Received March 10, 2015, accepted March 30, 2015, date of publication April 14, 2015, date of current version April 25, 2015.

Digital Object Identifier 10.1109/ACCESS.2015.2422783

Application-Specific Customization of Dynamic Profiling Mechanisms for Sensor Networks

LU DING[1,2], (Member, IEEE), ADRIAN LIZARRAGA[3], (Member, IEEE), ASHISH SHENOY[1,4], ANN GORDON-ROSS[5], (Member, IEEE), SUSAN LYSECKY[3,6], (Member, IEEE), AND ROMAN LYSECKY[3], (Senior Member, IEEE)

[1]University of Arizona, Tucson, AZ 85721, USA
[2]Western Digital Technolc Inc., Irvine, CA 92612, USA
[3]Department of Electrical and Computer Engineering, University of Arizona, Tucson, AZ 85721, USA
[4]Riverbed Technology Inc., San Francisco, CA 94107, USA
[5]Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32611, USA
[6]Zyante Inc., Los Gatos, CA 95033, USA

Corresponding author: A. Lizarraga (adrianlm@email.arizona.edu)

This work was supported by the National Science Foundation under Award CNS-0834080 and Award CNS-0834102.

ABSTRACT To reduce the complexity associated with application-specific tuning of wireless sensor networks (WSNs), dynamic profiling enables an accurate view of an application's runtime behavior, such that the network can be reoptimized at runtime in response to changing application behavior or environmental conditions. However, the dynamic profiling must be able to accurately capture application behavior without incurring significant runtime overheads. Since application- and sensor-specific constraints dictate the profiling requirements and tolerated overheads, designers require design assistance to quickly evaluate and select appropriate profiling methodologies. To increase designer productivity, we formulate profiling methodology design guidelines based on extensive evaluation and analysis of a variety of profiling methodologies suitable for dynamically monitoring WSNs with respect to network traffic overhead, power, and code impacts associated with each method. While energy consumption increases are reasonable, ranging from 0.5% to 2.6%, network traffic, code size, and computation time overheads can be as high as 66.2%, 75.9%, and 136.6%, respectively. Our results show that these overhead variations are highly application specific, and a single profiling method is not suitable for all types of application behavior, thus necessitating, application-specific profiling methodology customization. To facilitate rapid development of these profiling methodologies, we present a profiler-customization methodology consisting of a code generator module, overhead estimation module, and profile data management module. Using our profiling-customization methodology, designers can rapidly evaluate the overhead of different profiling methodologies, and automatically integrate the most appropriate methodology into the application at design time.

INDEX TERMS Adaptive algorithm, dynamic profiling and optimization (DPOP), dynamic profiling, embedded software, wireless sensor networks (WSN).

I. INTRODUCTION

The rapid proliferation of increasingly capable wireless sensor networks (WSNs) with massive numbers of constituent nodes/platforms has enabled a wide range of new application possibilities. With each different application, designers have a unique set of application requirements and constraints, such as lifetime, responsiveness, reliability, throughput, etc., that must be adhered to. For example, a disaster response application requires high responsiveness and reliability to survey damage or detect survivors, but may only require a lifetime of days or weeks. Conversely, an automated vineyard irrigation system would have a longer lifetime requirement since this system would operate on the order of years.

To achieve various application requirements, designers can tune/specialize configurable node-/platform-level parameters, such as voltage level, operating frequency, sensing frequency, processor mode, communication baud rates [26], etc. Designers can also consider numerous protocol-level design choices, such as power cycling to sensing units [43], data aggregation and filtering [24], etc. While the effects that various parameter configurations have on high-level application requirements/constraints (i.e., *design metrics*) have been well documented, balancing numerous competing design metrics remains challenging (e.g., increasing the sensing frequency (responsiveness) typically reduces lifetime).

To further complicate WSN design, design metrics are highly application specific and accurately predicting application behavior at design time is extremely challenging. Tuning the underlying platform to inaccurate application behavior estimations can yield either suboptimal results or negatively impact design metrics.

To alleviate some of the complexity associated with application-specific tuning and design of WSNs, we have begun to develop a fundamental dynamic profiling and optimization (DPOP) framework [7], [8], [50]. DPOP not only reduces designer effort but also increases accessibility to application experts (i.e., platform users) by abstracting much of the underlying platform-specific knowledge. While platform designers are typically engineers with the requisite understanding of the hardware and software required to develop the WSN platform, application experts that use these platforms, as defined in [31], are often not engineers, but rather scientists, biologists, or teachers that simply provide an initial software implementation of a WSN application. Thus, to maximize the applicability and usability of design frameworks, these frameworks must be cognizant of wildly varying usage scenarios, and provide design tools and interfaces for any end users (e.g., platform designers, application experts, etc.

To address these challenges, DPOP employs a flexible and reconfigurable profiling methodology for application-specific tuning and design of WSNs. The profiling methodology is composed of five configurable modules, where each module has multiple options. Users can easily create a customized profiling methodology based on application-specific behavior to collect sensor status and application behavior data during runtime. This profile data can then be used by platform designers to optimize the sensor network architecture or by application experts to analyze and adjust application behavior and/or functionality. DPOP also estimates incurred profiling overheads, which can enable users to evaluate the profiling's impact on design metrics. DPOP's features eliminate the need for application experts to possess any technical expertise while enabling an accurate view of the deployed application's behavior, precluding the challenging and lengthy effort to create an accurate simulation environment. Additionally, this dynamic profiling enables monitoring how the application responds to changes in environmental conditions or changes in the underlying WSN (e.g., failed nodes, newly introduced nodes, changes to the network topology, etc.), which provides opportunities for dynamically re-optimizing and updating the underlying platform accordingly.

In prior work, we investigated dynamic profiling of sensor-based systems [8] with initial efforts focused on evaluating several profiling methods for dynamically monitoring sensor-based platforms and analyzing the associated network traffic, energy consumption, and code impacts of these profiling methodologies. Network traffic overheads ranged from 7.9% to 32.2%, while energy and code size overheads remained reasonable with a maximum overhead of only 0.06 milliamp-hours and 1.4 kilobytes (or 3.5%), respectively. In other works, we have begun to consider various optimization methodologies [3], [7], [49] to quickly and efficiently determine an appropriate system configuration, with dynamic optimization of sensor nodes using the DPOP framework, resulting in up to an 83% improvement in overall design costs compared to a statically optimized node configuration.

Whereas these prior works provided sound foundations towards a complete DPOP framework, many challenges remain to be addressed. As compared to these prior works, in this paper we present novel contributions and significant enhancements to the existing DPOP framework with respect to the methods and tools targeted towards dynamic profiling. Specifically, the contributions include integrated tools to enable an application expert to explore various profiling methodologies, estimate the corresponding overheads incurred by these methodologies, and automatically generate the instrumented application code to include the desired profiling methodology's functionality. Section II and Section III provide necessary background in related works and a short overview of the existing DPOP framework, while the remainder of the sections elaborate on this paper's major contributions. Section IV enumerates the various profiling metrics observed and strategies employed within the DPOP framework and how these configurations impact application profiling. Section V details the components within the profiler module, and Section VI presents experiments across a variety of profiling methodologies and benchmark applications to evaluate the resulting overheads in terms of network traffic, battery, code size, and computational overhead. Finally, Section VII summarizes our conclusions and future work.

II. RELATED WORKS

Previous work on WSN design and optimization presents various approaches for estimating WSN behavior and/or performance. These approaches can be broadly classified as offline (static, design time) or online (dynamic, runtime). Offline approaches, such as Beretta et al. [23], leverage analytical models of network or node behavior to estimate various design metrics at design time. In an alternative approach, Bai et al. [32] augments WSN applications with statistical models that are generated offline to enable estimation of various performance metrics, such as lifetime, at runtime. However, since these performance models are generated offline, these models are typically only valid for a fixed set of network configurations. Runtime optimization of WSNs requires online profiling approaches that can accurately collect application behavior while considering dynamic application execution characteristics.

Dynamic optimization relies on accurate profiling results collected at runtime. To the best of our knowledge, there exists no holistic, accurate, robust method to capture external application-specific stimuli. While many dynamic profiling techniques exist, these techniques are highly system-specific, low level, not generally portable, and cannot provide real time status of nodes within a network. For example, working set analysis [1] monitors the current set of executing instructions to determine changes in system execution behavior. Changes in cache requirements can be determined using counters embedded within the cache structure [47], and simple methods can observe idle periods [17]. Whereas idle period observation is a generalized, high-level mechanism to profile a system when applied to WSNs, little information on overall system behavior can be inferred.

EnviroLog [35], which is used to achieve repeatability of asynchronous events in WSNs, logs all issued function calls and the call's parameters to record module events. Marionette [30] and L-SNMS [19] are tools that allow a PC to access the functions and variables of a statically-compiled program executing on a sensor node at runtime. However, such low level information about function calls and variables cannot be easily used to analyze status of the nodes and network. SNMS [21] uses a querying and logging system to collect user-selected attributes/behavior and unexpected events. This method requires users to manually retrieve this information, since each node maintains a local log, and thus this method is not suitable for dynamic optimization. PAD [56] uses a lightweight packet marking scheme, inference model, and inference engine to generate a fault report for the entire WSN, which does not provide specific data about node status and application behavior.

The distributed nature of WSNs complicates adoption of existing profiling methods. One of the major challenges of dynamically profiling sensor-based platforms is accurately capturing application behavior without incurring significant overhead or significantly altering system behavior. In many simulation frameworks, application experts must specify application-behavior via an input file [28], a mathematical model [18], or through synthetic data generation [57]. WSN emulators can enable control of particular sensor nodes providing controllability and repeatability for testing, evaluating, and comparing networks [13]. However, an emulation and profiling framework is better suited to developing and benchmarking WSNs, since this framework would incur significant overhead in deployed systems. Application layer tools can also be used to assist users in developing complex applications on heterogeneous WSNs. An important inclusion in these tools is support for runtime monitoring of the deployed WSN, and providing data collection and visualization capabilities. While these runtime monitoring techniques are applicable to monitoring a deployed system, as the authors themselves point out, the overhead of these techniques can be significant [34].

Since battery life is a dominant constraint in many WSNs, many simulation frameworks integrate power estimation. Discrete-event simulators typically estimate energy consumption by tracking transitions in components (e.g., CPU, transceiver, etc.), operating modes, and using known current and voltage values associated with each mode to incrementally determine the component's dissipated power. The SENSE [20] and PAWiS [15] simulators, for example, incorporate this component power dissipation model with a simple battery model in order to simulate each sensor node's power profile. In another power profiling technique, current draw is pre-measured for a variety of CPU modes as well as the sensor board and EEPROM [55]. These values are integrated into to an event-driven simulator for TinyOS applications [44] to determine how much time each component spends in a particular operation mode, thereby calculating energy consumption of individual nodes. Quanto [45] similarly tracks power by integrating current consumption information into the core OS and driver files of TinyOS. By recording the transfer of hardware power states and high-level activities of all nodes in the network, Quanto can provide a detailed breakdown of energy consumption over time for individual nodes or the entire network. Similarly, network-level energy consumption can be estimated through a combination of COOJA and MSPSim [25]. In addition to node-level power estimation, which similarly combines time spent in different operating modes with pre-measured consumption of these components, a network simulator is integrated into the framework to account for communication between nodes based on external emulation of the radio chip, sensor boards, and flash memory.

Simulation frameworks can also provide information pertaining to low-level hardware and network parameters. TOSSIM [44] tracks statistics, such as packet loss, CRC failure rates, as well as the length of send queues. Avrora [11] monitors hardware interrupts, I/O registers, and memory usage. The ATEMU framework [28] additionally provides platform designers with insight into the number of backoffs performed after transmission collisions.

