Exploiting Processor Heterogeneity For Energy Efficient Context Inference On Mobile Phones

Chenguang Shen, Supriyo Chakraborty, Kasturi Rangan Raghavan,
Haksoo Choi, and Mani B. Srivastava
Networked and Embedded Systems Lab
University of California, Los Angeles
{cgshen, supriyo, kasturir, haksoo, mbs}@ucla.edu

ABSTRACT
In recent years we have seen the emergence of context-aware mobile sensing apps which employ machine learning algorithms on real-time sensor data to infer user behaviors and contexts. These apps are typically optimized for power and performance on the app processors of mobile platforms. However, modern mobile platforms are sophisticated system on chips (SoCs) where the main app processors are complemented by multiple co-processors. Recently chip vendors have undertaken nascent efforts to make these previously hidden co-processors such as the digital signal processors (DSPs) programmable. In this paper, we explore the energy and performance implications of off-loading the computation associated with machine learning algorithms in context-aware apps to DSPs embedded in mobile SoCs. Our results show a 17% reduction in a TI OMAP4 based mobile platform's energy usage from off-loading context classification computation to the DSP core with indiscernible latency overhead. We also describe the design of a run-time system service for energy efficient context inference on Android devices, which takes parameters from the app to instantiate the classification model and schedules the execution on the DSP or app processor as specified by the app.

Categories and Subject Descriptors
C.1.4 [**Computer Systems Organization**]: Parallel Architectures—*Mobile processors*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Software libraries*

General Terms
Design, Measurement

Keywords
Context inference, digital signal processor, mobile phone

1. INTRODUCTION
The continuous stream of richly annotated real-time data made available by tapping into the spectrum of sensors on a smartphone has led to the emergence of a new class of context-aware apps. These apps typically employ a suite of machine learning algorithms to extract semantically meaningful inferences from sensory data. Unfortunately, these algorithms are computationally intensive and power hungry, and moreover these context-aware apps often need to run in the background, continually monitoring user contexts and behaviors for just-in-time feedback, notifications, and interventions. On current mobile platforms such always-on context-aware apps run on the main app processor, and represent a significant portion of overall workload and power consumption. According to Ju et al. [12], example apps consume 4% - 22% of CPU cycles to execute the inference, a situation that will only get worse as apps with even more sophisticated context inference capabilities emerge.

While apps on mobile devices run on the main app processor, i.e. the CPU (typically a high-end ARM processor core), under the hood, however, mobile processor chips are not simply app processors. Rather, they are sophisticated system-on-chips (SoCs) where the main app processors are complemented by embedded processors, digital signal processors (DSPs) and graphic processing units (GPUs) that handle specialized work such as media processing and low-level sensor I/O. However, these auxiliary processors are usually hidden from the programmer developing application software, and are instead limited to running prebuilt firmware provided by the platform manufacturer. Meanwhile, there is now a conscious effort being undertaken by various mobile processor vendors to expose the co-processor heterogeneity to the app developers and provide them with the ability to program the DSPs. The Qualcomm Hexagon DSP is a representative example of such embedded processors on mobile platform SoCs. The recently released Qualcomm Snapdragon 800 series SoC [3] includes a 600 MHz Hexagon DSP with custom programmability and operates in the ultra low power range. Likewise, the TI OMAP4 SoC [5] has a 500MHz C64x DSP and two low-end Cortex-M3 ARM cores on board.

