Technical Report

Inter-application Redundancy Elimination in Sensor Networks with Compiler-Assisted Scheduling VG-CAS-11

Vikram Gupta

Eduardo Tovar

Karthik Lakshmanan

Raj Rajkumar

SIES-RED-2012

Version:

Date: 6/1/2012

Inter-application Redundancy Elimination in Sensor Networks with Compiler-Assisted Scheduling VG-CAS-11

Vikram Gupta, Eduardo Tovar, Karthik Lakshmanan, Raj Rajkumar

IPP-HURRAY!

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail: vigup@isep.ipp.pt, emt@isep.ipp.pt, ,

http://www.hurray.isep.ipp.pt

Abstract

Most current-generation Wireless Sensor Network(WSN) nodes are equipped with multiple sensors of varioustypes, and therefore support for multi-tasking and multipleconcurrent applications is becoming increasingly common. Thistrend has been fostering the design of WSNs allowing severalconcurrent users to deploy applications with dissimilar requirements. In this paper, we extend the advantages of a holisticprogramming scheme by designing a novel compiler-assistedscheduling approach (called REIS) able to identify and eliminateredundancies across applications. To achieve this useful high-leveloptimization, we model each user application as a linear sequenceof executable instructions. We show how well-known string-matching algorithms such as the Longest Common Subsequence(LCS) and the Shortest Common Super-sequence (SCS) canbe used to produce an optimal merged monolithic sequenceof the deployed applications that takes into account embeddedscheduling information. We show that our approach can help inachieving about 60% average energy savings in processor usagecompared to the normal execution of concurrent applications.

Inter-application Redundancy Elimination in Wireless Sensor Networks with Compiler-Assisted Scheduling

Vikram Gupta[†‡], Eduardo Tovar[†], Karthik Lakshmanan[‡], Ragunathan (Raj) Rajkumar[‡]
[†]CISTER Research Center, ISEP, Polytechnic Institute of Porto, Portugal
[‡]Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, USA
vikramg@ece.cmu.edu, emt@isep.ipp.pt, {klakshma, raj}@ece.cmu.edu

***Abstract*—Most current-generation Wireless Sensor Network (WSN) nodes are equipped with multiple sensors of various types, and therefore support for multi-tasking and multiple concurrent applications is becoming increasingly common. This trend has been fostering the design of WSNs allowing several concurrent users to deploy applications with dissimilar requirements. In this paper, we extend the advantages of a holistic programming scheme by designing a novel compiler-assisted scheduling approach (called REIS) able to identify and eliminate redundancies *across* applications. To achieve this useful high-level optimization, we model each user application as a linear sequence of executable instructions. We show how well-known string-matching algorithms such as the Longest Common Subsequence (LCS) and the Shortest Common Super-sequence (SCS) can be used to produce an optimal merged monolithic sequence of the deployed applications that takes into account embedded scheduling information. We show that our approach can help in achieving about 60% average energy savings in processor usage compared to the normal execution of concurrent applications.**

***Keywords*-Wireless Sensor Networks; Energy Optimization; Scheduling;Compilers;**

I. Introduction

Recent advances in hardware and operating systems for Wireless Sensor Networks (WSNs) have enabled the support for multi-tasking and multiple concurrent applications on a sensor node. Most commercially available nodes are also equipped with several different types of sensors including, but not limited to, light, temperature, acceleration, humidity and audio. Such a diversity in sensors allows several users with different requirements to concurrently use a given sensor networking infrastructure. Moreover, a large percentage of applications for wireless sensor networks is designed around sensing the physical environment and transmitting a processed data value to the user. We call the paradigm for such applications as *Sense-Compute-Transmit (SCT)*. In such applications, there is a high possibility of redundancy as they may contain several independent requests for sampling the same sensors. In this paper, we propose an approach for eliminating this redundancy to save energy in the processor usage on each sensor node and the network.