These simulators can be broadly categorized by the granularity of simulation available, scalability to larger networks, as well as the underlying models utilized within the framework. While Avrora and ATEMU are cycle accurate, instruction level simulators, TOSSIM is an interrupt level discrete event simulator. Furthermore, the addition of energy models to these simulators, like AEON [42] to Avrora and PowerTOSSIM to TOSSIM, enables estimation of energy consumption.

While it is clear that simulation frameworks are an essential part of a designer's tool kit to evaluate and test prototype designs, designers must still make assumptions or predictions about the deployment environment, which can lead to inaccurate application behavior. Furthermore, Handziski et al. [54] recognized that the lack of a wide range of protocol models also adds uncertainty to simulation results. To avoid many of these challenges and error-prone methods, our work performs profiling during runtime after the WSN has been deployed into the WSN's intended environment.

Each of the presented related works explores a subset of the functionality required for runtime profiling of WSNs in the

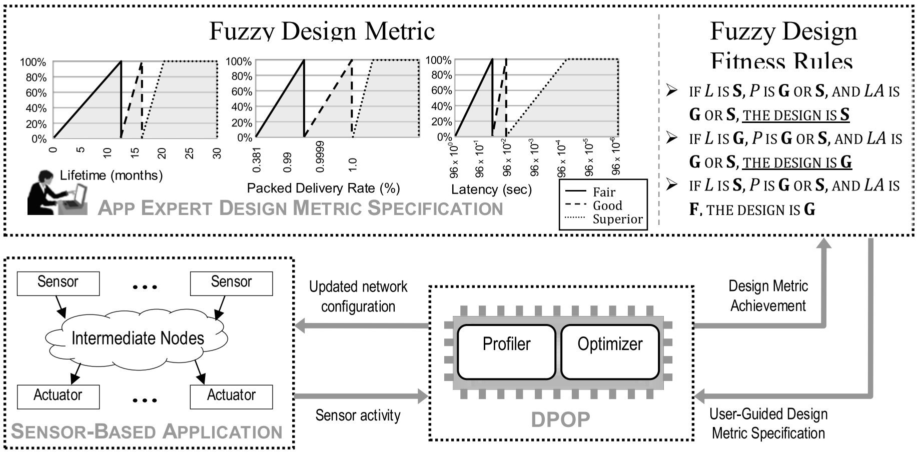

Fuzzy Design Metric
Lifetime (months)
Packed Delivery Rate (%)
Latency (sec)
Fair
Good
Superior
APP EXPERT DESIGN METRIC SPECIFICATION
Fuzzy Design Fitness Rules
IF L IS S, P IS G OR S, AND LA IS G OR S, THE DESIGN IS S
IF L IS G, P IS G OR S, AND LA IS G OR S, THE DESIGN IS G
IF L IS S, P IS G OR S, AND LA IS F, THE DESIGN IS G
Sensor
Intermediate Nodes
Actuator
SENSOR-BASED APPLICATION
Updated network configuration
Sensor activity
Profiler
Optimizer
DPOP
Design Metric Achievement
User-Guided Design Metric Specification

FIGURE 1. Dynamic profiling and optimization (DPOP) framework highlighting various tasks including specifying design metric evaluation equations and assigning design metric weights to indicate the relative importance between each design metric.

DPOP framework. The various models and techniques used within these works can be leveraged to design an accurate profiling mechanism with low overhead. However, given that the analysis of profiling techniques in these works is not always a primary focus, but rather just a small part in the larger development process, it remains difficult to determine which profiling method is most suitable, or how to customize existing methods, for a specific WSN application. The task of properly comparing and evaluating the performance of profiling approaches is additionally hindered by the large diversity in implementations. That is, the metrics for evaluating the various approaches are largely dependent on the underlying application and the type of parameters that are profiled.

III. DPOP FRAMEWORK

The DPOP framework [7], [8], [50] supports the profiling and optimization of WSNs given application expert-specified design metrics. Note that while we provide an overview of the entire DPOP framework in this section, this paper focuses on the profiling methodologies supported by the Profiler module, which we discuss later in this section. Fig. 1 illustrate the proposed DPOP framework, which is composed of three main components: the *Sensor-Based Application*, the *Application Expert Design Metric Specification*, and the *DPOP module*.

The *Sensor-Based Application* is the physical deployment of the application within the intended environment and consists of sensor nodes, intermediate processing and routing nodes, and actuator nodes, working together to achieve the desired application functionality.

In addition to how the underlying platform implements the application, an application expert may also be interested in the performance of the platform in terms of high-level design metrics such as the expected lifetime of a node or sensor network, the time required to process a single packet, or the time required to process and respond to a sensor event.

The *Application Expert Design Metric Specification* allows an application expert to define which design metrics are of importance to a particular application, and of those design metrics, what are the acceptable or unacceptable values of each, thereby providing a method to interpret the resulting system achievement within the context of a given application. First, for each design metric, an application expert creates a fuzzy-logic inspired classification function that relates a raw metric value (i.e., lifetime of 2 months) to a fuzzy classification term. Although the selection of which fuzzy terms are utilized for a given system could be arbitrarily defined by the application expert, the current DPOP framework utilizes the following three classifications for specifying the fitness of individual design metrics: *Fair*, *Good*, and *Superior*, as shown in Fig. 1. Using this classification mechanism, an application expert simply needs to specify the range of values that correspond to a *Fair*, *Good*, and *Superior* design for that given metric. As application experts are unlikely to be experts in optimization methods, this fuzzy classification scheme provides a relatable method for mapping design metric values to relative rankings using common terminology. To determine the relative importance of each design metric and how the metrics relate to the overall design quality, the application expert specifies a set of fuzzy design fitness rules. These fuzzy design fitness rules are specified using English sentences that map the fuzzy classifications of the design metrics to a fuzzy classification of the overall design.

The *DPOP Module* is a separate component—implemented within the base station node or as a separate sensor node—dedicated to the profiling and optimization of the underlying sensor-based platform, as the platform interacts within the intended environment.

The *Optimizer Module* is provided with a set of configurable parameters for a given platform. In our case, we considered the Crossbow IRIS platform [39] and defined the processor frequency, processor voltage, RF output power, RF frequency, and data rate as configurable parameters.

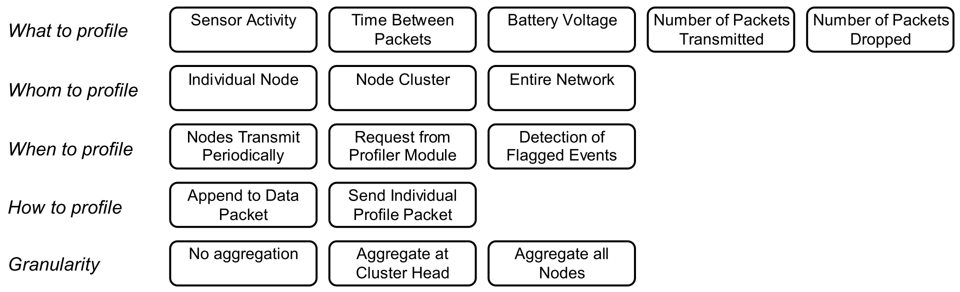


FIGURE 2. **Profiling metric overview, including what, whom, when, and how to profile, and the profiling granularity, which can be used for different profiling methodologies, and the metrics' configuration options.**

The *Optimizer Module* is responsible for evaluating possible node configurations (particular settings for each configurable parameter) within the design space to determine which configuration best meets the application requirements as specified during *Application Expert Design Metric Specification*. Given the design metric evaluation specification and dynamic profile data, the *Optimizer Module* uses an equation-based estimation methodology that estimates each design metric using both the node configuration and profile data. The *Optimizer Module* then explores the design space by evaluating various node configurations to determine which node configuration is best suited for a given application (i.e., the configuration yielding the lowest overall design cost). Details of the implementation of the *Optimizer Module*, design metric specification, and evaluation of the underlying optimization algorithms can be found in [3], [5], and [6]. As the application expert revises design metric goals, or as the application stimuli changes, the *Optimizer Module* will re-optimize the platform by selecting a configuration to adapt to these changes.

The *Profiler Module*, which is the focus of this work, is responsible for dynamically monitoring the application behavior while the sensor-based system is deployed, tracking statistics of interest to the application expert, and providing these observations to the *Optimizer Module* to determine how to configure the underlying platform to best meet user-defined goals. The following sections detail the types of metrics observed by the profiler, the mechanisms utilized to dynamically collect the corresponding profile data at runtime, and the underlying implementation of the profiler module.

IV. DYNAMIC PROFILING METRICS AND CONFIGURABLE MODULE OPTIONS

Within the DPOP framework, dynamically profiling a sensor-based application requires profiling methods to be incorporated within each node to monitor the execution behavior for individual sensor nodes. Additionally, in order to optimize a sensor-based system, a global view of the entire system is needed. As such, the resulting node-level profile data must eventually be transmitted and analyzed by the system-level *Profiler Module*. Numerous profiling strategies can be employed to collect the pertinent application-level information. Fig. 2 highlights the profiling metrics that the profiling methodologies can consider: 1) what application-level parameters need to be profiled; 2) whom (which nodes) to profile within the network; 3) when to perform profiling; 4) how to transmit the profile data; and 5) what granularity of profile data needs to be maintained. Each profiling metric has several configuration options, which enable profiling specialization.

A. WHAT TO PROFILE

WSN literature provides numerous design metrics and tunable parameters considered by platform designers and application experts in their optimization efforts, with lifetime being one of the most prominent design metrics [2], [9], [12], [14], [37], [38], [53]. In addition to lifetime, designers also seek to balance competing metrics such as latency (e.g., the time to transmit a packet over one hop), packet delivery rate [38], throughput (e.g., the number of bytes transmitted per node per second) [53], transceiver sleep/active states to mitigate the time necessary to transition between power states and respond to an event [9], [14], coverage area (e.g., the physical area monitored by sensors) [22], [33], etc. While numerous design metrics can be considered and could easily be incorporated, currently DPOP supports lifetime, reliability, and throughput goals.

Based on these design metrics, the *Profiler Module* collects low-level execution statistics (e.g., sensor sampling rate, packet transmissions, battery charge, etc.) such that the *Optimizer Module* can evaluate how well the current platform configuration meets user-defined goals. Determining what low-level metrics to profile within a sensor-based platform is thus related to both the high-level design metrics of interest and the estimation method utilized to evaluate those design metrics. Within the current DPOP framework, lifetime, reliability, and throughput are defined as functions of the configurable parameters and application profiling parameters. Depending on which high-level design metric is being evaluated, the *Profiler Module* observes a subset of parameters from the following application profile parameters: sensor sampling rate, time between successive packets, current battery voltage, number of packets transmitted by an individual nodes, and the number of packets dropped by an individual node. While some application parameters can be measured statically, such as the energy consumption to process a sensor event or transmit a predefined number of bits, the *Profiler Module* focuses on dynamic events that are difficult to measure at design time. For example, while the sensor sampling rate may be set within the application code, there may be different rates for different modes within the application. Determining when these modes are triggered or the duration of these modes may not be known at design time and are instead observed after the application is deployed.

We note that these *what* to profile configurable options are by no means an exhaustive list of low-level execution statistic, rather as the set of design metrics grows or as the underlying estimation mechanisms changes, these parameters will also change.

B. WHOM TO PROFILE

Profiling all individual sensor nodes is not always necessary since, for instance, closely neighboring nodes may experience nearly identical input stimuli and environmental conditions. Therefore, the profiling methodology must consider who (which nodes) is profiled within the network. Options include profiling an individual node, a cluster of nodes, or profiling the network as a whole. The selection of whom to profile is affected by both application and network topology. For example, the profiler may want to profile only the nodes tasked with forwarding packets since these nodes may have higher energy consumption for which optimizing lifetime would be of critical importance. Alternatively, the profiler may profile a single sensor node that is within a known high activity area (e.g., wildlife monitoring) to determine the minimum sampling rate required by the application.

