An Efficient Dataflow Execution Method for Mobile Context Monitoring Applications

Younghyun Ju, Chulhong Min, Youngki Lee, Jihyun Yu, Junehwa Song
Computer Science, KAIST, Daejeon, Korea
{yhju, chulhong, youngki, jihyun, junesong}@nclab.kaist.ac.kr

***Abstract*—In this paper, we propose a novel efficient dataflow execution method for mobile context monitoring applications. As a key approach to minimize the execution overhead, we propose a new dataflow execution model, *producer-oriented model*. Compared to the conventional *consumer-oriented model* adopted in stream processing engines, our model significantly reduces execution overhead to process context monitoring dataflow reflecting unique characteristics of context monitoring. To realize the model, we develop DataBank, an execution container that takes charge of the management and delivery of the output data for the associated operator. We demonstrate the effectiveness of DataBank by implementing three useful applications and their dataflow graphs, i.e., MusicMap, FindMyPhone, and CalorieMonitor. Using the applications, we show that DataBank reduces the CPU utilization by more than 50%, compared to the methods based on the consumer-oriented model; DataBank enables more context monitoring applications to run concurrently.**

Keywords-context monitoring, dataflow execution, performance

I. INTRODUCTION

Context monitoring applications [4][5][6] are increasingly emerging and becoming a major workload of smartphones. The applications continuously monitor contexts of users to provide situation-aware services. Their core is to transform high-rate raw sensing data to context information through a complex series of processing steps. Such a series of processing is commonly represented as a dataflow graph of operators. For example, SoundSense [4] continuously collects audio data from a microphone at 8 kHz, and subsequently applies more than 20 feature extraction and classification operations such as fast Fourier transform (FFT), mel-frequency cepstral coefficients (MFCC), and Gaussian mixture model (GMM).

The execution of complex context monitoring dataflow would impose significant overhead on smartphones, especially considering the resource scarcity of the smartphones. Naive implementation would not be effective in supporting continuous monitoring upon high-rate input sensing data. It easily causes frequent invocation of computation-intensive operators. It may also incur significant overhead in executing and managing the dataflow graphs such as for operator scheduling and data management. It is critical to minimize the execution overhead of context monitoring dataflow. With the high CPU overhead, only a small number of context monitoring applications can concurrently run on a smartphone.

To address the challenge, we develop a new dataflow execution model, a *producer-oriented* model. This model is distinguished from a *consumer-oriented* model, adopted in conventional server engines for stream processing [1][2][3]. In the consumer-oriented model, operators consuming data, i.e., consumers, separately collect and manage their input data; producers play a rather simple role to generate and pass their output data to the consumers immediately. In contrast, in the proposed model, the producers play the key role and manage the data in an integrated fashion for multiple consumers sharing the output. (See Fig. 1 for an example graph and its execution in the two models.)

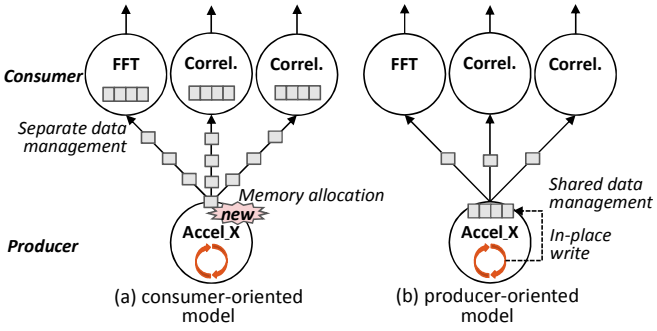


Figure 1. Execution of an example graph

Our producer-oriented model is advantageous in reducing the overhead of execution. First, it reduces the number of data pass and scheduling operations through lazy data delivery; the data are passed just when consumers actually need them. Since many operations are performed over a window of contiguous data rather than individual sensing data or feature, it is often of no use to immediately deliver the results. Second, the integrated management of data by a producer eliminates redundancy in the separate management performed by individual consumers. For many context monitoring applications, it is common that the results of an operator are shared by multiple consumers. Finally, the model avoids repetitive memory allocation and deallocation upon new data generation by allowing producers to directly write new data on its pre-allocated management buffer.

