SymPhoney: A Coordinated Sensing Flow Execution Engine for Concurrent Mobile Sensing Applications

Younghyun Ju, Youngki Lee, Jihyun Yu, Chulhong Min, Insik Shin†, Junehwa Song

Computer Science Department, KAIST, Daejeon, Republic of Korea

{yhju, youngki, jihyun, chulhong, junesong}@nclab.kaist.ac.kr, †insik.shin@cs.kaist.ac.kr

Abstract

Emerging *mobile sensing applications* are changing the characteristics of smartphone workloads. Whereas typical mobile applications run alone in the foreground interacting with users, sensing applications concurrently run in the background, providing unobtrusive monitoring services. Such concurrent sensing workloads raise a new challenge incurring severe resource contention among themselves and with other foreground applications. To address the challenge, we develop SymPhoney, a coordinated sensing flow execution engine to support concurrent sensing applications. As its key approach, we develop a novel *sensing-flow-aware coordination*. We first introduce the new concept of *frame externalization* i.e., to identify and externalize semantic structures embedded in otherwise flat sensing data streams. Leveraging the identified frame structures, SymPhoney develops *frame-based coordination* and *scheduling* mechanisms, which effectively coordinates the resource use of concurrent contending applications and maximize their utilities even under severe resource contention. We implemented several sensing applications on top of the SymPhoney engine and performed extensive experiments, showing effective coordination capability of SymPhoney.

Categories and Subject Descriptors

C.3 [Special-Purpose and Application-based Systems]: Real-time and embedded systems

Keywords

Concurrency, Coordination, Scheduling, Resource, Sensing flow, Allocation, Mobile Sensing, Dataflow, Smartphone

1. Introduction

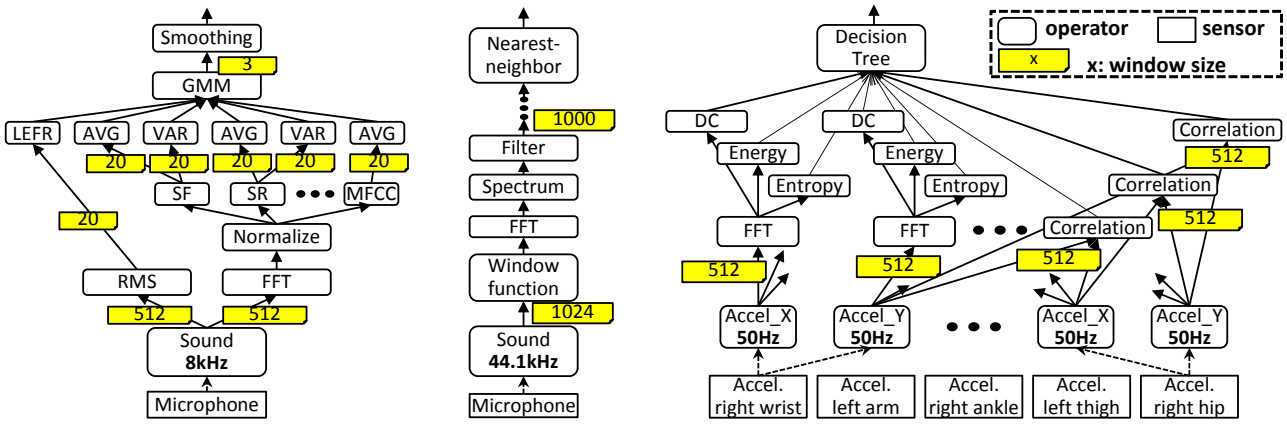
Emerging *continuous mobile sensing applications* [1][2][3] will significantly change workload patterns imposed on smartphones. Going beyond the confines of typical user-interactive mobile applications such as web browsers and games, they continuously run in the background and provide autonomous, situation-aware services without a user's intervention. This user-unobtrusive nature enables a smartphone to serve multiple sensing applications at the same time in spite of its small display and user mobility. As diverse, useful sensing applications are emerging, a smartphone will concurrently serve more of them, accompanying a conventional foreground application.

Such concurrent workloads will raise an unprecedented challenge, incurring severe resource contention on resource-scarce smartphones. The contention is aggravated due to the continuous and heavy CPU consumption of individual sensing applications to process high-rate sensor data; in our study, example applications consume 4%~22% of CPU cycles to run multi-step operations of sensing, feature extraction and classification (See Section 2.2). More important, such sensing and processing workloads should be handled near real-time to provide timely services. Even worse, the total resource availability might be limited further, deteriorating the contention; users won't exhaust the whole CPU cycles and battery only for background applications. Under such contentious situation, greedy resource use by an application may result in serious degradation of service qualities of the other applications, e.g., dragged interval, abrupt delays, and inaccurate context results. It could also degrade the performance of other daily use of smartphones, for instance, increasing loading time of web documents.

To address the challenge, we develop SymPhoney, a novel sensing flow execution engine for concurrent mobile sensing applications. It coordinates contentious concurrent workloads on the whole, and effectively resolves potential imbalances in the service qualities of applications. Developers easily build sensing applications without concerning severe resource contention and the dynamics caused by concurrent applications. Then, the engine coordinates the resource use of contending applications while maximizing their utilities under given resource conditions. Moreover, it dynamically adapts to the fluctuating resource availability from foreground applications, and minimizes the performance degradation of the interactive applications.

A simple approach to handle concurrent workloads is to make each sensing application as a process or a thread, delegating the coordination job to the mobile OS. This could significantly compromise efficiency and fairness and incurs unexpected starvation or delays. Since the OS deals with the processes without application-level information, it

SenSys'12, November 6-9, 2012, Toronto, Canada.



(a) ChildMon **(b) IndoorNavi** **(c) CalorieMon**

Figure 1. Sensing flows of motivating applications

is hard to identify and allocate right amount of resources for the applications. An alternative is that each application adapts their resource use to counter contentious resource situations [17]. However, proper adaptation is difficult in an application level, as applications have limited view on the other applications' resource use, and hardly negotiate with the others. Moreover, it would be a burden for developers to implement effective adaptation methods to prepare for diverse resource situations.

We develop the *flow-aware coordination* approach, which enables a system to exploit the internal structure and resource use patterns of sensing applications for effective resource coordination. We first introduce the novel concept of *frame externalization*, i.e., to identify and externalize semantic structures embedded in otherwise flat sensing data streams (See Figure 10); thus extracted knowledge on framing structure provides valuable hints to address various challenges of complex system design for sensing applications. For flow coordination, we specifically focus on two common types of frames, namely, *context-frame* (*c-frame*), a sequence of sensing data to produce a context result and *feature-frame* (*f-frame*), a sequence to execute the first-staged feature extraction operations; based on the frames, we design the new methods of *c-frame-based flow coordination* and *f-frame-based flow execution*.

The *c-frame-based flow coordination* leverages the c-frame as the basic unit of resource allocation and flow coordination. The use of each tiny portion of the allocated resources effectively contributes to the delivery of semantically meaningful results, and in turn, influences the quality of service as perceived by a user. As such, it enables a system to best coordinate the concurrent flows contending for limited system resources, maximizing the application utilities. The *f-frame-based flow execution* pipelines the complicated steps of sensing and processing in the unit of the f-frame, considerably reducing potentially lengthy delay and enhancing the schedulability of concurrent applications.

SymPhoney is a mobile sensing engine, clearly distinguishing itself from recently proposed mobile sensing systems [4][5][6][7][25]. The most important feature is the coordination among concurrent sensing applications and with typical smartphone applications. Jigsaw [5] and Kobe [7] provide optimization and adaptation techniques in the viewpoint of a single sensing application. SeeMon [4] and Orchestrator [6] deal with concurrent sensing applications and provide useful solutions. SeeMon aims at achieving processing and energy efficiency for multiple applications, while SymPhoney coordinates their resource usage. Orchestrator also supports the coordination among concurrent applications. SymPhoney delves into the resource coordination on a smartphone.

The contribution of this paper is summarized as follows. First, we propose SymPhoney, a sensing flow execution engine for emerging mobile sensing workloads, featuring the coordination of concurrent applications. Second, we introduce the concept of frame externalization and based on the concept, we devise effective c-frame-based flow coordination and f-frame-based flow execution mechanisms. Third, we support sensing flow design process based on a dataflow programming model, and implement several interesting sensing applications, ChildMon, IndoorNavi, and CalorieMon, inspired by recent works [1][2][3]. Finally, we report extensive experimental results on the coordination capability of the engine using the applications.

The rest of this paper is organized as follows. In Section 2, we present three example applications and motivating experiments. Section 3 shows the programming abstraction of SymPhoney and Section 4 explains our key approach. In Section 5, we describe the system design and the coordination mechanisms in detail. Then, we present our implementation in Section 6 and Section 7 shows experimental results. We introduce related work in Section 8 and finally conclude the paper in Section 9.

