Reflex: Managing Sensor Data Processing in Mobile Systems

Technical Report 03-15-2010

Xiaozhu Lin[1] and Lin Zhong[1,2]

[1]Department of Computer Science and [2]Department of Electrical & Computer Engineering
Rice University, Houston, TX 77005

Abstract

Many emerging mobile services leverage sensors available on mobile systems to acquire information regarding the physical world through *sensory data processing*, in order to serve human users intelligently and ubiquitously. While the sensors themselves can be quite low-power, sensor data processing has proven to be a key bottleneck in the system energy efficiency, which consequently hinders the adoption of sensor-based mobile services.

In this work, we present the design and realization of Reflex, a programming and system framework to address the challenge of efficient sensor data processing. Reflex provides a programming and operating abstraction, called *channel*, and a runtime system that manages the execution of channels, called *channel manager*. Reflex allows multiple applications to share a channel in order to minimize redundancy in sensor data processing. It further allows channels to be executed in microcontrollers increasingly available on their corresponding sensors. Such a decentralized Reflex, or dReflex, improves the system efficiency by disengaging the mobile system from high duty-cycle sensor data processing.

We provide a prototype of Reflex and dReflex based on Nokia N810 with both wired and wireless sensors, including accelerometer and camera. Measurement with the prototype shows that sharing sensor data processing can reduce the power consumption of N810 by up to 55%. Delegating sensor data processing to the sensors can further reduce the power consumption by up to 90%. A user study with developers also demonstrates that Reflex is easy to learn and use.

1. Introduction

Modern mobile systems have embraced a variety of sensors, including cameras, microphones, accelerometers, and GPS. Personal-area networking technologies such as Bluetooth provide even more wireless sensors. These sensors can supply rich information about the physical world, including their human users, or *sensory information*, to the mobile device. They have fueled creative developments in what to do with sensory information, or sensory applications.

In contrast, there is a lack of development in how to process the sensor data or acquire sensory information in the platform, operating system (OS), and programming interface.

Existing major mobile platforms treat sensors as data suppliers and process sensor data in an application-specific manner with a centralized paradigm. This has led to excessive inefficiency that challenges battery lifetime and thermal management, as acknowledged by many authors [1-8]. Not surprisingly, most deployed sensory applications are on-demand, such as user interfaces and games, as versus periodic or continuous applications like Google Latitude. Periodic applications are rare and often limited to information that can be acquired efficiently, e.g. assisted GPS. And users still complain about their sizable negative impact on the battery lifetime [9]. Continuous or periodic applications with other sensors, e.g. cameras and microphones, largely remain as research prototypes.

We address the efficiency of sensor data processing with a programming and system framework, called Reflex[1]. Reflex provides two small sets of APIs to allow developers to effectively separate the use and processing of sensor data and encapsulates the latter in an abstraction called *channel*. A channel takes sensor data as input and outputs sensory information, ranging from processed sensor data to recognized events. Reflex provides a runtime, called *channel manager*, to manage the execution of all channels and to improve the execution efficiency as follows. First, the channel manager executes channels from all applications as internal threads and therefore minimizes process context switches due to periodic sensor data processing by applications. Second, Reflex allows multiple applications to share a channel by subscribing to it. Finally, Reflex provides a sensor runtime to allow the channel manager to execute a channel outside the system, namely in the programmable microcontroller of the corresponding sensor, disengages the system from high duty-cycle sensor data processing, and realizes decentralized sensor data processing, or dReflex.

We have prototyped Reflex for Nokia N810, a Linux-based Internet Tablet and prototyped dReflex with both wired and wireless sensors, including Rice Orbit sensors and the CMUCam3. Our implementation of Reflex consists of more than 2000 lines of C code. The dReflex sensor runtime based on µC/OS-II contributes additional 2000 lines of C

---

[1] Reflex arc is an important part of the biological nervous system and allows animals to react to stimuli without involving the brain. It is an example of decentralized sensor data processing in biological systems.

1

code and requires only 2KB RAM and 15KB Flash. Using the prototype, we are able to evaluate both Reflex and dReflex against conventional sensory applications. Our measurement demonstrates that Reflex improves execution efficiency of sensor data processing when there are multiple sensory applications running simultaneously. Furthermore, channel sharing reduces the power consumption of N810 by up to 55%. Finally, delegating the channel to an MSP430-based sensor reduces the power consumption by up to 90%. An informal study with sensory application developers shows that Reflex APIs are easy to use and existing code can be rapidly ported to Reflex.

On the other hand, our experiments also show Reflex incurs slight overhead in handling sensory events and running a single sensory application in a mobile system. Such limitation is inherent to the fundamental design tradeoff Reflex makes between processing of sensor data and delivery of acquired information to the application. Reflex leverages a key observation of emerging sensory applications that acquired information only needs to be delivered to the application occasionally but the sensor data processing must run periodically or continuously for a continual "sense."

As a programming and system framework, Reflex complements existing technologies in energy-efficient sensory application design, such as sensor selection [5-8] and data fidelity adaptation [3, 4]. Moreover, the effectiveness of dReflex suggests a platform change to allow programmability into digital sensors currently available on mobile systems.

The rest of the paper is organized as follows. We examine sensory applications, analyze existing support for them, and distill platform requirements for sensory computing in Section 2. We present the overall design of Reflex in Section 3. We discuss the channel and channel management of Reflex in Sections 4. We present the decentralized Reflex or dReflex in Section 5. We report prototypes of Reflex and dReflex based on Nokia N810 in Section 6 and an extensive evaluation in Section 7. We discuss the limit and future work of Reflex in Section 8 and related work in Section 9. We conclude in Section 10.

2. Sensory Applications on Mobile Devices
2.1 Characteristics of Sensory Applications
Based on our own experience, a survey of sensory applications reported in literature, and a careful examination of existing systems, we make the following observations regarding emerging sensory applications:

Firstly, we observe that increasingly more applications are intended to serve users for an extended period of time, or even continuously, e.g. [6, 9, 10]. To conserve energy, these applications typically run at regular intervals in the background. We call such applications *periodic*, in contrast to on-demand. They are critical to truly ubiquitous mobile services.

Secondly, Sensory applications access sensors with a relatively high duty cycle due to usability considerations and the nature of sensory information. A 10% or higher duty cycle, e.g. three seconds access every 30 seconds, is common.

Thirdly, events detected from sensor data and interested by applications happen rarely (sparsity) and are difficult, if possible, to predicted (asynchrony). Therefore, although a sensory application has to monitor a sensor periodically, most of the sensor data will be uninterested to applications. This is in contrast with conventional peripherals such as disk drive and network interface card.

Finally, the same sensory information may be required by multiple applications. For example, the direction of movement can be simultaneously required by the user interface [11], wireless interface [12], video codec [13], a game [14], and a health monitoring service [2].

2.2 Design Principles of Reflex
The characteristics of sensory applications as analyzed above naturally lead to the following principles that drive our design and realization of Reflex.

First, the system must execute sensor data processing for multiple applications efficiently because sensory applications are likely to be resource-hungry through their high duty cycle sensor data processing. Existing systems incur frequent process context switches between the sensor data processing of multiple applications, leading to inefficient execution.

Moreover, the system should manage sensor data processing as a resource that can be shared in a way transparent to the application and application developers. Modern systems have provided solutions for system entities to share and manage sensor data only, e.g. zero copy [15]. Therefore, each application has to process the data separately in existing systems.

Finally, the system should disengage the host from sensor data processing with high duty cycles. Existing systems employ a centralized paradigm that engages the host with all sensor operations. This leads to over a six-fold power increase when the accelerometer is used on Android G1 even with a duty cycle of 10%. While it might be acceptable for on-demand, occasional use, it is excessive for periodic applications.

