresses already contain live kernel state.

**Home-brew tools** In developing K2, we have built the following tools. We hope that they can facilitate future multikernel OS research.

- **func-lookup.** Generate the lookup table for different addresses of the same kernel function.

- **ds-diff.** Compare the size and member offsets of same data structures in different kernels. Suggest padding to make data structure identical on different kernels.

- **sym-diff.** Verify that kernel symbols, in particular global variables, have identical loading addresses in both kernels.
6.5 Memory Management

In the next three sections, we describe the core components of K2: memory management in §6.5, interrupt management in §6.6, and scheduling in §6.7. Among them, we will focus on memory management, which is the core mechanism in providing state coherence and unified resource management. In this section, we describe the major building blocks of the K2 memory management: the layout of kernel virtual addresses, the physical memory management, and the software coherence.

6.5.1 Unified Kernel Address Space

K2 creates a unified address space for both kernels, while preserving the key assumption made by the Linux kernel virtual memory. In this section, we briefly recap the background of the Linux kernel virtual memory, describe the design constraints faced by K2, and then present our design.

Linux maps a large portion (sometimes all) of physical memory into kernel space directly. The mapping is linear and permanent: any virtual address is mapped to a physical address that only differs by a constant offset. Such direct-mapping greatly simplifies kernel virtual memory design: accessing direct-mapped memory will never trigger a page fault and conversions between kernel virtual and physical addresses are fast. As a result, direct-mapped memory usually hosts all important kernel data structures. †

In managing virtual memory for multiple kernels, K2 has to satisfy the following constraints:

†We are aware of the SPARSEMEM feature for servers, which, however, is rarely enabled for mobile devices.
1. Memory objects shared between kernels must have identical virtual addresses in both kernels. In addition, private memory objects should reside in non-overlapping address ranges to help catch software bugs.

2. For each kernel, the assumption of linear mapping holds for the entire direct-mapped memory.

3. Contiguous physical memory should be maximized, in particular for the main kernel which usually needs large physical memory blocks for multimedia applications.

K2 arranges its kernel address space as shown in Figure 6.4. From each kernel’s view, its available physical memory consists of two regions, both direct-mapped. (i) A small local region hosts executable code (in local ISA) and the memory objects statically allocated for private or independent OS services (e.g., those in bss and data sections). (ii) All the rest of the physical memory belongs to the global region, which hosts the memory objects for shared OS services and all free physical pages for dynamic allocation. The global region is typically from several hundred MB to one GB.
From the start of physical memory (the left side of Figure 6.4), K2 populates all local regions: first for the shadow kernel and then for the main kernel. From the end of the last local region, the global region spans to the end of physical memory. By putting the main kernel’s local region right before the global region, K2 avoids memory holes in the main kernel. With this memory layout, K2 keeps both kernels’ virtual-to-physical offsets identical, thus essentially creating a unified virtual address space.

**Temporary mapping**  In addition to the direct-mapped memory discussed above, the OS may need to establish temporary mapping and thus make changes to the unified kernel address space. In supporting so, K2 treats two major types of temporary mappings differently. First, the OS may need temporary mappings for accessing IO memory. As creations and destructions of such mappings are infrequent, K2 adopts a simple protocol between two kernels for propagating page table updates from one to the other. Second, on platforms with abundant memory resources, the amount of physical pages may exceed that can be directly mapped into kernel space. Thus, temporary mappings are needed on-demand in accessing the extra pages (i.e., `highmem` in Linux’s term). In the current implementation for the 32-bit ARM, K2 does not support `highmem`; as a workaround, it increases the size of kernel virtual space from the default 1GB to 2GB, which is large enough for directly mapping physical memory available on today’s mainstream mobile devices. We do not expect this to be a fundamental limitation, as the emerging 64-bit ARM allows kernel to directly map a much larger amount of physical memory.
6.5.2 Physical Memory Management

Physical memory, as managed by OS page allocator, is one of the most important resources. The page allocator is frequently invoked, performance-critical, with its state among the hottest data structures in the OS. In managing physical memory for multiple coherence domains, K2 should:

1. Require minimum inter-domain communication in memory allocation and free.

2. Enable both kernels to dynamically share the entire pool of free pages.

3. Avoid disruptive changes to the existing Linux page allocator, which is already a mature design with sophisticated mechanisms and policies.

4. Minimize physical memory fragmentation.

An overview of the K2 memory management is shown in Figure 6.5. To achieve the first objective above, K2 applies the shared-most model and creates instances of page allocator in both the main and the shadow kernel, with separated state. Allocation requests are always served by the local instance on the same coherence domain; free requests are redirected asynchronously to the instance which allocated the pages, based on a simple address range check implemented as a thin wrapper over the existing free interface.

We next describe how K2 coordinates multiple page allocator instances to achieve objective 2-4 above, by first discussing the mechanism (balloon drivers) and then the policy (the meta-level manager).

**Balloon drivers** K2 retrofits the idea of balloon driver from virtual machines [92] for controlling the amount of contiguous physical memory available to individual
kernels. To individual kernels, balloon driver creates the key illusion of on-demand resizable physical memory.