2. Mobile Sensing Workloads

2.1 Example Applications

We implement three sensing applications on SymPhoney to motivate and evaluate its effectiveness: (1) ChildMon,

(2) IndoorNavi, and (3) CalorieMon. The core sensing flows of the applications are presented in Figure 1.

ChildMon allows working parents to be aware of real-time activities of their kindergarten child during classes and fieldtrips. It is inspired by our previous works to support kindergarten education with sensing technology [14][15]. It monitors children's activities like talking and playing using a backpack-attached phone, and notifies their parents of the distinguished activities. For the activity detection, we adopt and modify the logic of SoundSense [1]. Figure 1 (a) shows the core sensing flow of the application.

IndoorNavi helps a mobile user to navigate large-scale building complexes. For the navigation, the application continuously localizes itself in the level of an office or a room. The core sensing flow for the indoor localization is inspired by BatPhone [2] and specified in Figure 1 (b).

CalorieMon monitors a user's physical activities during daily exercise and estimates real-time caloric expenditure of the user. Its sensing flow is shown in Figure 1 (c), which is adopted from [3]. This application utilizes five bi-axis accelerometers placed in different body limbs.

2.2 Limitations of Current Mobile OS

Current mobile OSs such as Android and iOS hardly handle resource contentions by concurrent sensing applications, which motivates us to design a new engine. To show the limitation of current OSs, we conduct several experiments on Android Nexus One phones.

Experimental settings: we execute the three sensing applications; each runs as an Android *background service*. In addition, for foreground workloads, we use two common applications, a Web browser and a video player [10]. The Web browser represents user-interactive applications that generate intermittent and dynamic workload. We implement a simple benchmark that loads a desktop version of Amazon.com page every 20 seconds. The video player represents the applications that consistently impose heavy workloads. We play a 720p video file encoded in Xvid.

Results and observations: we first investigate the CPU utilization of the applications over time. Figure 2 shows the results. IndoorNavi alone consumes over 20% of CPU cycles to continuously execute heavy localization logics over high-rate sensing data, 44.1 kHz. Such continuous and significant CPU use could incur severe resource contention with other applications. An interesting observation is that all the applications show periodic patterns in their CPU use. This is because sensing applications usually repeat a series of operations over a sequence of sensing data to extract a user context. In the perspective of background operations, we can find a similarity with Android background services such as *netd* and *vold* which handle occasional I/O events and interrupts. However, they utilize CPU less than 1%.

To identify the resource contention with foreground mobile workloads, we measure the CPU utilizations when the sensing applications run together with the video player.

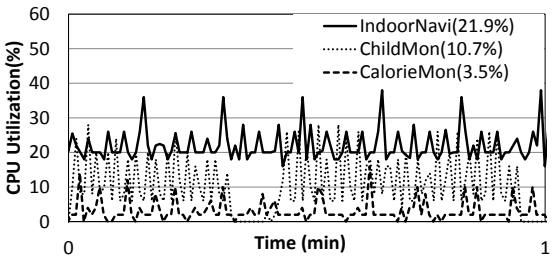


Figure 2. CPU utilization of sensing applications

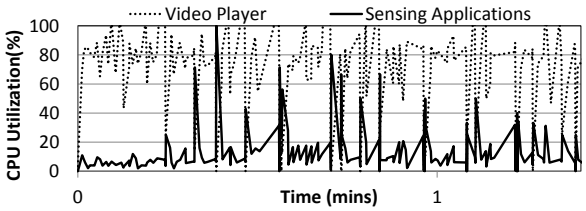


Figure 3. CPU utilization of sensing applications with a video player

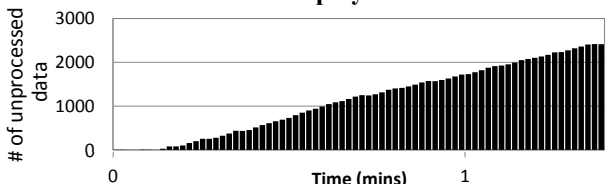


Figure 4. Unprocessed data of sensing applications with a video player

As shown in Figure 3, due to the consistent CPU use of the video player, the sensing applications are allocated under 10% of CPU resource. This is much less than their resource demand, i.e., about 35% of CPU resource in total. For better understanding, we also measure the total number of unprocessed sensor data over time. As shown in Figure 4, the resource shortage makes the applications pile up the streaming sensor data, which finally results in memory overflow and the potential termination of the applications.

Figure 5 shows the results when the sensing applications run together with the Web browser. In this case, the CPU utilization of the sensing applications dynamically changes according to the behavior of the browser. More specifically, the allocated CPU resource is significantly reduced while the browser is loading web pages. Such reduction degrades the performance of the sensing applications, especially in terms of the delay to produce context results. In worst case, the result delivery is delayed by about 7 seconds. The sacrifice of the sensing applications occurs because today's mobile OSs assign a much higher priority to an activated foreground application compared to other background ones.

A naïve solution might simply increase the priority of the sensing applications. Figure 7 shows the results when their priority is set to the same priority with the Web browser. The sensing applications are allocated with more sufficient CPU resource and work stably. However, the Web browser slows down and the average time to load pages increases from 4.7 to 6.7 seconds.

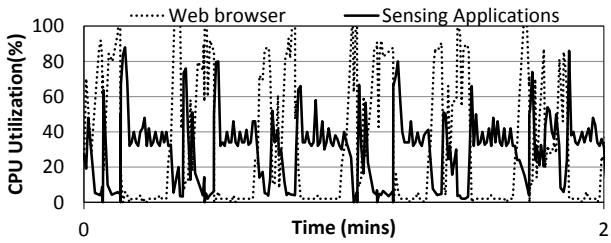


Figure 5. CPU utilization of sensing applications with a Web browser

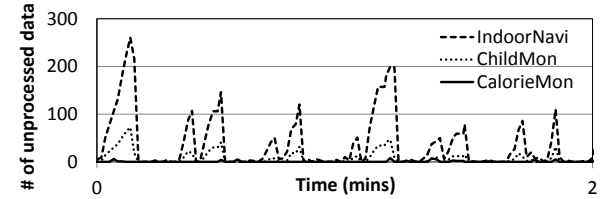


Figure 6. Unprocessed data of sensing applications with a Web browser

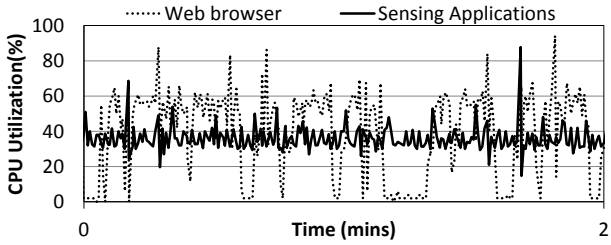


Figure 7. CPU utilization in the case of the same priority

To examine the resource distribution within the sensing applications, we measure the number of unprocessed data in each sensing application, in the same setting as in Figure 5. As shown in Figure 6, the unprocessed data in IndoorNavi increases more rapidly than others under contention. This is because it requires more CPU resource than the others while Android equally distributes the CPU resource to the applications under contentious situations. Such simple distribution regardless of the differences in resource demand could result in significant imbalance of service quality among applications. Moreover, a user often has her own preference for different sensing applications and their resource use. For instance, even under resource contention, parents would not want to compromise the quality of ChildMon, whereas teenagers at a commercial complex need higher responsiveness of IndoorNavi. Current OSs do not consider such diverse preferences and different resource demands of mobile sensing applications, yet.

3. Programming Sensing Flows

Without programming supports from a system like SymPhoney, it involves multi-lateral challenges to develop practical mobile sensing applications from scratch [7]. A primitive challenge is to design sensing flows that capture the contexts of interest precisely. For example, a *ChildMon* developer should carefully select a meaningful feature set among plenty of sound-extracted features, and an effective classification algorithm. The developer further requires

Table 1. Example operators provided by SymPhoney

Operator types	SymPhoney built-in operators
Sensing operators	Sound, Accel., Gyro., GPS, …
Feature extractors	FFT, MFCC, RMS, Correlation, Energy, Average, Entropy, …
Classifiers	GMM, HMM, kNN, Decision tree, …

significant time and efforts, going through learning and testing iteratively, to find optimal parameters. More challenging, such designed sensing flows should be crafted further, considering scarce resources of smartphones. Developers need to estimate reasonable range of resource availability in advance, and undergo serious optimization process. Also, they may need to adopt complicated adaptation mechanism to prepare for dynamic changes in resource availability.