Let us consider a simple case of a sensor network deployed across an office building with each node having a temperature and a humidity sensor. A building manager may be interested in collecting the temperature values from the sensors for a fine-grained temperature control, and a civil engineer may want to find the correlation between temperature and humidity for optimizing the building's HVAC system. Such applications can be executed concurrently on the sensor network infrastructure. Both the building manager and the civil engineering researcher sample the temperature sensor for their independent applications, which provides an opportunity for sharing the sensed value among both the applications. It turns out that reading a sensor value typically involves accessing the Analog-to-Digital Converter (ADC) module on the microprocessor, for converting the analog sensor value into a digital format, and storing into a register. This process of sampling a sensor can consume about $2-3$ orders of magnitude more processor cycles than a simple memory-based instruction. With the increase in the number of applications deployed on a sensor network, the overhead because of sampling the sensors can also increase dramatically. Hence, by sharing sensing requests among applications, a significant percentage of resource-usage and energy can be saved on a sensor node. In this paper, we propose a solution able to achieve such energy-savings through a compile-time approach. There are several challenges involved in such an approach and are discussed next.

Computer science researchers have long focused on designing compiler optimizations to remove redundancies and dead-code in a program. Several simple optimizations are standard features in most modern compilers; complex features can also be enabled for specific optimizations based on overall program logic [1]. In general-purpose computing systems (e.g. desktop computers or data-centers), independent applications may have similar logic but it is very less likely that they share the same data as well. This makes inter-application redundancy elimination a less-explored research area, as the possibility of energy savings is quite low. For instance, two independent users may want to use a distributed system to compute Fast Fourier Transform (FFT) over large datasets. Even though the computation module of FFT is the same for both the users, it is highly unlikely that the dataset will be the same as well. Hence, the provisions of sharing the same result among the two users may not be beneficial in terms of energy savings. In sensor networks, however, the data of interest typically is the sampled values of the physical quantities, and it is significantly more likely that different applications may require sampling of the same sensors. We show in this paper that sharing those samples can achieve considerable energy savings.

As most sensing applications are periodic in nature and have low duty-cycles, eliminating redundant sections in case of mismatching periods can be difficult, and may not provide significant gains if elimination is carried out using simple temporal overlap detection. Secondly, the applications can sample the sensors multiple times at different intervals and in different order. Compiler support is a practical and effective technique for identifying such requests and optimizing them for finding better overlap. Finally, redundancy elimination at each node at run-time can add significant complexity to the scheduler on the sensor node. The scheduler in this case will have to pre-profile the execution of the program to identify the overlapping sections.

In this paper, we propose a novel solution to the problem of finding overlapping sensing requests issued by network-wide applications created by independent users. We model each application as a linear sequence of executable instructions, and find a merged sequence of multiple applications through the use of well-known string-matching algorithms. In particular, we use the Longest Common Subsequence (LCS)[2] and the Shortest Common Super-sequence (SCS)[3] techniques. Our proposed solution creates a monolithic task-block resulting from the optimized merging of user applications with embedded scheduling information. This scheme is particularly advantageous in cases where the relative order of sensing requests is important, and simply caching the values may not help. One such case can be envisaged in an application where multiple sensors are sampled at different intervals but in a specific order to infer patterns of target behavior as may be the case in assisted living scenarios, or sensor-fusion based localization. We show that our approach can help in achieving about 60% average energy savings in processor usage as compared to the execution of several applications without eliminating the redundancies.

The organization of the rest of this paper is as follows. First, we provide an overview of our approach in Section II. Section III and Section IV provide the details of the modeling of applications and the proposed redundancy elimination approach, respectively. We evaluate our approach in Section V. The background research and related work is presented in Section VI. We then conclude the paper with a section on future work, the conclusions and the limitations of our approach.

II. Overview of the Approach

We assume that the users develop network-level sensing applications using a higher-level programming framework. The application code written by the users can either be at an abstract network-level using a macro-programming language like Regiment [4] or it can use node-specific virtual-machines (for example Matè [5]). In both these cases, the programming framework creates node-level intermediate code based on the application logic specified by the user. Our approach is based on a machine-language like intermediate code, generally referred to as *bytecode*. The architecture of such a complete system is shown in Figure 1, where the user applications are converted into bytecode by a parser, such that each output instruction is either an indivisible subexpression or a special