A balloon driver is a pseudo device driver and each kernel has its own private instance. Controlled by K2, the driver’s job is to occupy contiguous physical memory to keep the memory from the local page allocator. As shown in Figure 6.5, to K2, a balloon driver provides two primitives that operate on physically contiguous blocks of pages, or page blocks: *deflate*, i.e., the driver frees a page block to the local page allocator, which essentially transfers the ownership of the page block from K2 to the local kernel; *inflate*, i.e., the driver allocates a page block from the kernel, by forcing the kernel to evacuate pages from that page block, which essentially transfers the ownership of the page block from the local kernel back to K2. In the early stage of kernel boot, balloon drivers of both kernel are initialized to occupy the entire shared region so that K2 owns all physical pages in that region.

We implement K2 balloon drivers based on Linux’s contiguous memory allocation framework. Balloon drivers require no change to the Linux page allocator: from each
kernel’s perspective, a balloon behaves like a common device driver that reserves a large contiguous memory region at boot time and later allocates and frees page blocks within that region autonomously.

**Meta-level manager**  On top of the balloon drivers, K2 provides a meta-level manager to decide when to take and give page blocks from and to kernels. The meta-level manager is implemented as a set of distributed probes, one in each kernel. Each probe monitors local memory pressure with hooks inserted into the local kernel page allocator; the probes coordinate through hardware messages, and take actions by invoking local balloon drivers. Generally, when memory pressure increases and before the local kernel reacts to the pressure (e.g., activating the Android low-memory killer), the meta-level manager instructs the balloon driver to free page blocks to alleviate the pressure; when free pages run low in the entire system, the meta-level manager instructs a balloon driver to inflate for reclaiming pages. Like the Linux kernel swap daemon, the meta-level manager performs operations in the background when OS is idle in order to minimize performance impact on individual allocations.

**Optimizations**  K2 has applied a few optimizations in physical memory management, as shown in Figure 6.5. First, to reduce inter-domain communication, the meta-level manager operates on large-grain page blocks, which are 16MB in the current implementation. Second, to maximize the size of physically contiguous area available to the main kernel, the meta-level manager instructs balloons to deflate from the two ends of the free portion in the shared region, and inflates in the reverse directions. Thus, blocks allocated to the main kernel grow from right after its private region. Third, to maximize the chance of successfully reclaiming page blocks from kernels, K2 commands local page allocators to place movable pages (e.g., those con-
taining user data) close to the ‘frontier’ of the balloon with best efforts. The efforts are likely to succeed, as movable pages usually constitute 70%-80% of total pages based on our experiments with mobile systems. As a result, when the balloon inflates, those movable pages can be evacuated from the requested page block to elsewhere.

6.5.3 Software Cache Coherence

As mentioned in §6.4.2, K2 provides transparent coherence for shadowed services. This is implemented with software cache coherence, a classic approach to hide distributed program state from programmers [46], which also has been applied to recent incoherent architectures such as CPU-GPU systems [32] and smartphones [48]. Like most software cache coherence designs, the K2 software cache coherence maintains the key one-writer invariant [46], by automatically translating memory accesses to inter-domain communication for coherence. While most prior designs are designed to back user applications [79, 46], the one used by K2 backs OS services. Thus, we carefully construct fault handlers and coherence communication to avoid interference or lockup with other OS components.

Basic design  The K2 software cache coherence implements sequential consistency. Although many software cache coherence designs [48, 32] implement relaxed consistency which relies on applications’ correct locking behaviors, K2 neither relaxes the consistency model assumed by the existing OS, nor makes any assumption of OS locking behaviors which may be complicated. In implementing sequential consistency, the K2 software cache coherence performs coherence operations upon each access fault and keeps the order of coherence messages. The K2 software cache coherence adopts a page-based granularity, using 4KB page as the smallest memory unit that is kept
coherent. This is for leveraging MMU to trap accesses to shared memory, as page is the unit used by the MMU for enforcing protection.

The K2 software cache coherence has adopted a simple, standard two-state protocol. For each shared page, every kernel keeps track of its state, being Valid or Invalid. Kernels communicate with two types of coherence messages: GetExclusive and PutExclusive. When a page is Valid, a kernel can perform read or write of the page. Before performing read or write of an Invalid page, a kernel must send GetExclusive to the other kernel who currently owns the page; upon receiving GetExclusive, the latter kernel flushes and invalidates the requested page from its local cache, and acknowledges with a PutExclusive message. After that, the former kernel can proceed to access the memory.

The K2 software cache coherence detects accesses to shared memory as follows: when a page transits from Valid to Invalid state, the software cache coherence design modifies the corresponding page table entry to be ineffective and handles the resulting page fault triggered by the subsequent access to the page. The fault handling is transparent to the OS code that made the memory access.

**Coherence communication** For performance, the coherence communication directly leverages the OMAP4 hardware mailboxes. Each message is 32-bit, the size of a hardware mail, with 20 bits for page frame number, 3 bits for message type, and the rest for message sequence number. The mailbox will guarantee that messages are delivered in order.

We carefully construct coherence communication to avoid lockup. The communication must be synchronous to the requester: this is because interrupt handlers, which cannot sleep, may access shared state and thus initiates communication. Thus,
when a requester kernel sends out a *GetExclusive* message, it spins waiting until the
destination kernel sends back a *PutExclusive* message. For the same reason, handling
*GetExclusive* must avoid sleeping as well, e.g., it must use atomic memory allocation.
More importantly, handling *GetExclusive* must avoid accessing shared state which
may trigger new page faults, resulting in an infinite request loop between kernels.