The contribution of this paper can be summarized as follows. First, we propose a novel dataflow execution method for mobile context monitoring applications. The method exploits a new producer-oriented execution model to minimize dataflow execution overhead. To realize the model, we develop DataBank, an execution container that takes charge of the management and delivery of the output data for the associated operator. Second, to show the effectiveness of our model, we develop three interesting applications based on the execution model inspired by recent works in mobile and sensor systems [4][5][6]. Finally, we report extensive evaluation results using the three applications. Our results

978-1-4673-0258-6/12/$31.00 ©2012 IEEE

show that our method reduces CPU consumption by more than 50% compared with traditional methods.

The rest of this paper is organized as follows. In Section II, we present related work and Section III describes the three motivating applications and their dataflow graphs. Section IV explains the data structure and operations of DataBank. Then, we show the experimental results in Section V and finally conclude the paper in Section VI.

II. Related Work

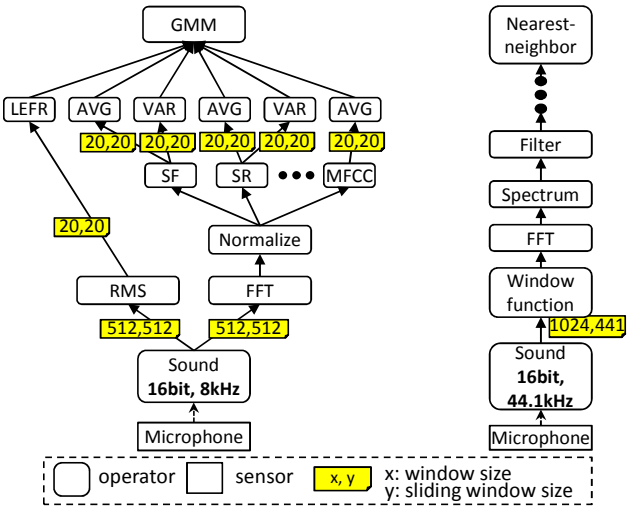
Context monitoring applications have been developed in diverse application domains, e.g., motion and activity analysis [6][7], affective and cognitive applications [8][13], gesture recognition [12], and indoor location tracking [5]. Also, Jigsaw [9] has been proposed to provide a reusable mobile sensing engine for three typical applications. They mainly focus on devising accurate and efficient context monitoring dataflow specific for target applications. On the other hand, our focus is to develop a general method to minimize the execution overhead of such dataflow. Diverse types of context monitoring applications can adopt our execution method to optimize their performance.

There has been some research to develop a general context monitoring platform, SeeMon [8][14], MobiCon [15] and Orchestrator [10]. They provide a high-level declarative query language so that applications simply specify their queries in a context level, e.g., activity. To process high-level queries efficiently, they carefully determine processing methods and necessary resources, e.g., the set of sensors. On the other hand, our work focuses on optimizing the execution of given dataflow graphs rather than selecting the best processing method.

In the field of data stream processing, active research has been conducted to optimize the execution of dataflow graphs in server environments [1][2][3][11]. They mostly target dataflow graphs composed of relational operators, such as *selection*, *join*, and *aggregation*. Our work is a new trial to develop an efficient dataflow execution method for smartphones, specialized for mobile context monitoring. It has several key differences from the conventional methods. First, the types and semantics of target operators are different; instead of relational operators, it deals with diverse operators for sensing, feature extraction (e.g., FFT, MFCC), and context inference (e.g., GMM, Decision tree). Second, we propose a novel producer-oriented execution model, reflecting the characteristics of context monitoring dataflow. Recently, a server engine, XStream [3], has been proposed to support operators for signal processing. However, it would have a performance limitation to apply for mobile context monitoring since it is based on the consumer-oriented model.