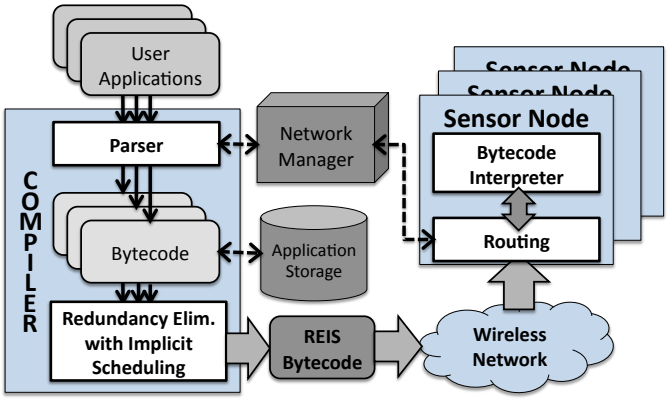


Fig. 1. Overview of the approach for redundancy elimination

function for accessing the hardware (including sensing, GPIO access or packet transmission). Bytecode corresponding to all applications are converted to a monolithic code by the *Redundancy Eliminator with Implicit Scheduler (REIS)* module. This monolithic code, which we call *REIS-bytecode* and ρ-code in short, is a merged sequence of all the applications with the redundancies eliminated according to the temporal overlap of the sensing requests. REIS-bytecode is then sent over the wireless network to each sensor node where the applications are to be executed. A bytecode interpreter at the sensor node executes the received REIS-bytecode.

Our approach assumes that a data link-layer and a suitable routing layer are already implemented on the sensor node and our solution is transparent to it as long as end-to-end packet delivery is supported. A network manager module handles the responsibility of dynamically updating the routing tables, and maintaining network topology information. As users issue applications to the system independently, our approach requires an application storage database to store the bytecode and merge them using the REIS module whenever a new application is submitted. The semantics of each user application is embedded within the REIS-bytecode such that maximum sharing of sensing requests and radio transmissions is obtained. Bytecode from different applications share non-overlapping variable and address space, which removes any need for context switching, and the interleaving of bytecode provides an implicit schedule of execution.

The motivation behind the sharing of sensing requests can be justified based on the comparison of the time taken for reading a sensor sample into memory with a simple memory-based instruction. Figure 2 shows an oscilloscope capture of this comparison on a WSN platform with an Atmel ATMEGA1281 processor. This comparison is obtained by toggling a GPIO pin just before and after the execution of a sensor sampling instruction (shown by Trace 1) and a memory-based loading of a 16-bit value into a register (Trace 2). The former takes about 500 microseconds but the latter instruction takes only 10 microseconds. Please note that this time comparison also includes the time taken for toggling the I/O pins. As the ATMEGA1281 ($8MHz$) processor on the sensor node has on-chip memory, a load instruction takes a maximum of 3

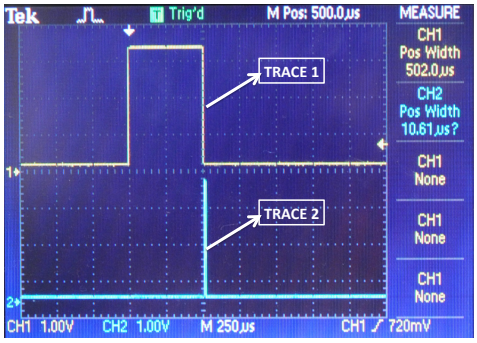


Fig. 2. Oscilloscope screenshot showing the comparison of the time taken for reading a sensor against a memory-based operation

cycles that correspond to 375 nanoseconds. A majority of the time consumed in the case of Trace 2 is because of the pin toggling. Hence, a sensor sampling instruction consumes up to $\frac{(500-10)\times10^{-6}}{375\times10^{-9}} = 1306$ times more power. This factor, which we refer to as ϕ (*time-factor*), is specific to the platform and the operating system. However, the order of magnitude of ϕ can be assumed to be similar across most systems.

In addition, radios on newer System-on-Chip (SoC) solutions like the ATMEL ATmega 128RFA1 [6] support 2 Mbps data rate, compared to 250 Kbps from the commonly used CC2420 [7] radio for about 20% less power consumption. This implies that the power per packet can be reduced by a factor of 8, bringing the power consumption of radio closer to that of the processor. Hence, optimizations at the processor level are bound to play a significant role in reducing total energy consumption, in contrast to the majority of research efforts focussing mainly on energy savings at the radio-level.