3. Overview of Reflex
We first provide an overview of Reflex.

3.1 Major Components and Concepts
Reflex consists of four key components: channels, a channel manager, and two APIs. Reflex encapsulates sensor data processing in a channel. A channel is essentially a piece of code that process sensor data and output sensory information that interests an application. It runs periodically. Sensory information can range from processed sensor data to
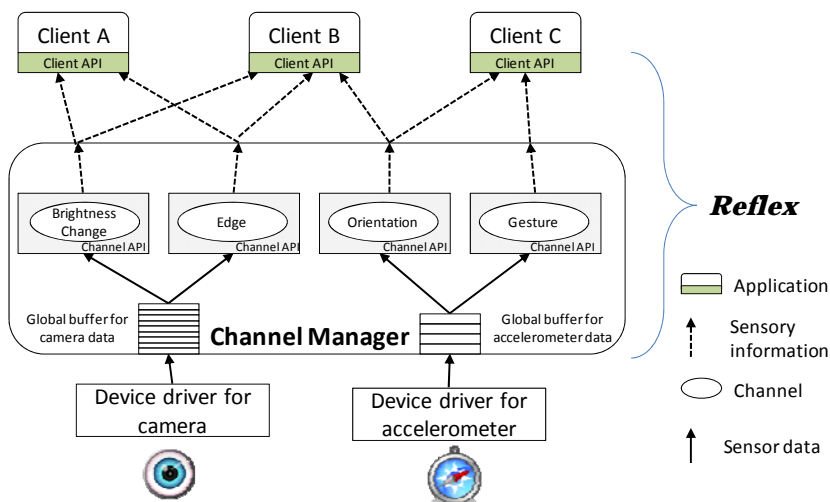
Figure 1. An overview of the Reflex framework. Reflex encapsulates sensor data processing as a channel and provides a channel manager to manage the execution of all channel instances. The channel manager also maintains a data buffer for each sensor and allows multiple clients to share a channel. A client and a channel employ the client and channel APIs, respectively, to interact with the channel manager

events meaningful to sensory applications. Applications can obtain sensory information by subscribing to the right channel as clients. A channel can either be provided by a client at run-time or preinstalled in the system as part of a library.

The channel manager manages the execution of all channels. The basic function of the channel manager is to create an instance of a new channel when a client subscribes to it. It allocates a global data buffer for each sensor used and share the buffer among all channels of the sensor to minimize data copying. More importantly, the manager provides mechanisms to share channels among clients and determine the details of their execution.

Reflex provides two APIs for software development: one, called *client API* and implemented as a library (libclient), for sensory applications to interact with the channel manager and access channels; the other, called *channel API* and implemented as a library too (libchannel), for a channel to interact with the channel manager and access system resources.

Figure 1 provides an overview of how different components of Reflex work together.

3.2 Programming with Reflex

Reflex employs event-driven programming and divides the development into two parts: code to process sensor data, or *channel code*, and code to use the sensory information, or *application code*. Note that the developers of the two parts can be the same or different.

Using the channel API, Reflex supports channel code development using C with syntax restrictions and a set of annotations for safety and efficiency. We will address channel code development in detail in Section 4.1.

Reflex supports application development with the client API. Figure 2 shows example code to illustrate the structure of a sensory application with Reflex. Note that we name client API using lowercase words, e.g. `channel_list`. The application firstly checks if the interested channel is already created (line 3 and 4).If not, it creates a new channel instance by pointing to where the channel package is (line 8). If the channel is already created, the application only needs to call a client API function, `channel_subscribe`, to register a callback `event_handler` that expects the sensory information from the channel of ID (line 9).

3.3 Efficiency Benefits of Reflex

By separating the acquisition and use of sensory information and encapsulating of the former in a channel, Reflex provides the following key benefits to the efficiency of sensory applications.

First of all, by executing all channels as threads within the channel manager, Reflex reduces process context switches when running multiple sensory applications. Secondly, Reflex allows multiple clients to share a channel by subscribing to it and therefore minimizes redundancy in sensor data processing. Finally, Reflex allows a channel to be executed by the programmable microcontroller on its sensor and therefore disengages the system from sensor data processing of high duty cycles. Our prototype-based experiments demonstrate all three benefits as will be addressed in Section 7.

4. Reflex Channel and Channel Management

Channels are abstractions of sensor data processing. In this section, we describe how a channel can be specified and the internals of the channel manager.

```
1    // Using channel is straightforward
2    SensoryApp::Main() {
3        ChannelList = channel_list();
4        //Look for a channel in ChannelList...
5        if (NO_FOUND)
6            //create an instance
7            ID =
8                channel_create(CHANNEL_PACKAGE);
9        channel_subscribe(ID, EventHandler);
10       //enter event-loop;
11   }
12
13   SensoryApp::EventHandler(event, data) {
14       //the use of sensory information
15   }
```

Figure 2. Example application code for creating a channel instance, and using its sensory information with `EventHandler().`

4.1 Channel Code

The developer of a channel must provide its program code and a set of channel properties as the channel metadata that facilitates channel management.

In designing the programming space of Reflex channel code, we consider execution *safety* and *efficiency* as the two primary objectives, because a channel is executed by the channel manager and it has a high duty-cycle. The two objectives are even more important if the channel execution is delegated to a programmable sensor with primitive hardware support such as in dReflex to be discussed in Section 5. To address these two issues, at compile time we build *sandbox* for the channel code to assure its safety while retaining high performance. More specifically, we construct the sandbox by restricting programming language syntax and the channel API. We disfavor a run-time sandbox such like virtual machine because of its obvious computational overhead in executing channels.

4.1.1 Programming Support

Reflex requires channel programmers develop their channels using C language, with syntax restrictions for both code safety and resource accountability. The restrictions and annotations include:

- No pointers, goto statement or any inline assembly code;
- All loops must be annotated with maximum counts;
- No nonstandard library function call, except those provided by the channel API; and
- Access to sensor data only through the channel API.

We have implemented a channel code preprocessor that checks the syntax restrictions and converts the source code into standard C code. We note that there were existing safe-C research works [16][17][18] and our adopted approach is just for engineering convenience.

4.1.2 Channel API

Reflex employs the *channel API*, a collection of five functions, to support channel code with its interaction with the channel manager. Reflex ensures that calls to channel API

```
1    init () { /* ... */}
2    cleanup () { /* ... */}
3
4    /* definition of static variables */
5    int channel_state, ...
6
7    /* standard channel code entry */
8    int ChannelEntry () {
9        int i = 0, has_err = 0, x, y;
10       while (!has_err) { /* MAX_COUNT=10 */
11           i++;
12           /* Access pixels in the i-th frame ... */
13           pixel_value = GetSample(i, x, y);
14           /* More processing follows ... */
15       }
16       ReportEvent(MY_EVENT_ID);
17       return STATUS_OK;
18   }
```
(a) Code

```
1    <?xml version="1.0" encoding="UTF-8" ?>
2    <channel name="edge">
3        <IO>
4            <sample_rate> 20 </sample_rate>
5            <input_buf_len> 5 </input_buf_len>
6        </IO>
7        <resource>
8            <WCET> 10 </WCET>
9            <stack_depth> 100 </stack_depth>
10       </resource>
11       <schedule> ... </schedule>
12       <misc> ... </misc>
13   </channel>
```
(b) Metadata

Figure 3. An example channel

functions are safe for the channel manager. The channel API functions do not accept nor return any pointer; nor do they change the internal state of the channel manager. In addition, channel API functions all take deterministic time to complete, which facilitates channel executable resource modeling.

Rather than directly implementing channel API functions in the channel manager, we implement them with a lightweight wrapper library *libchannel* (less than 100 lines of C code) to be statically linked with the channel code. Through statistically linking, libchannel ensures the safety and efficiency of the interaction between the channel code and the channel manager. First, it provides a trusted middleware layer for the channel code to access internal data structures of the channel manager. Moreover, during compilation, it inline-expands performance-critical channel API calls as direct readings and writings regarding buffers managed by the channel manager for higher efficiency.

Figure 3. (a) provides example channel code that demonstrates the use of channel API functions. The code provides initialization, cleanup routines (line 1 and 2) that are invoked during channel creating and destroying, respectively. It provides one entry function (line 8) that is called by the channel manager in each period. The channel entry function accesses sensor data in channel's input queue, (`GetSample`, a channel API function, line 13) and report any acquired sensory information to the channel manager (`ReportEvent`, a channel API function, line 16).

4.1.3 Executable Generation

Reflex takes two steps to generate the final channel executable. In the first step, it employs a preprocessor to enforce Reflex restrictions and expands special annotations into C source code. The preprocessor also inserts run-rime checking code for array access before producing the standard C code with equivalent function as the channel code. In the second step, Reflex compiles the output of the preprocessor, links it with libchannel as described above, and generates the channel executable as a shared library so that the channel manager can dynamically load for execution when it creates an instance of the channel.

4.2 Channel Metadata

In addition to the channel code, the channel developer also provides a set of metadata to facilitate resource management by the channel manager and promote sharing. The metadata defines key properties of one channel, including:

- **I/O Specification:** The sampling rate of sensor data, the length of channel's input queue, and the length of the output buffer.
- **Resource Model:** The channel period. Parameters that characterize the time and space computation resource requirements of a channel, including the size of program segments, the maximum depth of stack, and Worst-Case-Execution-Time (WCET).
- **Identities:** Channel's Universal Unique ID, which uniquely indentified a channel to facilitate sharing; textual description of channel function; the information of the channel developer; channel licensing information; version information, etc.