We optimize the communication latency in favor of the main kernel, by setting
their priorities of handling coherence messages differently. The main kernel handles
*GetExclusive* in bottom halves, and will further defer the handling if under high work-
loads; in contrast, the shadow kernel handles the request before any other pending
interrupt. Through this, K2 reduces the impact of weak cores on system performance.

**Optimize memory footprint**  The K2 software cache coherence incurs a low over-
head in storing protocol information, asking for three bits per page. Furthermore,
although shared memory areas have to be mapped in 4KB granularity, the K2 soft-
ware cache coherence allows non-shared areas to be mapped in larger grains (1 MB or
16 MB) as supported by the ARM hardware, thus reducing the size of page tables and
alleviating TLB pressure. To achieve this, the K2 software cache coherence turns on
software coherence for memory addresses and replaces the existing large-grain map-
ing on-demand, only when an address (or its neighbouring area) has been accessed
by both kernels.

**An alternative design**  While we are aware of a more common three-state protocol
[85] that supports read-only sharing, our choice of the two-state protocol is based on
the hardware limitation of the Cortex-M3 MMU on OMAP4.

In general, supporting read-only sharing requires the software cache coherence
design to use MMU for differentiating memory reads from writes in order to handle
them separately. However, this is expensive on the OMAP4 Cortex-M3, which relies on the first level of its two cascading MMUs for read/write permissions. The first-level MMU has no page table but just a software-loaded TLB which only contains ten entries for 4KB pages. According to our experiment, using the MMU for read access detection puts a high pressure on its TLB and leads to severe thrashing. In contrast, without supporting read-only sharing, the K2 software cache coherence detects both read and write accesses solely with the second-level MMU, which has a larger TLB and a hardware page table walker.

### 6.6 Interrupt Management

Multiple coherence domains share interrupts generated by IO peripherals. Although any interrupt signal is physically delivered to all domains, K2 must ensure that it is only handled by exactly one kernel. If multiple kernels compete for the same interrupt signal, peripherals may enter incorrect states or cores may have spurious wakeups. The choice of kernel in handling interrupts does not affect OS correctness, thanks to the K2 software coherence; however, it does affect performance and efficiency.

K2 coordinates its two kernels in handling shared interrupts, by following two simple rules. First, for energy efficiency, K2 does not allow shared interrupts to wake up the strong domain from an inactive power state, in which case the shadow kernel should handle the interrupts. Second, for performance, when the strong domain is awake, K2 lets the main kernel handle all shared interrupts.

K2 has implemented the rules on the OMAP4 platform, where coherence domains have private interrupt controllers. K2 inserts a few hooks into the Linux power management code to configure interrupt controllers. When the shadow kernel boots on a weak domain, it masks all shared interrupt locally. When a strong domain
transits to an inactive state, K2 unmaskes all shared interrupts on the weak domain and masks them on the strong domain; when the strong domain is woken up from the inactive state, K2 reverses such operations, by masking shared interrupts on the weak domain and unmasking them on the strong domain.

6.7 Nightwatch Threads

As mentioned in §6.4.2, K2 provides an abstraction called NightWatch thread for application developers to implement light tasks. NightWatch threads are pinned on a weak domain; after being created, a NightWatch thread enters the shadow kernel runqueue for execution. From a developer’s view, a NightWatch thread is identical to a normal thread: for instance, they share a unified process address space and a single system image. The only exception is that in order to limit multi-domain parallelism (the third aspect of the shared-most model in §6.3.3), K2 enforces that:

A NightWatch thread will only be considered for scheduling when all normal threads of the same process are suspended.

The scheduling strategy, together with the single system image, distinguishes NightWatch threads from other abstractions that also encapsulate code for targeting heterogeneous hardware [48]. It is worth noting that K2 does not change the mechanism or policy of the Linux scheduler; all normal threads are scheduled as they are in Linux.

Preempt NightWatch threads K2 preempts the execution of all NightWatch threads in a process when any normal thread of the same process is about to execute. After the main kernel decides to schedule-in a normal thread, it examines if any NightWatch threads belong to the same process. If so, the kernel sends a SuspendNW
hardware message to the shadow kernel. Interrupted by the SuspendNW message, the shadow kernel immediately responds with an AckSuspendNW message and then removes all NightWatch threads of the same process from the local runqueue by flagging them. On receiving the AckSuspendNW, the main kernel lands the normal thread on a core for execution.

To reduce latency, the main kernel overlaps the wait for AckSuspendNW with the context switch to the schedule-in thread: after sending SuspendNW, the main kernel proceeds to context switch, and only waits for AckSuspendNW after the context switch is done, before returning to user space. Given that a message round trip takes around 5 $\mu$s and a context switch usually takes 3-4 $\mu$s, the extra overhead for the main kernel is 1-2 $\mu$s for every context switch.

**Resume NightWatch threads** When the main kernel finds that all normal threads of a process blocked, e.g., waiting for IO, it sends a ResumeNW message to the shadow kernel. On receiving the message, the shadow kernel removes flags from all NightWatch threads of the given process and will consider them in future scheduling.