SymPhoney supports both the design and the optimization process of sensing flows in such complex programming process. Many different models can be adopted for the flow design stage. The mechanisms and approaches for sensing flow coordination proposed in this paper are generally applied to many models. Currently, it provides a dataflow programming model in an XML interface. It helps developers flexibly compose customized sensing flows and hence, enables rapid prototyping of complex sensing flows, significantly reducing the time and effort for design iteration. The dataflow model is well-suited to represent sensing flows, usually composed of a series of pipelined computations over sensing data. SymPhoney provides a set of widely-used operators so that developers can readily use them. Table 1 lists some operators provided.

More importantly, SymPhoney takes charge of resource-aware operation of specified sensing flows on behalf of developers. Developers just need to register application-level requirements like a desired monitoring interval and a tolerable delay. SymPhoney then automatically adapts the execution of the flows to the runtime resource availability and concurrent applications' resource use while meeting their execution requirements. Furthermore, it efficiently executes the flows with minimal execution overhead.

Specifically, developers implement their applications by using the following API.

registerMonitoringRequest (*flow*, *requirements*)

In the parameter, the *flow* represents a sensing flow such as a dataflow graph used for monitoring (See the examples in Figure 1). A dataflow graph is specified as an XML document, as shown in Figure 8. It consists of operators and edges connecting the operators. Each operator in the graph represents a unit of computation or I/O. The edges of the graph represent data dependencies between operators. An edge includes the information about how many outputs of the previous operator is required as an input of the next operator.

```
<Operator ID="0" Type="SensingSound" Parameter="8000"><..
<Operator ID="1" Type="RMS" Parameter=""></Operator>
<Operator ID="2" Type="FFT" Parameter="512"></Operator>
…
<Edge From="0" To="1"  WinSize="512"></Edge>
<Edge From="0" To="2" WinSize="512"></Edge>
…
```

Figure 8. Example XML specification of ChildMon

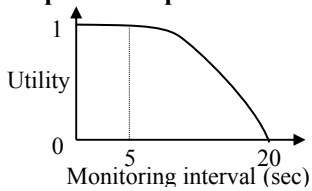


Figure 9. Example utility function of CalorieMon

As for the *requirements* parameter, SymPhoney supports the monitoring interval and delay.

Monitoring interval represents how often the application needs to monitor the user's situation. For example, CalorieMon may require capturing the user's physical activity every several seconds to compute the total calorie expenditure of a day. Typically, the shorter the interval is, the higher the utility of the application is. Applications specify the preferred monitoring interval tied with utility values for each interval, which is expressed as a utility function. Figure 9 shows an example utility function for CalorieMon. SymPhoney attempts to maximize the utility under changing resource conditions.

Monitoring delay is specified in terms of a maximum tolerable delay of the application, where the delay means the time to deliver final context results to the application from the moment of sensing. The freshness of the results is important for some applications to provide timely and responsive services. For example, *CalorieMon* requires less than two seconds of delay to provide a runner with real-time calorie expenditure. The engine aims to meet the delay requirement of the applications as much as possible.

Another important requirement is the accuracy of monitoring results. We suppose that it is developers' role to design a sophisticated sensing flow that can meet a desired accuracy requirement. SymPhoney prevents the accuracy from being undesirably compromised at runtime due to the abrupt drop of sensing data or temporary fluctuation in sensing frequency.

4. Flow-aware Coordination of Sensing Applications

SymPhoney plays a key role to coordinate the resource use of mobile sensing applications: (1) among themselves, and (2) with other foreground applications. A main goal of the coordination is to maximize the utility of sensing applications even under high CPU contention, and prevent skewed resource use either by sensing or other foreground applications. For effective resource coordination of sensing applications, we develop the new *flow-aware coordination* approach. A core of the approach is structuring continuous sensing streams into meaningful units by looking into regular processing structure of the sensing applications.

4.1 Frame Externalization of Sensing Data Streams

A sensing data steam is generated as a flat sequence of data. However, taken by an application, it is interpreted with a tailored structure. That is, data samples at different positions may give distinct meanings to, and hence, differently used by the application. For example, three consecutive samples may belong to separate semantic units, one used to generate a result and the following two used for the next result. Thus, a flat sensing stream can be considered to have a virtual structure, given by an application.

While the structure of a sensing stream is implicit within an application, identifying and externalizing the structure provide great potential for system design. Knowledge on the structure gives useful hints in executing an application and planning its resource use, especially when a system should deal with multiple concurrent applications under contentious situations.

Consider an application A takes a sensing stream S as input. The virtual structure of S can be extracted by inspecting the flow of A. We call the process of externalizing the virtual structure of S with respect to A as *framing* or *frame externalization* of S with A. Then, we call the resulting structural entity of S as a *frame*. A stream is framed differently by different applications. Also, one can be framed differently even for the same application. Figure 10 shows that a sound stream of 8 kHz is framed by the ChildMon application in three layers. In the first layer, a sequence of 512 consecutive samples is framed for feature extraction, and in the second, 20 of such frames are combined for classification. Lastly, in the third layer, three of the second-layer frames are aggregated for post-processing, constructing a frame of 512×20×3 which is the unit to generate a final recognition result. The issues of structuring sensing data streams are distinct from those in well-structured media stream processing. In the latter, input streams such as structured video and audio have explicit and usually standardized frame structures while in the former, those should be identified and externalized with respect to individual applications.

Focusing on the purpose of coordination and execution of contending sensing applications, we identify two structural layers, i.e., feature extraction and result generation layers (See Figure 10) that are common to many sensing applications. We then utilize the two corresponding frames: (1) *context-frame* (*c-frame*), a sequence of sensing data to produce a context result and (2) *feature-frame* (*f-frame*), a subsequence of a c-frame to execute the first-staged feature extraction operations. Based on such structuring, we develop mechanisms for flow coordination and execution.

Framing sensing data streams requires understanding on the internal processing structures of applications. Diverse analysis methods can be designed to inspect different representations of applications. For a dataflow or a directed graph, the c-frame and f-frame are identified by a quick depth-first traversal of its graph from the initial sensing operators through the subsequent processing operators. The concept of externalizing the frame structure of a sensing data stream can be generally applied to handle various system design issues other than efficient flow execution and coordination.

4.2 Frame-based Coordination and Execution

c-frame-based flow coordination: the approach first takes a *c-frame* as the basic unit of data processing and resource allocation. This is based on the observation that collecting a c-frame and performing subsequent operations can hardly be ceased in the middle, in order to generate an accurate and timely context result. On the other hand, it is often tolerable to temporarily pause the whole operations once a c-frame is completely processed. Accordingly, the approach first regards 'collecting and processing a c-frame' as a unit of resource allocation. Each c-frame is allocated with the right and deserved amount of resources to collect and process it. Then, a flow is coordinated by assigning a *c-frame interval*, i.e., the interval between successive c-frames, in a way to guarantee the processing of the c-frame in each interval with the given resource availability. Figure 11 shows an example. Moreover, for multiple concurrent flows, the coordination is performed to assign different c-frame intervals for the different flows and to maximize the total utility of corresponding applications under the given resource availability. Figure 12 shows an example with two concurrent applications. In this way, the approach enables a system to best utilize its resources and maximize application utilities under a contentious situation.

The c-frame-based coordination is advantageous in several ways. First of all, the allocation made in the unit of a c-frame is directly mapped to the provision of a result meaningful to an application. For sensing applications, the service quality perceived by a user is generally affected by the frequency of result delivery. Thus, the direct mapping between a result and resource usage allows a system to easily reflect the requirements from the application under dynamic resource situations. Second, the approach allows a system to efficiently utilize its resources without wastage. As it allocates the deserved amount for each c-frame, all of the allocated resources contribute to the provision of results and accordingly to the service quality; by ensuring the deserved amount, it provides applications with timely results, and does not compromise the accuracy of results even under contentious situations. As shown in Figure 11, it also gives chances to save resources by deactivating unnecessary sensing and corresponding processing during the intervening periods between assigned c-frames.

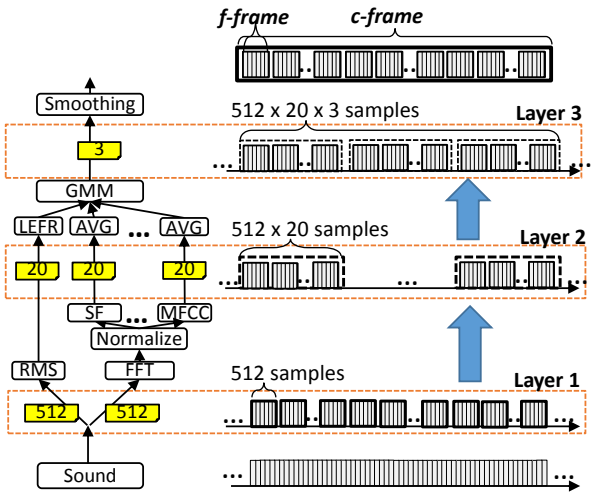


Figure 10. Framing a sound stream into c-frames and f-frames with ChildMon example: a sound stream of 8 kHz is framed in three layers. A c-frame of 512×20×3 samples is for the inference of a child's activity. An f-frame of 512 samples is for the first-stage feature extraction operators.