Reflex saves the channel metadata in XML format (See Figure 3(b) for example). Most of the metadata is provided directly by the channel developer. Only the resource model must be provided by a trusted agent because it is critical to channel scheduling and resource allocation. With the programming restrictions, existing techniques in worst-case execution time (WCET) analysis (e.g. [19]) and maximum stack depth analysis (e.g. [20]) readily provide the resource model based on either source code, intermediate representation, or even binary. After the channel is created, the channel manager makes its metadata visible for other potential clients to query in order to maximize the chance of sharing the channel.

4.3 Channel Management

Reflex manages channels through the channel manager. For channels, the channel manager sets up runtime environments, allocates computation resource and executes them. In this section, we address other key functions in Reflex channel management, including channel execution, comparison, and sharing as well as access control.

Sensor Data Acquisition: The channel manager creates an input queue of sensor data for each channel according to its metadata. To minimize data copying, the channel manager keeps a global buffer of sensor data and only put indexes to the global buffer entries into the input queue of a channel, according to the channel's sampling rate requirement.

Channel Scheduling: Reflex executes a channel instance as a thread inside the channel manager and therefore delegates the scheduling task to the native OS. Because a channel runs periodically, the manager marks the channel's scheduling status properly. This one-thread-per-channel has a drawback in the scalability in comparison to a single-threaded design that maintains a set of channel executables and schedule them accordingly, similar to how Linux kernel executes the bottom-halves of interrupt handlers [21]. However, we decide the simplicity of the channel manager is more important when the number of channels is not very large; we therefore adopt the one-thread-per-channel design.

Channel-Client Communication: A channel needs to contact its clients when having acquired sensory information interested by applications, in the form of event or processed data. Since the subscriber lists are maintained inside the channel manager, channel code communicates with its clients by making requests to the channel manager, with the support of three API functions. Output() writes processed data to channel's private output buffer. The channel manager later makes the written data available to all clients, by sending in IPC messages or sharing memory across processes. ReportEvent() notifies the occurrence of a sensory event by multicasting an asynchronous signal to all its clients. ReportSensorData() reports raw sensor data in the input buffer, for providing backward compatibility to legacy applications. On the client side, the corresponding signal handlers, registered at the time of subscribing, will be invoked. As a result, clients will learn the occurrence of event and then access processed data.

5. Decentralized Reflex

By separating the acquisition and use of sensory information, Reflex allows them to be executed on different hardware other than the host mobile system. We observe that computing resources are increasingly available on sensors. Not only computationally-intensive sensors such as cameras already have dedicated hardware, but also lightweight sensors like accelerometers, e.g. [22, 23]. Some already provide limited programmability with a few predefined methods, e.g. angle calculation [22]. By making such resources programmable at runtime by the host mobile system, we can delegate the channel execution to them and realize a decentralized Reflex, or dReflex.

dReflex will not only reduces the sensor data traffic between the host and sensor but also disengage the host from sensor data processing of high duty cycles. Such decentralized realization acquisition resembles natural nervous systems, which usually provide both centralized and peripheral support for sensor data processing [24].

5.1 Sensor System Design

In view of the extremely limited computing resources available on sensors, we adopt a *minimalist* design principle and

leave most of the management complexity in the host. This minimalist design also facilitates the sensory application development because it shifts most code development to the host. According to the minimalist design principle, we only leave the following functions to the sensor: sensor data acquisition, channel execution, and communication with the host. Using a prototype reported in Section 6, we show that such functions can be realized with the following hardware primitives:

- Interrupt handling;
- A timer;
- Access to digitalized sensor data, e.g. through a built-in ADC or a digital interface with the sensing apparatus;
- A bi-directional digital interface with the host for loading code and interrupting the host. The interface can be wired or wireless; and
- A very small memory requirement: as small as about 15KB flash and 2KB RAM for sensor software.

We observe such hardware primitives are available on most 16-bit and even some 8-bit microcontrollers.

The three functions essentially demand the sensor software stack to be multitasking. Therefore, we adopt uC/OS-II as the underpinning kernel and leverages its Earliest-Deadline-First (EDF) scheduling. While Reflex creates a thread inside the channel manager for each channel instance, a dReflex sensor creates a lightweight process for each delegated channel. Because that the channel manager stays in the host, the sensor runtime carries out management of the global buffer for sensor data according to a schedule from the channel manager.

The sensor utilizes a process called *proxy* to communicate with the host on behalf of sensor and it manages resources according to host instructions. Because the proxy process works under the dictation from the host, its implementation is straightforward and therefore incurs little computational overhead.

It is important to note that the host must ensure that the sensor can afford the resources required by a channel before delegating it using the resource model of the channel and knowledge of the sensor system. This is important for both sensor resource management and safety.

5.2 Channels for Decentralized Reflex
Executing a channel outside the host invites new problems in channel generation and execution. While addressing these problems, we keep the channel programming model and channel API unchanged.

Compilation: When the channel manager determines that a channel should be executed on the sensor, the channel manager must have the binary native to the sensor system.We favor the native binary due to efficiency consideration. However, channel developers may not know the sensor

hardware at the time of development. Even worse, they may refuse to distribute channels as source code for intellectual property (IP) consideration. Reflex addresses this challenge by distributing channels in an intermediate format for IP protection and by producing the native binary at the time of delegation in the channel manager.

Dynamic Link and Loading: Recall that Reflex compiles the channel executable as dynamically linked library to glue it with the channel manager at run-time, as discussed Section 4.1.3. This poses a challenge to delegating channels to the programmable sensors. When there are multiple channels to delegate, the order of delegation can be arbitrary, depending on the order of channel creations by clients. As a result, only at the time of delegation can the memory location of channel executable be resolved and can addresses of channel executable symbols be determined. Therefore, the channel manager of dReflex links a channel executable right before the channel is delegated to determine the addresses of symbols. We decide against compiling the channel executable as Position Independent Code for efficiency consideration.

5.3 Sensor-Host Communication
Under dReflex, we call the mobile system the *host* of the programmable sensor. The sensor communicates with its host communication bi-directionally. The host needs to download channel executables to the sensor and dictates its resource management; on the other hand, the sensor notifies the host when events interested by applications are detected and report acquired sensory information. We call the former host-initiated communication and the latter sensor-initiated communication.

Host-initiated Communication. The host initiates communication with the sensor to control the operations of the sensor with commands and to delegate a channel. Because operation control requires high reliability and error-recovery and the sensor is usually resource limited, we design the communication of commands to be synchronous. That is, the proxy on the sensor always listens on the communication port, immediately acknowledges the reception of a command from the host, and piggy-back responses to the host's inquiry. The host will retransmit the command if it does not receive acknowledgement upon timeout. It can even reset/reprogram the sensor if it determines that the sensor is stalled or faulty.

Because a channel executable can be much larger than commands, the host splits the communication to ensure the reliability. The host first instructs the proxy to stop sensor data acquisition so that the transfer of channel executable will not be disrupted. Then, the host instructs the proxy to set up the input queue and output buffer needed by the new channel. Next, the host ships channel metadata packed in a header structure. After the proxy acknowledges that the header is saved and the receiving buffer is prepared, the host transmits the channel executable binary. The proxy gets the

executable, sets up its segments in the memory, at addresses specified in the channel metadata. Finally it creates a lightweight process to wrap the new channel executable. After the proxy acknowledges that all above is successful, the host instructs the proxy to resume sampling on the sensor.

Sensor-initiated Communication. Proxy on the sensor initiates communication with the host to notify the latter that there are data, i.e. channel output, available for it. To maximize the opportunities for the host to sleep, proxy only initiates communication when reporting new sensory information by interrupting the host. While the host-initiated communication is synchronous, we design the sensor-initiated communication to be asynchronous because the host is much more resourceful to deal with bursty sensory information than then sensor. In reporting an event, the sensor sends a textual message with the channel instance ID and the event identifier. To transmit processed data, the sensor packs data as binary in the same message. Interrupted by the reception of such message, the native device driver on the host copies the message to the channel manager. The channel manager in turn parses that message and further dispatches the information to corresponding subscribers.