It is also possible that every node in the entire network must be profiled. In this situation, profiling overhead must be carefully considered since per-node profile data must traverse the communication network. To limit communication overhead, nodes may aggregate or average the collected profiling data as the data is forwarded to the *Profiler Module*. However, profiling at different levels of granularity provides the ability to tradeoff profiling detail with profiling overhead. We note that each node is still responsible for obtaining and transmitting the corresponding profile data. By changing *whom* to profile, the profiling for some nodes may be deactivated.

C. WHEN TO PROFILE

Given the desired profile data to be collected, the frequency at which profiling is performed directly impacts both the accuracy of the profiling data as well as the intrusiveness of the profiling methodology. Profiling can be performed *periodically* at each node or cluster of nodes using an internal timer to indicate when the profile data should be forwarded. Although the performance and energy consumption overhead of periodic profiling is easily predicted, dynamic activity patterns of individual nodes or across the WSN may be unpredictable and periodically collected profile data may not accurately capture the current execution behavior.

Alternatively, nodes can directly detect any event that is related to the required profile data and directly transmit upon *detection of flagged events* to the *Profiler Module* as these events occur. This method provides the advantage of highly accurate profile data, but at the expense of potential increases in code size due to the need to detect flagged events and packet transmission overhead.

An alternative method to control when to profile is by requiring the *Profiler Module* to explicitly send a profile request packet to the nodes. While a packet transmission overhead is incurred to transmit the profile request packet, the *Profiler Module* can dynamically control how often these requests are sent based on the currently collected profile data or observed behavior patterns.

D. HOW TO PROFILE

The method of transmitting the collected profile data back to the *Profiler Module* impacts the WSN's network traffic load as well as nodes' energy consumptions, since the radio subsystem must remain active for longer durations to transmit/forward the profile data packets. Currently, our profiler implementation provides support for either transmitting profile data as separate profile packets or appending (i.e., piggybacking) the profile data to existing packets already transmitted by the application. Requesting nodes to send separate profiling packets may increase overall network traffic, since each dedicated profiling packet must also include the packet header. Instead, by piggybacking the profile data onto existing data packets, the profiling data can be transmitted without requiring an additional packet header. However, piggybacking profile data onto existing data packets may require an individual sensor node to store the profile data until the sensor node transmits a data packet. Currently, profile data is piggybacked on existing data packets if the application data requires fewer bytes than available in the payload. Thus, if a large amount of data is being transmitted within the network, the profile data may experience delays before arriving at the *Profiler Module*.

E. PROFILING GRANULARITY

Finally, the profiling granularity denotes the level of aggregation of the profiling data within the network. In the case of no aggregation, all profile data collected at the node level is forwarded to the *Profiler Module*. By forwarding all of the profile data, the *Profiler Module* can maintain a detailed record of each node, enabling a detailed view of the entire network. However, the tradeoff is an increase in the network traffic and energy consumption, particularly for intermediate nodes, since these nodes must transmit additional packets. An alternative strategy is to aggregate at the cluster head, averaging the profile data collected for each node within a cluster before forwarding a single profile packet to the *Profiler Module*. This aggregation methodology limits the number of packets transmitted, but increases the computational complexity of the nodes tasked with consolidating the node-level statistics. Furthermore, by aggregating the profiling data, the accuracy of detected events may be diminished. Thus, aggregation would be appropriate

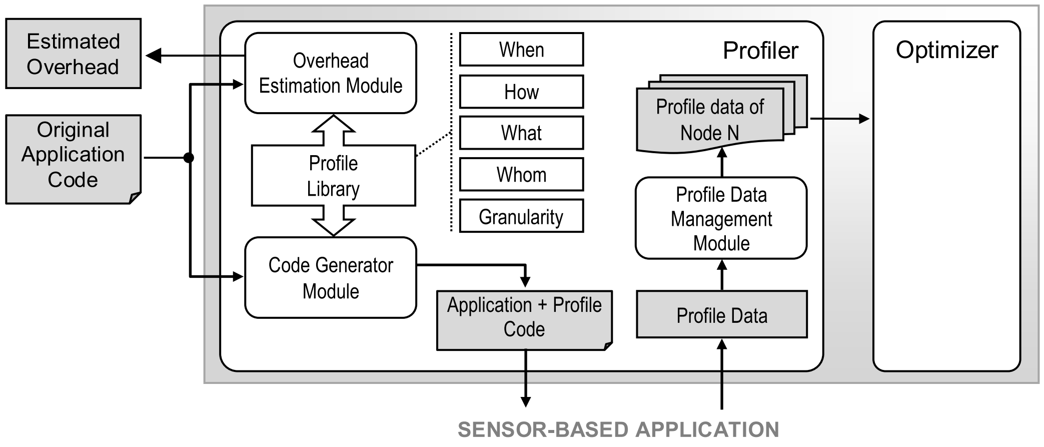


FIGURE 3. The *Profiler Module* is composed of an estimation module, code generation module, and a data processing module.

for applications that do not require such a detailed view of node or WSN behavior. In the extreme situation, the profile data may be aggregated for all nodes. Thus, instead of tracking activity based on nodes or cluster, all profile data arriving at the *Profiler Module* is aggregated into a single value.

F. INCORPORATING PROFILING CONFIGURATION OPTIONS

The profiling metrics and the metrics' configuration options available within the *Profiler Module* (Fig. 2), enable a variety of permutations to customize how the dynamic profile data is collected. Incorporating the profiling methodologies into the target application does not directly impact the functionality of the application. Rather, the profile code is inserted within the underlying software infrastructure for packet transmission/reception and sensor interfaces and non-intrusively monitors the deployed application. The following sections highlight the underlying implementation of this module.

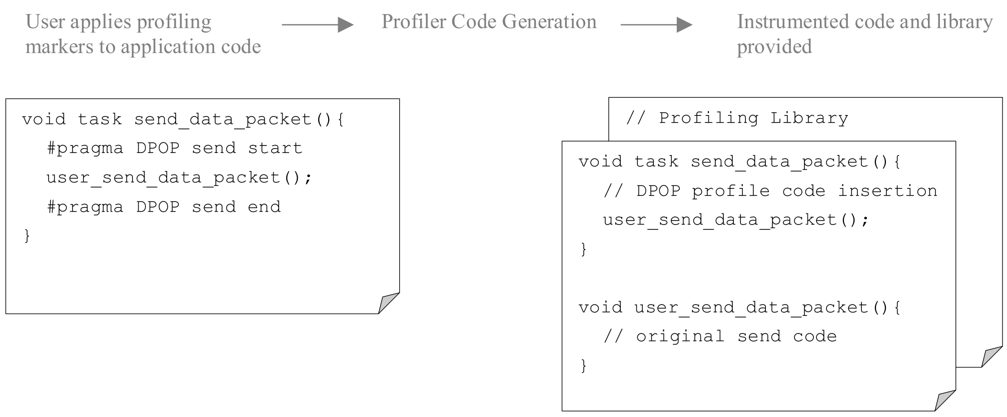
V. PROFILER MODULE IMPLEMENTATION

Although these dynamic profiling methodologies provide powerful methods to monitor and optimize WSN deployments, implementation and evaluation of these methodologies must be accessible to application experts. Fig. 3 depicts the Profiler *Module*, which comprises three components—a code generator, an estimation module, and a profile data management module—that aid application experts in the customization and evaluation of profiling methodologies. At design time, the code generator is used to simplify the task of incorporating the desired profiling methodology within the existing application code. With the newly augmented application, the estimation module determines the resulting profiling overheads to help an application expert analyze the expense of profiling the WSN. If the incurred overheads are acceptable, the application is deployed. If the incurred overheads are not acceptable, the application expert can either revise the application requirements and/or functionality, or revise the profiling configuration options (Section IV). At runtime, the profile data management module receives profile data packets generated from the sensor nodes, cluster heads, and base station, and parses each profile packet, aggregating the profile data into an intermediate format required by the *Optimizer Module*. Details of each component are discussed in the following subsections.

A. CODE GENERATOR MODULE AND PROFILING LIBRARY

Based on the configurable options available within the *Profiler Module* (Fig. 2), a variety of permutations can be selected to customize how the profile data is collected. Integration of profiling methodologies within the application code involves modifying the original application code by adding in profiling-specific variables, data structures, and functions. To ensure integration of the profiling mechanisms into the target application does not directly impact the functionality of the application, the profile code is inserted within the underlying software infrastructure for packet transmission/reception and sensor interfaces. A code generator and profiling library has been integrated within the DPOP framework to not only provide a clear separation between the initial application development and integration of the profiling mechanisms, but to reduce development effort and expertise needed by application experts.

Fig. 4 depicts a sample of the code generation process. To automate the integration of the profiling methodologies, the code generator needs to identify basic regions within the application code, such as variable declaration, function calls, macro definitions, along with NesC-specific regions, such as the signature blocks, implementation of signal events, and wiring in the configuration. Compiler directives—profile markers—are provided in the form of `#pragma DPOP`, which are inserted by the application expert in the corresponding locations. The code generator then parses the original application code and inserts the appropriate profile code in these regions. For example, an application expert may have


User applies profiling markers to application code
Profiler Code Generation
Instrumented code and library provided

```
void task send_data_packet(){
  #pragma DPOP send start
  user_send_data_packet();
  #pragma DPOP send end
}
```

```
// Profiling Library
```

```
void task send_data_packet(){
  // DPOP profile code insertion
  user_send_data_packet();
}

void user_send_data_packet(){
  // original send code
}
```

FIGURE 4. User application code containing the appropriate profiling markers in addition to the existing application code.

TABLE 1. DPOP profile markers inserted into the original application code by the application expert are translated by the code generation module to integrate profiler-specific code.

Markers	Interpretation
`#pragma DPOP interface declaration`	Inserts interfaces used by the profiler.
`#pragma DPOP variable declaration`	Inserts global variables used by the profiler.
`#pragma DPOP function declaration`	Inserts variables associated with function used by the profiler.
`#pragma DPOP timer`	Inserts timer code utilized by profiler.
`#pragma DPOP send start`	Inserts wrapper around user's packet transmission code.
`#pragma DPOP send end`	
`#pragma DPOP read sensor start`	Inserts wrapper around user's sensor read code.
`#pragma DPOP read sensor end`	
`#pragma DPOP receive start`	Inserts wrapper around user's packet receive code.
`#pragma DPOP receive end`	
`#pragma DPOP end`	Marks end of user's application code, profiler code inserted to collect statistics and send to optimizer module.
`#pragma DPOP profiler configuration`	Inserts new components used by profiler.
`#pragma DPOP profiler define`	Inserts macro definitions used by profiler.
`#pragma DPOP profiler message`	Inserts message struct used for profiling.

code that is executed each time a data packet is transmitted within a `send_data_packet` function (Fig. 4). In order to profile an execution statistic, such as the time between packets, profile code needs to be executed on every occurrence of this event within the `send_data_packet` function. Rather than directly modify the initial function, the code generator moves the original code located between the `#pragma DPOP send start` and `#pragma DPOP send stop` markers to a new `user_send_data_packet` function. The `send_data_packet` function then becomes a wrapper that integrates the necessary profiling structures and code for a given profiling methodology and calls the original user code.

Table 1 depicts similar profile marker abstractions provided in the DPOP framework for profiling the required events within the profiling methodologies. We note that an application expert does not need to be aware of the specific profiling methodology in use to annotate the code. Rather, generalized profile markers are provided to annotate regions of code, such as variable or function declarations, as well as code for packet transmission, reception, and sensor reading. The code generator is responsible for analyzing the customized profiling methodology selected by the application expert to determine the profile code that must be inserted into the application code.