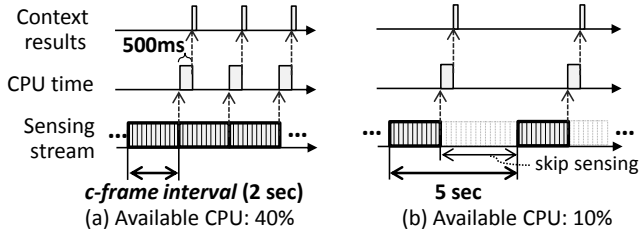


Figure 11. c-frame-based resource allocation: When CPU availability is enough, the c-frame interval is set to its minimum value, 2sec and results in 25% CPU usage; processing a c-frame takes 500ms of CPU time. However, as the available CPU is reduced to 10% (due to contention), the interval is increased to 5sec and results in 10% as in (b).

As an alternative to adjust the resource use of sensing applications, we can apply down-sampling of sensing data. As shown in Figure 13 (b), simply reducing the sampling rate of sensors, however, could result in inaccurate processing results since a sensing flow is typically designed and optimized for a certain sampling rate [7]. Even if the flow can be re-configured for different sampling rates, a reduced rate could severely compromise the accuracy of the resulting contexts [17]. Our approach can be considered as a kind of sensor duty-cycling method, *frame-aware duty cycling* (Figure 13 (d)), crafted for concurrent mobile sensing applications. It is differently designed from those used in typical traditional sensor networks [24], where a single application runs alone and manages sensing data as a flat sequence without externalizing its frame structure (Figure 13 (c)). We argue that the frame structures of the subsequent flows serve as critical information to duty-cycle a sensor, turning off the sensor without compromising data

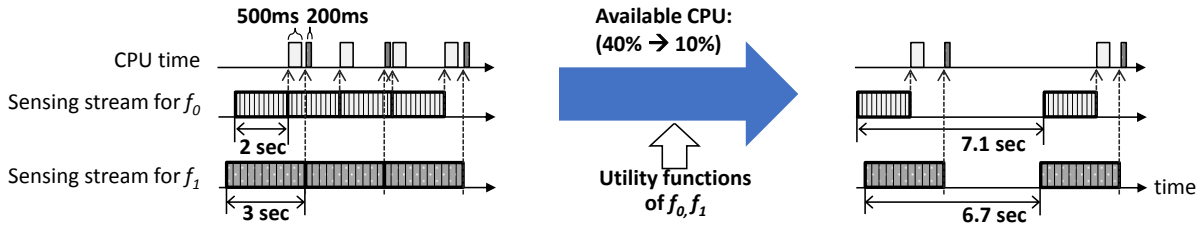


Figure 12. c-frame-based flow coordination: This shows the case that two flows (f_0 and f_1) run concurrently. As the CPU availability is dropped to 10% (right figure), the intervals are extended to 7.1 for f_0 and 6.7 seconds for f_1; they are coordinated to meet 10% availability (500ms / 7.1sec + 200ms/6.7sec = 10%) while maximizing the total utility of f_0 and f_1.

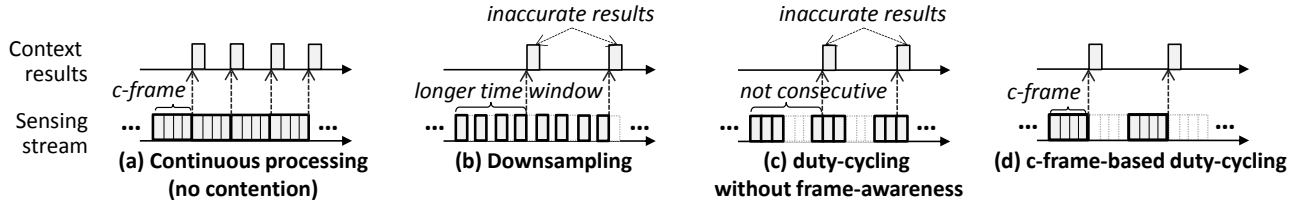


Figure 13. Comparison of alternative methods to adjust sensing applications' resource use: in the alternative methods described in (b) and (c), the c-frames of the sensing flow are compromised and accordingly, inaccurate results are generated.

integrity to create a context result. Especially, identifying the frame structure is important to coordinate the sensor use of concurrent applications in a system level.

f-frame-based flow execution: SymPhoney further adopts the *f-frame-based execution* of a flow. The idea is to leverage the characteristics of the workloads one step further, looking into the internal structure of a c-frame; a c-frame can be decomposed to a sequence of *f-frames*, *feature-frames*. With the f-frame-based execution, the actual execution of a flow is made in the unit of an f-frame while the allocation is made for each c-frame.

The f-frame-based execution provides the engine with the flexibility in the scheduling of the flow execution. For some applications, the size of a single *c-frame* would be large (e.g., about 4MB with IndoorNavi), requiring significant time for processing (about 1sec). In such a case, processing the whole c-frame in a single step after its collection may cause sudden CPU usage peaks and induce quite a long delay. SymPhoney makes it possible to reduce delays in processing and delivering a context result through a pipelined processing of a c-frame in the unit of the *f-frame* before the whole *c-frame* is ready. Also, processing in the unit of smaller-size f-frames enables the engine to flexibly schedule concurrent applications in order to meet their delay requirements under contentious situation.

Structuring sensing data streams into c-frames and f-frames also helps design the structure of the engine for the efficient execution of sensing flows. Naïve handling and processing high-rate data samples may incur considerable overhead to a system, such as frequent operator scheduling and data management. To reduce the overhead, SymPhoney further develops an efficient flow execution method exploiting the structural characteristics of the sensing data streams (See Section 5.2).

4.3 Coordination between Sensing and Foreground Applications

Running on a mobile OS, SymPhoney takes charge of the resource management for all sensing applications instead of the OS. It directly coordinates and controls the sensing applications' resource use. However, SymPhoney alone is not able to coordinate between the foreground and sensing applications. If most of the CPU time is allocated to a foreground application, the whole SymPhoney engine could starve. SymPhoney collaborates with the mobile OS to avoid such situations.

For example, when running on the Android platform, the OS allocates and guarantees the minimum CPU quotas for both foreground applications and the SymPhoney engine, e.g., 20% of total CPU cycle for the engine. While the OS guarantees the minimum quota, the actual CPU use of the foreground applications will change over time. SymPhoney identifies the amount of available CPU and dynamically re-coordinates the multiple sensing applications to maximize the application utilities under continuous CPU changes. In addition, it enforces an energy constraint, e.g., 5% of battery per hour, to prevent quick battery drain caused by continuous sensing and processing. CPU and energy quota can be specified according to users' preferences.

5. SymPhoney Design

5.1 Architecture Overview

We design the SymPhoney architecture as shown in Figure 14. The engine serves as a middleware between sensing applications and OSs. Developers specify sensing application flows with their QoS requirements on execution intervals and delay requirements. The *flow analyzer* inspects the flows and identifies their frame structures. Based on the frame structures, the *flow execution planner*

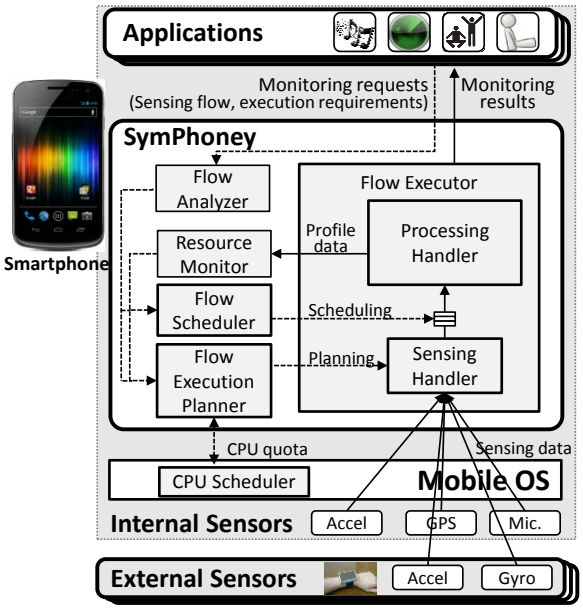


Figure 14. SymPhoney architecture

determines their c-frame intervals under changing resource conditions and accordingly allocates resources to the concurrent flows. Then, the *flow executor* executes each flow in the unit of an f-frame according to its c-frame interval and delivers the results to the applications. During the execution, the *flow scheduler* determines the order of execution of the f-frames, in a way to meet the applications' delay requirements. The *resource monitor* continuously monitors the changing resource availability and demands at runtime. The flow planner and the flow scheduler dynamically adapt the intervals and the scheduling orders.