6. Prototyping Reflex

We have implemented Reflex with C and C++ for Nokia N810 Internet Tablet and N900 smartphone. We next only report the prototype with Nokia 810.

6.1 Nokia N810

Nokia N810 features a TI OMAP2420 processor, 128MB RAM, 256MB main flash, and rich peripherals including Bluetooth, USB and WiFi interface. Although it does not come with a cellular interface, its application system is similar to that of modern smartphones. We choose N810 for two reasons. Hardware-wise, N810 exposes an RS232/UART interface and is one of the few mobile devices that expose a digital interface with interrupt capability on both ends. It is worth noting that USB 1.0/2.0 is not interrupt capable and a USB host needs to poll the bus to receive data. Software-wise, it is open-source and cross-platform GCC ld and GNU binutils are available, facilitating the realization of dReflex. According to our measurement, N810 consumes about 40mW when in the deep-sleep mode and about 200-500mW when active, without display, Bluetooth or WiFi.

6.2 Reflex Implementation

The Reflex prototype consists of two user libraries libclient (about 300 lines of C code), libchannel (about 100 lines of C code) and the channel manager (about 2000 lines of C++ code) as a stand-alone system daemon. The prototype implements the communication between clients and the channel manager based on dbus, an IPC mechanism widely available on mobile systems, and implements sensory event notification using POSIX-style signals. The prototype is open-source and available from our SVN repository [25].
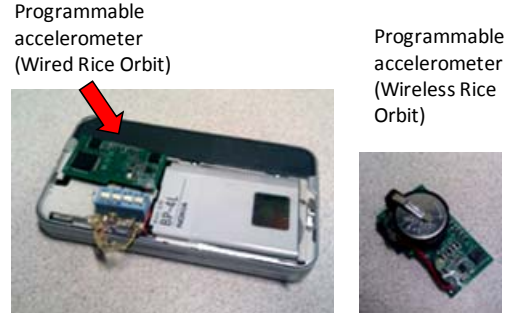

Figure 4. **Reflex prototype implemented with Nokia N810 and two programmable sensors for dReflex prototype.**

We implement the channel manager with three well-defined layers: the interface layer, the resource layer and the execution layer. The *interface* layer exports a set of dbus methods that enable channel access in a unified way. The *resource* layer allocates and maintains all comsputational resource used for channel execution and maintains a subscriber lookup table for each channel. The *execution* layer actually acquired sensor data from the native device driver, maintains input queues and output buffers, and creates threads for channel instances. A compact and clean interface is defined between the resource and execution layers so that only the execution layer must be adapted for the dReflex sensor platform and the upper two layers stay on the host system without any change.

Reflex minimizes the overhead in delivering the processed data from a channel to subscribers. After the channel outputs processed data, the channel manager saves such data in its own buffer as the unique copy and shares it among all subscribers with read-only memory mappings. Therefore, a subscriber can read the processed data without copying, unless it needs to run an in-place algorithm on the data.

6.3 dReflex Implementation

Based on the Reflex prototype, we have further realized dReflex. With N810 as the host system, we implement dReflex sensor runtime for two programmable sensors: the Orbit accelerometer and the CMUCam3 camera. The two programmable sensors represent two extremes in terms of data intensity and the computation in sensor data processing. For the Orbit accelerometer, both wired and wireless connections with the host are realized and evaluated.

Programmable Accelerometer: We build the programmable accelerometer prototype using the open-source, Orbit sensor platform [26]. An Orbit board carries an ultra low-power 16-bit TI MSP4301612 microcontroller, a KXM52 three-axis accelerometer and a KC21 Bluetooth interface. The Orbit accelerometer can talk with N810 over the UART serial port and Bluetooth. Compared to commercial digital accelerometers such as [23], the Orbit accelerometer is quite power-hungry, consuming 30 mW when active and 18 mW when idle in a low-power mode.

Table 1. Benchmarks of sensor data processing

Sensors	Channels
Accele-rometer	**Orientation**: detect the device orientation change from acceleration data. If the change exceeds a preset threshold, report an event as the acquired sensory information. With about 40 lines of C code.
	Gesture: match the user gesture with preset patterns. The algorithm used is simplified Dynamic Time Warping from our prior work [28]. Report the gesture matching event as the acquired sensory information. With about 300 lines of C code.
	Data: report raw reading at 32Hz.
Camera	**Brightness**: detect the overall brightness change in captured image frames. If the change exceeds a preset threshold, report an event as the acquired sensory information. With about 55 lines of C code.
	Edge: detect edge in horizontal, vertical or diagonal direction, using Sober operator with Hough transform. Report the detected edge as the acquired sensory information. With about 100 lines of C code.
	Data: report image frame at 10Hz.

Programmable Camera: We build the programmable camera using the open-source CMUCam3 [27]. CMUCam3 is a small embedded platform that features an ARM7 micro-processor (LPC2106). We connect CMUCam3 and N810 through their UART/RS232 interfaces. The CMUCam3 consumes about 510mW power when active according to our measurement.

We note that the power consumption of both our programmable sensors is much higher than that of their commercial counterparts, especially digital sensors on mobile systems. We choose them only because existing commercial mobile devices do not allow any programmability into their sensors. While the high power consumption of our programmable sensors undercuts the significant power reduction achieved in their host mobile device, i.e. the Nokia N810, it is due to the non-optimized design of the sensors, instead of the limit of Reflex or dReflex. Indeed, the sensors' programmability is achieved by very efficient processors, TI MSP430 for Orbit (6.5 mW when active) and ARM7 LPC2106 (54 mW when active) for CMUCam3.

Sensor Runtime: We port μC/OS-II to both Orbit and CMUCam3 and implement the sensor runtime with about 2500 lines of C code in addition to μC/OS-II code. We compile the sensor runtime with μC/OS-II together, using msp430-gcc for Orbit and arm-none-eabi-gcc for CMU-Cam3. We also cross-compile *binutils* in corresponding GCC tool chains for N810 to link channel executable for sensor native binary at run-time. The whole software stack for Orbit requires 15KB Flash and 2KB RAM; that for CMUCam3 requires 17.5KB Flash and 8KB RAM.

7. Experimental Evaluation and Analysis

We design experiments to evaluate both Reflex and d-Reflex with the prototype realizations. In addition, we report a study with programmers to evaluate the learning curve and usability of Reflex.
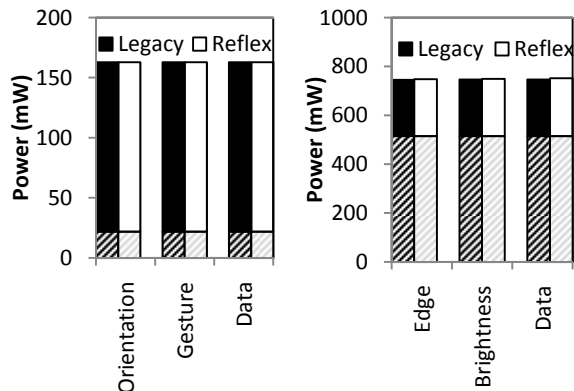

Figure 5. System power consumption when running a single application of each benchmark: (Left) Accelerometer and (Right) Camera. It shows that Reflex introduces <1% overhead even when there is no sharing or delegation. For each column, the bottom part shows the power of the programmable sensor and the top part shows that of N810

7.1 Experimental Setup and Benchmarks

We measure the power consumption of N810 and sensors at 100Hz with a data acquisition system (USB-2533) from Measurement Computing. We include the power consumption by both N810 and the sensor when reporting. Because the sensors are not optimized for power, we distinguish their contribution to the power so that the power impact on N810 is apparent. In particular, in the column charts of system power, the bottom and top parts are the contributions by the sensor and N810, respectively.