A generic profiling library is provided to support the large number of profiling methodologies as well as the code generation process. The profiling library defines all required data structures and functions needed to support all possible profiling configuration options (Fig. 2).

Each function has function-specific properties, including the function prototype, function input, function outputs, global and local variables utilized by the function, as well as the interfaces, macros, and header files. In the profile code generation process, the code generator gathers the properties of these functions based on the profiling methodology, and generates five files containing: profiler headers, profiler variables, profiler macros, profiler interfaces, and profiler function implementations. The first four of these files specify the customized set of header file includes, variables declarations, macro defines, and function interfaces that will be inserted into the original application code. The last file provides the customized implementation of the selected profiling methodology. Note that these files can also provide users with a method to estimate the extra program code required for the selected profiling methodology. Although the current profiling library and integration framework has been initially developed for TinyOS, the generic profile markers and profiling library can be readily adapted to other operating systems and platforms.

The code generator module combines various functions within the profiling library to construct customized profiling methodologies. Integration of profiling within the application will require the use of additional resources to support these profile functions. As sensor nodes are resource constrained, integration of these functions should not only guarantee that the functionality of the new application remains unchanged, but must ensure the performance of the sensor nodes with the instrumented application code will not experience unacceptable degradation. Thus, the overhead estimation module component is designed to help the application expert evaluate the overhead incurred by the customized profiling methodology selected.

B. OVERHEAD ESTIMATION MODULE

The overhead estimation module determines the corresponding overheads for a given profiling methodology. The estimated overheads vary depending on the application and profiling methodology selected. Currently, we consider five types of overhead: network overhead, energy consumption overhead, code size overhead, and computation time overhead.

Since the DPOP framework's current configuration options (Fig. 2) offer over 6,000 different profiling methodologies, measuring the corresponding overheads for each profiling methodology at runtime is not practical or feasible. Instead, the estimation module estimates overheads by characterizing how each configuration option within the profiling methodology (selection of what, whom, when, how, and granularity) contributes to each of the overhead metrics:

$$Overhead_{PM_X} = f(O_{what} O_{whom} O_{when} O_{how} O_{granularity}). \quad (1)$$

where $Overhead\ _{PMx}$ is the combined overheads of the configurable options.

1) NETWORK TRAFFIC OVERHEAD

The network overhead is the percentage of extra packets transmitted in the network, measured by computing the number of bytes transmitted within a profiled network compared to the total number of bytes transmitted within the original application code. Lower network traffic overheads are desired, since this implies that the integration of the profiling does not significantly increase the burden of the underlying network by increasing the number of collisions, increasing radio usage, or significantly altering the energy consumption. If the original application already has heavy network traffic, the addition of profile data has a smaller impact than an application with sparse network traffic. Thus, the methodology utilized must be carefully selected.

The *what* and *how* profiling metrics dictate the number of bytes needed to capture the profile data. The *what* profiling metric indicates which low-level execution statistics need to be monitored, and each design metric monitored requires 2 bytes. The *how* profiling metric impacts how profile data is transmitted to the *Profiler Module*. Using piggybacking, the number of packets within the network does not increase since the profile data is appended to existing data packets, but the packet size increases due to the additional profile data. This packet size increase is accounted for in the overhead evaluation. When using separate profile packets, the overhead must also account for the packet header attached to the profile data. The network overhead incurred (L) in terms of the number of extra bytes for a given profiling methodology can be estimated as:

$$L = N_{metric} \times 2 + l_{header}\ (bytes) \quad (2)$$

where N represents the number of metrics profiled, l_{header} is the length of the packet header, and α specifies if a separate profiling packet is sent ($1 =$ separate packet, $0 =$ piggybacking). In the case of TinyOS, l_{header} is equal to 13 bytes. In addition, the number of packets transmitted in a given time must also be considered. The *whom*, *when*, and *granularity* profiling metric impact the number of transmissions (N):

$$N = n_{nodes} + \sum_i \beta_i \times c_{cluster_i} + (m - \sum_i \beta_i) \quad (3)$$

where n_{nodes} is the number of nodes actively profiling (which send one profile data packet to the cluster head every profile period), $c_{clusteri}$ is number of sensor nodes within a particular cluster ($i = 1, \ldots, m$, where m is number of clusters), and β_i indicates whether the i^{th} cluster head is responsible for aggregating the profile data (no aggregation versus aggregation). Thus, $\Sigma_i\ \beta_i * c_{clusteri}$ represents the profile data packet directly forwarded by cluster, and $(m - \Sigma_i\ \beta_i)$ represents the aggregated profile packets sent by the cluster head. Thus, the network overhead for a particular profiling methodology is $L * N$.

2) ENERGY CONSUMPTION OVERHEAD

Since most sensor nodes have stringent power constraints, the energy consumption of the application will significantly

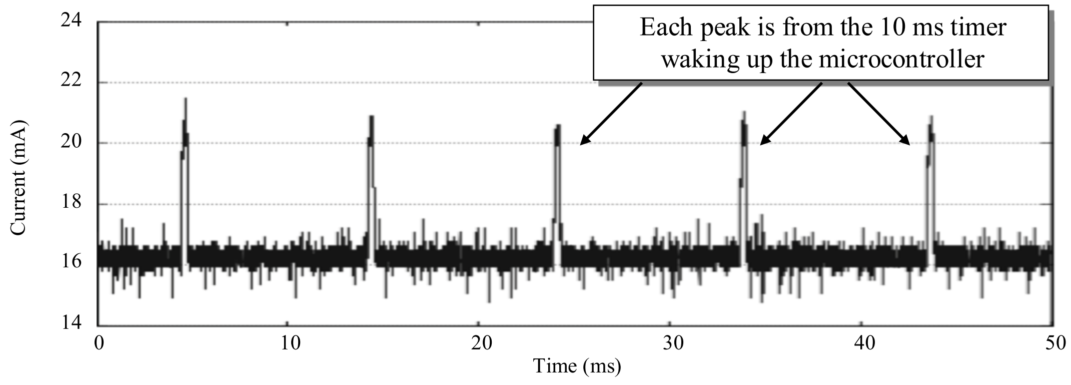


FIGURE 5. Current consumption for the microcontroller.

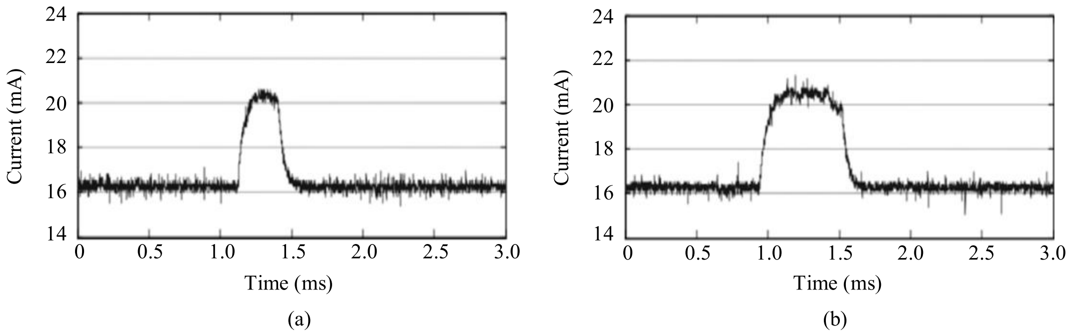


FIGURE 6. Current consumption of (a) the original application and (b) application with profiling. (a) Original Sensor activity with no profiling. (b) Active state extended by logging profile data.

affect WSN's performance. The energy consumption overhead provides the application expert with a preview of the energy requirements of the instrumented application as compared to the energy requirements of the original application. The energy consumption overhead incurred by adding profiling to the application can be primarily attributed to receiving and sending profiling packets, and collecting the profile data. While applications have varying application-specific requirements, the length of a profiling packet, methodology configuration packet, and code needed to collect the profile data are application independent. Thus, we can measure the energy consumption of each event at design time, and estimate the runtime energy consumption based on the selected profiling methodology.

In an application written in NesC, power is managed by TinyOS. Thus to analyze an application's energy consumption and the profiling methodologies, an understanding of the TinyOS power management strategy for platform's components is necessary. For each node, the microcontroller remains in an idle state if no events need processing; the radio remains in receive mode to monitor the physical channel and only switches to transmit mode when the application requests to send data; and the sensor board draws a constant current when accessed by microcontroller. Additional considerations are required for applications using profiling. Profiling requires the microcontroller to access the sensor board (if needed) more frequently than the original application to acquire and process the desired profile data and the radio also switches to transmit mode more often due to sending profile data (as in the case of separate data packets).

The energy consumption measurements are based on the Crossbow IRIS platform [39]. To detect the profiling overheads, current variations within the sensor node must be tracked. The current can be recorded by placing a resistor between the sensor node and power supply. We measured the voltage across the resistor using a National Instruments USB-6361X data acquisition module [40] to observe current consumption trends, from which the power consumption was determined. Since profiling only affects the microcontroller's active state and the radio's transmit mode, all sensor node idle states can be disregarded.

Through analysis of the current consumption of the sensor node in the active state (Fig. 5) and transmit mode (Fig. 7), and the overall application (Fig. 6), we identified three main deviations between the original application and instrumented application. First, an application with profiling requires the microcontroller to wakeup every 10 ms (Fig. 5), since the application expert may want to profile the time interval between two events or take periodic samples. A timer is used to provide a 10 ms resolution wakeup period, and a counter is incremented each time the microcontroller transitions to

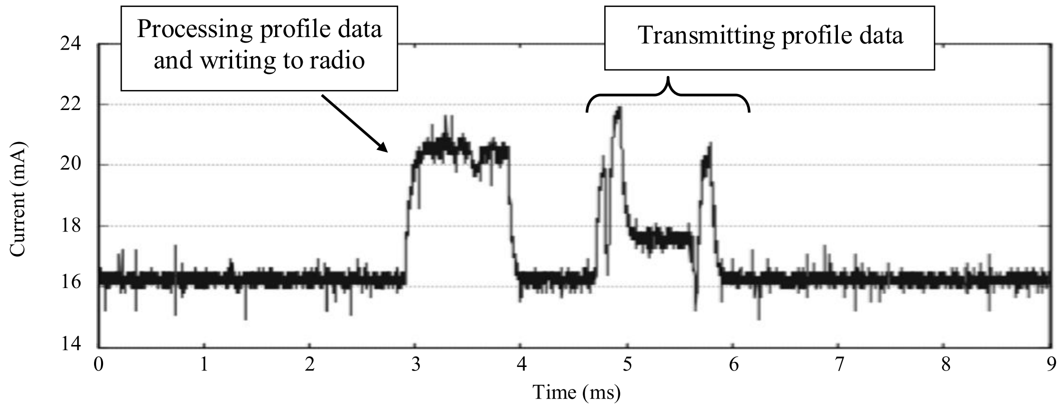


FIGURE 7. Current consumption for transmitting one profile data packet.

the active state. If the monitored event occurs faster than 10 ms, the base timer can be augmented, however the resulting energy consumption overhead incurred by the profiling will increase. Similarly, the time resolution can be increased to minimize the frequency at which the microcontroller transitions to the active state. However, if a timer is used within the application code, and the event to be monitored occurs at a frequency greater than the existing timer, the energy consumption overhead incurred by the timer can be eliminated since the profiling methodology can use the application's timer instead of introducing a dedicated profiling timer.

The second contributor to increased energy consumption occurs at the end of the profile period, illustrated in Fig. 7. At this time the microcontroller transitions to the active state to process the profile data, pack the profile data into a profile data packet, and transmit the profile packet. Sending the profile packet requires the radio to switch to transmit mode once more as compared to the original application.

Finally, the duration of the microcontroller's active state is increased (Fig. 6). In the original application code, the microcontroller enters the active state to access data from the sensor board or transmit sensor data. In an application with profiling, the microcontroller needs to additionally execute the profile code to log pertinent profile data in addition to the existing application behavior. Thus, the profiling extends the duration of the active state.