5.2 Flow Execution

In this section, we present how each flow is executed by the flow executor, enforcing the given execution intervals and scheduling orders. For the ease of understanding, we present the flow execution before we describe the flow execution planning and the flow scheduling in Section 5.3 and 5.4. Figure 15 illustrates the design of the flow executor.

The flow executor consists of two major components, the *sensing handler* and *processing handler*. The sensing handler manages the sensing operators of the registered flows, each of which runs in a separate thread. A sensing operator takes a key role in enforcing the framing of an input sensing stream. Specifically, it repeatedly reads a c-frame from the stream at the given interval. Throttling the data early at the foremost operator enables to eliminate unnecessary sensing as well as processing in the other operators upstream. It reads the c-frame in the unit of an f-frame and passes each to the f-frame queue. This reduces the communication overhead between operators, compared to the case of delivering individual samples. For energy efficiency, the operator deactivates the corresponding

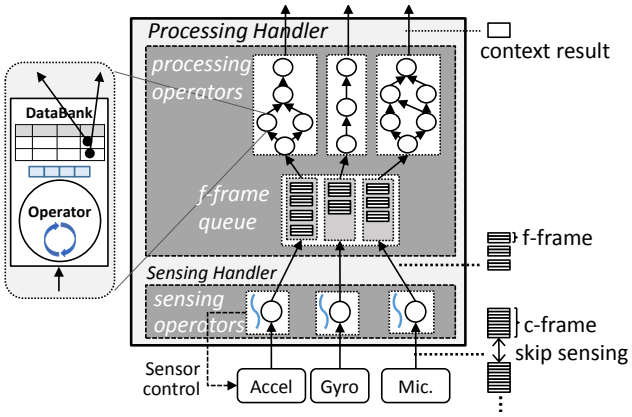


Figure 15. Flow executor

sensor module for the periods between consecutive c-frames.

The processing handler takes charge of the execution of the subsequent processing operators. When the flow scheduler picks an f-frame of a flow, the processing handler takes it from the flow's f-frame queue and processes it. It executes the operators by using a highly-optimized execution container, called DataBank [11]. The container significantly reduces the execution overhead of sensing flows through structured execution and shared data management.

The decoupling of the sensing operators from the processing operators facilitates the engine to support sharing of sensing sources by multiple sensing flows. The flow executor allocates a sensing operator for each physical sensor and the operator mediates the concurrent sensing requests on the sensor from multiple flows. If the flows require different sampling rates and different sizes of frames, it collects the data at the highest rate, re-samples and re-packages them according to each flow's respective requirements.

5.3 Flow Execution Planning

The flow execution planner realizes the *c-frame-based flow coordination* for the use of the CPU and battery, which are the major resources competed by concurrent sensing applications. As described in Section 4.2, it allocates the resources to concurrent sensing flows while keeping their CPU and battery use under the availability and maximizing the application utilities. It adapts the c-frame intervals upon changes in system resource status.

For clear understanding, we first define the coordination problem as follows. Given concurrent sensing flows, $\{flow_i\}$, the problem is to determine their c-frame execution intervals, $\{p_i\}$, with the following objective function and constraints.

Objective. maximize $\sum u_i(p_i)$

Constraints. $\sum ct_i / p_i \leq 1 - CPU_f$ -- (*C1*)

$\sum ec_i / p_i \leq E_{limit}$ -- (*C2*)

In the formula, $u_i(p_i)$ is a utility function of $flow_i$ that takes an interval, p_i, as its input, and returns the corresponding utility value. The first constraint checks if the total CPU use by all flows does not exceed the current CPU availability. The CPU availability is specified as 1-CPU_f, where CPU_f is the CPU portion taken by foreground applications. In the left side, ct_i represents the CPU time required to process a c-frame of $flow_i$ and thus ct_i / p_i means the CPU utilization of $flow_i$ when it is executed at the interval of p_i. Similarly, the second constraint stipulates the energy consumption rate of all flows, $\sum ec_i / p_i$, within the maximum allowance of consumption rate, E_{limit}.

Note that SymPhoney can flexibly incorporate diverse coordination policies according to user preferences. For example, to differentiate the applications with distinct importance, the objective can be set with weights on the utilities. Similarly, to reflect fairness of applications, the objective can be set to maximize the minimum of the application utilities.

5.3.1 Resource Monitoring and Profiling

A practical challenge to perform the c-frame-based flow coordination lies in monitoring and measuring resource availability and demands to evaluate the CPU and energy constraints. First, it should be performed at runtime in a light-weight way to continuously reflect the changes in CPU and energy availability, influenced by foreground applications. Second, the system should be able to capture resource demands of diverse, customized sensing flows, when running at different execution intervals. In this section, we present our method to monitor the availability and demand of CPU and energy.

CPU Monitoring. Monitoring the CPU time to execute c-frames in the OS level ensures accurate measurements, but, in practice, incurs much overhead due to preemptive scheduling; it requires keeping track of the allocation of the CPU in every time slice. To address the challenge, we use a light-weight method and infer it from the elapsed time for the c-frame execution, i.e., the time between the start and end of the c-frame execution. Under contention, the elapsed time is longer than the CPU time for the execution since it includes the time taken by the other applications in the middle. We compensate the excess under the assumption that foreground applications use the CPU cycles uniformly over time at a constant rate.

For each coordination decision, $\{p_i\}$ should be chosen while meeting $\sum ct_i / p_i \leq 1 - CPU_f$. Let's denote the elapsed time for the c-frame execution of $flow_i$ as et_i. Applying the above idea, ct_i can be approximated as $et_i \times (1 - CPU_f)$; in et_i, the time taken by other applications would be $et_i \times CPU_f$. Hence, the condition (*C1*) can be simplified as follows:

$$\sum et_i \times (1 - CPU_f) / p_i \leq 1 - CPU_f$$

$$\sum et_i / p_i \leq 1$$

Leveraging the final derived inequality, SymPhoney can perform coordination by measuring $\{et_i\}$ without monitoring $\{ct_i\}$ or CPU_f. To avoid over-utilization caused by the approximation error, the right side of the inequality needs to be set to less than 1. In the current implementation, we set the value to 0.8 after iterative experiments.

Energy Profiling. It is difficult to measure ec_i at runtime, since off-the-shelf smartphones provide battery monitoring functionality only in a process-level and a coarse-grained way. Instead, we use an offline profiling method to calculate ec_i, adopted from our previous work [6][18]. It pre-builds energy profiles for sensors, CPU, and network interfaces on smartphones, which are major hardware components used by sensing applications; the network interfaces are used to communicate with external sensor devices. Then, it estimates ec_i based on the profiles by adding the energy consumed to perform the operations to collect and process a c-frame of $flow_i$. For instance, the energy consumption of IndoorNavi is calculated by correlating the profile for a microphone sensor with 44.1 kHz, and that for CPU with 21%. Such modeling-based energy estimation is well adopted for mobile devices [19].

5.3.2 Coordination and Adaptation

SymPhoney finds the execution intervals of flows, $\{p_i\}$, as defined by the above coordination problem. In many cases, the utility functions specified in the objective function are represented as decreasing linear or concave utility functions of intervals. In such cases, the coordination problem becomes a well-known concave maximization problem. SymPhoney solves the problem by using the Newton's method [20], which finds better approximation values successively. We could further adopt or design other algorithms for different optimization goals. Note that such optimization algorithms are not the main focus of this paper.

In addition, SymPhoney continuously adapts $\{p_i\}$ to changing resource availability mainly due to foreground applications. It is important since any optimal $\{p_i\}$ derived at a moment would cause severe under/over utilization of resources upon the changes. Specifically, Symphony triggers adaptation when the actual gap between the availability and demands goes above certain threshold. Internally, it is evaluated by monitoring et_i and checking if $\Sigma\ et_i / p_i$ exceeds or falls below certain threshold values. To avoid frequent adaptation, we set the upper and lower threshold to 0.8 and 0.6 in the current implementation, respectively. Once triggered, SymPhoney re-executes the Newton's method to find a new solution, $\{p_i\}$.

5.4 Flow Scheduling

Even though the planner coordinates concurrent flows' CPU use in a long term, the CPU allocation for a flow might be delayed in a short term by temporarily yielding up CPU cycles to others. Naïve ordering in the short-term CPU allocation might compromise freshness of the results for some sensing flows, to which the fresh result is essential.