III. Context Monitoring Dataflow

The context monitoring logic of applications is commonly represented as a dataflow graph of operators, which applies a series of processing on continuous sensing data. A dataflow graph consists of operators and edges. Each operator represents a unit of computation or I/O, and each edge represents data dependencies between two operators. We define operators and edges as follows. An operator, op_i, is defined as a 3-tuple (*id*, *type*, *parameter*) where *id*, *type*, and *parameter* are the identifier, the type, and the set of configuration parameters for op_i, respectively. In context monitoring dataflow graphs, there are two kinds of operators: *sensing operators* which generate sensing data and *processing operators* which take input data and generate new output data. An edge from a producer op_i to a consumer op_j is defined as a 4-tuple (*prod_id*, *cons_id*, *wsize*, *sl_size*), where *prod_id* and *cons_id* are the identifiers of op_i and op_j, and *wsize* and *sl_size* are the data requirement of op_j on op_i, i.e., window size and sliding window size used by op_j.



(a) Ambient sound monitoring (b) Indoor place monitoring

Figure 2. Dataflow graphs of motivating applications

We implement three applications and their dataflow graphs to motivate and evaluate the effectiveness of our method: (1) *MusicMap*, (2) *FindMyPhone*, (3) *CalorieMonitor*. The context monitoring dataflow of the applications contains tens of processing operators fed by high-rate sensing operators (from 50 Hz to 44.1 kHz). The processing operators continuously take high-rate input data, generate new processing results, and pass them to next operators.

MusicMap collects and shares the information of played music in diverse city-wide places, such as cafés, restaurants, and streets. This is done by participatory sensing from users' smartphones. A smartphone continuously collects ambient sound data, processes them, and reports the musical genre once music is detected. We implement the core *ambient sound monitoring* graph based on a previous work [4]. The dataflow graph is shown in Fig. 2 (a). As a data source, a sound sensing operator continuously samples audio data at 8 kHz from a phone-embedded microphone. Audio samples are delivered to the next two operators, which calculate RMS and FFT over a window of 512 samples, respectively. Their results are further processed through a series of operators until the musical genre is classified by GMM.

FindMyPhone enables a smartphone user to trace her phone when she lost it inside buildings. To trace the location of the phone, the phone continuously localizes itself at the level of an office or a room, and reports its location to a

designated server. The core logic for *indoor place monitoring* is inspired by BatPhone [5] and specified as a dataflow graph as shown in Fig. 2 (b). Although the graph is in a form of a simple chain, the operators in the graph are frequently executed to process very high-rate sound data generated at 44.1 kHz.

CalorieMonitor continuously recognizes a user's activities such as walking, sitting, and bicycling and estimates caloric expenditure from the activities. For the core *activity monitoring* logic, we adopt the activity recognition algorithm using five external accelerometer sensors from [6].

IV. Efficient Dataflow Execution with DataBank

It is critical to minimize the execution overhead of context monitoring dataflow since high overhead causes severe performance problem. To address the problem, we proposed a producer-oriented execution model in Section I. We realize the model by developing *DataBank*, the execution container for an operator. Given a dataflow graph, DataBanks are constructed for individual operators, and an *execution network* for the graph is built as a network of the DataBanks. A DataBank serves as the basic unit in the execution of the network. It encapsulates the execution of the contained operator. More importantly, it takes charge of the management of the output data as well as the delivery of the data to the consumers upstream in the network.

The DataBank separates the pure processing logic from the logic for dataflow execution and data management. By adopting the DataBank, developers can easily develop the execution network of a graph, realizing the producer-oriented model; for an operator, they focus on its processing logic itself without the burden of the data management.

A. Structure of DataBank

Fig. 3 illustrates the structure of a DataBank. In addition to the associated operator, the DataBank maintains two data structures: *shared output buffer* and *flow table*.