Because the host and the programmable sensors have independent clock sources, to study the overhead in latency, we measure the hardware clock offset between them at the beginning of the experiment. We measure the latency by inserting code that logs timestamps of critical operations. Timestamps have a resolution of 1ms.
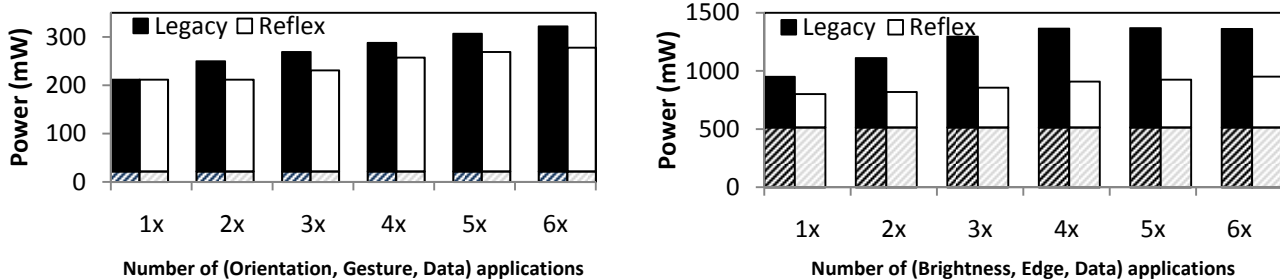
Figure 6. System power consumption when running applications of benchmarks without sharing: (Left) Accelerometer (Right) Camera. For example, 2x of (Orientation, Gesture, Data) means that running two applications of each of the three benchmarks together. For each column, the bottom part shows the power of the programmable sensor and the top part shows that of N810
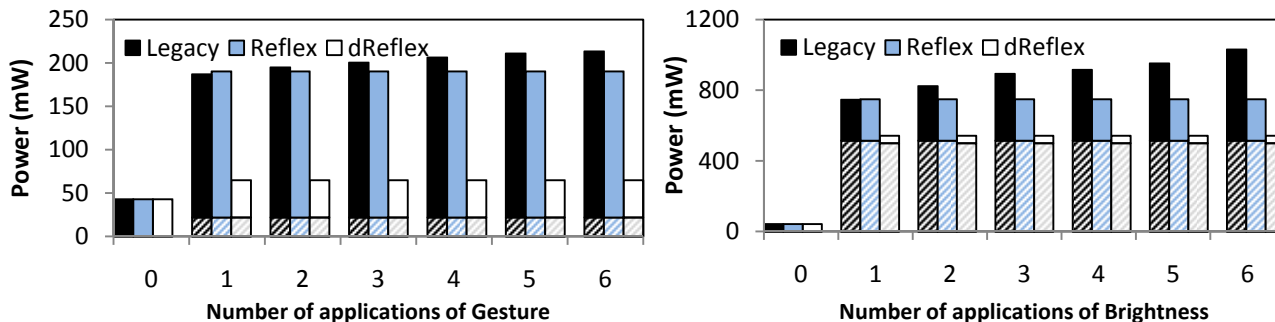

Figure 7. System power consumption when running an increasing number of dummy applications of Gesture (Left) and Brightness (Right). Due to sharing, the power of Reflex and dReflex remains largely unchanged as the number of applications increases, whilst that of Legacy keeps increasing. It also shows that dReflex allows N810 to sleep and therefore further reduces system power consumption. For each column, the bottom part shows the power of the programmable sensor and the top part shows that of N810

We evaluate Reflex and dReflex with six benchmarking sensor data processing: three for accelerometer and three for camera, as shown in Table 1. For comparison, we implement six dummy applications, one for each benchmark, for each of the three cases: without Reflex (Legacy), Reflex, and dReflex. We write a dummy Legacy application as a monolithic program that acquires sensor data from the serial port in a blocking way, buffers data in its internal queue and processes the data using the corresponding benchmark. We write a dummy Reflex/dReflex application as described in Section 3. It creates and subscribes to a channel of the corresponding benchmark and lets the channel manager mange the channel. The application does nothing but get the information output by the channel. For fair comparison, we keep all other aspects of the benchmarks identical for Legacy, Reflex, and dReflex.

In order to evaluate the overhead in running multiple instances of channels, we need to run multiple Legacy applications using the same sensor simultaneously. However, there is no Linux support for such simultaneous multiple access. As a workaround, we only grant one Legacy application access to the real device and emulate others favorably. That is, we let them read from their own local buffers, instead of from the actual sensor. Because reading from the local buffer is much more efficient than reading from the

real device, this emulation favors the Legacy applications in comparison with their counterparts with Reflex and dReflex.

7.2 Overhead and Benefit of Reflex Execution

Because Reflex employs the channel manager to execute sensor data processing, a natural question is if it incurs overhead. Figure 5 presents the measured system power when running one application of each benchmark for Legacy and Reflex. For the accelerometer benchmarks, there is no measurable degradation for Reflex; for camera ones, there is a 1% increase. This shows Reflex does incur a very small overhead for executing a single instance of data-intensive sensor data processing, possibly coming from the thread context switch between the channel manager main thread and the thread executing the channel.

That the channel manager executes all instances of channels as threads also have potential benefits of reduced context-switches between applications. Figure 6 presents the measured system power in running an increasing number (1 to 6) of application groups for Legacy and Reflex. Each application group consists of three applications, one for each benchmark. To focus on execution, sharing is not used in Reflex. That is, the Reflex channel manager runs from 1x3 to 6x3 (18) channels simultaneously. Figure 6 clearly shows that Reflex outperforms legacy applications in term of pow-
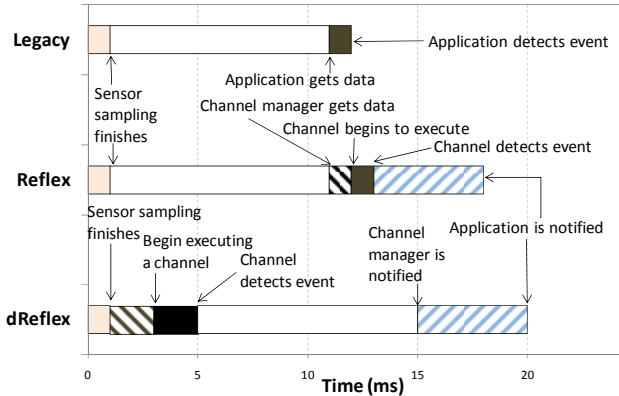
Figure 8. The breakdown of latency in sensory event handling, for Legacy, Reflex, and dReflex

er efficiency, up to 15% for accelerometer applications and up to 48% for camera applications, even though we are emulating legacy applications favorably. Importantly, the benefit of Reflex execution increases as the number of applications increases; the benefit is more for compute-intensive benchmarks, i.e. camera ones than accelerometer ones. We attribute such increasing advantage to that Reflex incurs fewer process context switches in execution. In summary, Figure 5 and Figure 6 show that Reflex provides increasing benefit in efficiency when there are more sensory applications and more compute-intensive sensor data processing.

7.3 Channel Sharing

A key benefit of Reflex by design is that multiple sensory applications can share a channel and therefore minimize redundant processing. We take one benchmark for each sensor to demonstrate such benefit: Gesture for accelerometer and Brightness for camera.

Figure 7 presents the measured system power of Legacy, Reflex, and dReflex. When there is no application, all three cases have the same power consumption as N810 in sleep and sensor powered off. As the number of applications increases from one to six, Reflex and dReflex shows almost no changes in system power because all because all applications subscribe to the same channel for sensor data processing. In comparison, the Legacy case consumes more power as the number of applications increases. The power reduction in N810 due to Reflex is up to 55%. And the increase is more prominent with the camera whose benchmark is more compute-intensive than that of the accelerometer.

7.4 Channel Delegation

dReflex and delegation provide another key benefit of Reflex by design. Figure 7 clearly demonstrate it. dReflex reduces the power by N810 by 60%~80% compared with Reflex and by 75%~90% compared with Legacy. It is also worth noting that dReflex significantly reduces sensor-to-host data transfer and therefore reduces the power consumption by the sensor too. Not surprisingly, dReflex also inhe-

rits the benefit of channel sharing from Reflex and incurs little power increase with more applications.

7.5 Overhead of Event Handling

While Reflex significantly improves the efficiency of channel execution, it introduces overhead when information interested by applications is acquired and corresponding applications must be notified. This is a fundamental design tradeoff made by Reflex to improve the efficiency of tasks that run often (sensor data processing) at the cost of the efficiency of tasks that run rarely (sensory event handling). The rationale is that events interested by applications happen much less frequent than the execution of the channel, or sensor data processing. We next use the prototypes to quantify the overhead in event handling. Note that we use event here to broadly refer to the output of sensor data processing that is interesting to the application. It can be data or actual event.

Latency: We define the latency in sensor data processing as the time elapsed from when the last sample is acquired by the sensor to the moment that the application is notified. Reflex introduces extra stops for the acquired data; dReflex further adds stops on the programmable sensor. We break the latency into fine-grained steps and compare the latency breakdown among Legacy, Reflex, and dReflex. We use the Orientation benchmark of accelerometer with one application as the example and illustrate the latency breakdown results in Figure 8. The figure shows that the major extra latency of Reflex comes from channel-to-subscriber IPC communication as expected. dReflex incurs more latency by executing the benchmark channel on the slower MSP430 controller (4ms), that the host CPU takes extra time to wake up from sleep mode when interrupted by the event (10 ms according to our measurement), and channel-to-subscriber IPC communication (5ms). Overall, Reflex and dReflex brings extra latency of about 20 ms in handling an event.