In this paper, our contribution is twofold. First, we explore the energy and performance implications of off-loading the computation associated with machine learning algorithms in context-aware apps to DSP cores. We have implemented two frequently used classification algorithms and created their power consumption profiles on the DSP of the TI Pandaboard ES [6], which is a TI OMAP4 mobile processor based embedded board capable of running Android. Our results indicate that off-loading on the DSP produced a 17% decrease in measured board-level energy consumption, with negligible effect on application latency. To the best of our knowledge, this is the first work leveraging a powerful co-processor for context-aware inference. Second, to provide developers with the ability to easily develop and run continuous inference while leveraging the DSP, we propose the design of a run-time framework named `EINF` (`Efficient INFerence`). Prior work such as [9], [15] and [12] have conceptualized continuous context inference engines to support the building and execution of context-aware apps. However, each introduced a programming paradigm considerably different from that of Android hindering their adoption within the developer community. By contrast, `EINF` provides app developers with a context inference infrastructure and an associated programming model that maintains the familiar abstracts adopted by other Android system services. `EINF` is designed to be flexible in terms of configurability by allowing developers to subscribe to the service and specify parameters for the classification models for their context inference needs. The system service in turn instantiates the classifiers, and schedules them as tasks on the CPU or the DSP based on app-provided execution parameters and the system state.

2. RELATED WORK

Several researchers have recently explored leveraging low-power processors in mobile sensing apps. Priyantha et al. [16] used an external MSP430 microcontroller for off-loading frequent sampling of sensor data. Ra et al. [17] examined how to partition different modules of a sensing app between the app processor and the low-power processors to reduce the overall energy consumption. However, the low-power processors used in these experiments were low-end microcontroller class processors optimized for sampling and buffering, and not capable of meaningfully doing more complicated context inference. Moreover, such low-end processors cores are deeply embedded in the hardware fabric and unlikely to be exposed to app developers. Philosophically, such work targets off-loading of simple frequent tasks from the main app processor, whereas our work explores off-loading of computationally intensive tasks.

Among other work in this space is Lin et al. [14] which compared heterogeneous loosely coupled processor cores, e.g. Cortex-A9 + Cortex M3, with tightly coupled processor cores with identical instruction set architectures but different power-performance operating points, e.g. ARM big.LITTLE [10]. They proposed distributing both app and OS workloads over loosely coupled processors such as the TI OMAP SoC. They also implemented Distributed Shared Memory (DSM) between the Cortex-A9 and Cortex-M3 [13], which enables efficient data exchange between the two asymmetric processors. They proposed a set of compiler and runtime for programming transparency.

In commercial market, Apple's new iPhone5s comes with a dedicated M7 motion coprocessor [1] that continually measures sensor data and offers a API that provides apps with phone contexts. On the Android side there is the Renderscript API [4] which enforces a unified programming model where native code can be compiled and executed on different processors. Both have shown the industrial effort to leverage heterogeneous co-processors for off-loading computation in mobile platforms. Moreover, Google proposes its activity recognition API [2] offering a convenient interface for developers to access physical activity of the phone. `EINF` combines the above ideas of computation off-loading and adding programming support for context inference, but provides app developers with a higher level of configurability where they provide model parameters for inference classifiers.

3. OFF-LOADING INFERENCE TO DSP

We start by describing the off-loading of the *classification* phase of inference onto the DSP. In a typical context-aware app, the app running on the app processor subscribes to sensor data streams, and the hardware fabric, the device drivers, and system services work in concert to pass the sensor samples to subscribing apps. The app then extracts features to reduce the dimensionality of sensor data, and performs classification over the stream of extracted features to return context labels (i.e. which class the current context falls under). While feature extraction itself can be computationally intensive in many apps, our initial focus is on the classification.

3.1 Classification Algorithm

We have chosen Support Vector Machine (SVM) and Gaussian Mixture Model (GMM) as representative models to explore initially, since they are used by a variety of apps. Our implementation of SVM classification is based on LIBSVM [8]. The GMM classification code in C is based on Voicebox [7]. As DSP is still a resource-constrained processor, we have optimized the running time and memory footprint of the classification algorithms. Both algorithms use a set of mathematical operations such as multiplication, addition, and exponentiation. As DSPs typically do not have specialized floating-point processing units, we have transformed most of the floating-point arithmetic in the original implementations to be fixed-point, while largely preserving the accuracy of the results. This has led to reduced latency and run-time memory footprint. Even in DSPs that do support floating point, such as those found in the latest generation of Snapdragon processors, the memory constraints remain and moving away from floating-point to the more compact fixed-point representation is beneficial.