A **shared output buffer** contains the processing results of the associated operator. The producer-side shared buffer provides two key advantages. First, it maintains the outputs of the operator in a single unified buffer, thereby eliminating redundant data management by multiple consumers. Second, it enables the operator to write processing results directly on the buffer. Such *in-place write* avoids frequent memory allocation and deallocation overhead for new processing results.

The shared output buffer is implemented as a fixed-size circular buffer statically allocated on contiguous memory region. It stores output data in the order of generation. Such a circular buffer is well-suited to contain the output of context monitoring operators which are periodically generated in fixed forms and used only for a short duration. Also, the contiguous buffer facilitates windowing operations over the containing data frequently used for context monitoring. DataBank maintains the current buffer index, idx_c, to point the buffer position to be written next.

A **flow table** connects the associated operator to its consumers. Each table entry contains the information about

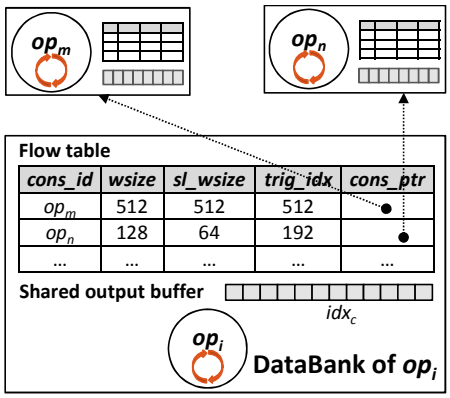


Figure 3. Structure of DataBank

the data requirement for a consumer, e.g., window size and sliding window size. Based on the information, the Databank delivers the results of the producer on time, just when the data are actually required by consumers. In this way, the DataBank reduces the number of data pass and scheduling operations, realizing the lazy data delivery of the producer-oriented model.

For a consumer, op_m, the table entry is formally described as a 5-tuple (*cons_id*, *wsize*, *sl_wsize*, *trig_idx, cons_ptr*).

- *cons_id* is the identifier of op_m
- *wsize* and *sl_wsize* keep the window size and sliding window size used by op_m.
- *trig_idx* is the buffer index that points the end position of the next data window for op_m; it is used to identify if the buffer is filled up with the produced data, and signal the consumer op_m to begin its operation.
- *cons_ptr* is the pointer to the DataBank of op_m.

This representation of the flow table mainly targets operators requiring count-based windows, composed of a certain number of data items. Such window semantics is commonly used for many context monitoring operators for which input data are regularly-sampled sensing data or their processing results; for instance, the FFT operator in MusicMap takes a window of 512 samples from a sound sensing operator with 8 kHz of sampling rate (Fig. 2(a)). The structure of the flow table can be extended to support other window semantics such as time-based window, where a window consists of data within a certain time period.

B. Execution Network of DataBanks

The DataBank for an operator op_i is constructed based on the edges e_k's between op_i and its consumers. For an edge e_k, a table entry te_l = <e_k.*cons_id*, e_k.*wsize*, e_k.*sl_wsize*, e_k.*wsize*, *ptr*(*D_Bank*(e_k.*cons_id*))> is created where *ptr*(*D_Bank* (e_k.*cons_id*)) is the pointer to the DataBank for the consumer e_k.*cons_id*. Then, the shared output buffer is statically allocated at the initialization time; its size is set to the maximum window size of the consumers, i.e., max {te_l.*wsize* for all table entries, te_l's}.