Based on these observations, each time the timer triggers the microcontroller to wake up, the microcontroller leaves the idle state for approximately 400 us. For the aforementioned hardware, this will consume an extra 4 μJ compared to remaining in the idle state. Fig. 7 shows the extra active state and transmission operation at the end of each profile period, resulting in an additional 15.6 μJ of energy consumption. Fig. 6 is a comparison between the active state in the original application compared and the active state in the application with profiling. As shown in the plots, profiling prolongs the active state from 400 μs to 650 μs due to logging the profile data. Thus, the final energy consumption overhead during one profile period can be estimated as:

$$E = E_i \left(\frac{P}{T}\right) + E_s + E_c N_e, \tag{4}$$

where E_t is the energy consumption incurred each time the timer triggers profiling, P is the profile period, and T is time required to profile. Thus, P/T indicates the number of additional active states incurred in one profile period. E_s is the energy consumption of processing the profile data and sending the profile data packet at the end of profile period, E_c is the extra energy used to log profile data, and N_e is the number of times a user-defined event occurs in one profile period.

3) CODE SIZE OVERHEAD

An additional consideration when evaluating various profiling methodologies is the code size overhead, defined as the size of the additional code required to perform profiling as compared to the size of the original application code. Code size overhead is related not only to the actual profiling methodology, but is also a function of the code size of the original application. For example, integrating a profiling methodology that is responsible for monitoring the battery voltage may need to integrate a driver to sample the battery voltage. However, if the battery voltage interface has already been used in the original application code, the code size overhead will be smaller compared to an application in which the driver was not initially used. Therefore, code size overhead is not only determined by the specific profiling methodology, but is impacted by the original application's basic functionality. Therefore, the code size is profiling-specific and application-specific, and overhead estimates from one application cannot be applied to any other application. Thus, we only need to use a subset of the profiling methodologies to measure the profiling overhead for an application. This information is then abstracted to estimate the code size overhead for the remaining profiling methodologies.

TABLE 2. Overview of the seven profiling methodologies considered.

	PROFILER CONFIGURATION				
	How	**When**	**Whom**	**What**	**Granularity**
PM1	Piggybacking	Periodic	All nodes	All metrics	No aggregation
PM2	Separate Packets	Periodic	All nodes	All metrics	No aggregation
PM3	Piggybacking	Profiler Request	All nodes	All metrics	No aggregation
PM4	Piggybacking	Flagged Event	All nodes	All metrics	No aggregation
PM5	Separate Packets	Profiler Request	All nodes	All metrics	No aggregation
PM6	Separate Packets	Periodic	One cluster	Sensor activity, Battery, # Packets Dropped	Aggregate at cluster head
PM7	Piggybacking	Flagged Event	One cluster	Sensor activity, Battery, # Packets Transmitted	Aggregate at cluster head

To determine a base estimation of the code size overhead, we used seven of the profiling methodologies (Table 2), which evaluate each of the configurable options available within the current profiling library. The corresponding code size overhead is determined by comparing the ROM usage for the original application to the ROM usage for the instrumented application, using the AVR compiler [10]. To estimate the code size overhead of a different profiling methodology, the overhead estimation module determines which of the seven base cases most closely matches the new configuration, denoted as CS_O.

Next, the overhead estimation module determines the differences between the most similar base case (CS_O) and the new profiling methodology (CS). In the new configuration, for every parameter (i) that has a different value than a base case, the overhead estimation module determines an additional base case (CS_i) that is most similar with respect to parameter i and computes the difference (d_i) in the measured code size overhead between the CS_O and CS_i base cases. The summation of these differences is added to the most similar base code size overhead to estimate the code size overhead for the new profiling methodology:

$$CS = CS_0 + \sum_i d_i. \tag{5}$$

4) COMPUTATION TIME OVERHEAD

Integration of profiling methodologies within the application code will result in additional computation for all nodes in the WSN, ranging from sensor nodes that use timers to determine when to collect profiling data, to intermediate nodes that may aggregate profile data, to base stations that may need to transmit profile request packets. Thus, the impact of computation time overhead must also be considered when evaluating different profiling methodologies. The computation time overhead is defined as the percentage of extra CPU cycles spent executing the profile code. The computation time overhead has a direct impact on the microcontroller's energy, since more computationally-intensive tasks require the microcontroller to remain in an active state for a longer duration.

Since the DPOP framework's profiling methodologies are implemented using functions in the profiling library, evaluating the computation time of a profiling methodology can be determined by measuring the computation time of individual functions in the profiling library that are also used by the profiling methodology. The computation time overhead (CT) can be estimated as:

$$CT = \sum_{i=1}^{n} t_i \times (1/m_i), \tag{6}$$

where n is number of functions used by the profiling methodology, t_i represents the computation time of an individual profiling function, and m_i represents the execution period of a function.

To determine the computation time of functions in the profiling library, we embedded each function in a specialized testing application that is instrumented to obtain time stamps with a microsecond precision before and after executing each profile function. The differences in the timer readings provide the time contributed by these profile functions, which are translated to the number of CPU cycles. We ensure that each application is executed long enough such that nondeterministic activities, such as interrupts, become negligible. The computation times for the profile functions are stored in the profiling library to aid with estimating computation time overhead. Table 3 provides the computation times for a subset of the most common profile functions.

C. PROFILE DATA MANAGEMENT MODULE

The code generator module and overhead estimation module help application experts to determine a suitable customized profiling methodology at design time. At runtime,

TABLE 3. Computation time of profile functions.

Events	*Computation Time (μs)*
Extracting the profiling methodology from the profile request packet	271
Sending a separate profile packet	61
Sending a piggybacked profile packet	63
Collecting profile data	225
Detecting flagged events (decreasing of battery voltage exceeds threshold set by user)	54
Detecting flagged events (time interval between two successive data packets)	29
Detecting flagged events (time interval between two successive sensor activities)	10
Detecting flagged events (number of packets transmitted by node)	10

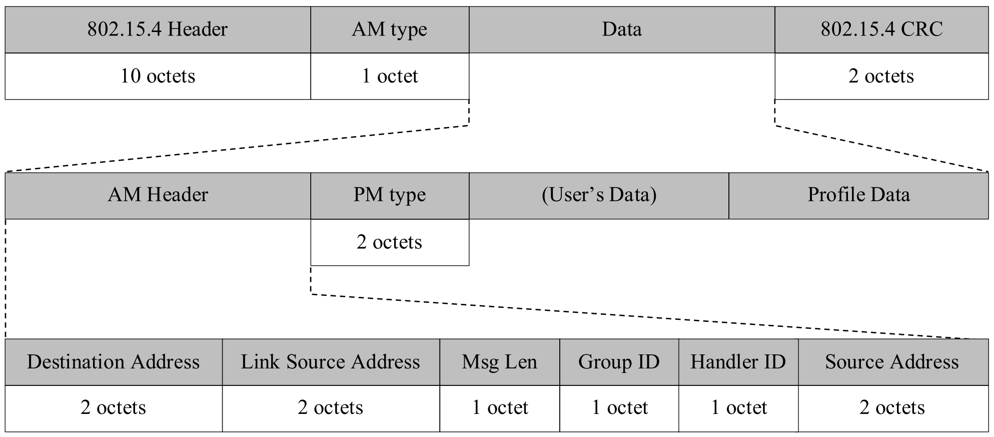


FIGURE 8. Profile data packet format.

profile data collected by the profile code is inserted into profile packets using a pre-determined format defined by the *Profiler Module*. The profile data management module parses the various fields within the profile packets and stores the accumulated profile data to be viewed by the application expert or to be fed into the *Optimizer Module*.

Fig. 8 depicts the profile packet's format. The 802.15.4 Header, AM type, and 802.15.4 CRC are fields added by radio drivers and are transparent to the application expert. The profile data management module uses the AM header that is added by TinyOS to identify the source and destination address of the profile packet. The PM field, which is added by the *Profiler Module*, indicates whether this is a profile packet and the profiling methodology used. When the profile data management module receives a profile packet, the profile data management module first parses the PM field to determine which profiling methodology is used by the application. Then, the binary data contained within the profile data field is converted to intermediate—human readable—format used by the *Optimizer Module*. Meanwhile, the source address in the AM header field is used to separate the received profile data and create profiles for each node/cluster considered within the WSN. The profile data, profiling methodology, and time stamps are appended to the recorded profile data. The application expert and/or the *Optimizer Module* can then easily use these profiles to retrieve pertinent information from any profiled node(s).

VI. EVALUATION

Depending on which profiling methodology is used in conjunction with the application's execution characteristics and requirements, the resulting overheads incurred will vary greatly. To provide a generalized and holistic evaluation of our DPOP framework that is applicable to a wide range of application characteristics and WSN deployment scenarios, we began our evaluations using a variety of profiling methodologies selected from the configurable options to better understand how each profiling methodology impacts the resulting overhead. This experimental data was also used to validate the estimated overhead metrics. To evaluate the feasibility of the proposed profiling methodologies and functionality of the *Profiler Module*, we developed five general benchmark applications that are representative of the communication and computation requirements of a wide variety of deployed WSN applications. Each of the benchmark applications and profiling methodologies were implemented on the Crossbow IRIS platform [39] using a network of 14 sensor nodes, each equipped with various sensors (e.g., temperature, light, humidity, etc.) a 16 MHz processor, and a 2.4 GHz RF transceiver.

A. BENCHMARK APPLICATIONS

The High Sample-Transmission Rate (HSTR) benchmark application implements a sensing-dominant application that requires high sampling and packet transmission rates.

The forest fire detection system deployed by Zhang et al. [29] is one example of a sensing-dominant application due to the high sensor sampling rates required to actively track a fire as it spreads. The HSTR benchmark application samples a single sensor input every two seconds and transmits data packets every four seconds.

The Multi-Sensor (MSEN) benchmark application is similar to the HSTR benchmark application, but reads multiple sensor inputs with a slightly reduced transmission rate of five seconds. The WSN deployed on Great Duck Island [4], which consists of nodes capable of sensing temperature, pressure, and humidity, is a deployed example of an MSEN application.

The Dual-Mode Power Saving (DMPS) benchmark application dynamically switches between a low-power sleep mode and high-power, high-speed monitoring mode. The energy-efficient surveillance system developed by He et al. [52] is an example of a DMPS application that must dynamically switch to a low-power, low communication mode in order to track vehicles in a stealthy manner. In the DMPS benchmark's low-power sleep mode, sensor nodes turn off their radio and sensors for prolonged duration (e.g., one minute) to reduce energy consumption. During the monitoring mode, the base station receives data packets sent by all nodes at a rate of one sample per second.

The Communication Intensive (COMM) benchmark application implements a sensor application with heavy network traffic, such as the WSN employed by Biagioni and Bridges [16], which collects data on endangered species by transmitting high-resolution images across the network. The COMM benchmark application samples and transmits sensor readings every five seconds and transmits a 20∗20 pixel grayscale image once every minute.

The Computation Intensive (COMP) benchmark application implements an application with high computational requirements. For example, Chu et al. [36] used computationally-intensive image processing techniques to track objects with the deployed WSN's environment. The COMP benchmark collects sensor data every 500 milliseconds and filters the sensed data using a 32-point FFT.

B. PROFILING METHODOLOGIES

Using these benchmark applications, we defined, implemented, and evaluated the various profiling methodologies in Table 2. These methodologies were selected to ensure that the corner cases were considered, as well as ensure that each of the different configuration options appeared in at least one of the profiling methodologies explored.