3.2 Experimental Platform

In order to verify the feasibility of off-loading classification algorithms to DSP and show potential gain in energy efficiency, we implemented the above mentioned classification algorithms on a TI Pandaboard ES [6] running Android 4.1.2 (JellyBean). The OMAP 4460 SoC on board includes a Cortex-A9 ARM CPU at 1.2GHz and a TI C64x low-power DSP at 500MHz. The DSP is running its own SYS/BIOS real-time OS (RTOS). In order to execute our implementations on the DSP, we use TI's RPC infrastructure called RPMsg, which enables IPC calls from the kernel running on the CPU to the RTOS on the DSP. Since the SVM and GMM implementations are in C, they can be compiled for native execution both on the CPU and the DSP with the latter being accessed by the apps using RPMsg.

For power measurement, we use two external Agilent 34410A digital multimeters connected to a Agilent E3631A DC power supply to measure the total energy consumption of the Pand-

Processor	Latency(SVM)	Latency(GMM)
CPU	1.320ms	0.043s
DSP	9.232ms	0.704s

Table 1: Performance overhead

aboard. We also use two on-board resistors to measure the energy consumption of the CPU and DSP subsystems.

3.3 Example Application

We use publicly available datasets to generate the classification models for latency and energy profiling. For SVM, we use the daily activity dataset collected by TU Darmstadt [11], which includes accelerometer features over a 7-day time period. We have trained an SVM model to classify the action scenario of each sample as one of {*dinner, commuting, lunch, work, undefined*}. Each sample is obtained over a 400ms window.

For GMM, we use the classical speaker recognition application, where a GMM model is trained for each speaker. For each sound clip a likelihood score is calculated which represents the probability of the sample being generated by the GMM. The speaker corresponding to the maximum likelihood is set as the output. We use the TIDIGITS dataset as the input, and a set of MFCC features on 3s sound clip at 8KHz is used as one sample.

3.4 Profiling Result

We first run the SVM and GMM classification code on CPU or DSP individually, showing the change in latency. Table 1 shows the profiling result for SVM and GMM. Here latency refers to the time to classify one sample. Although running the classification on DSP instead of CPU generates certain overhead in terms of latency, it is still well within the range that is acceptable from app perspective. Since the features of a sample are computed over a sliding window of sensor data, the actual classification is duty-cycled by the window size. The current classification result will be valid until the next complete window. Therefore the latency is acceptable as long as it is smaller than the window size. In the activity recognition app, the user can hardly discern the increase of latency from 1ms to 9ms because each sample is calculated from a 400ms sliding window. In the GMM app, the window length is 3s, therefore the execution on DSP is still sufficient because it takes less than 3s to recognize one clip.

Next, we explore the energy implication brought by off-loading computation to DSP with the intuition that despite the increased latency, the specialized ISA of the DSP will result in overall improvement in energy. Our first experiment measures CPU and DSP subsystem energy consumptions. To classify one sample using SVM, the CPU and the DSP use $1.229 * 10^{-3}J$ and $1.171 * 10^{-3}J$ of energy on average, respectively, indicating that off-loading classification to DSP offers a 4.72% reduce in subsystem energy consumption.

The second experiment profiles the energy saving of the entire platform, showing the total energy consumption of the Pandaboard with SVM classification continually running on CPU or DSP separately. Currently the RPC call to DSP brings overhead, and we are classifying 300 samples as a group to amortize the overhead. This is also common in real-world apps because the inference often operates on a sliding window, and the classification algorithms are running on blocks of samples and not on individual samples.. As we can see from Table 2, the Pandaboard consumes about 17% less energy when the continuous execution of classification is on the DSP instead of the CPU. If we subtract the board baseline power 1.993W, the DSP execution actually consumes about 50% less energy compared to the CPU. For a 24-hour period of running, off-loading classification to the DSP will save about $12.8 Watt \cdot Hour$ of energy. We note that the measured platform level energy saving is larger than the subsystem level saving (4.72%) in the first experiment. This is because the continuous execution on CPU in the second experiment also activates other modules on the board, such as the voltage regulation and thermal control modules, adding extra energy consumed by subsystems other than the CPU. Moreover, in continuous execution mode the frequency and voltage of CPU will be increased by the Dynamic Voltage Frequency Scaling (DVFS) governor in the Android OS to accommodate the high CPU utilization, resulting in more board-level energy consumption, whereas in the first case the execution is not continuous so that such increase will not be triggered.