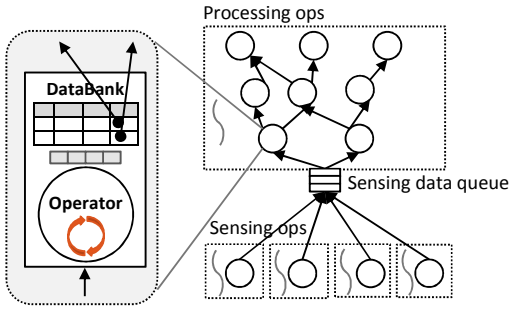


Figure 4. Execution network of DataBanks

Fig. 4 shows an execution network of DataBanks. DataBanks are connected via the pointers in the flow table. A DataBank passes control and data to a consumer's DataBank via a direct function call; it invokes a method in the consumer's DataBank through the pointer in the table. Such a design choice is advantageous in two aspects. First, it achieves low overhead in inter-operator communications as it avoids expensive scheduling and queuing operations. Second, the direct transfer of control results in synchronous execution of operators, which facilitates the sharing of a single data buffer between a producer and multiple consumers.

Exceptionally, the DataBanks of sensing operators are connected to their consumers' DataBanks via a *sensing data queue*. This is due to the asynchronous nature of sensing operators; different from processing operators, the sensing operators should be triggered and executed at a certain sampling rate. The DataBanks of sensing operators run in different threads and push generated data to the sensing data queue.

C. Execution of DataBanks

The execution of DataBank involves the three steps: (1) *operator execution* and *in-place buffer write*, (2) *data requirement check*, and (3) *data delivery to consumers*. The pseudo code for the execution mechanism is presented in Fig. 5.

First, the associated operator is executed with input data (*line 3*). During the execution, the operator writes processing results via *write*() interface provided by the DataBank. Each result is directly written on the shared output buffer at the current buffer index, idx_c. idx_c is updated whenever the write operation is performed. After the execution of the operator, the flow table is looked up to check if the data is ready to signal a consumer, i.e., the data window for the consumer is filled up (*line 4-7*). For a table entry te_i, the consumer with $te_i.cons\_id$ is signaled if the current buffer index idx_c crosses over $te_i.trig\_idx$; the consumer's DataBank is directly invoked via the pointer in the entry, $te_i.cons\_ptr$ (*line 8-14*). If the associated operator is a sensing operator, the data are delivered through the sensing data queue.

The data are delivered by using a *DataWindow* abstraction. DataWindow is an abstract data type that represents a window of data. It encapsulates the internal buffer structure of the producer. It also prevents consumers from writing on the buffer by providing read-only interfaces.

```
Function: execute()
Input   : input_dw, a DataWindow passed from previous operator
1. idx_p ← idx_c   // idx_p is set to the previous buffer index
2. buf_size ← the size of the shared output buffer
3. operator.process (input_dw)
4. foreach entry of the flow table te_i
5.    trig_idx ← te_i.trig_idx
6.    // check the data requirement of the corresponding consumer
7.    if {(idx_p < idx_c) && (idx_p < trig_idx) && (trig_idx ≤ idx_c)}||
        {(idx_p ≥ idx_c) && {(idx_p < trig_idx) || (trig_idx ≤ idx_c)}}
8.       // if the requirement is met, a DataWindow is created
9.       DataWindow out_dw
10.      out_dw.buffer_ptr ← the pointer to its output buffer
11.      out_dw.start_idx ← (te_i.trig_idx – te_i.wsize) % buf_size
12.      out_dw.length ← te_i.wsize
13.      // deliver the DataWindow via the consumer pointer
14.      te_i.cons_ptr.execute (out_dw)
15.      // update trig_idx of the entry
16.      te_i.trig_idx = (te_i.trig_idx + te_i.sl_wsize) % buf_size
```
Figure 5. Pseudo code for DataBank execution

It provides two interfaces for sequential and random data access. A DataWindow is represented internally as <*buffer_ptr*, *start_idx*, *length*>, which refers to a contiguous section of the shared output buffer starting from the *start_idx*. After delivery, *trig_idx* of the entry is updated for the next data window (*line 15-16*).

D. Shared Output Buffer Management

The shared output buffer of a DataBank is accessed by multiple consumers as well as the associated operator. In the case of processing operators, the shared use is simply realized without complication since the operators are synchronously executed. None of the operators try to access the shared buffer simultaneously.