### 6.8 Experimental Results

We evaluate how well K2 meets the design goals we set in §6.3.1 by reporting our efforts in refactoring Linux and testing the energy efficiency harvested by K2. We then evaluate the benefits and overheads of the shared-most model by benchmarking the physical page allocator and the DMA device driver.
<table>
<thead>
<tr>
<th>Existing Implementations</th>
<th>Changed SLoC</th>
<th>Original SLoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exception handling</td>
<td>1151</td>
<td>395</td>
</tr>
<tr>
<td>Page allocator, interrupt, scheduler</td>
<td>205</td>
<td>12119</td>
</tr>
<tr>
<td>New Implementations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSM</td>
<td>883</td>
<td></td>
</tr>
<tr>
<td>Memory management</td>
<td>468</td>
<td></td>
</tr>
<tr>
<td>Bootstrap</td>
<td>1306</td>
<td></td>
</tr>
<tr>
<td>OMAP4-specific Cortex-M3</td>
<td>772</td>
<td></td>
</tr>
<tr>
<td>Debugging, etc.</td>
<td>1362</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>4791</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.2: Source code changes and additions to the Linux kernel 3.4, which contains around 10 million SLoC. Much of the OMAP4-specific Cortex-M3 code is adopted from the linux-panda project [65].

6.8.1 Efforts in Refactoring

In building K2, we have achieved the goal of reusing mature OS source at the refactoring cost shown in Table 6.2. K2 introduces small changes to the Linux source. The biggest portion of changes is exception handling, where K2 handles page faults for software cache coherence operations and dispatches function pointers. To core OS components such as the page allocator, K2 only adds a small portion of source lines. Device drivers, such as the one for DMA, can be reused by K2 with few modifications. K2 introduces a set of new software modules to implement its core components like software cache coherence and memory management. It also includes extensive debugging support to help ourselves understand Linux.
<table>
<thead>
<tr>
<th>Allocation Size</th>
<th>Main</th>
<th>Shadow</th>
</tr>
</thead>
<tbody>
<tr>
<td>4KB</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>256KB</td>
<td>5</td>
<td>45</td>
</tr>
<tr>
<td>1024KB</td>
<td>13</td>
<td>146</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Balloon</th>
</tr>
</thead>
<tbody>
<tr>
<td>deflate</td>
</tr>
<tr>
<td>inflate</td>
</tr>
</tbody>
</table>

Table 6.3: Latencies of physical memory allocations in K2, in μs. The performance of K2’s main kernel allocator has no noticeable difference from the Linux page allocator.

### 6.8.2 Independent Service

In order to evaluate our model of independent core services, we study the performance of the K2 physical memory management, with a microbenchmark to exercise the independent page allocators as well as the meta-level manager. In the benchmark, we measure the latency in allocating memory of different sizes, and in balloon deflating and inflating.

As shown in Table 6.3, K2 preserves the memory allocation performance of Linux. In comparing the memory allocation latency of the K2 main kernel with that of Linux, we find no noticeable difference: with separate state, the allocators of two kernels can operate independently without communication for most of the time. For each allocation, the pressure-monitoring probes inserted by the K2 meta-level manager incur less than twenty instructions, which is negligible compared to the allocation time. A balloon operation is more expensive: it intensively updates the page allocator state and moves pages, introducing around ten milliseconds latency that may be
perceivable to users. This validates our design of triggering them asynchronously in the background. With allocation and free operations interleaved in practice and K2’s large-grain page blocks, we expect that balloon operations are triggered infrequently and thus have their overheads amortized over a large number of memory allocations.

To contrast with K2’s independent page allocators, we attempted but found it infeasible to implement the page allocator as a shadowed service. The contention between coherence domains is very high, incurring four to five coherence page faults in every allocation, leading to a 200x slowdown. What is even worse, OS lockups happen frequently and are difficult to debug.

### 6.8.3 Shadowed Service

With transparent coherence provided by software cache coherence, K2’s shadowed OS services are able to achieve performance very close to Linux.
Table 6.5: The major operations of the DMA controller driver, as well as the key data structures they access. The worst-case overhead due to software cache coherence is also shown. Delay in $\mu$s.

<table>
<thead>
<tr>
<th>DMA Ops</th>
<th>Accessed Data Structures</th>
<th>#Misses</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>probe</td>
<td>chan config chan desc callback flag</td>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>setup</td>
<td>× ×</td>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>irq</td>
<td>× × × ×</td>
<td>3</td>
<td>150</td>
</tr>
</tbody>
</table>

**Microbenchmark** We first show the latency of a single coherence page fault and its breakdown in Table 6.4. Although further optimizations are possible (e.g., local fault handling and protocol execution contribute more than 30% of the latency), the software cache coherence provides acceptable performance to most extended OS services where sharing usually happens at a time scale of milliseconds or seconds.

Next we study a series of three shadowed services, in order to understand how K2’s OS-level software cache coherence interplays with the OS state, thus affecting the service performance. In order to do so, we dissect each service, examine their key data structures (i.e., OS state), and check how many shared misses may occur when OS execution paths access those data structures.

**The DMA Controller Driver**

The DMA driver is probably one of the most important and busiest drivers in the Linux kernel. It is used in almost all bulk IO transfers, e.g., for flash storage and WiFi. On the OMAP4 platform, the driver executes DMA transfers by operating the OMAP4 DMA engine.

There are four types of key data structures used by the DMA driver:
• **chan_config.** The global metadata that defines the configuration of the DMA controller, e.g., how many DMA channels are available in total.