Processor	Power	24-hour Energy
CPU	3.062W	73.488 $Watt \cdot Hour$
DSP	2.528W	60.672 $Watt \cdot Hour$

Table 2: Energy consumption of a Pandaboard running continuous classifications

3.5 Discussion

Although we are using the TI OMAP4 in our experiment, the concept of leveraging low-power DSP is not limited to this specific architecture. Recently Qualcomm has also opened access to its Hexagon DSP in the Snapdragon SoC, which is used by more mainstream smartphones. We believe that off-loading computations to DSP for energy efficiency would be applied in a broader contexts for more applications, and our proposed framework described in Section 4 is necessary to simplify the development of these apps.

Note that the 17% empirical energy saving in Section 3.4 is achieved by off-loading *unoptimized* classification code to the DSP. Our ongoing work focuses on off-loading feature extraction, as well as optimizing these algorithms using signal processing libraries and primitives provided by the Qualcomm DSP, which would potentially offer more energy saving.

4. FRAMEWORK DESIGN

A framework for building and executing context-aware apps should ideally (1) provide improved energy efficiency by off-loading (part of) the execution of inference onto the DSP, while maintaining latency overhead that are acceptable; (2) introduce no additional burden on the developers to program the DSP; (3) allow programmers to specify execution parameters of the inference and enforce them.

The profiling results in Section 3.4 suggest that DSP is a promising target for off-loading the classification execution, since overall energy saving is achieved with acceptable latency overhead. To let the programmer access the DSP, we propose a run-time framework named `EINF`, which adds a *ClassificationService* to the Android framework. Apps written in Java and running on Android's Dalvik VM can subscribe to ClassificationService in a fashion similar to the way they access other system services currently available in Android.

Given that the RPC call to DSP is typically in standard C/C++, the naive solution of leveraging the DSP would be to create an RPC wrapper library and allow the apps to use the library to talk directly to the DSP. However, to perform efficient scheduling, the OS needs to have global knowledge of the various app requests. The ClassificationService in EINF acts as an intermediary broker which receives requests for instantiating background context classifiers from all the apps. ClassificationService instantiates and schedules the classifications based on app-provided execution parameters, such as latency and energy, as will be discussed in Section 4.2. This ClassificationService centered architecture can in future be extended to incorporate other scheduling mechanisms that would take into account other platform-specific information, e.g. battery state.

4.1 Architecture of `EINF`

The architecture of `EINF` is shown in Figure 1. The ClassificationService consists of three parts: a Java part and a JNI part running in the app process, and a C++ part running in the system process. The C++ part implements a library of popular classification algorithms, such as SVM and GMM, and will be initialized together with other system services during boot-up, loading the implementations into the memory of both the CPU and the DSP.