III. Application Modeling

Our proposed optimizations are aimed at applications whose main goal is to sample sensors, process the sensor data for more meaningful results, and then transmit the results towards a gateway node through the network tree.

Each bytecode instruction contains a list of hex opcodes, and is of the form: `<TYPE OP1 OP2 OP3 ...>`, where `TYPE` defines the kind of operation, and operand `OP<K>` can have specific usage based on the bytecode. For the sake of clarity, example formats of some relevant bytecodes are provided in Table I. Specific implementation can vary based on the design of the Parser and the Bytecode Interpreter.

TABLE I
EXAMPLE BYTECODE STRUCTURE FOR SOME RELEVANT SUBEXPRESSION INSTRUCTIONS

Operation	Opcode	Details
Sense	`S t VAR`	Sample sensor `t` and copy the value in VAR
Assign	`AEQ VAR1 VAR2`	Assign `VAR1 = VAR2`
Transmit	`T DEST VAL1 VAL2`	Transmit `VAL1` & `VAL2` to `DEST` node
Compute	`C VAR1 VAR2 VAR3`	`VAR1 := VAR2 'C' VAR3`

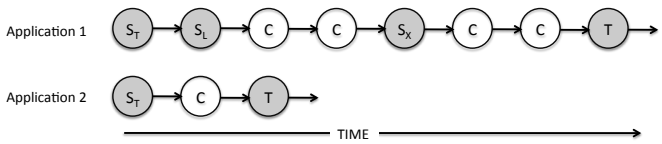


Fig. 3. An example showing a linearized execution sequence for one instance of two applications. Application 1 samples three different sensors, and Application 2 samples the temperature and transmits its scaled-down value.

A. Conversion to a sequence of nodes

Most sensor networking applications are of the form: *Sense-Compute-Transmit (SCT)*, as the users are typically interested in sampling one or more sensors, processing the data from sensors and collecting the processed results at a gateway node. Such applications can be modeled as a string of nodes where each node represents a sub-expression in the *bytecode*, β, as shown in Figure 3. S_t represents a sensing request for sensor type t, where t can be either be temperature (T), light (L), accelerometer (X, Y, Z) or any other sensor available on board. C denotes nodes with algebraic computation. As most sensor nodes typically have one kind of radio for communication, we use T to denote nodes corresponding to packet transfer via the radio. This conversion of bytecode subexpressions to nodes is captured by the function `create_node()` in Algorithm 1. As algebraic computations are generally data-dependent, finding the overlap across C nodes is considerably less plausible. Moreover, there are no significant energy savings by eliminating such overlap, as these instructions typically consume a small (about 1 to 2) number of machine cycles, particularly on a sensor network platform having a RISC processor and on-chip memory. Hence S_t and T-type nodes participate in finding the overlap across applications, and are called *Anchor Nodes*.

Conditional statements in an application may not allow it to be converted into a linear string. We present the techniques for modeling applications having at least one anchor node inside the conditional statements in the next subsection. The conditional statements without an anchor node can be trivially mapped to a C type node.

B. Modeling Conditional Statements

As it cannot be known at compile-time which execution path can be taken in case of a conditional statement, it is not possible to create a ρ-code (REIS-bytecode) from the input bytecodes based on a linear application model as described in Section III-A. We propose an algorithm to create a functionally equivalent code with a maximum possible number of sequential nodes, such that the conditional statements in the output bytecode sequence β_η are purely computational. Algorithm 1 provides a solution where the anchor nodes (sensing requests) are moved to before the beginning of the outermost conditional statement in case of nested if-loops. Please note that the sensing requests are data-independent instructions; moving them to a previous point in the code cannot impact the application logic. An `assign` instruction is inserted in the place of the original instruction, which loads the value returned by the sensing request into the variable originally designed to

read the output of sensing instruction, as shown in lines 19-21 in the algorithm. An example scenario is shown in Figure 4, where the original sampling request inside an `if`-condition is moved to before the outermost `if`-statement and the sampled value is stored in a temporary variable `var1_temp`. The original variable, `var1`, is assigned the value of `var1_temp` at its original location in the bytecode.