In the first five profiling methodologies (PM1 through PM5) all configuration options for the *what* profiling metric (Fig. 2) are monitored, including the sensor activity, time between packets, battery voltage, number of packets transmitted, and the number of packets dropped. Additionally, all nodes in the network are monitored without any aggregation. PM2 and PM3 expose how piggybacking and separate packets affect the entire network. PM1, PM3, and PM4 compare the difference between three configurable options of *when* to profile. PM6 and PM7 investigate the resulting overheads for different scenarios, where only a subset of the profiling metrics are profiled and address how aggregation may decrease the resulting overhead.

We anticipated that PM5 and PM7 would exhibit overhead extremes. PM5 should show the highest overhead because of the number of additional bytes required to transmit both profile request messages and separate profiling packets. Alternatively, PM7 should incur the smallest overhead since only a few nodes are being profiled, thereby contributing to fewer profiling bytes being inserted into the network.

C. EVALUATION OF DYNAMIC PROFILING METHODOLOGIES

In this section, we evaluate the performance for the different profiling methodology (seven) and benchmark application (five) combination scenarios (35 total scenarios) in terms of network traffic overhead, energy consumption overhead, code size overhead, and computation time overhead (Section V.B). We evaluate these overheads incurred by the profiled application as compared to the original application. Each overhead is reported as a percentage such that the performance variation across the different profiling methodologies, profiling metrics, and benchmark applications can be easily quantified.

1) NETWORK TRAFFIC OVERHEAD

To determine the network overhead, we logged all transmitted packets within a 5 hour execution interval using a packet sniffer that monitored all radio transmissions and recorded the contents of all packet transmissions on a host computer. The profile data packet length varied from 14 bytes to 138 bytes depending on which profiling methodology was used, and which application was profiled. The *DMPS* benchmark application was set to periodically switch between high-power monitoring mode and low-power off mode.

Fig. 9 shows the resulting network traffic overhead for each profiling methodology and benchmark application scenario. As expected, PM2 incurred the highest network overhead for all benchmark applications ranging from a 66.2% overhead for *DMPS* to a 0.44% overhead for *COMM*. PM7 incurred the lowest overhead across all benchmark applications because PM7 only profiles a subset of the profiling metrics and only when flagged events are triggered, which are sparse for most nodes. However, for *COMM*, PM7 and PM4 incurred higher overheads compared to the other profiling methodologies due to the heavy network traffic caused by transmitting images. One of the flagged events supported by our profiling methodologies is detecting if the time between two successive data packets is longer than the user-defined threshold. For example, this flagged event could be used to detect the delay in transmission of periodic temperature samples due to the extended transmission of the image data for this application.

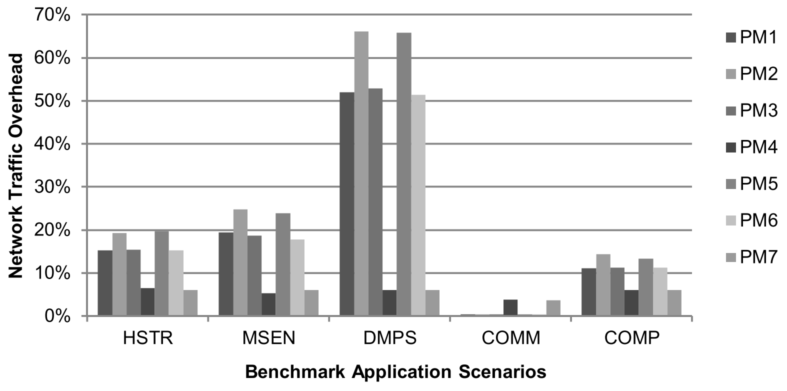


FIGURE 9. Network traffic overhead, in terms of additional bytes transmitted within the network to perform profiling, for all profiling methodology and benchmark application scenarios.

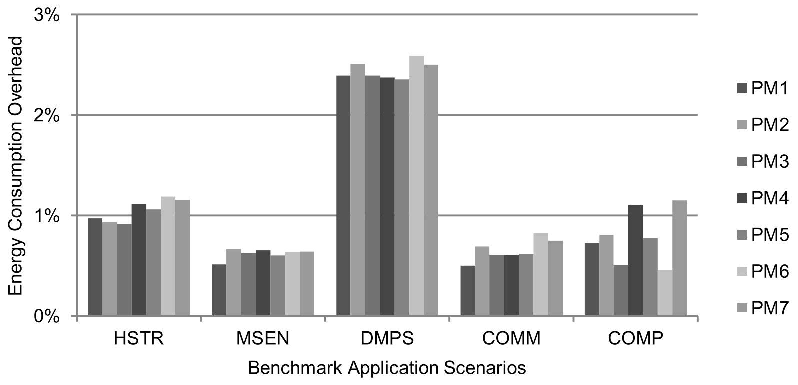


FIGURE 10. Energy consumption overhead, in terms of additional CPU cycles, for all profiling methodology and benchmark application scenarios.

We also observed similar network traffic overheads between PM1 and PM3, as well as between PM2 and PM5. This similarity is due to the formats of the profile data packets defined and the profiling frequency mandated by the *Profiler Module's* configuration. In each instance, profile data is transmitted every 60 seconds regardless of the frequency of the external stimuli. Thus, the profiling methodologies using the *what* and *how* profiling metrics will incur almost the same network traffic overhead.

In the original *DMPS* benchmark application, network traffic consists of 6 bytes for synchronization. Thus, profile packets sent by nodes contribute to a large portion of the network overhead in the profiled application. Conversely, the profiling methodologies examined only generate $<1\%$ of the network traffic in *COMM* due to heavy network traffic in the original application. Thus, the overheads incurred by the various profiling methodologies cannot be evaluated independently, and are highly application-specific.

2) ENERGY CONSUMPTION OVERHEAD

Integrating a profiling methodology into an application inevitably increases the energy consumption of that application due to the collection of profile data and transmission of profiling packets to the *Profiler Module*. Thus, to evaluate a profiling methodology's power overhead, we measured the current consumed by the node using a high-resolution data acquisition device (Section 4.2.2). Each application was configured to execute a fixed number of iterations and the current readings were then averaged over this time period. Fig. 10 shows the resulting energy consumption overhead for all of the profiling methodology and benchmark application scenarios based on the physical measurements of the original application code and the corresponding profile-instrumented application code.

The energy consumption overheads across all profiling methodology and application benchmark scenarios remained modest, ranging from 0.5% to 2.59%. Three main profile activities contribute to the energy consumption overhead. Compared with the power consumed by the timer, the power consumed by logging profile data, and sending profile packets is negligible. The timer triggers 6,000 times per minute and consumes 0.138 J per minute when profiling *HSTR*. However, the logging and transmission of profile data consumes less than 0.002 J. Thus, the various profiling methodologies considered incurred about the same energy consumption overhead due to the timer in isolation. Generally, profiled applications consumed an additional 1.7 J as compared to the original application within

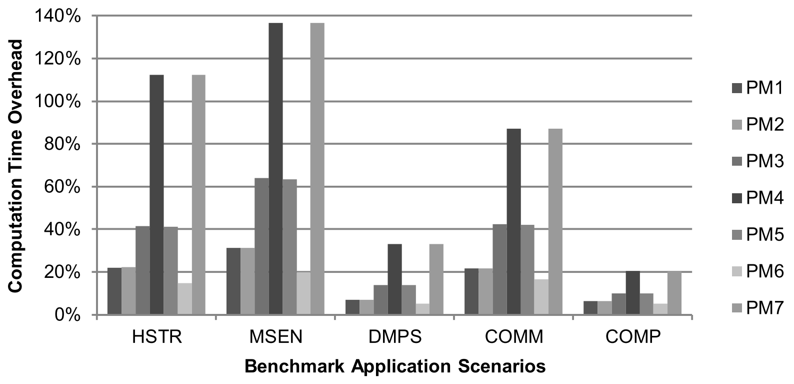


FIGURE 11. **Computation time overhead, in terms of additional CPU cycles, for all profiling methodology and benchmark application scenarios.**

a one hour timeframe. The slight differences among the profiling methodologies can be attributed to voltage fluctuations and noise in the experimental setup. In the case of *DMPS*, we observed a slightly higher energy consumption overhead that is attributed to the low energy of the original application.

3) COMPUTATION TIME OVERHEAD

Collecting profile data, receiving and parsing profile request packets, sending profile data packets, and detecting flagged events are the main contributors to computation time overhead. The computation time overhead is important since the computation time has a direct impact on the energy consumed by the microcontroller (i.e., a more computationally-intensive task will require the microcontroller to remain in a higher power state for a longer duration).

Fig. 11 shows the resulting computation time overhead for all of the profiling methodology and benchmark application scenarios. Across all scenarios, PM4 resulted in the highest computation time overhead, ranging from 20.32% for *COMP* to 136.59% for *MSEN*. While few profile data packets are sent by sensor nodes, the computational complexity stems from the need to periodically detect flagged events. Alternatively, PM6 had the smallest computation time overhead across all benchmark application scenarios, ranging from 5.19% for *COMP* to 23.06% for *DMPS* because fewer metrics are being collected as part of the profiling, and the computation overhead is reduced. In addition, a reduction in the number of profiling statistics tracked reduces the size of the packets being transmitted back to the *Profiler Module*, which yields a reduction in computational complexity by limiting the amount of data processed. Finally, PM3 resulted in a higher overhead than PM1 for all benchmark application scenarios because of the need to parse the profile request packet from *Profiler Module*.

The percentage of computation time overhead enables a comparison of the computation time between different profiling methodologies and the original application code. However, if the original application does not have computationally intensive tasks, a higher computation time overhead does not necessarily indicate that profiling operations and events are computationally intensive. For example, on average, the CPU is active for approximately 25μs during each 1s period in the original *HSTR* application. Although profiling methodologies that use flagged event detection incur the highest percentage of computation time overhead, the active time for the CPU only increased to 36μs within a 1s period. For all methodologies measured, increases in the CPU active time varied from 2μs for *MSEN* with PM3 to 13μs for *COMP* with PM6. Thus, application experts need to be able to evaluate the resulting overheads using different profiling methodologies, profiling metrics, and profiling metric configuration options to determine which customized profiling methodology is best suited for a given application.

4) CODE SIZE OVERHEAD

The code size overhead provides an additional metric to evaluate how well each profiling methodology performs in terms of the size of the additional code required to perform profiling as compared to the original application. The code size overhead is an important consideration since the profiling methodologies require additional memory, which may be a constrained resource depending on the application requirements. In the case of the IRIS platform, code size is currently limited at 128 kB [39], and our benchmark application sizes range from 10 - 24 kB. Fig. 12 illustrates the resulting code size overhead for all profiling methodologies and benchmark application scenarios obtained from the ROM usage reported by the AVR compiler. For PM6 and PM7, the overheads of profiled nodes were isolated and reported, since other nodes within the network do not contain additional profile code overheads.

D. ESTIMATION MODULE EVALUATION

To validate the accuracy of the overhead estimation module, we additionally performed estimations for each of the profiling methodology and benchmark application scenarios. Table 4 presents an overview of the maximum, minimum, and average estimation errors for all scenarios compared to the corresponding measured overheads.

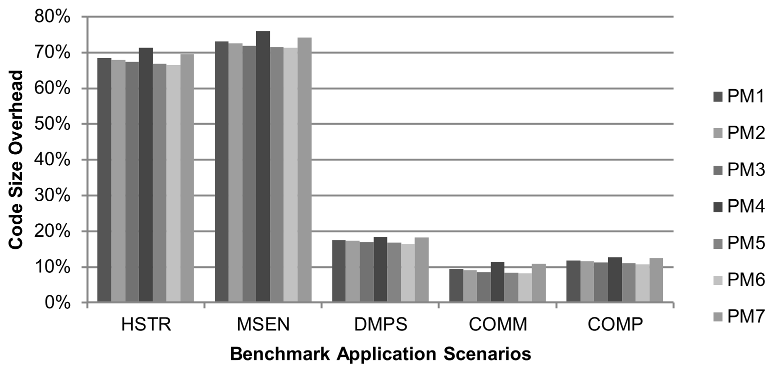


FIGURE 12. Code size overhead, in term s of additional bytes storage required, for all profiling methodology and benchmark application scenarios.