The DataBank for a sensing operator requires additional management for the shared use of its buffer. First, it should keep track of the use of DataWindows passed to the consumers. Before writing a new data at the current buffer position, it checks if there exists a DataWindow that contains the buffer position and has not been processed yet. If the data at the current position has not been processed, the buffer space is extended.

V. Evaluation

A. Experimental Setup

Alternative methods: For comparison, we implement two alternative methods based on the consumer-oriented execution model: *Cons_Queue* and *Cons_DF*. In these methods, operators are executed without the support of DataBank; the data generated by producers are immediately passed to their consumers and are managed by individual consumers. The difference of the two methods is in the way to connect operators. In Cons_Queue, the operators are connected by a message queue as in many stream processing engines [1][2]. On the other hand, Cons_DF connects the processing operators via function calls as in DataBank.

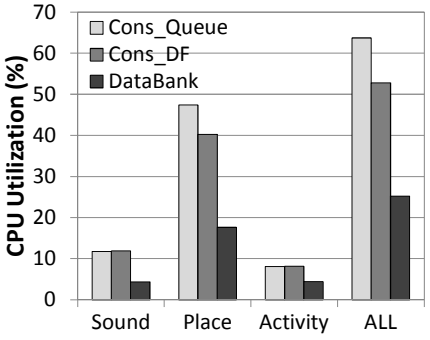


Figure 6. Processing cost

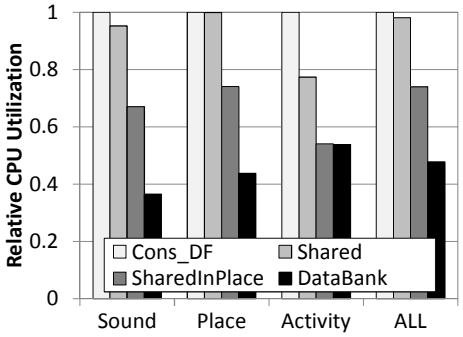


Figure 7. Relative CPU utilization

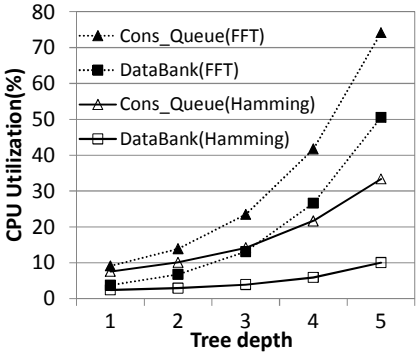


Figure 8. Scalability

While Cons_Queue allows diverse scheduling policies, we use the depth-first traversal used in DataBank and Cons_DF to eliminate the effect of scheduling policy.

Workloads: We evaluate the performance of DataBank based on three applications' dataflow graphs presented in Section III: ambient sound monitoring (Sound), indoor place monitoring (Place), and activity monitoring (Activity). In addition, we evaluate the performance when all of the three dataflow graphs are executed (ALL) concurrently.

Metric: We measure the performance of DataBank in terms of CPU utilization. We measure and present the average CPU utilization over 300 seconds. All experiments are conducted three times and the average value is presented with negligible variance, i.e., 0.1. The CPU information is obtained from '/proc/*pid*/stat' in procfs.

Implementation and hardware setup: We have implemented DataBank and example dataflow graphs with Java using Android SDK 2.3. We use the Google Nexus One running Android OS 2.3.3 with 1GHz Scorpion processor and 512MB RAM. Most recent smartphones adopt dynamic voltage and frequency scaling (DVFS) technique, which adjusts the CPU frequency according to workload for energy saving. To fairly compare the CPU cost of the methods, we fix the CPU frequency to the maximum one of Nexus One, i.e., 998MHz.