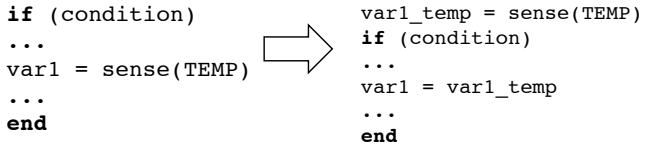


Fig. 4. An example showing the modeling of an if-condition

Algorithm 1: convert_app(A_i): convert an application to bytecode β_η

Input : A_i: A user created application
Output: (β_η, N_{an}): Bytecode node sequence, Number of moved anchor nodes

1 Parse A_i to bytecode β using the parser
2 INITIALIZATIONS:
3 $\beta_{\eta:} = \emptyset$; $if\_index := \emptyset$; $node := \emptyset$
4 $if\_depth := 0$; $N_{an} = 0$;
5 **foreach** *sub-expression* $\eta \in \beta$ **do**
6 $i = IndexOf(\eta)$
7 **if** η *is an if-clause* **then**
8 $if\_depth++$;
9 $if\_index \cdot append(i)$;
10 $\beta_\eta \cdot append(\eta)$;
11 **else if** $\eta = S$ **then**
12 **if** *if_index·isEmpty()* **then**
13 $index = i$;
14 **else**
15 $index = if\_index(i)$;
16 $N_{an}++$;
17 **end**
18 $node := create\_node(\texttt{type}(\eta), var)$;
19 $\beta_\eta \cdot insert(index, node)$;
 // move S node before the beginning of outermost if-condition
20 $node := create\_node(\texttt{assign}, var, \texttt{op2}(node))$;
21 $\beta_\eta \cdot append(node)$;
22 **else if** η *is an endif-clause* **then**
23 $if\_depth--$;
24 $if\_index \cdot pop\_back()$;
25 $\beta_\eta \cdot append(\eta)$;
26 **else**
27 $\beta_\eta \cdot append(\eta)$;
 // Non anchor nodes remain at the same relative location
28 **end**
29 **end**

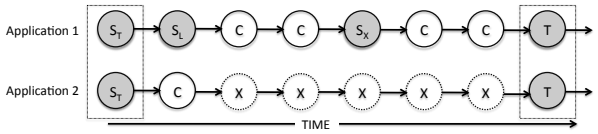


Fig. 5. Application 2 modified to be aligned with Application 1 for sharing sensing requests and packet transmission (based on example in Figure 3)

C. Merging Packet Transmissions

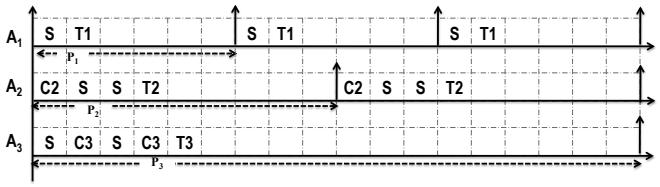
It can be claimed that, for better power savings, the transmit nodes T should also be moved towards the end of the bytecode sequence to obtain better overlap of radio usage across applications. We, however, do not take such an approach since a solution was already proposed in [8] to harmonize packet transmissions from different applications. Instead of transmitting whenever the applications request, the packets are queued in a local buffer and are transmitted at instants that provide maximum overlap of radio-transmissions. As radio is a shared resource among applications, such a queue based mechanism can help in achieving what is aimed by our proposed approach. Many other solutions (such as [9]) have been proposed to optimize the network-wide scheduling of packets. For brevity purposes, we skip further details of packet transmission optimization in this paper.

IV. Redundancy Elimination with Implicit Scheduling

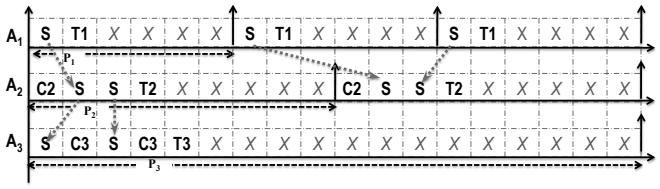
A. String-Matching Algorithms

Once an application is modeled as a sequence of nodes as described in the previous section, the problem of finding overlapping sections among two or more applications can be reduced to that of finding a common subsequence between a pair of applications. The Longest Common Subsequence (LCS) is a technique commonly used to find the overlap between a pair of strings of symbols such that the relative order of common symbols is the same in both the input strings. LCS provides one such common sequence having the longest possible length. Consider the two following string sequences: `SENSOR` and `NETWORK`. The longest common subsequences are {`N,O,R`}, {`E,O,R`} but the Longest Common Sub-String (LCSS) would just be {`O,R`}. A longest common substring is always a subset of the longest common subsequence, but the opposite may not be true. There are some commonly available solutions [2] that are guaranteed to return *a* longest ordered subsequence between a set of input strings.