We emphasize that the extra latency is inherent to the designs of Reflex and dReflex, rather than flaws in implementation. Reflex's basic design principle, separation of the processing of sensor data and the use of sensory information, necessitates IPC communication overhead. By delegating channels to sensors, dReflex inevitably slows down their execution.

It is worth nothing that since the programmable sensor and the channel manager are multitasking and can deal with events in an overlapped manner, Reflex/dReflex actually can achieve higher rate of event notification than the latency may suggest. Indeed, we have verified that our dReflex accelerometer can deal with an event rate as high as 128Hz, instead of 50Hz suggested by the 20ms latency.

Energy Overhead: In Reflex/dReflex, energy overhead in event handling comes from the channel manager's extra operations in looking up for right subscribers and notifying them. Since it is difficult to accurately measure such overhead, we estimate it pessimistically as the whole system

energy consumption starting from channel detects the event until the subscriber is notified. Because our measurement shows that the duration is less than 7 ms, we estimate the energy overhead is about 1.3 mJoule, which is acceptable given that events are sparse.

7.6 dReflex with Wireless Sensor

Our dReflex prototype also works with the Orbit accelerometer over a Bluetooth connection. Our measurement shows that even delegation to such a Bluetooth sensor can reduce the overall system power consumption. The Bluetooth sensor and N810 consume 80mW and 70mW when connected in the Sniff mode, respectively, giving a total system power of 150mW for dReflex. The total power is lower than the 190mW system power consumption of the Legacy case as shown in Figure 7. However, the use of the Bluetooth Sniff mode will introduce new latency due to the Sniff period (about 400ms in our case) during which the sensor and N810 cannot communicate. On the other hand, if the Sniff mode is not used, the delegation will not be profitable. However, we expect emerging ultra-low power Bluetooth technology (Wibree) to significantly bring down the power cost of Bluetooth and make dReflex with wireless sensors more attractive.

7.7 Programming with Reflex

Because Reflex requires changes to the sensory application programming, we evaluate its learning curve and usability with an informal study with nine programmers from the first author's social network. The participants include employees by IT companies, graduate students, and free-lancer developers. They have been involved in software development with hundreds to ten thousand lines of code.

We ask them to read a user guide of Reflex and implement two small sensory programs with Reflex: using accelerometer to detect free-fall event and using camera to detect brightness change. The guide provides one example code and we also provide them with pseudo code that realizes the sensor data processing algorithms.

All participants learned Reflex and finished the programming tasks correctly within one hour, though with minor flaws or typo. Eight of them rated programming with Reflex as easy to learn and use; only one had a neutral opinion. All indicated that they understood the programming model. Two of them felt that their coding flexibility is limited by Reflex restrictions, as described in Section 4. Three expected more features of the programming model, such like more support from libchannel.

While the feedbacks are encouraging to a new programming paradigm like Reflex, we acknowledge that Reflex achieves safety at the cost of programming flexibility. To make the programming model more usable, we must further abstract sensor data processing and enrich channel API in future work.

8. Discussions

Our evaluation clearly demonstrates the strength and limitation of Reflex in efficient sensor data processing. Reflex significantly improves the efficiency of sensor data processing itself, or the processing of sensor data. Yet it does so with a small sacrifice in the efficiency of handling sensory events, or delivering acquired information to interested applications. As a result, Reflex is not for every sensory application, in particular those interested in events that appears at high probability when the sensor turns on.

Moreover, as our study with programmers shows, the programming restrictions for channel code also limit the function of channels or the sophistication of sensor data processing. Nevertheless, Reflex can be combined with conventional tools to improve the efficiency of sophisticated sensor data processing. For example, although it is inadequate for developing an entire camera-based computer vision application, Reflex can be employed in the early stages of camera data processing.

We note Reflex's support of delegation is more of a mechanism than policy. When channels are few per sensor and the sensor is wired, delegating a channel to its sensor is obviously more efficient. However, when there are more channels than the sensor can accommodate or the communication cost between a sensor and its host is expensive, e.g. Bluetooth, there is a need to determine which channels to delegate to balance the communication cost and host engagement in sensor data processing under the resource constraints on sensors.

We also note that Reflex provides a useful framework to develop even more mechanisms for efficiency enhancement for sensory applications. For example, many have considered mobile devices as "sensors" [10, 29-31] to collect massive amounts of data regarding the physical world so that the cloud can provide previously impossible services. Reflex can similarly improve the efficiency of the mobile devices for such services, which can be considered the clients of channels as the cloud being host and the mobile device being its "sensor". With dReflex, a cloud service can delegate a channel for sensor data processing to the mobile device to reduce the wireless traffic from the device. Furthermore, multiple cloud services from different providers can share a channel through multicast, leading to reduction in not only processing but also wireless traffic from the mobile device. Interestingly, the cloud platform also presents three tiers of computing resources for Reflex: wired and wireless sensors serve a mobile embedded system, which in turn serves the cloud.

9. Related Work

We next discuss solutions related to supporting sensor data processing and the Reflex framework.

9.1 Sensory Information Management

We find Symbian S60 provides the most advanced support for sensor data processing. For instance, it provides "accele-

11

rometer" and "orientation" channels from the same physical accelerometer. By allowing multiple applications to subscribe to the same channel, it allows limited sharing. Yet S60 is primitive in its support for sensor data processing. First of all, it only supports a few predefined sensors and a few predefined channels. This severely limits the opportunities for sharing and reuse. Furthermore, its fundamental design is expensive for data-intensive sensors, e.g. cameras, since it invokes significant data movement and frequent inter-process communication (IPC). Finally, it only supports user-space clients, however other system entities such as wireless interface card may also need sensory information [11, 32].

Many from context-aware computing have studied the middleware or management framework for sensory information from sensors, e.g. [33-36]. Their frameworks provide APIs to use sensory information through abstractions that relate, store, and retrieve acquired sensory information. Without providing the system support for its acquisition, however, they still address the use of sensory information.

9.2 Sensor Design and Selection

Sensors are a key component to wireless sensor networks (WSN). Their design is related to dReflex's sensor system design. Recent work in sensor system design has provided many efficient solutions for continuously vigilant operation, e.g. passive or hierarchical wakeup [37-40] and dual-processor design [41-43]. The authors of [3, 4] studied the tradeoffs between sensor data resolution and energy efficiency. The authors of [5-8] studied the selection of an energy-efficient subset of sensors. All these solutions improve the sensor data acquisition on the sensor system. They are complementary to the sensor system design of dReflex, as presented in Section 6. Moreover, most of them are specific to their applications or sensors. In contrast, Reflex is intended to provide generic support in programming and system management for sensory programs.

9.3 Work Related to dReflex

Related to dReflex, leveraging the computing resources outside the central processing unit (CPU) has been studied for various peripherals. Active disk [44], active network [45], and TCP offloading [46] delegate tasks to the hard disk, network router, and network interface card (NIC) respectively, in order to improve throughput.

In the same spirit, Weinsberg et al. [47] proposed to leverage computing resources available in peripheral devices, in particular network interface card (NIC), disk controller, and graphics adapter. Their solution is a framework called Hydra with programming and runtime support for developers to write code that can be delegated to peripheral devices. Despite their similarity in leveraging peripheral resources, Hydra differs from dReflex in the following important ways. Firstly, their goals are completely different. dReflex's goal is to improve the system efficiency by delegating high duty-cycle operations of sensor information acquisition to its sen-

sor so that the host can sleep. In contrast, Hydra's goal is to improve the system performance by harvesting the resources in peripherals. Secondly, dReflex only delegates sensor data processing to its sensor while Hydra aims at delegating general computing to any programmable peripherals. While seemingly more sophisticated, Hydra indeed will not address the technical challenges of delegating sensor data acquisition. First of all, Hydra provides no support for programmable sensors to deal with multiple periodic tasks (channels). It uses a peripheral device more as an accelerator that executes a single task. More importantly, Hydra's ambitious goal to delegate general computing also leads to a complicated and expensive design that is inadequate for sensor data processing on mobile systems. While not much information about Hydra's implementation and hardware requirement was reported [47], all the peripheral devices employed in the experiments, i.e. NIC, disk controller, and graphics adapter, are based on 32-bit embedded processors, much more resourceful than the 16-bit microcontroller targeted by dReflex.