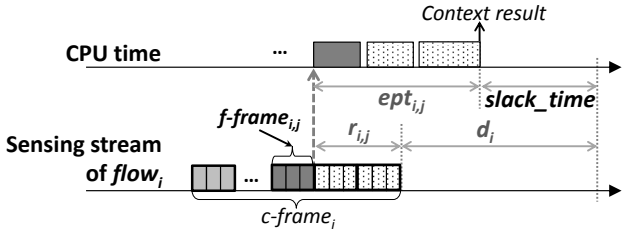


Figure 16. Slack time estimation of $f\text{-}frame_{i,j}$

Note that, in smartphone environments, it is a newly arising problem to schedule the execution order of concurrent, time-sensitive workloads.

To address the challenge, the flow scheduler carefully determines the execution order of the f-frames piled up in the queues, with the following objective function:

$$\textbf{maximize } \Sigma\ satisfy(c_i),\ \forall\{c_i\},$$

where, c_i denotes a context result for $flow_i$ generated by processing the f-frames in the queues, and $satisfy(c_i)$ is a binary function to evaluate if c_i is generated within a tolerable delay; $satisfy(c_i)$ is 1 if the delay of c_i is less than its tolerable delay and 0, otherwise. The delay is defined as the time taken to generate the context result from the moment that the final f-frame in a c-frame is ready.

To solve the problem, we adopt the least slack time (LST) scheduling algorithm [22], which is widely used for task scheduling in time-sensitive, real-time systems. It calculates the slack times for all queued f-frames and dispatches the f-frame with the least slack time, one at a time; meanwhile, when a new f-frame is queued, its slack time is calculated and reflected in the next scheduling turn.

A key to apply the LST algorithm for f-frame scheduling is to compute the slack time of an f-frame. It is notable that our slack time calculation reflects the position of an f-frame in relation to the whole c-frame and effectively approximates the actual remaining time to the deadline. Suppose that $f\text{-}frame_{i,j}$ is the j^{th} f-frame of $c\text{-}frame_i$ that is currently processed for $flow_i$. The slack time of $f\text{-}frame_{i,j}$ can be calculated by subtracting the processing time of remaining f-frames in $c\text{-}frame_i$ from its delivery deadline. For clear understanding, we represent the slack time of $f\text{-}frame_{i,j}$ in Figure 16 and formulate it as follows:

$$slack\_time(f\text{-}frame_{i,j}) = d_i + r_{i,j} - ept_{i,j},$$

where d_i is the tolerable delay of $flow_i$, $r_{i,j}$ is the remaining time to collect the whole *c-frame* with upcoming f-frames, and $ept_{i,j}$ is the expected time to process the unprocessed f-frames in $c\text{-}frame_i$., i.e., from $f\text{-}frame_{i,j}$ to the final f-frame.

Now, the deadline can be simply calculated as $d_i + r_{i,j}$ by its definition, and the slack time can be acquired by subtracting $ept_{i,j}$ from the deadline, $d_i + r_{i,j}$. SymPhoney computes $r_{i,j}$ based on the position of $f\text{-}frame_{i,j}$ in the $c\text{-}frame_i$, the number of data in the f-frame, and the sampling rate. For $ept_{i,j}$, SymPhoney continuously profiles processing times of f-frames. It estimates $ept_{i,j}$ from the profiling results for the f-frames of the previous c-frame.

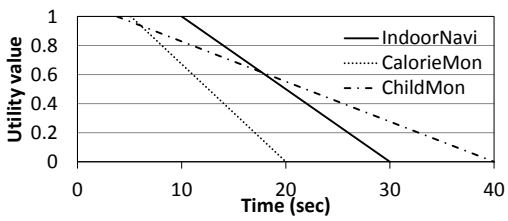


Figure 17. Utility functions of sensing applications

6. Implementation

We have implemented the prototype of SymPhoney engine on the top of Android phones using Android SDK 2.3. The implementation is in Java and about 12,000 lines of code. The engine runs as a *service* on the Android platform. Mobile sensing applications specify their monitoring requests via pre-defined Android service interfaces provided by the engine. The engine delivers processing results by using *intents* which are provided by Android for inter-process communication.

On top of the engine, we have easily implemented three example applications described in Section 2. The engine incorporates more than 50 types of operators commonly used in mobile sensing applications. The core sensing flows of the applications are concisely specified in the XML format by composing the built-in operators; the sensing flows for IndoorNavi, ChildMon, and CalorieMon are only 20, 99, and 190 lines, respectively.

We have modified Android to guarantee the minimum CPU quota for the SymPhoney engine. The functionality is implemented on the current CPU scheduler in Android, namely Completely Fair Scheduler (CFS) [16], without any kernel modification. The CFS provides the functionality to divide CPU usage into given portions for multiple process groups. Through */proc* interface, we designate the engine as a separate group, and allocate the minimum CPU quota as a user specifies.

7. Experiments

We performed extensive experiments with the prototype system in the following three aspects. First, we look into the effectiveness of our sensing-flow-aware coordination mechanisms: c-frame-based flow coordination and f-frame scheduling. Then, we examine how SymPhoney coordinates resource use between background sensing applications and foreground applications, in collaboration with a mobile OS. Last, we investigate the overhead of the SymPhoney engine.

7.1 Experimental setup

Mobile sensing workloads: we use CalorieMon, ChildMon, and IndoorNavi as sensing workloads. By default, all the three applications run simultaneously. We set the linear utility functions for the applications as shown in Figure 17. We set a minimum interval of an application to the duration required to collect a c-frame, which is most frequently executable. At this interval, an application provides its service at its best quality, where the utility

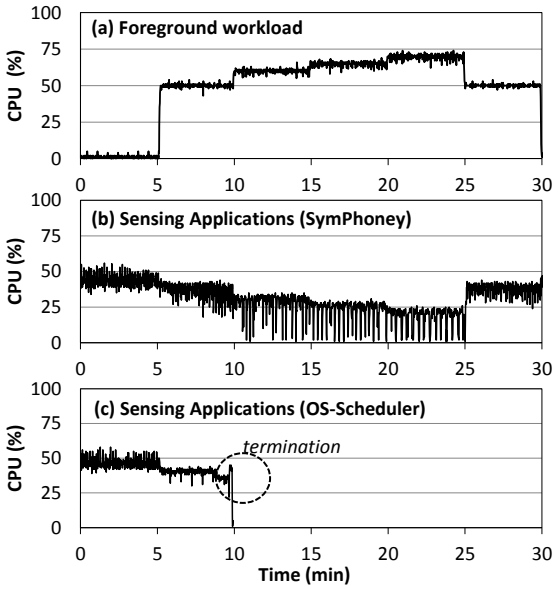


Figure 18. CPU usage on SymPhoney and OS-Scheduler

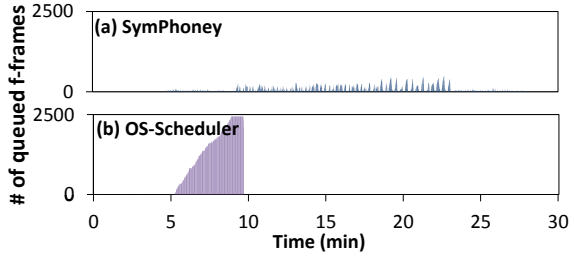


Figure 19. Number of queued f-frames over time

value is set to one. A maximum interval is decided considering application characteristics; beyond this interval, we assume that the application is of no use, and thus its utility value is set to zero.

Foreground workloads: We consider two types of foreground workloads with different characteristics: *dynamic* and *constant*. To generate the foreground workloads, we develop a tool that occupies the specified amount of CPU cycles for a given duration. In the case of the dynamic workload, we increase or decrease the CPU use every 5 minutes (See Figure 18 (a)). For the constant one, we fix the CPU utilization to 70% to continuously generate resource contention situation.

SymPhoney-related setting: we conduct the experiments by using the Google Nexus One with 1GHz Scorpion processor and 512MB RAM. We set the CPU quota for SymPhoney to 25%, by default, which would be reasonable considering users' smartphone usage behavior.