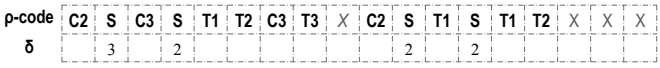
LCSS can help in finding redundant anchor nodes that appear consecutively in the input sequences. As an improvement over LCSS, LCS finds a subsequence with maximum overlap such that the relative order of nodes is not sacrificed. One or more of the input applications may be 'stretched' at various points, as illustrated in Figure 5 after applying LCS to the applications shown in Figure 3. An optimal merger of input sequences can be obtained by using an approach related to LCS called *Shortest Common Super-sequence (SCS)*[3].



(a) An example execution scenario showing three applications with different periods



(b) Process of locating overlapping sensing requests. The pattern repeats every hyper-period



(c) One possible output of the Algorithm 2, along with the degree of overlap of each shared sensing request

Fig. 6. Identifying overlap in sensing instructions in three different applications and creating a merged ρ-code using Algorithm 2

Definition 1: Given input sequences X and Y, the shortest common super-sequence, $Z = SCS(X, Y)$, is the shortest sequence such that both X and Y are subsequences of Z.
In the case of two input sequences, it is trivial to find the SCS if the LCS is known. For more than two sequences, finding a SCS is not a direct application of the LCS solution.

Algorithm 2: REIS(Γ): Generate a monolithic ρ-code with implicit scheduling from an input set of applications

Input : Γ: a set of n applications $< A_1, A_2, \ldots A_n >$ each with period P_i for i^{th} application
Output: ρ-code: a monolithic bytecode sequence
// From Equation 1
$P_H := LCM(P_1, P_2 \ldots P_n)$
INITIALIZE:
for $i = 1 : n$ **do**
  $\beta_{new,i} := \emptyset$;
end
foreach *application* $A_i \in \Gamma$ **do**
  $(\beta_{\eta,i}, N_{an,i}) = convert\_app(A_i)$;
  for $j = 1 : \frac{P_H}{P_i}$ **do**
    // create new strings
    $\beta_{new,i} := concatenate(\beta_{new,i}, \beta_{\eta,i})$;
  end
end
ρ-code $= SCS(\beta_{new,1}, \beta_{new,2}, \ldots \beta_{new,n})$;

One important aspect of applications designed to operate on sensor networks is periodicity. Applications are typically designed as tasks that repeat periodically with low duty-cycles. Different applications deployed on a sensor network may have unequal periods. This adds further complexity to the redundancy detection and elimination across applications. Let us assume that an application A_i has a period P_i; the harmonizing period P_H is given by:

$$P_H = LCM(P_1, P_2, \ldots P_n) \tag{1}$$

where LCM stands for the Least Common Multiple of the input values.

B. Algorithm for generating a ρ-code (REIS-bytecode)

Let us consider a set Γ of n independent applications, where each application is denoted by A_i and $i = 1, 2, \ldots n$. The period of an application A_i is P_i. First, each application is converted into a sequence of bytecodes as described in Algorithm 1. The output of Algorithm 1 contains nodes within each periodic execution. As the periods can mismatch, the minimum length of time for which the overlap among two or more applications should be calculated is equal to the harmonizing period, P_H. A new sequence is created from each input bytecode sequence β_η by self-concatenating it $\frac{P_H}{P_i}$ times to create a new sequence β_{new}. After this operation, all the sequences are of an equal length of P_H. Thereafter, the Shortest Common Supersequence (SCS) solution is applied to find a merged sequence ρ-code from the concatenated input bytecode. This approach is expressed through Algorithm 2. This may result in the size of a merged application being quite large as the concatenated code corresponds to P_H. However, it should be noted that there may be several repeating code blocks in the merged sequence that can be compressed significantly using simple compression approaches to save both the radio power and the memory footprint. This issue is beyond the scope of this paper, and will not be considered.