Nightingale *et al.* [48] presented Helios, an operating system intended for heterogeneous platforms. With a similar goal as Hydra, Helios integrates programmable peripherals much more tightly and therefore further simplifies their programming. However, similar to Hydra, Helios aims to provide unified runtime environments on all processors, therefore it requires peripheral processors have at least 32MB RAM and a few hundred MHz, much higher than that a sensor of a mobile system can afford. In contrast, dReflex, targeting at smartphone sensing tasks, requires processors with a few KB RAM and a clock speed of a few MHz.

Researchers have integrated heterogeneous computational resource into single sensor network node to improve its energy-efficiency[39, 49, 50]. Although they leverage the same principle to save system energy as Reflex does, the well-defined tasks in sensor networks enable programmers to statically assign them to different computational resource at design time, which is impossible for the smartphone platform. Many tools [51, 52] have also enabled modularized design of sensor network motes . However, their focus is code reuse, while Reflex focuses on the reuse of code *execution*.

The final group of related work shares the same goal of dReflex: maximize the opportunities for a host to sleep with a low-power cohort. Its solutions include Wake-on-Wireless [53], Turducken [54], and Somniloquy [55]. All three employ a low-power cohort to represent a host system on the network and perform tasks on behalf of the host. They enable a power-managed host to be alert of the *network*; in contrast, dReflex enables a power-managed host to be alert of the *physical world*. Only Turducken may program a sensor node to process sensor data for the host. However, all three solutions consider the cohort as an autonomous and independently programmed system. While this simplifies the cohort system design by leveraging existing systems such as

TinyOS sensor nodes and ARM-based embedded systems, it leads to much more complicated and expensive cohort systems than the programmable sensors of dReflex. More importantly, it requires developers to develop the code for the cohort system separately, usually implementing the same service on both the host and the cohort independently. In contrast, developers of dReflex-based sensory programs do not even need to know the sensor hardware, as described in Section 6.

In summary, dReflex realizes decentralized sensor data processing with a more efficient and more developer-friendly design.

10. Conclusions

We presented Reflex, a programming and system framework that separates the acquisition and use of sensor data processing in both programming and system operation. Reflex provides three mechanisms to improve the efficiency of sensory applications. First, it executes sensor data processing by multiple applications within a single runtime and therefore obviates process context switches with high frequency. Second, it allows multiple applications to share sensor data processing and therefore minimizes redundancy in sensor data movement and processing in the system. Finally, it enables a system to delegate high-duty cycle sensor data processing to programmable resources in the corresponding sensor.

We described a prototype of Reflex and its decentralized realization, dReflex, with the Nokia N810. Our measurements clearly showed the efficiency advantage provided by the three Reflex mechanisms mentioned above. Power reduction of up to 48%, 55%, and 90% can be achieved with them, respectively. An informal study with programmers also provided encouraging feedbacks regarding the learning curve and usability of Reflex. On the other hand, our evaluation also indicated that Reflex is certainly not intended for all sensory applications due to its overhead in handling events and its programming restrictions.

Acknowledgement

References

[1]   J. Wang, S. Zhai, and J. Canny, "Camera phone based motion sensing: interaction techniques, applications and performance study," in *Proc. ACM Symp. User interface software and technology (UIST)* Montreux, Switzerland, 2006.

[2]   S. Consolvo, P. Klasnja, D. W. McDonald, D. Avrahami, J. Froehlich, L. LeGrand, R. Libby, K. Mosher, and J. A. Landay, "Flowers or a robot army?: encouraging awareness & activity with personal, mobile displays," in *Proc. ACM Int.*

[3]   I. Constandache, S. Gaonkar, M. Sayler, R. R. Choudhury, and L. Cox, "EnLoc: energy efficient localization for mobile phones " in *Proc. IEEE Conf. Computer Communication (INFOCOM) Mini* Rio de Janeiro, Brazil, 2009.

[4]   A. Krause, M. Ihmig, E. Rankin, D. Leong, S. Gupta, D. Siewiorek, A. Smailagic, M. Deisher, and U. Sengupta, "Trading off prediction accuracy and power consumption for context-aware wearable computing," in *Proc. Int. Symp. Wearable Computers (ISWC)*, M. Ihmig, Ed., 2005, pp. 20-26.

[5]   S. Kang, J. Lee, H. Jang, H. Lee, Y. Lee, S. Park, T. Park, and J. Song, "SeeMon: scalable and energy-efficient context monitoring framework for sensor-rich mobile environments," in *Proc. ACM/USENIX Int. Conf. Mobile Systems, Applications, and Services (MobiSys)* Breckenridge, CO, 2008.

[6]   Y. Wang, J. Lin, M. Annavaram, Q. A. Jacobson, J. Hong, B. Krishnamachari, and N. Sadeh, "A framework of energy efficient mobile sensing for automatic user state recognition," in *Proc. ACM/USENIX Int. Conf. Mobile Systems, Applications, and Services (MobiSys)* Krakow, Poland, 2009, pp. 179-192

[7]   E. I. Shih, A. H. Shoeb, and J. V. Guttag, "Sensor selection for energy-efficient ambulatory medical monitoring," in *Proc. ACM/USENIX Int. Conf. Mobile Systems, Applications, and Services (MobiSys)* Krakow, Poland, 2009, pp. 347-358

[8]   K. Murao, T. Terada, Y. Takegawa, and S. Nishio, "A context-aware system that changes sensor combinations considering energy consumption," in *Proc. Int. Conf. Pervasive Computing (PERVASIVE)*, 2008, pp. 197-212.

[9]   F. R. Bentley and C. J. Metcalf, "Sharing motion information with close family and friends," in *Proc. ACM SIGCHI Conf. Human Factors in Computing Systems (CHI)* San Jose, CA, 2007, pp. 1361-1370.

[10] H. Lu, W. Pan, N. D. Lane, T. Choudhury, and A. T. Campbell, "SoundSense: scalable sound sensing for people-centric applications on mobile phones," in *Proc. ACM/USENIX Int. Conf. Mobile Systems, Applications, and Services (MobiSys)* Krakow, Poland, 2009.

[11] A. Rahmati, C. Shepard, and L. Zhong, "NoShake: content stabilization for shaking screens of mobile devices," in *Proc. IEEE Int. Conf. Pervasive Computing and Communications (PerCom)* Galveston, TX, 2009, pp. 1-6.

[12] H. Dumanli, A. Amiri Sani, L. Zhong, and A. Sabharwal, "BeamSwitch: system solution for energy-efficient directional communication on mobile devices," *Technical Report 0616-09, Rice University,,* 2009.

[13] X. Chen, Z. Zhao, A. Rahmati, Y. Wang, and L. Zhong, "SaVE: sensor-assisted motion estimation for efficient H.264/AVC video encoding," in *Proc. ACM Int. Conf. Multimedia* Beijing, China, 2009.

[14] Super Monkey Ball, *http://www.sega.com/games/super-monkey-ball/.*

[15] D. Stancevic, "Zero copy I: user-mode perspective," *Linux Journal,* vol. 2003, p. 3, 2003.

[16] G. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "CCured: Type-safe retrofitting of legacy software," *ACM Transactions on Programming Languages and Systems (TOPLAS),* vol. 27, p. 526, 2005.

[17] J. Morrisett, T. Jim, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C," in *USENIX Annual Technical Conference*, 2002.

Conf. Ubiquitous Computing (UbiComp) Seoul, Korea, 2008, pp. 54-63.

[18] R. Kumar, E. Kohler, and M. Srivastava, "Harbor: software-based memory protection for sensor nodes," in *Proceedings of the 6th international conference on Information processing in sensor networks* Cambridge, Massachusetts, USA: ACM, 2007.

[19] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem: overview of methods and survey of tools," *ACM Trans. Embedded Computing Systems,* vol. 7, pp. 1-53, 2008.

[20] K. Chatterjee, D. Ma, R. Majumdar, T. Zhao, T. A. Henzinger, and J. Palsberg, "Stack size analysis for interrupt-driven programs," *Information and Computation,* vol. 194, pp. 144-174, 2004.

[21] D. Bovet and M. Cesati, *Understanding the Linux kernel*, 3rd ed.: O'Reilly Media, Inc., 2005.

[22] OKI Semiconductor, *ML8953A: MEMS 3-axis accelerometer*: http://www.okisemi.eu/Products/Sensors/mems_accelerometer.html.