7.2 Coordination among Sensing Applications

7.2.1 Effect of c-frame-based flow coordination

We first examine the effect of our c-frame-based flow coordination mechanism. We compare it with two alternative methods: *OS-Scheduler* and *SymPhoney-Equal*. In the former, a sensing flow is executed in a separate thread, delegating the resource allocation and scheduling to

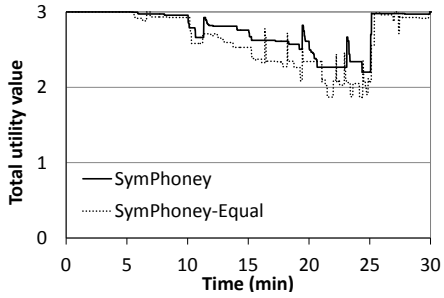


Figure 20. Total utility over time

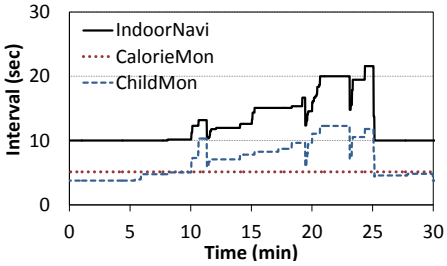


Figure 21. Execution intervals on SymPhoney

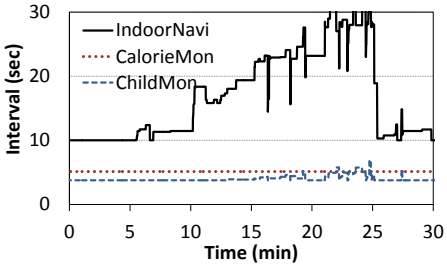
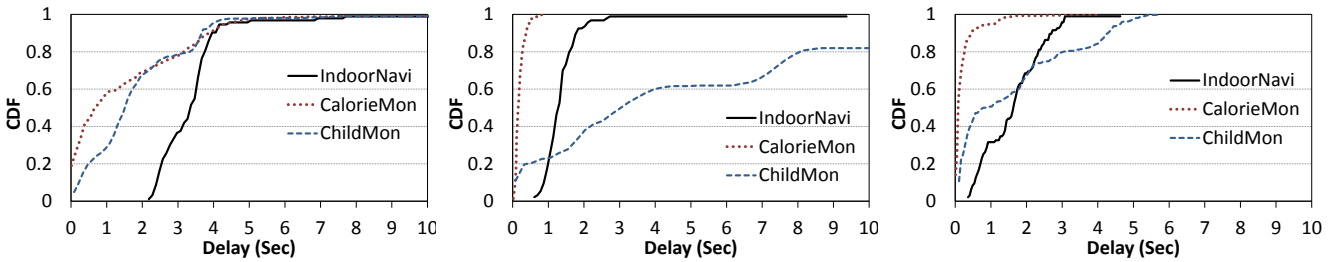


Figure 22. Execution intervals on SymPhoney-Equal

the Android CPU scheduler. The method equally distributes available CPU quota to the applications running in the background; if some applications underutilize given resources, the scheduler re-distributes the remaining cycles to the others. Each flow is executed in its minimum interval to maximize its utility. SymPhoney-Equal is a modified version of SymPhoney, which has the same resource allocation policy as the OS-Scheduler, i.e., equal allocation. Different from the OS-Scheduler, however, it adapts the execution interval of each flow to the allocated resources based on c-frames.

We first evaluate SymPhoney's resource coordination capability under different CPU availability, in comparison with OS-Scheduler. We control the available CPU by using the *dynamic* foreground workload (See Figure 18 (a)). We first investigate the coordination behavior in a macroscopic viewpoint by measuring the CPU utilization and the number of unprocessed f-frames over time. Figure 18 and Figure 19 show the results, respectively. In Figure 18 (b), we can observe that SymPhoney properly throttles the CPU use of sensing applications under the increasing foreground workload, whereas OS-Scheduler does not, undesirably terminating the sensing applications at 10 minutes in Figure 18 (c). When the foreground workload starts to take 50% CPU time from 5 minutes, the sensing applications are not



(a) FIFO (b) Priority (c) SymPhoney

Figure 23. CDF of delays on diverse scheduling methods

allocated with enough resources to run at their minimum intervals. From this time, applications cannot process all the continuously streaming f-frames by using OS-scheduler, and their internal queue is accumulated, as shown in Figure 19 (b). At 10 minutes, the queue is full by unprocessed 2500 f-frames, resulting in undesirable buffer overflow.

On the other hand, SymPhoney effectively coordinates the resource use by increasing the execution intervals of c-frames for sensing applications. Accordingly, the number of unprocessed f-frames are rapidly flushed although they are piled up temporarily upon the increase of the foreground workload (Figure 19 (a)). In addition, SymPhoney quickly recovers the CPU use of sensing applications as the foreground workload decreases, for example, to 50% at 25 minutes. It again reduces the execution intervals of c-frames to increase the application utility, fully leveraging the whole available resources. We also performed the same experiment using SymPhoney-Equal. SymPhoney-Equal also avoids the system overload by adapting the execution intervals of c-frames and shows similar behavior with SymPhoney; we do not plot it in the figure.

Next, we look into the utility of applications to investigate the coordination among the sensing applications. Figure 20 shows the total sum of utility values for the three applications for SymPhoney and SymPhoney-Equal; the experiments are performed in the same setting with the previous experiment. Most importantly, SymPhoney mostly provides the higher utility value compared to SymPhoney-Equal, especially under contentious situation (10 min ~ 25 min). This is because SymPhoney considers applications' resource demand and requirements in a holistic view, when determining the c-frame execution interval of each flow.

For more detailed understanding, we additionally plot the execution intervals of the three applications adjusted by SymPhoney and SymPhoney-Equal (Figure 21 and Figure 22). The two show different behaviors in coordinating multiple applications under contentious situations. From 10 minutes, the sensing applications should share 40% of the CPU cycles since the foreground takes 60%. In the case of SymPhoney-Equal, each is allocated 13%. From 10 to 15 minutes, SymPhoney-Equal selects only IndoorNavi and increases its c-frame execution intervals since it takes more cycles than the allocated amount while the other two still operate well with the 13%. At around 20 minutes, the system starts to increase the intervals of ChildMon slightly which requires about 11% for the best operation because the available CPU gets under 30%, each allocated about 10%. On the contrary, SymPhoney considers the resource requirements of different applications as well as their utility functions. Under contention, it increases the intervals of ChildMon together with the ones of IndoorNavi, since the utility of ChildMon is less sensitive to the increase. In this way, it minimally reduces the total utility value of the system. Note that SymPhoney can support diverse policy functions, although this experiment focuses on the policy to maximize the total sum of applications' utilities.

7.2.2 Effect of f-frame scheduling

In this section, we investigate the effect of our slack time-based f-frame scheduling method. For comparison, we develop two alternative scheduling methods as follows:

- **FIFO**: simply, it first schedules the f-frame that is generated first. We consider this method as a baseline.
- **Priority**: it prioritizes f-frames in the queues according to the tolerable delays of the associated sensing flow; it first schedules an f-frame of which the associated flow has the shortest delay requirement.

We execute CalorieMon, IndoorNavi, and ChildMon together with the *constant* foreground workload; their delay requirements are set to 2, 3, and 5 seconds, respectively. We fix the execution interval of each flow to clearly understand the effect of the different scheduling methods.

Figure 23 shows the distribution of the delays of context results. Overall, SymPhoney better meets the delay requirements of the applications than the other methods. As shown in Figure 23 (a), FIFO is not able to meet the delay requirements of CalorieMon and IndoorNavi for 30% and 63% of their context results, respectively. While f-frames for IndoorNavi are executed to generate a result, CPU consumption quickly goes up, and hence the processing of f-frames of all the applications is delayed (up to 4 seconds). On the other hand, as shown in Figure 23 (b), Priority meets the tight delay requirements of CalorieMon and IndoorNavi, while ChildMon experiences significant delay, i.e., above 8 seconds for 20% of its results. This is because Priority always schedules f-frames of CalorieMon and IndoorNavi ahead of those of ChildMon, significantly delaying the processing of ChildMon. Finally, Figure 23 (c) shows that

SymPhoney well meets the delay requirements of all the applications. It intelligently schedules f-frames based on the actual slack time up to the tolerable delay, rather than only the static delay requirement of an application. For example, it schedules f-frames of ChildMon first when f-frames of the other applications have enough slack time to meet their delay requirements, reducing the delays of ChildMon.

7.3 Coordination between Sensing and Foreground Applications

In this section, we examine how SymPhoney coordinates resource use between sensing and foreground applications, in collaboration with Android CFS scheduler. We vary the CPU quota for SymPhoney and look into the performance trade-off between sensing and foreground applications. For a realistic foreground workload, we use VideoPlayer described in Section 2.

Table 2 shows the CPU utilization of SymPhoney and VideoPlayer with diverse CPU quota setting, together with the total utility value of the sensing applications. When the CPU quota for SymPhoney is enough, i.e., more than 40%, SymPhoney provides the applications with high utilities, fully utilizing the available CPU. On the other hand, the CPU utilization for VideoPlayer decreases to less than 56%, resulting in video frame drops. When the CPU quota for SymPhoney is reduced, e.g., 20%, VideoPlayer runs smoothly by using the increased available CPU. In that case, SymPhoney degrades the utility of the applications to accommodate them with the limited resource.

7.4 System overhead

We look into the cost of resource coordination of SymPhoney. We observe three major causes of overheads.

Decision of flow coordination: SymPhoney performs the flow coordination whenever a c-frame of an application is processed. Even in the case that the execution intervals are set to the minimum, the coordination is performed occasionally, at the rate of about 0.5Hz, in our setting. It takes only 0.6ms per coordination on average.