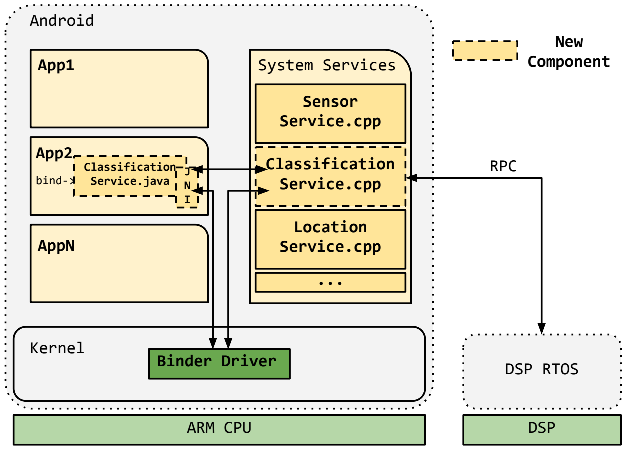

Figure 1: Architecture of `EINF`

On the app side, programmer writes Java code to bind to the ClassificationService. *ClassificationService.java* provides a wrapper of the JNI part, and the JNI code in the app process communicates with the *ClassificationService.cpp* in the system process via the binder kernel driver. A feature of `EINF` is that programmer specifies the parameters of inference classifier as opposed to providing code implementing the classifier. Compared to Reflex [13], where app code is compiled and executed on CPU and peripheral processors, `EINF` provides a library of algorithms optimized and compiled for both CPU and DSP, while the developer will continue to use the Android programming model to subscribe to system service and get event update. This works well since most context-aware apps use only a small set of common classification algorithms, and differ only in the model parameters and features used. Instead of requiring the developer to implement these algorithms, providing an optimized library not only ensures the efficiency of the execution, but also enables the OS to better schedule different workloads.

Parameter	Possible Value
Priority	ENERGY_FIRST, LATENCY_FIRST, FLEXIBLE
Target	CPU, DSP, ANY
Latency	Real number (Second)
Energy consumption	Real number ($W \cdot Hour$)
Accuracy	Real number between 0 and 1

Table 3: Possible execution parameters

To use the ClassificationService, the app first calls *setModel (Model x)* to initialize the desired classification model on both CPU and DSP. Then it can use *getPrediction(Sample d, ExecParams p)* to push new samples to the service and get the classification result back. The C++ part takes care of the actual execution, whether on CPU, or on the DSP through RPC, according to the execution parameters *ExecParams* described in Section 4.2. We are currently using TI's RPMsg for RPC, which is synchronous and will add overhead to the calling app thread. The event-based message passing implemented by Reflex to achieve asynchronous communication among heterogeneous processors would be a valuable asset to our framework, and could also help reduce RPC overhead in the off-loading.

4.2 Execution Parameters

As discussed in Section 3.4, the execution on DSP although energy efficient incurs a latency overhead. Given that the choice of CPU and DSP impacts latency and energy in app dependent ways, the ClassificationService needs to make scheduling decisions that are informed both by guidance from app and run-time evidence. For example, a critical health-monitoring app should be executed with minimum latency, regardless of the energy consumption. Other kinds of apps, such as a data logger, do not have strict requirement on latency and can therefore be executed on the DSP to save energy.

The variety of app requirements and different characteristics of CPU and DSP motivate the importance of letting app specify execution parameters to provide evidence for scheduling. When an app invokes *getPrediction(Sample d, ExecParams p)* method provided by ClassificationService for inference result, it also specifies one or more execution parameters as *ExecParams*, with the possible values shown in Table 3. The service then schedules the model execution on either the CPU or the DSP based on these parameters. For example, the app can pass in a single *Priority* parameter. If latency is the primary concern (*LATENCY_FIRST*), the classification should be executed on the CPU, otherwise DSP should be used for energy efficiency (*ENERGY_FIRST*). It can also set Priority to be *FLEXIBLE*, and the service will choose idle processors to schedule the execution. Another possibility is that programmer directly sets the *Target* to be CPU or DSP, and the Target preference will override other parameters. Without those parameters, ClassificationService cannot be aware of the app requirements. Moreover, the service tries to accommodate detailed inference requirements such as latency, accuracy, or energy consumption, using heuristics from previous execution. To enforce those requirements, the classification algorithms implemented in the service expose knobs for controlling the execution accuracy and latency. For instance, floating-point computation will result in better accuracy, but longer latency, therefore the choice of floating-point or fixed-point arithmetic is used to control the execution.

5. CONCLUSION AND DISCUSSION

In this paper, we have shown that considerable energy savings can be achieved by off-loading the execution of classification algorithms to the DSP. The `EINF` framework offers a unified programming model for programmers to develop context-aware apps, and exploit both the CPU and DSP for the execution. It is also the first attempt to make classification an OS service optimized for execution on heterogeneous processors. We expect to release the `EINF` implementation in the near future.