[23] STMicroelectronics, *MEMS motion sensor 3-axis - ± 2g/± 8g smart digital output piccolo accelerometer*: http://www.st.com/stonline/products/literature/ds/12726/lis302dl.htm.

[24] A. R. Moller, *Sensory systems: anatomy and physiology*: Academic Press, 2003.

[25] RECG SVN Repository for Reflex, http://svn.rice.edu/r/dreflex.

[26] Rice Open-Source Orbit Platform, http://www.ruf.rice.edu/~mobile/orbit.htm.

[27] CMUcam3, "Open Source Programmable Embedded Color Vision Platform," http://www.cmucam.org/.

[28] J. Liu, Z. Wang, L. Zhong, J. Wickramasuriya, and V. Vasudevan, "uWave: accelerometer-based personalized gesture recognition and its applications," in *Proc. IEEE Int. Conf. Pervasive Computing and Communications (PerCom)* Galveston, TX, 2009, pp. 1-9.

[29] P. Mohan, V. N. Padmanabhan, and R. Ramjee, "Nericell: rich monitoring of road and traffic conditions using mobile smartphones," in *Proc. ACM Conf. Embedded Networked Sensor Systems (SenSys)* Raleigh, NC, 2008, pp. 323-336

[30] M. Mun, S. Reddy, K. Shilton, N. Yau, J. Burke, D. Estrin, M. Hansen, E. Howard, R. West, and P. Boda, "PEIR: the personal environmental impact report, as a platform for participatory sensing systems research," in *Proc. ACM/USENIX Int. Conf. Mobile Systems, Applications, and Services (MobiSys)* Krakow, Poland, 2009.

[31] E. Miluzzo, N. D. Lane, K. Fodor, R. Peterson, H. Lu, M. Musolesi, S. B. Eisenman, X. Zheng, and A. T. Campbell, "Sensing meets mobile social networks: the design, implementation and evaluation of the CenceMe application," in *Proc. ACM Conf. Embedded Networked Sensor Systems (SenSys)* Raleigh, NC, 2008.

[32] A. Rahmati and L. Zhong, "Context-for-Wireless: context-sensitive energy-efficient wireless data transfer," in *Proc. ACM/USENIX Int. Conf. Mobile Systems, Applications, and Services (MobiSys)* San Juan, Puerto Rico, 2007, pp. 165-178.

[33] J. E. Bardram, "The Java Context Awareness Framework (JCAF) – a service infrastructure and programming framework for context-aware applications," in *Proc. Int. Conf. Pervasive Computing (PERVASIVE)*, 2005, pp. 98-115.

[34] O. Riva, "Contory: a middleware for the provisioning of context information on smart phones," in *Proc. ACM/IFIP/USENIX Int. Middleware Conf. (Middleware)*, 2006, pp. 219-239.

[35] M. Addlesee, R. Curwen, S. Hodges, J. Newman, P. Steggles, A. Ward, and A. Hopper, "Implementing a sentient computing system," *IEEE Computer,* vol. 34, pp. 50-56, 2001.

[36] J. Froehlich, M. Y. Chen, S. Consolvo, B. Harrison, and J. A. Landay, "MyExperience: a system for *in situ* tracing and capturing of user feedback on mobile phones," in *Proc. ACM/USENIX Int. Conf. Mobile Systems, Applications and Services* San Juan, Puerto Rico, 2007, pp. 57 - 70

[37] D. Prabal, M. Grimmer, A. Arora, S. Bibyk, and D. Culler, "Design of a wireless sensor network platform for detecting rare, random, and ephemeral events," in *Proc. ACM Int. Symp. Information Processing in Sensor Networks (IPSN)*, 2005, pp. 497-502.

[38] S. Jevtic, M. Kotowsky, R. P. Dick, P. A. Dinda, and C. Dowding, "Lucid dreaming: reliable analog event detection for energy-constrained applications," in *Proc. ACM Int. Conf. Information Processing in Sensor Networks (IPSN)* Cambridge, MA, 2007.

[39] M. Malinowski, M. Moskwa, M. Feldmeier, M. Laibowitz, and J. A. Paradiso, "CargoNet: a low-cost micropower sensor node exploiting quasi-passive wakeup for adaptive asynchronous monitoring of exceptional events," in *Proc. ACM Int. Conf. Embedded networked Sensor Systems (SenSys)* Sydney, Australia, 2007.

[40] L. Gu and J. A. Stankovic, "Radio-triggered wake-up for wireless sensor networks," *Real-Time Systems,* vol. 29, pp. 157-182, 2005.

[41] R. Pon, M. A. Batalin, J. Gordon, A. Kansal, D. Liu, M. Rahimi, L. Shirachi, Y. Yu, M. Hansen, J. K. William, M. Srivastava, G. Sukhatme, and D. Estrin, "Networked infomechanical systems: a mobile embedded networked sensor platform," in *Proc. ACM Int. Conf. Information Processing in Sensor Networks (IPSN)* Los Angeles, CA, 2005.

[42] D. McIntire, K. Ho, B. Yip, A. Singh, W. Wu, and W. J. Kaiser, "The low power energy aware processing (LEAP) embedded networked sensor system," in *Proc. ACM Int. Conf. Information Processing in Sensor Networks (IPSN)*, K. Ho, Ed., 2006, pp. 449-457.

[43] D. Lymberopoulos, N. B. Priyantha, and F. Zhao, "mPlatform: a reconfigurable architecture and efficient data sharing mechanism for modular sensor nodes," in *Proc. ACM Int. Conf. Information Processing in Sensor Networks (IPSN)* Cambridge, MA, 2007, pp. 128 - 137.

[44] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle, "Active disks for large-scale data processing," *Computer,* vol. 34, pp. 68-74, 2001.

[45] D. L. Tennenhouse and D. J. Wetherall, "Towards an active network architecture," *ACM SIGCOMM Computing Communication Review,* vol. 26, pp. 5-17, 1996.

[46] J. C. Mogul, "TCP offload is a dumb idea whose time has come," in *Proc. Conf. on Hot Topics in Operating Systems* Lihue, Hawaii: USENIX Association, 2003.

[47] Y. Weinsberg, D. Dolev, T. Anker, M. Ben-Yehuda, and P. Wyckoff, "Tapping into the fountain of CPUs: on operating system support for programmable devices," in *Proc. ACM Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)* Seattle, WA, USA, 2008.

[48] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt, "Helios: heterogeneous multiprocessing with satellite kernels," in *Proc. ACM SIGOPS Symp. Operating Systems Principles (SOSP)* Big Sky, Montana, USA: ACM, 2009.

[49] B. Schott, M. Bajura, J. Czarnaski, J. Flidr, T. Tho, and L. Wang, "A modular power-aware microsensor with >1000X dynamic power range," in *Proceedings of the 4th international symposium on Information processing in sensor networks* Los Angeles, California: IEEE Press, 2005.

[50] C. Han, M. Goraczko, J. Helander, J. Liu, B. Priyantha, F. Zhao, and N. Computing, "Comos: An operating system for heterogeneous multi-processor sensor devices," MSR-TR-2006-117, 2006.

[51] B. Greenstein, E. Kohler, and D. Estrin, "A sensor network application construction kit (SNACK)," in *Proceedings of the 2nd international conference on Embedded networked sensor systems* Baltimore, MD, USA: ACM, 2004.

[52] L. Girod, N. Ramanathan, J. Elson, T. Stathopoulos, M. Lukac, and D. Estrin, "Emstar: A software environment for developing and deploying heterogeneous sensor-actuator networks," *ACM Trans. Sen. Netw.,* vol. 3, p. 13, 2007.

[53] E. Shih, P. Bahl, and M. J. Sinclair, "Wake on Wireless: an event driven energy saving strategy for battery operated devices," in *Proc. Int. Conf. Mobile Computing and Networking (MobiCom)* Atlanta, Georgia, USA, 2002, pp. 160-171.

[54] J. Sorber, N. Banerjee, M. D. Corner, and S. Rollins, "Turducken: hierarchical power management for mobile devices," in *Proc. ACM/USENIX Int. Conf. Mobile Systems, Applications, and Services (MobiSys)* Seattle, WA, 2005, pp. 261-274.

[55] Y. Agarwal, S. Hodges, R. Chandra, J. Scott, P. Bahl, and R. Gupta, "Somniloquy: augmenting network interfaces to reduce PC energy usage," in *Proc. USENIX Symp. Networked Systems Design and Implementation (NSDI)* Boston, MA, 2009.