• **chan_desc.** The per-channel metadata that describe the channel-specific configuration, e.g., the ID and status of the channel.

• **callback.** The per-request metadata that is passed between the invoker and the interrupt handler to describe the request information.

• **flag.** The flag that indicates the completion of a DMA transfer.

These key data structures and key major execution paths that access them are shown in Table 6.5. Note that the data structures can be accessed in both process context and interrupt context. According to interrupt handling in K2, an interrupt can be handled on either strong or weak domain, depending on their power states. In the table, we have shown the worst case overhead – the request is generated from the weak domain and the resultant interrupt is handled by the strong domain, causing the shared state to bounce between the two domains.

We examine the performance of shadowed services by running a DMA driver benchmark on the main and the shadow kernel concurrently. In each run of the benchmark, both kernels repeatedly invoke the DMA driver to execute transfers at full speed with a given batch size. The run is repeated for multiple times with different batch sizes. As a comparison, we also run the same benchmark using the original Linux kernel that uses the strong domain only. The DMA throughputs of the Linux and the K2 kernels, together with their differences, are summarized in Table 6.6.

As shown in the results, in backing the DMA shadowed service with software coherence, K2 incurs low overhead, reducing the throughput by up to 5.5%. Since K2
Table 6.6: DMA throughputs of K2 when the DMA driver is invoked in both kernels concurrently, in MB/Sec. The throughput differences as compared to the original Linux are also shown.

<table>
<thead>
<tr>
<th>DMA BatchSize</th>
<th>4K</th>
<th>128K</th>
<th>256K</th>
<th>1M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>37.8</td>
<td>40.3</td>
<td>40.3</td>
<td>40.5</td>
</tr>
<tr>
<td>K2</td>
<td>35.7</td>
<td>39.9</td>
<td>40.5</td>
<td>43.1</td>
</tr>
<tr>
<td>K2:Main</td>
<td>35.6</td>
<td>28.4</td>
<td>28.6</td>
<td>28.8</td>
</tr>
<tr>
<td>K2:Shadow</td>
<td>0.1</td>
<td>11.5</td>
<td>11.9</td>
<td>14.3</td>
</tr>
</tbody>
</table>

runs two kernels from two domains simultaneously, the throughput reduction seen by the main kernel is contributed by two factors: software coherence overhead incurred by K2 and the DMA engine bandwidth allocated to the shadow kernel. With small batch sizes such as 4KB, the benchmark is CPU-bound: the contention for shared kernel state is relatively high and the DMA engine is not fully utilized. Due to the asymmetry in processor performance and in K2’s design, the coherence overhead is low – a 50 μs coherence page fault happens around every 10 transfers, or 1200 μs. As batch size increases, the benchmark is becoming more IO-bound; the shadow kernel starts to get better chances in competing for the bandwidth of DMA engine. In this situation, as the rate of accessing shared state decreases, the coherence overhead is even lower – one coherence page fault in every 18 ms. Thus, the DMA engine serving two domains has a higher utilization, leading to a small increase in throughput (by up to 6%) as compared to the Linux case.
Table 6.7: The major operations of the Linux ramdisk driver, as well as the key data structures they access. The worst-case overhead due to software cache coherence is also shown. Read or write touches one page of data. Delay in $\mu$s

The Ramdisk Driver

Widely used in embedded and mobile systems, the Linux ramdisk driver is a virtual block device that emulates the disk behaviors using physical memory. Same as other block devices, e.g., hard disks, a ramdisk device acts as an array of blocks, each block being one physical page; accordingly, the interface provided by the ramdisk driver to the upper-level kernel subsystem (e.g., kernel IO scheduler) is a set of block-based operations. Although most block device drivers process queues of requests as the performance of block devices may benefit from batching request, the ramdisk driver directly processes individual requests, as batching is not beneficial to ramdisk.

Simply put, the major function of ramdisk driver is to convert IO block requests to reads or writes of the corresponding physical pages that back up the ramdisk. In doing that, the driver maintains mappings between blocks to memory locations. To make this happen, the ramdisk driver is built around three types of key data structures:

- **brd_device.** The metadata that describe a ramdisk device, including a series
of pointers to all key data structures associated with the device.

- **brd_disk.** The metadata that describe a ramdisk device as a generic hard disk.

- **brd_pages.** The radix tree that maps any given block in a ramdisk to the underlying backing page.

- **backing pages.** The actual physical pages that act as the backing store of ramdisk devices.

As shown in Table 6.7, the worst-case overhead of the major ramdisk operations is around tens to a few hundreds of microseconds. As we stated previously, this overhead is directly related to the number of memory pages one operation may touch. We will discuss the implication on performance optimization and memory allocation design in Section 6.9.

**The Linux Virtual File System**

In a Unix-like OS such as Linux, the virtual file system (VFS) is the foundation of the namespace of OS objects: most OS objects have been exposed to the user space as files, and it is the VFS that maps files to the underlying OS objects. Thus, the performance of VFS is crucial to the unified OS namespace across multiple loosely coupled processors.

There are four types of key data structures internally used by VFS:

- **super block.** This data structure contains the top-level metadata of one entire file system that has been mounted. For actual file systems, the data structure has a corresponding on-disk image.
The major operations of the Linux VFS, as well as the key data structures they access. The worst-case overhead due to software cache coherence is also shown.