B. *Performance of DataBank*

We evaluate the performance of DataBank compared to the methods based on the consumer-oriented model. Fig. 6 depicts the average CPU utilization with different workloads. Overall, DataBank shows significantly lower CPU utilization than Cons_Queue and Cons_DF. For example, in the case that all three applications concurrently run (ALL), DataBank shows 28% of CPU utilization while Cons_Queue and Cons_DF show 65% (2.3 times) and 52% (1.9 times) of utilization, respectively. By adopting the producer-oriented model, DataBank significantly reduces the execution overhead of context monitoring dataflow for data passing, memory allocation and deallocation, and buffer management. Such lower CPU cost enables the smartphone to support more number of context monitoring applications concurrently. Cons_DF shows slightly better performance than Cons_Queue. This is because Cons_DF reduces inter-operator communication overhead by using function calls rather than message queues. However, DataBank achieves much better performance than Cons_DF.

C. *Performance Benefit Breakdown*

We quantify and analyze the performance benefits by using DataBank. As described in Section IV, DataBank improves performance by reducing the execution overhead in three ways: (1) shared data management, (2) in-place writing, and (3) lazy data delivery. To identify the effect of individual factors, we implemented different versions of our method in which we apply each improvement in turn on Cons_DF. These versions are named as follows:

- *Shared*: interposes a specialized window operator that performs the shared data management between a producer and multiple consumers.
- *SharedInPlace*: the same as above, but producers write their processing results directly on their internal buffer.
- *DataBank*: the version that takes all the three advantages of DataBank.

Fig. 7 shows the relative CPU utilization of the variants, regarding the CPU utilization of Cons_DF as 1.0. This shows the relative performance improvement of each variant compared to Cons_DF. The difference between Cons_DF and Shared represents the effect of the shared data management. The effect is most clearly observed in Activity; with Shared, the relative CPU utilization decreases by 0.23. In the Activity dataflow, accelerometer data are shared by tens of feature extraction operators such as FFT and correlation. Shared eliminates the redundant data management by multiple feature extraction operators. In the case of Place, the operators in the dataflow graph only have a single consumer, thus takes no benefit from the shared management.

The effect of in-place writing is shown by the difference between Shared and SharedInPlace. With SharedInPlace, the relative CPU utilization further decreases by up to 0.28 compared to that of Shared. The effect is clearly shown in all workloads. For example, the relative utilization of ALL decreases from 0.98 to 0.74; the absolute CPU utilization is reduced from 51.8% to 39.1%. The in-place writing reduces the repetitive memory allocation that results in frequent garbage collection in Android.

Finally, we can see the effect of lazy data delivery comparing SharedInPlace to DataBank. DataBank further reduces the relative CPU utilization by up to 0.3 compared to SharedInPlace. The effect becomes larger as data passes between operators are more frequent; for example, the effect is large in the cases of Sound and Place that start from a high-rate sound sensing operator.

D. Scalability

In this section, we evaluate the scalability of DataBank. For the evaluation, we generate dataflow graphs in the form of a full binary tree, where the root node is an 8 kHz sound sensing operator and the other nodes are processing operators. We use two types of processing operators, i.e., hamming window and FFT operators, each of which represents simple and complex processing operators, respectively. The window and sliding window size of the operators are set to 512.

Fig. 8 shows the average CPU utilization of DataBank and Cons_Queue while increasing the depth of the tree. DataBank shows much less CPU utilization than Cons_Queue even when there are a number of operators. For both types of operators, it reduces the CPU utilization by 24% when the depth is 5 and the number of operators is 63. The relative gain of DataBank is larger when the processing logics of operators are simpler. When the depth of the tree is 5, the relative CPU utilization of DataBank is 70% smaller than that of Cons_Queue for simple hamming window operators and 32% smaller for complex FFT operators. Note that even for the complex FFT operators the performance gain of DataBank is considerable.