Slack time calculation and scheduling: we investigate the cost of f-frame scheduling. The scheduling is triggered frequently, upon the generation of an f-frame, i.e., 125Hz in our setting. However, the cost is relatively small; each scheduling decision takes 0.07ms per scheduling on average. Overall, it consumes under 1% of CPU cycles.

Runtime CPU monitoring: instead of tracking CPU time at kernel level, SymPhoney records the elapsed time of f-frame processing. This is negligible, under 1% of CPU.

8. Related Work

Initial trials to provide a common platform for mobile sensing applications were made in SeeMon [4] and Orchestrator [6]. They support concurrent applications, and provide a high-level declarative query language so that applications simply specify their queries in a context level. On the other hand, SymPhoney provides a more generic way of specifying sensing applications as dataflow. As a platform, the major objective of SeeMon is to efficiently support multiple sensing applications, whereas SymPhoney focuses on the coordination between them rather than efficiency. Orchestrator provides coordination functionalities similar to SymPhoney, but it focuses on the coordination on external sensor devices, not effectively dealing with resource contention on a smartphone.

Table 2. Quality tradeoff

Quota for SymPhoney		50%	40%	30%	20%
VideoPlayer	Avg. CPU	50.4%	55.6%	67.6%	75.3%
SymPhoney	Avg. CPU	44.2%	39.3%	29.1%	20.2%
	Total utility	3	2.94	2.65	2.13

Jigsaw [5] and Kobe [7] were also proposed to provide system supports for mobile sensing applications. Jigsaw focuses on supporting three types of widely-used contexts, and devises optimization techniques for each context. On the other hand, SymPhoney targets far wider and highly customized sensing applications. Kobe conducts runtime optimization for an application by identifying the best configuration based on the tradeoff between energy, latency, and accuracy. Different from SymPhoney, however, it does not deal with the issues of concurrent sensing applications running with other typical smartphone applications.

In mobile and sensor systems, there have been considerable efforts to reduce resource consumption for continuous sensing and data processing [8][9]. A variety of techniques have been proposed, such as an energy-fidelity tradeoff [8], user interest-based sensor management [4] and hierarchical sensor management [9]. Recently, an approach is proposed to leverage in-situ collaboration among nearby users [18], They focus on optimizing resource use for a single sensing application whereas SymPhoney newly aims to coordinate resource uses in consideration of multiple concurrent applications. These techniques designed for individual applications can be incorporated into SymPhoney and further improve its performance supporting concurrent applications more efficiently.

In the field of data stream processing, several dataflow execution engines have been proposed in server environments [12][13]. They mostly focus on the dataflow graphs composed of relational operators, such as selection, join, and aggregation. SymPhoney is a new trial to develop a sensing flow execution engine for a smartphone, specialized for sensing applications; it deals with diverse operators for sensing, feature extraction (FFT, MFCC), and context inference (GMM, Decision tree). Recently, a server engine, XStream [13], has been proposed to support operators for signal processing. However, it does not deal with the resource contention problem arising when executing concurrent application flows.

Scheduling workloads subject to deadlines has been substantially studied for several decades [21][22]. Many successful scheduling techniques such as EDF and LST

have been introduced for a variety of scheduling objectives, constraints, and system models. Building upon such theoretical results, SymPhoney adopts the principle of LST into sensing flow scheduling. Interestingly, sensing applications are associated with their own characteristics, such as soft real-time, continuous and patterned workloads, and energy constraints. This raises new challenges in development of general real-time scheduling theories to accommodate such sensing workloads on smartphones.

9. Conclusion

In this paper, we show that concurrency and coordination is a key to accelerate the deployment of mobile sensing applications, which is different from conventional sensing applications. This is mainly because smartphones are used as a generic personal computing platform, whereas other sensor platforms such as motes are mainly designed for a single specialized application. As we capture such differences, we propose SymPhoney, a novel sensing flow execution engine to support concurrent sensing applications. Through the sensing-flow-aware coordination approach and methods, it coordinates the resource use of sensing applications effectively, maximizing the service qualities with respect to utility and delay. We demonstrate the coordination capability of our engine through in-depth experiments.

10. Acknowledgements

This work was supported by the National Research Foundation of Korea grant (No. 2012-0005733 and No. 2012-0006422) funded by the Korean Government (MEST), and the SW Computing R&D Program of KEIT (2011-10041313, UX-oriented Mobile SW Platform) funded by the Ministry of Knowledge Economy of Korea. The authors thank Bupjae Lee for his support for experiments.

11. References

[1] H. Lu, et al., Soundsense: scalable sound sensing for people-centric applications on mobile phones, Proceedings of The 7th International Conference on Mobile Systems, Applications, and Services (Mobisys '09).

[2] S. P. Tarzia, et al., Indoor Localization without Infrastructure using the Acoustic Background Spectrum, Proceedings of The 9th International Conference on Mobile Systems, Applications, and Services (Mobisys '11).

[3] L. Bao, et al., Activity Recognition from User-Annotated Acceleration Data, Proceedings of The Second International Conference on Pervasive Computing (Pervasive '04).

[4] S. Kang, et al., SeeMon: scalable and energy-efficient context monitoring framework for sensor-rich mobile environments, Proceedings of The 6th International Conference on Mobile Systems, Applications, and Services (Mobisys '08).

[5] H. Lu, et al., The Jigsaw continuous sensing engine for mobile phone applications, Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems (SenSys '10).

[6] S. Kang, et al., Orchestrator: An Active Resource Orchestration Framework for Mobile Context Monitoring in Sensor-rich Mobile Environments, Proceedings of The IEEE Pervasive Computing and Communication (PerCom '10).

[7] D. Chu, et al., Balancing Energy, Latency and Accuracy for Mobile Sensor Data Classification, Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems (SenSys '11).

[8] B. D. Noble, et al., Agile application-aware adaptation for mobility, Proceedings of The ACM Symposium on Operating Systems Principles (SOSP '97).

[9] T. Park, et al., E-Gesture: A Collaborative Architecture for Energy-efficient Gesture Recognition with Hand-worn Sensor and Mobile Device, Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems (SenSys '11).

[10] VPlayer, https://play.google.com/store/apps/details?id=me.abitno.vplayer.t&hl=en.

[11] Y. Ju, et al., An Efficient Dataflow Execution Method for Mobile Context Monitoring Applications, Proceedings of The IEEE Pervasive Computing and Communication (PerCom '12).

[12] D. Carney, et al., Monitoring streams: a new class of data management applications, Proceedings of 28th International Conferences on Very Large Data Bases (VLDB '02).

[13] L. Girod, et al., XStream: A signal-oriented data stream management system, Proceedings of the 24th International Conference on Data Engineering, (ICDE '08).

[14] I. Hwang, et al., Leveraging Children's Behavioral Distribution and Singularities in New Interactive Environments: Study in Kindergarten Field Trips, Proceedings of The Tenth International Conference on Pervasive Computing (Pervasive '12).

[15] H. Jang, et al., RubberBand: Augmenting Teacher's Awareness of Spatially Isolated Children on Kindergarten Field Trips, Proceedings of The 14th International Conference on Ubiquitous Computing (Ubicomp '12).

[16] Completely Fair Scheduler , http://en.wikipedia.org/wiki/Completely_Fair_Scheduler.

[17] A. Krause, et al., Trading off prediction accuracy and power consumption for context-aware wearable computing, Proceedings of The International Symposium on Wearable Computers (ISWC '05).

[18] Y. Lee, et al., CoMon: Cooperative Ambience Monitoring Platform with Continuity and Benefit Awareness, Proceedings of The 10th International Conference on Mobile Systems, Applications, and Services (Mobisys '12).

[19] L. Zhang, et al., Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones, Proceedings of The International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '10).

[20] Newton's method, http://en.wikipedia.org/wiki/Newton%27s_method.

[21] C. L. Liu, et al., Scheduling Algorithms for Multi-programming in a Hard-Real-Time Environment, Journal of the ACM, Vol. 20, No. 1, January 1973.

[22] J. Leung, A New Algorithm for Scheduling Periodic, Real-Time Tasks, Algorithmica, Vol. 4, 1989.

[23] J. Lester, Validated caloric expenditure estimation using a single body-worn sensor, The 11th International Conference on Ubiquitous Computing (Ubicomp '09).

[24] Y. Gu, et al., Data Forwarding in Extremely Low Duty-Cycle Sensor Networks with Unreliable Communication Links, Proceedings of the 5th ACM Conference on Embedded Networked Sensor Systems (SenSys '07).

[25] Y. Lee, et al., MobiCon: Mobile Context-Monitoring Platform, Communications of the ACM, March 2012.