The depth of file path is three. Delay in $\mu$s.

- **inode.** The index node structure contains the metadata for a given file. Same as above, this data structure has on-disk image for actual file systems.

- **dentry.** Directory entry caches the mapping from a file path to the corresponding inode.

- **file,** which contains the state of a opened file. The state is specific to the process that opens the file. In other words, if multiple processes open the same file, there will be multiple different file data structures.

It is worth noting that, each data structure is fully loaded with function pointers that point to operations specific to the data structure. For instance, file structure contains a vector of function pointers that manipulate per-process file state.

Although conceptually simple, the implementation of VFS is complicated and full of performance optimizations. For instance, the path name lookup function (namei.c) takes around 3500 lines of C code, and the entire VFS layer takes more
than 20000 lines of C code. The kernel state is scattered in a large number of objects that are connected to each other by pointers.

As shown in Table 6.8, the worst-case overhead of the major VFS operations is around tens to a few hundreds of microseconds. Execution paths that act upon on file descriptors, such as read, write, and close, are relatively simple and efficiency. Since the caller process provides the file descriptor, it is not necessary to walk the directory hierarchy in order to locate the data structure of inode. Instead, the VFS only has to examines the exact dentry and inode as pointed by the file being requested. As a result, these paths usually touch at most two to three pages, incurring a worse-case overhead of a few hundred microseconds.

Execution paths that act upon on path names, such as open, are more expensive, because they involve pathname resolution in the per-process namespace. Typically, all dentrys along the path will be touched; if the path goes across file system boundary (e.g., another file system is mounted on a directory on the path), extra data structures such as vfsmount will have to be consulted.

It is worth noting that the shown performance is for the worst case. That is, a case when all memory pages needed by one execution path are owned by the other kernel. In practice, we expect the worst case to occur rarely. The actual performance depends on many factors: kernel memory allocation, the actual file system, and the workloads. Thus, full understanding of the VFS performance on top of the K2 software cache coherence requires future comprehensive benchmarks.

As in the ramdisk case, we have observed that kernel objects accessed by a single VFS paths are rather spreaded out in physical memory, thus affecting the performance negatively. We will discuss the implication of the observation on OS design in Section 6.9.
Table 6.9: Power consumptions of the heterogeneous OMAP4 cores, in mW. Both cores consume less than 0.1 mW when inactive. The frequencies with asterisks are used in the benchmarks for testing energy efficiency.

<table>
<thead>
<tr>
<th></th>
<th>Active</th>
<th>Idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cortex-M3 (200MHz)*</td>
<td>21.1</td>
<td>3.8</td>
</tr>
<tr>
<td>Cortex-A9 (350MHz)*</td>
<td>79.8</td>
<td>25.2</td>
</tr>
<tr>
<td>Cortex-A9 (1200MHz)</td>
<td>672</td>
<td>25.2</td>
</tr>
</tbody>
</table>

6.8.4 Energy Efficiency Benefits

We show that K2 greatly improves energy efficiency for light tasks, by running a series of OS benchmarks to exercise K2 and the original Linux 3.4. In the benchmarks, we measure power consumption of coherence domains by sampling current on their separate power rails [27]. We measure elapsed time using hardware performance counters when cores are active, and using a 32KHz platform timer when idle.

In the benchmarks, K2 is able to use the weak core for OS execution, while Linux can only use the strong core. As summarized in Table 6.9, we configure the platform to favor the energy efficiency of Linux: we fix the strong core at its most efficient operating point while the weak core has to run at its least efficient operating point. This is because the OMAP4 implementation limits DVFS on the weak core, by coupling its voltage with that of the system interconnect and RAM. There is no way to scale down the voltage without crashing the entire SoC, to the best of our knowledge.

Our benchmarks test energy efficiency of three representative OS services: device driver, file system, and network stack. The benchmarks mimic mobile light tasks:
in each run of a benchmark, cores are woken up, execute the workloads as fast as possible, and then stay idle until becoming inactive. We set the core inactive timeout as 5 sec, as reported in a recent study of mobile device power [96].

**DMA driver** As shown in Figure 6.6(a), we pick a set of DMA transfer sizes typical to light tasks. Each run of the benchmark repeatedly invokes the DMA driver to execute multiple memory-to-memory transfers, where each transfer copies $BatchSize$ bytes and for a total of $TotalSize$ bytes copied. In each transfer, the DMA driver clears the destination memory region, looks for empty resources, programs the DMA engine and initiates the transfer. When the transfer is done, the DMA driver is interrupted.
to free resources and complete the transfer. We calculate the energy efficiency as the amount of transferred bytes per Joule.

**ext2 file system** We choose ext2, a file system widely used for flash storage, to test the energy efficiency of file system workloads. We use ramdisk as the underlying block device, as the SD card driver of K2 is not yet fully functional. This favors the energy efficiency of Linux: ramdisk is a much faster block device than real flash storages; using it shortens idle periods that are more expensive to strong cores. We design the benchmark to mimic a light task synchronizing contents from the cloud. As shown in Figure 6.6(b), for each run, a NightWatch thread operates on eight files sequentially: it creates a file, writes to it and closes it. We vary the write size to represent different content types: 1KB (emails), 256KB (pictures), and 1MB (short videos). We calculate the energy efficiency as the number of bytes written per Joule.