VI. CONCLUSION

In this paper, we proposed a novel dataflow execution method for mobile context monitoring applications. Our method exploits a new producer-oriented execution model to process context monitoring dataflow in a highly optimized way. To realize the model, we develop DataBank, an execution container that takes charge of the management and delivery of the output data for the associated operator. Through experiments, we show that DataBank reduces CPU consumption by more than 50% compared with the methods based on the conventional consumer-oriented model. It shows that the producer-oriented model and DataBank technique clearly reduces the execution overhead, utilizing the unique characteristics of context monitoring dataflow. The performance improvement enables smartphone users to double the number of executable context monitoring applications.

ACKNOWLEDGEMENTS

This work was supported by a Korea Research Foundation Grant (KRF-2008-220-D00113) and the National Research Foundation of Korea grant (No. 2011-0018120) funded by the Korean Government.

REFERENCES

[1] D. Carney, U. Cetintemel, M. Cherniak, C. Convey, S. Lee, G. Seidman, et al., "Monitoring streams: a new class of data management applications," Proc. VLDB, 2002.

[2] R. Motwani, J. Window, A. Arasu, B. Babcock, S.Babu, M. Data, C. Olston, J. Rosenstein, and R. Varma, "Query processing, approximation and resource management in a data stream management system," Proc. CIDR, 2003.

[3] L. Girod, Y. Mei, R. Newton, S. Rost, A. Thiagarajan, H. Balakrishnan, and S. Madden, "XStream: A signal-oriented data stream management system," Proc. ICDE, 2008.

[4] H. Lu, W. Pan, N. D. Lane, T. Choudhury, and A. T. Campbell, "Soundsense: scalable sound sensing for people-centric applications on mobile phones," Proc. MobiSys, 2009.

[5] S. P. Tarzia, P. A. Dinda, R. P. Dick, and G. Memik, "Indoor Localization without Infrastructure using the Acoustic Background Spectrum," Proc. MobiSys, 2011

[6] L. Bao, and S.S. Intille, "Activity Recognition from User-Annotated Acceleration Data," Proc. Pervasive, 2004.

[7] K. Lorincz, B. Chen, G. W. Challen, A. R. Chowdhury, S Patel, P. Bonato, and M. Welsh, "Mercury: a wearable sensor network platform for high-fidelity motion analysis," Proc. SenSys, 2009.

[8] 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," Proc. MobiSys, 2008.

[9] H. Lu, J. Yang, Z. Liu, N. D. Lane, T. Choudhury, and A. T. Campbell, "The Jigsaw continuous sensing engine for mobile phone applications," Proc. SenSys, 2010.

[10] S. Kang, Y. Lee, C. Min, Y. Ju, T. Park, J. Lee, Y. Rhee, and J. Song, "Orchestrator: An Active Resource Orchestration Framework for Mobile Context Monitoring in Sensor-rich Mobile Environments," Proc. PerComm, 2010.

[11] B. Gedik, H. Andrade, K. Wu, P. Yu, M. Doo, "SPADE: The System S Declarative Stream Processing Engine", Proc. SIGMOD, 2008.

[12] T. Park, J. Lee, I. Hwang, C. Yoo, L. Nachman, and J. Song, "E-Gesture: A Collaborative Architecture for Energy-efficient Gesture Recognition with Hand-worn Sensor and Mobile Devices", Proc. SenSys, 2011.

[13] M. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan, "Odessa: Enabling Interactive Perception Applications on Mobile Devices," Proc. MobiSys, 2011.

[14] S. Kang, J. Lee, H. Jang, Y. Lee, S. Park, and J. Song, " A Scalable and Energy-efficient Context Monitoring Framework for Mobile Personal Sensor Networks ", IEEE Transactions on Mobile Computing (TMC), Vol. 9, No. 5, pp. 686-702, May 2010.

[15] Y. Lee, S.S. Iyengar, C. Min, Y. Ju, S. Kang, T. Park, J. Lee, Y. Rhee, and J. Song, "MobiCon: A Mobile Context-Monitoring Platform", to appear in Communicatoins of the ACM (CACM), March 2012.