The experience with this work also motivates the following questions about architectural support for sensor data acquisition and processing needed in mobile processors for efficient support of continually running context-aware apps:

Direct access to sensors from DSP Currently, the apps fetch sensor data through the app processor prohibiting longer sleep cycles. We envision that in the future the DSP too will support interfaces to gain direct access to sensor data, allowing longer sleep duration for the app processor and also off-loading of the feature extraction to the DSP.

Use DSP for learning In `EINF` we only consider off-loading the classification phase to DSP. However, due to its frugal energy consumption the DSP can also be used for the classifier training. An ideal framework would thus be one where the DSP collects sensor data in the background, and reinforces the model periodically. This will further automate the building and execution of context-aware apps, with no extra burden on the app processor. Moreover, on-device learning will enable more immediate personalization and customization of models in response to changing contexts.

Profile other workloads We plan to explore workloads other than context-aware inferences to better characterize the energy profile of the DSP and other co-processors available, and conduct app-level energy profiling to show how complete apps can benefit from the off-loading.

6. ACKNOWLEDGMENTS

This material is based in part upon work supported by the NSF under award # 0910706, 0905580, and 1029030. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF. The authors would like to thank Lucas Wanner and Newton Truong for their help on this work.

7. REFERENCES

[1] Apple iPhone5s. `http://goo.gl/1LUSj0`.
[2] Google Activity Recognition API. `http://goo.gl/mYJn84`.
[3] Qualcomm Snapdragon. `http://goo.gl/ZFTm0`.
[4] Renderscript. `http://goo.gl/W1jGz`.
[5] TI OMAP. `http://goo.gl/9Z5R4`.
[6] TI Pandaboard. `http://goo.gl/ujdiL`.
[7] VOICEBOX: Speech Processing Toolbox for MATLAB. `http://goo.gl/wakDY`.
[8] C.-C. Chang and C.-J. Lin. Libsvm: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):27, 2011.
[9] D. Chu, N. D. Lane, T. T.-T. Lai, C. Pang, X. Meng, Q. Guo, F. Li, and F. Zhao. Balancing energy, latency and accuracy for mobile sensor data classification. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems (SenSys'11)*, pages 54–67. ACM, 2011.
[10] P. Greenhalgh. Big. little processing with arm cortex-a15 & cortex-a7. *ARM White Paper*, 2011.
[11] T. Huynh, M. Fritz, and B. Schiele. Discovery of activity patterns using topic models. In *Proceedings of the 10th international conference on Ubiquitous computing (UbiComp'08)*, pages 10–19. ACM, 2008.
[12] Y. Ju, Y. Lee, J. Yu, C. Min, I. Shin, and J. Song. Symphoney: a coordinated sensing flow execution engine for concurrent mobile sensing applications. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems (SenSys'12)*, pages 211–224. ACM, 2012.
[13] F. X. Lin, Z. Wang, R. LiKamWa, and L. Zhong. Reflex: using low-power processors in smartphones without knowing them. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII),*. ACM, March 2012.
[14] F. X. Lin, Z. Wang, and L. Zhong. Supporting distributed execution of smartphone workloads on loosely coupled heterogeneous processors. In *Proceedings of the 4th Workshop on Power-Aware Computing and Systems (HotPower'12)*, October 2012.
[15] H. Lu, J. Yang, Z. Liu, N. D. Lane, T. Choudhury, and A. T. Campbell. The jigsaw continuous sensing engine for mobile phone applications. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems (SenSys'10)*, pages 71–84. ACM, 2010.
[16] B. Priyantha, D. Lymberopoulos, and J. Liu. Littlerock: Enabling energy-efficient continuous sensing on mobile phones. *Pervasive Computing, IEEE*, 10(2):12–15, 2011.
[17] M.-R. Ra, B. Priyantha, A. Kansal, and J. Liu. Improving energy efficiency of personal sensing applications with heterogeneous multi-processors. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing (UbiComp'12)*, pages 1–10. ACM, 2012.