**UDP loopback** To test network workloads efficiency, we design a UDP loopback benchmark to mimic the networking activities when light tasks fetch contents from the cloud. In the benchmark, a NightWatch thread creates two UDP sockets; it writes to one and reads from the other one for a total number of $TotalSize$ bytes at full speed. Once every $BatchSize$ bytes transferred, the thread destroys both sockets and recreates new ones. As shown in Figure 6.6(c), we choose sizes to represent typical contents as done for the ext2 benchmark. We calculate the energy efficiency as the amount of bytes sent per Joule.

**Results analysis** In all three benchmarks, K2 shows significant advantages of energy efficiency over Linux, by improving the energy efficiency by up to 9x, 8x, and 10x, respectively. Based such results and the mobile device usage reported in recent
literature [96], we estimate that K2 will extend the reported device standby time by 59%, from 5.9 days to 9.4 days.

Overall, K2 gains the advantages mainly by exploiting the much lower-power idle periods of the weak core, which exist within each OS invocation (such as file operations in the ext2 benchmark) as well as in the inactive timeout periods. This is shown by the fact that when workloads are more IO-bound (e.g., the batch size of DMA transfers increases) or when the total task size of each run is smaller (e.g., in UDP loopback with fewer total sent bytes), the advantages of K2 are even more significant. In addition, over all benchmarks, K2 is able to use the weak core to deliver peak performance that is 20%-70% of the strong core performance at 350MHz, which can accommodate a wide range of light tasks that have low to moderate performance expectations.

6.9 Observation: Need for Locality-aware Memory Allocation

Our case study with multiple kernel subsystems as extended services yield an important observation: the K2 DSM calls for kernel memory allocation that promotes page-based locality.

In K2, the memory coherence is ensured by software and is page-based. Thus, the performance of extended OS service highly depends on how frequent coherence operations are executed. This in turn depends on the size of the working set of the extended OS service, i.e., the locality of its memory accesses.

Unfortunately, Linux was not designed with the concern in mind: its abundant abstraction layers, although serve as an important way to factor out common code or to hide platform heterogeneity, hinder locality. For instance, a block device driver involves the generic kernel object layer, the generic hard disk layer, the power man-
agement layer, then the block device itself. Each layer is likely to allocate memory independently from different pages. As a result, an OS execution path is likely to traverse multiple layers, and therefore accesses objects scattered all over different pages, resulting in excessive coherence operations. In other words, Linux tends to organize memory objects according to the horizontal abstraction layers, rather than the vertical execution paths that go through those layers – thus the poor locality.

Motivated by the observation, we speculate a locality-aware memory allocator that optimizes the performance of K2. The memory allocator should be aware of which OS service is currently using it and act accordingly. As a result, during execution, the working set (i.e., number of pages touched) of one given OS service will be reduced, as well as the number of coherence operations. We see two ways to implement this:

First, one can modify the kernel API for memory allocation, so that it accepts a tag as the hint for which OS identity will use the allocated memory. For instance, the tag could refer to a particular block device, or a socket. Accordingly, the kernel memory allocator should do best effort to accommodate allocations for the same OS identity with minimum number of memory pages.

Second, the OS can employ online profiling. Through profiling, the allocator can identify memory objects (by the contexts in which they are allocated, e.g., enclosing function, source file name, etc.) and discover the set of memory objects that are accessed by the same OS identities, or by the same execution paths. In future executions, the allocator can attempt to co-locate this set of memory objects on same memory pages. A practical solution may be a combination of the above two approaches.
6.10 Summary

Multi-domain SoCs promise high energy efficiency to a wide range of mobile applications, yet are difficult to program. A missing system support is replicating OS services to multiple coherence domains and run them with minimum inter-domain communication. To address this challenge, we identify a shared-most OS model for the architecture, and argue for transparently maintaining state coherence for extended OS services while creating per-domain, shared-nothing instances of core OS services. By applying the model, we build K2, a prototype OS on the TI OMAP4 SoC. K2 presents a single system image and reuses most of the Linux kernel source. Although still in an early stage under development, K2 improves energy efficiency for light OS workloads by up to 10x and provides almost the same performance as Linux, showing that the shared-most model is promising. The source of K2 is available from http://www.k2os.org.
Chapter 7

Concluding Remarks

7.1 Design Hints

The thesis work yields multiple hints for system design on loosely coupled, heterogeneous processors, as discussed below.

1. **Software structure shall mirror hardware asymmetry.** Software on asymmetric processors shall be optimized for asymmetric objectives. For instance, code executed on strong processors should favor performance while code executed on weak processors should favor efficiency.

2. **Managing state using software policies on top of hardware mechanisms.** The hardware shall provide the mechanism that keeps a certain portion of state coherent across processors, and it is up to the software to decide which portion of state should be kept coherent.

3. **Considering refactoring legacy code before reimplementing from scratch.** As hardware architecture evolves rapidly towards loose coupling and high heterogeneity, it is usually costly or even infeasible to re-implement system software from scratch for the new hardware. In building system software, refactoring should be considered with a higher priority.

In our design, hint 1 ensures that the energy efficiency promised by the underlying loosely coupled resources is exposed to the higher level software; hint 2 limits the
communication overhead that is inherent in the loose coupling of resources; hint 3 reduces the engineering efforts in building system software for loosely coupled resources as an emerging architecture.