TABLE 4. Estimation error of the network traffic, computation time, energy consumption, and code size overheads for the *high sample-transmission rate (HSTR), multi-sensor (MSEN), dual-mode power saving (DMPS), communication intensive (COMM), and computation intensive (COMP) applications.*

	Network Traffic	Computation Time			Energy Consumption			Code Size		
		Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.
HSTR	<0.001%	0.29%	3.81%	1.60%	0.13%	0.52%	0.41%	0.03%	0.09%	0.05%
MSEN	<0.001%	0.55%	4.83%	2.69%	0.23%	0.73%	0.64%	0.02%	0.87%	0.29%
DMPS	<0.001%	0.54%	4.62%	3.53%	0.01%	0.18%	0.08%	0.19%	0.53%	0.30%
COMM	<0.001%	3.27%	14.57%	9.34%	1.68%	1.19%	1.77%	0.05%	0.85%	0.44%
COMP	<0.001%	1.15%	14.38%	3.21%	0.12%	0.51%	0.26%	0.03%	0.83%	0.42%

TABLE 5. Network overhead in terms of number of profile data packet sent by nodes.

Application	Estimation	PM1, PM2, PM3, PM5, PM6	PM4, PM7
HSTR	500	500	491
MSEN	500	500	495
DMPS	500	500	495
COMM	500	500	485
COMP	500	500	491

The estimation of network traffic overhead is constrained by the profiling methodologies that include flagged event detection. For network overhead, the estimation module accurately predicted overheads with error rates under 0.001%. The profiling methodologies configure nodes or base stations to periodically profile their status. The frequency of sending profile data and the length of the profile packets are controlled by *Profiler Module* and can be easily predicted. The error arises from missing profile request packets from the base station.

Table 5 presents the number of profile packets sent by nodes, comparing the *periodically profile* by nodes and the *base station requests profile* options. If individual nodes manage profiling, these nodes will not miss any profiling operations and events and therefore will transmit the anticipated number of packets. Alternatively, nodes using PM4 or PM7 may fail to start profiling due to the loss of profile request packets from base station. Additionally, benchmark applications that do not have computationally-intensive tasks and heavy network traffic have a low probability of losing and retransmitting packets. For profiling methodologies that profile tasks and user-defined flagged events, the estimation module would not be able to predict the resulting network overhead since these specific events always relate to the functionality of application.

The computation time overhead estimation error varies from 9.34% to 1.6% across all profiling methodology and benchmark application scenarios. Inaccuracy is caused by a difference in the functionally of the benchmark applications and the testing applications used (Section V.B.4) to 1) measure the computation time of individual profile functions and 2) estimate the computation time of collecting profile data. A negative error is obtained for some profiling methodologies for *COMM*, indicating the overhead was underestimated leading to a larger increase in computation time than estimated. This underestimate of computation time overhead can potentially be attributed to how the compiler optimizes the code. Given the large memory usage required for storing images, less memory is available for the program memory, and the compiler will need to optimize the code for memory size. This type of optimization can often lead to longer executions times. The lack of resources leads to longer execution times for the profile code. Thus, the characteristics of *COMM* resulted in higher computation time overheads for all profiling methodologies in comparison to the other benchmark applications. Thus, the computation times in the profiling library are not enough to estimate the computation time for a particular application, and feedback from nodes is required to improve computation time overhead accuracy.

Due to using the energy consumption measurements of specific profiling events, the estimated energy consumption overhead is approximately equal to the measured values, yielding an average error of 0.63%. Although the energy consumption of the benchmark applications may vary, energy usage of profiling events remains consistent across these benchmark applications. Thus, the premeasured data can be used to accurately estimate energy consumption overheads under diverse benchmark application scenarios.

The code size overhead estimates use the profiling methodologies (PM1-PM7) as the base profiling methodologies. Four additional profiling methodologies are then used to evaluate the accuracy of code size overhead estimation accuracy. To avoid similarity between base profiling methodologies and the additional profiling methodologies used for evaluation, at least two configuration options in the additional profiling methodologies are different from the seven base profiling methodologies. Because we use the actual ROM usage of the profiled application to estimate code size overhead, the estimation error is lower than 1%. However, the drawback of this method is that we must collect code size overheads for more profiling methodologies (different configuration options) to provide more information in the profiling library.

VII. CONCLUSIONS AND FUTURE WORK

Dynamic profiling of wireless sensor networks (WSNs) enables an accurate view of an application's execution behavior, but incurs increased network traffic, energy consumption, code size, and computation overheads. The collected profile data can be used by platform designers and application experts to quickly evaluate, select, and optimize appropriate profiling methodologies using the dynamic profiling and optimization (DPOP) framework. Since application-specific and sensor-specific constraints dictate the profiling requirements and tolerated overheads, this design assistance is required.

In this work, we significantly enhance the DPOP framework. We developed various methods for configuring the *Profiler Module* to implement a variety of dynamic profiling methodologies and analyzed the corresponding overheads. While energy consumption increases are low, ranging from 0.5% to 3%, network traffic, code size and computation time overheads can be as high as 66%, 76% and 137%, respectively, thus it is critical to evaluate and consider these overheads in the context of application requirements, goals, and constraints. At design time, the code generator module, overhead estimation module, and profile data management module work together to assist an application expert in choosing a suitable profiling methodology, evaluating that methodology with respect to the application, and rapidly integrating these profiling methodologies within the application to extract profile data at runtime. During runtime, application experts can use a reconfigurable profiling methodology to adapt the profile approach as needed to adjust to changing application execution and environmental stimuli.

Currently, the overhead of a profile function recorded within the profiling library is derived from measurements based on applications executing in an ideal experimental environment. Given our existing mechanism to accurately estimate profiling overhead, future work includes developing an automated methodology to assist application experts in determining which profiling methodology best suits a specific application and the application expert's design goals. This automated—or assisted—customization of the profiling methodology must be capable of evaluating the profiling overhead, the accuracy of the profile data, and the impact of the profiling method on the optimization of the underlying application, which is guided by the application expert's design goals and is a parallel research thrust. To support this tradeoff analysis, the presented profiling methods must be integrated with prior work in runtime optimization of sensor networks, and new methods will need to be developed to estimate the profiling accuracy, evaluate the adequacy of a profiling methodology with respect to the requirements of the optimization tools, and provide application experts with an intuitive method for understanding these tradeoffs.

REFERENCES

[1] A. S. Dhodapkar and J. E. Smith, "Managing multi-configuration hardware via dynamic working set analysis," in *Proc. Annu. Int. Symp. Comput. Archit. (ISCA)*, May 2002, pp. 233–244.

[2] A. Kurtkoti and B. Patel, "Evaluation metrics of MAC layer in wireless sensor network," in *Proc. 1st Int. Conf. Emerg. Trends Eng. Technol. (ICETET)*, Jul. 2008, pp. 250–254.

[3] A. Lizarraga, R. Lysecky, S. Lysecky, and A. Gordon-Ross, "Dynamic profiling and fuzzy-logic-based optimization of sensor network platforms," *ACM Trans. Embedded Comput. Syst.*, vol. 13, no. 3, pp. 1–29, Dec. 2013, Art. ID 51.

[4] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson, "Wireless sensor networks for habitat monitoring," in *Proc. ACM Int. Workshop Wireless Sensor Netw. Appl. (WSNA)*, Atlanta, GA, USA, 2002, pp. 88–97.

[5] A. Munir, A. Gordon-Ross, S. Lysecky, and R. Lysecky, "A lightweight dynamic optimization methodology for wireless sensor networks," in *Proc. IEEE 6th Int. Conf. Wireless Mobile Comput., Netw. Commun. (WiMob)*, Oct. 2010, pp. 129–136.

[6] A. Munir, A. Gordon-Ross, S. Lysecky, and R. Lysecky, "A one-shot dynamic optimization methodology for wireless sensor networks," in *Proc. Int. Conf. Mobile Ubiquitous Comput., Syst., Services (UBICOMM)*, Oct. 2010, pp. 287–293.

[7] A. Munir and A. Gordon-Ross, "An MDP-based application oriented optimal policy for wireless sensor networks," in *Proc. Conf. Hardw./Softw. Codesign Syst. Synth. (CODES+ISSS)*, 2009, pp. 183–192.

[8] A. Shenoy, J. Hiner, S. Lysecky, R. Lysecky, and A. Gordon-Ross, "Evaluation of dynamic profiling methodologies for optimization of sensor networks," *IEEE Embedded Syst. Lett.*, vol. 2, no. 1, pp. 10–13, Mar. 2010.

[9] A. Sinha and A. Chandrakasan, "Dynamic power management in wireless sensor networks," *IEEE Des. Test. Comput.*, vol. 18, no. 2, pp. 62–74, Mar./Apr. 2001.

[10] AVRFreaks. *WinAVR*. [Online]. Available: http://winavr.sourceforge.net/index.html, accessed May 2015.

[11] B. L. Titzer, D. K. Lee, and J. Palsberg, "Avrora: Scalable sensor network simulation with precise timing," in *Proc. 4th Int. Conf. Inf. Process. Sensor Netw. (IPSN)*, Apr. 2005, pp. 477–482.

[12] C. Alippi and G. Vanini, "Application-based routing optimization in static/semi-static wireless sensor networks," in *Proc. 4th Annu. IEEE Int. Conf. Pervasive Comput. Commun. (PerCom)*, Mar. 2006, pp. 46–51.

[13] C. Park and P. H. Chou, "EmPro: An environment/energy emulation and profiling platform for wireless sensor networks," in *Proc. 3rd Annu. IEEE Commun. Soc. Conf. Sensor Ad Hoc Commun. Netw. (SECON)*, Sep. 2006, pp. 158–167.

[14] C. Schurgers, V. Tsiatsis, S. Ganeriwal, and M. Srivastava, "Optimizing sensor networks in the energy-latency-density design space," *IEEE Trans. Mobile Comput.*, vol. 1, no. 1, pp. 70–80, Mar. 2002.

[15] D. Weber, J. Glaser, and S. Mahlknecht, "Discrete event simulation framework for power aware wireless sensor networks," in *Proc. 5th Int. Conf. Ind. Informat. (INDIN)*, vol. 1. Jun. 2007, pp. 335–340.

[16] E. S. Biagioni and K. W. Bridges, "The application of remote sensor technology to assist the recovery of rare and endangered species," *Int. J. High Perform. Comput. Appl.*, vol. 16, no. 3, pp. 315–324, Aug. 2002.

[17] F. Douglis, P. Krishnan, and B. N. Bershad, "Adaptive disk spin-down policies for mobile computers," in *Proc. Symp. Mobile Location-Independ. Comput.*, 1995, pp. 121–137.

[18] L. F. Perrone and D. M. Nicol, "A scalable simulator for TinyOS applications," in *Proc. Winter Simulation Conf.*, Dec. 2002, pp. 679–687.

[19] F. Yuan *et al.*, "A lightweight sensor network management system design," in *Proc. 6th Annu. IEEE Int. Conf. Pervasive Comput. Commun. (PerCom)*, Mar. 2008, pp. 288–293.

[20] G. Chen, J. Branch, M. Pflug, L. Zhu, and B. Szymanski, "SENSE: A wireless sensor network simulator," in *Advances in Pervasive Computing and Networking*. Berlin, Germany: Springer-Verlag, 2005, pp. 249–267.

[21] G. Tolle and D. Culler, "Design of an application-cooperative management system for wireless sensor networks," in *Proc. 2nd Eur. Workshop Wireless Sensor Netw. (EWSN)*, Jan./Feb. 2005, pp. 121–132.