We do not claim these hints to be rigorous guidelines for system design; rather, as many other well-known design hints for computer systems [43], they are meant to be helpful when facing design trade-offs. Beyond personal computing systems, we expect these hints to be useful for server systems where heterogeneous architectures are becoming prevalent, as the tension among hardware heterogeneity, resource asymmetry, and engineering efforts becomes a major concern there.

7.2 Scalability

K2 is designed for a typical mobile SoC which consists of a few heterogeneous domains. As compute resources keep increasing, we next discuss how K2 should be adapted for the following architecture trends.

First, individual domains will accommodate more cache-coherent cores. Since individual kernels of K2 already have the Linux SMP support, K2 can (almost) transparently scale with these additional cores.

Second, one system may embrace more, but not many, types of heterogeneous domains, as determined by the current spectrum of mobile workloads. For $N$ domains ($N$ being moderate), K2 can be extended without structural changes: the DSM (§6.5.3) will track page ownership among $N$ domains as in [48]; the unified kernel address space (§6.5.1) will host the additional $N-2$ local virtual regions; the global physical region will still be managed by balloon drivers (§6.5.2), yet the relative locations of $N$ local physical regions depend on the intended use of the added domains. Overall, the asymmetric aspects of K2 (e.g., page allocation favoring strong
domains) will remain.

Another possibility is that a system incorporates many coherence domains of the same type, as seen on some scalability-oriented systems including the 48-domain Intel SCC [62]. Our shared-most model (§ 6.3.3) will be useful, while new OS implementations should be engineered for scalability. However, we do not expect to see so many domains on a mobile device in the foreseeable future.

7.3 Recommendation to SoC architects

In developing K2, we find that the following architectural features will greatly benefit system performance and efficiency, yet are still missing in today’s multi-domain SoCs:

- **Direct channels for inter-domain communication that bypasses the system interconnect.** In a typical modern SoC, coherence domains usually communicate through system-level hardware components, e.g., hardware mailbox in OMAP4. This hardware architecture is easy for SoC vendors to engineer, yet slowing down the inter-domain communication. Therefore, we expect that domain-to-domain channels will greatly benefit software coherence as implemented by K2. Note that the channel can be lightweight: compact hardware messages that serve as signals will suffice.

- **Efficient MMUs for weak domains.** To the best of our knowledge, very few modern mobile SoCs equip weak domains with MMU. Although the OMAP4 SoC we used for prototyping has a MMU for its Cortex-M3 cores, this MMU is not meant for multi-user, multi-process environment: it has limited support for permission check and a small TLB that has to be loaded by software. As a result, despite our high engineering efforts, K2 still suffers from sub-optimal
virtual memory performance. Ideally, weak domains should be accompanied by MMU with normal permission check and hardware page table walker. For instance, ARM may define lower power (and lower performance) MMUs while still keeping the same interface as its standard MMUs.

- **Finer-grained power domains.** When the strong domain is idle and the weak domain is active, ideally all unused hardware components, including the strong domain, should remain in low power mode, e.g., clock gated or power off, to aggressively reduce static power consumption. However, in modern SoCs this is often difficult to achieve, as many of such unused hardware components share power domains with the active hardware components, thus could not enter lower power mode. We expect finer-grained power domains could address this need by allowing hardware components to enter low power mode independently. However, managing a larger number of power domains challenges both SoC and system software design.

7.4 Conclusion

To keep pace with the soaring demand for energy efficiency, personal computing devices have pioneered the adoption of highly heterogeneous, non-coherent architectures. This seriously challenges the current software stack which has assumed memory coherence at its lowest level. Traditionally, mobile systems abstract heterogeneous resources behind narrow APIs, which limit flexibility, code reuse, and energy efficiency. Parallel or distributed programs, when facing a similar challenge of distributed state, usually leverage domain-specific programming abstractions – an approach unlikely to be adopted by mass mobile developers. To this end, we redesigned the mobile
software stack, from the user level down to the OS level.

To ease user-level programming, we created Dandelion and Reflex. Dandelion is a RPC scheme that allows mobile applications to span across smartphone and the surrounding wearable sensors. Reflex is a compiler and runtime system that provides programming transparency. Its technical heart is a software distributed shared memory (DSM) designed for energy efficiency. As opposed to the conventional wisdom of classic DSMs, the Reflex DSM uses a low-power coherence domain to host shared memory objects, which essentially keeps frequently updated program state on the low-power domain, thus minimizing the activation of high-power domains. By enabling applications to efficiently tap into heterogeneous resources, Reflex improves their energy efficiency by up to 4x. Started in 2009, Reflex was a pioneering research project that forecast the heterogeneous architectures widely adopted in today’s mobile devices, including Apple iPhone 5s and Microsoft Surface.

The experiences with Reflex gave insights into an even more challenging problem – enabling OS workloads to exploit heterogeneous resources. To this end, we created K2, an experimental OS that replicates its services over heterogeneous coherence domains, while still presenting a single system image. The main insight of K2 is its shared-most model that: 1) creates per-domain instances for a small set of core OS services with no shared state, and 2) transparently enables the remaining OS services to span multiple domains, backed by software cache coherence. Among other goals, K2 reuses most Linux kernel source and improves energy efficiency for OS workloads by 8-10x.
Bibliography


[73] Qualcomm. Snapdragon MSM8660 and APQ8060 Product Brief.