[22] H. Zhang and J. C. Hou, "Maintaining sensing coverage and connectivity in large sensor networks," *Ad Hoc Sensor Wireless Netw.*, vol. 1, no. 1–2, pp. 89–124, 2005.

[23] I. Beretta, F. Rincon, N. Khaled, P. R. Grassi, V. Rana, and D. Atienza, "Design exploration of energy-performance trade-offs for wireless sensor networks," in *Proc. 49th ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2012, pp. 1043–1048.

[24] I. Kadayif and M. Kandemir, "Tuning in-sensor data filtering to reduce energy consumption in wireless sensor networks," in *Proc. Design, Autom., Test Eur. Conf. Exhibit. (DATE)*, Feb. 2004, pp. 852–857.

[25] J. Eriksson, F. Österlind, N. Finne, A. Dunkels, N. Tsiftes, and T. Voigt, "Accurate network-scale power profiling for sensor network simulators," in *Proc. 6th Eur. Conf. Wireless Sensor Netw.*, 2009, pp. 312–326.

[26] J. L. Hill and D. E. Culler, "MICA: A wireless platform for deeply embedded networks," *IEEE Micro*, vol. 22, no. 6, pp. 12–24, Nov./Dec. 2002.

[27] J. Polastre, R. Szewczyk, C. Sharp, and D. Culler, "The mote revolution: Low power wireless sensor network devices," in *Proc. Hot Chips Symp.*, pp. 1–20, 2004. [Online]. Available: http://webs.cs.berkeley.edu/papers/hotchips-2004-motes.ppt

[28] D. Blazakis, J. McGee, D. Rusk, and J. S. Baras, "ATEMU: A fine-grained sensor network simulator," in *Proc. IEEE Commun. Soc. Conf. Sensor Ad Hoc Commun. Netw. (SECON)*, Oct. 2004, pp. 145–152.

[29] J. Zhang, W. Li, Z. Yin, S. Liu, and X. Guo, "Forest fire detection system based on wireless sensor network," in *Proc. 4th IEEE Conf. Ind. Electron. Appl. (ICIEA)*, May 2009, pp. 520–523.

[30] K. Whitehouse *et al.*, "Marionette: Using RPC for interactive development and debugging of wireless embedded networks," in *Proc. 5th Inf. Process. Sensor Netw. (IPSN)*, 2006, pp. 416–423.

[31] L. S. Bai, R. P. Dick, and P. A. Dinda, "Archetype-based design: Sensor network programming for application experts, not just programming experts," in *Proc. Int. Conf. Inf. Process. Sensor Netw. (IPSN)*, Apr. 2009, pp. 85–96.

[32] L. S. Bai, R. P. Dick, P. H. Chou, and P. A. Dinda, "Automated construction of fast and accurate system-level models for wireless sensor networks," in *Proc. Design, Autom. Test Eur. Conf. Exhibit. (DATE)*, Mar. 2011, pp. 1–6.

[33] C. Liu and G. Cao, "Spatial-temporal coverage optimization in wireless sensor networks," *IEEE Trans. Mobile Comput.*, vol. 10, no. 4, pp. 465–478, Apr. 2011.

[34] L. Girod, N. Ramanathan, J. Elson, T. Stathopoulos, M. Lukac, and D. Estrin, "EmStar: A software environment for developing and deploying heterogeneous sensor-actuator networks," *ACM Trans. Sensor Netw.*, vol. 3, no. 3, Aug. 2007, Art. ID 13.

[35] L. Luo, T. He, G. Zhou, L. Gu, T. F. Abdelzaher, and J. A. Stankovic, "Achieving repeatability of asynchronous events in wireless sensor networks with EnviroLog," in *Proc. 25th IEEE Int. Conf. Comput. Commun. (INFOCOM)*, Apr. 2006, pp. 1–14.

[36] M. Chu, J. E. Reich, and F. Zhao, "Distributed attention in large scale video sensor networks," in *Proc. Intell. Distrib. Surveill. Syst.*, London, U.K., Feb. 2004, pp. 61–65.

[37] M. Holland, "Optimizing physical layer parameters for wireless sensor networks," M.S. thesis, Dept. Elect. Comput. Eng., Univ. Rochester, Rochester, NY, USA, 2007.

[38] M. Zimmerling, "Automatic parameter optimization of sensor network MAC protocols," M.S. thesis, Dept. Comput. Sci., Dresden Univ. Technol., Dresden, Germany, 2009.

[39] Memsic Corporation. (2010). *IRIS Wireless Measurement System*. [Online]. Available: http://www.memsic.com/products/wireless-sensor-networks/wireless-modules.html

[40] National Instruments. *NI USB-6361 X Series DAQ*. [Online]. Available: http://sine.ni.com/nips/cds/view/p/lang/en/nid/209073, accessed Jan. 2015.

[41] N. Kothari, R. Gummadi, T. Millstein, and R. Govindan, "Reliable and efficient programming abstractions for wireless sensor networks," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 200–210, Jun. 2007.

[42] O. Landsiedel, K. Wehrle, and S. Götz, "Accurate prediction of power consumption in sensor networks," in *Proc. IEEE Workshop Embedded Netw. Sensors*, May 2005, pp. 37–44.

[43] P. K. Dutta and D. E. Culler, "System software techniques for low-power operation in wireless sensor networks," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2005, pp. 925–932.

[44] P. Levis, N. Lee, M. Welsh, and D. Culler, "TOSSIM: Accurate and scalable simulation of entire TinyOS applications," in *Proc. Conf. Embedded Netw. Sensor Syst. (SenSys)*, 2003, pp. 126–137.

[45] R. Fonseca, P. Dutta, P. Levis, and I. Stoica, "Quanto: Tracking energy in networked embedded systems," in *Proc. 8th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, 2008, pp. 323–338.

[46] R. Sugihara and R. K. Gupta, "Programming models for sensor networks: A survey," *ACM Trans. Sensor Netw.*, vol. 4, no. 2, pp. 1–8, Mar. 2008.

[47] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: Exploiting generational behavior to reduce cache leakage power," in *Proc. 28th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2001, pp. 240–251.

[48] S. Kim *et al.*, "Health monitoring of civil infrastructures using wireless sensor networks," in *Proc. 6th Int. Symp. Inf. Process. Sensor Netw. (IPSN)*, Apr. 2007, pp. 254–263.

[49] S. Lysecky and F. Vahid, "Automated application-specific tuning of parameterized sensor-based embedded system building blocks," in *Proc. Int. Conf. Ubiquitous Comput. (UbiComp)*, 2006, pp. 507–524.

[50] S. Sridharan and S. Lysecky, "A first step towards dynamic profiling of sensor-based systems," in *Proc. 5th Annu. IEEE Commun. Soc. Conf. Sensor, Mesh Ad Hoc Commun. Netw. (SECON)*, Jun. 2008, pp. 600–602.

[51] T. Arampatzis, J. Lygeros, and S. Manesis, "A survey of applications of wireless sensors and wireless sensor networks," in *Proc. 13th Medit. Conf. Control Autom.*, Jun. 2005, pp. 719–724.

[52] T. He *et al.*, "Energy-efficient surveillance system using wireless sensor networks," in *Proc. MobiSys*, Boston, MA, USA, 2004, pp. 270–283.

[53] T. van Dam and K. Langendoen, "An adaptive energy-efficient MAC protocol for wireless sensor networks," in *Proc. 1st Int. Conf. Embedded Netw. Sensor Syst. (SenSys)*, 2003, pp. 171–180.

[54] V. Handziski, A. Kopke, H. Karl, and A. Wolisz, "A common wireless sensor network architecture," Tech. Univ. Berlin, Berlin, Germany, Tech. Rep. TKN-03-012, 2003.

[55] V. Shnayder, M. Hempstead, B.-R. Chen, G. W. Allen, and M. Welsh, ''Simulating the power consumption of large-scale sensor network applications,'' in *Proc. 2nd Int. Conf. Embedded Netw. Sensor Syst. (SenSys)*, 2004, pp. 188–200.

[56] Y. Liu, K. Liu, and M. Li, ''Passive diagnosis for wireless sensor networks,'' *IEEE/ACM Trans. Netw.*, vol. 18, no. 4, pp. 1132–1144, Aug. 2010.

[57] Y. Yu, D. Ganesan, L. Girod, D. Estrin, and R. Govindan, ''Synthetic data generation to support irregular sampling in sensor networks,'' *GeoSensor Netw.*, vol. 1, no. 4, pp. 211–234, 2004.

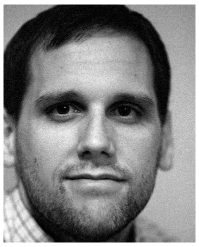
LU DING received the M.S. degree in electrical and computer engineering from the University of Arizona, in 2012. He is currently with Western Digital.

ADRIAN LIZARRAGA received the B.S. degree in electrical engineering from the University of Arizona, in 2010. He is currently pursuing the Ph.D. degree with the Electrical and Computer Engineering Department, University of Arizona. His research interests include runtime optimization of embedded systems and data-adaptable embedded systems. He is a recipient of the NSF Bridge to Doctorate Grant.

ASHISH SHENOY received the M.S. degree in electrical and computer engineering from the University of Arizona, in 2012. He is currently with Riverbed Technology.

ANN GORDON-ROSS (M'00) received the B.S. and Ph.D. degrees in computer science and engineering from the University of California, Riverside, USA, in 2000 and 2007, respectively. She is currently an Associate Professor of Electrical and Computer Engineering with the University of Florida, USA, and a member of the NSF Center for High Performance Reconfigurable Computing with the University of Florida. She is also a Faculty Advisor of the Women in Electrical and Computer Engineering and the Phi Sigma Rho National Society for Women in Engineering and Engineering Technology, and an active member of the Women in Engineering Proactive Network. Her research interests include embedded systems, computer architecture, low-power design, reconfigurable computing, dynamic optimizations, hardware design, real-time systems, and multicore platforms. She was a recipient of the CAREER Award from the National Science Foundation in 2010, best paper awards at the Great Lakes Symposium on VLSI in 2010 and the IARIA International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies in 2010, and the Best Ph.D. Poster at IEEE Computer Society Annual Symposium on VLSI in 2014. She is very active in promoting diversity in STEM fields, and has been a Guest Speaker at several international workshops/conferences on this topic, organizes workshops, and participates in local outreach programs at local K-12 schools.

SUSAN LYSECKY received the Ph.D. degree in computer science from the University of California, Riverside, in 2006. She was an Assistant Professor of Electrical and Computer Engineering with the University of Arizona. She is currently a Senior Content Engineer with Zyante, Inc. Her research interests include Web-native interactive learning, embedded system design, and human–computer interaction.



ROMAN LYSECKY received the B.S., M.S., and Ph.D. degrees in computer science from the University of California, Riverside, in 1999, 2000, and 2005, respectively. He is currently an Associate Professor of Electrical and Computer Engineering with the University of Arizona. He has co-authored five textbooks in *VHDL*, *Verilog*, *C*, *C++*, and *Java Programming*. His research interests focus on embedded systems, with an emphasis on runtime optimization, nonintrusive system observation methods for *in-situ* analysis of complex hardware and software behavior, data-adaptable system, and embedded system security. He was a recipient of the Outstanding Ph.D. Dissertation Award from the European Design and Automation Association for new directions in embedded systems in 2006, the CAREER Award from the National Science Foundation in 2009, and four best paper awards from the ACM/IEEE International Conference on Hardware–Software Codesign and System Synthesis, the ACM/IEEE Design Automation and Test in Europe Conference, the IEEE International Conference on Engineering of Computer-Based Systems, and the International Conference on Mobile Ubiquitous Computing, Systems, Services.