An example for demonstrating the merging of bytecode is shown in Figure 6. There are three input application bytecodes as shown in Figure 6(a). Please note that all applications only sample one type of sensor for the sake of simplicity. The periods of the applications are different, and, in this example, $P_H = P_3$. Application A_1 consists of S and T nodes occurring consecutively with a period of 6 units. A_2 is a sequence $< C, S, S, T >$ with a period of 9 units, and A_3 is $< S, C, S, C, T >$ with a period of 18 units. Non-anchor nodes across different application sequences are considered as *dissimilar nodes*. For example, C in A_2 is not the same as C in A_3, hence they are represented as $C2$ and $C3$, respectively. The SCS algorithm considers only S-type nodes as common across applications and merges, such that the length of the merged sequence is the shortest possible. Figure 6(b) shows a possible alignment of the S nodes, and Figure 6(c) shows a merged sequence with the overlapping S nodes omitted. The degree of overlap δ for each merged node is also shown.

For n applications to be executed on a sensor node, each with Worst Case Execution Time (WCET) C_1, C_2, ... C_n, respectively, the total execution time of the input applications per hyper-period is :

$$C_T = \sum_{i=1}^{n} \left(\frac{P_H}{P_i} \times C_i \right) \tag{2}$$

where P_H is also the period of the ρ-code.

In the case of m overlapping instructions (anchor nodes), each with an execution time of E_i, the total execution time of the ρ-code is given by:

$$C_{T,\rho} = \sum_{i=1}^{n} \left(\frac{P_H}{P_i} \times C_i \right) - \sum_{i=1}^{m} ((\delta_i - 1) \times E_i) \qquad (3)$$

where δ_i is the degree of overlap and is defined as the number of applications sharing a given anchor node.

C. Implicit Scheduling

The monolithic ρ-code obtained from the input applications is forwarded to the sensor nodes, where an interpreter executes it with a period equal to P_H. The design of the ρ-code is such that the constituent applications have explicitly non-overlapping variable space. The interpreter module has its own run-time stack to maintain its overall state, but it does not need to handle the responsibility of deciphering the individual applications inside the ρ-code. The schedule of each application is embedded in the sequence of instructions at the level of the hyper-period. If the total execution time without overlap, C_T, is less than the harmonizing period, the merged sequence ρ-code is guaranteed to finish the execution before the end of each period.

V. Evaluation

A. Comparison of Online vs. Proposed Solution

We compare the average power consumed by the radio of a sensor node with respect to the rate of reprogramming of the network. The comparison is shown in Figure 7. It is intuitive that more frequent reprogramming will consume more power. We compare the average power for the following scenarios.

1) The network is programmed using an online approach where a single application can be dynamically added.
2) Our proposed compile-time approach where a new monolithic ρ-code has to be sent to each node even if one application has been changed or added. The size of the monolithic ρ-code corresponds to 2 applications.
3) The ρ-code corresponds to 5 applications.

In this comparison, we assume that a node is only receiving application programming (bytecode) packets, and there is no other traffic in the network. We compare the average power consumption based on the assumption that the size of each application is equal to one data-packet of size 128 bytes and the power consumption of the radio is 56.4 mW (based on a CC2420 IEEE 802.15.4-compliant radio). We notice that the difference of power consumed between the online approach and the compile-time approach diminishes fairly quickly. For instance, even if the network is reprogrammed at a very high rate of every 100 seconds, the online approach will consume about $2\mu W$ on average, whereas our approach consumes about $11\mu W$ for a monolithic block of 5 applications. For more practical reprogramming rates of the order of days or weeks, the absolute difference in average power consumption between our approach and an online approach will be negligible even for very power-constrained sensor nodes. To put this comparison of power consumption in perspective, the average power consumed by a basic LPL-CSMA (Low Power Listen - Carrier Sense Multiple Access) medium access protocol (MAC) is about $138\mu W$ for a background operation of maintaining time synchronization within 5ms accuracy [10]. We can therefore infer that even for fairly frequent reprogramming at every 100 secs, the power consumed is at least an order of magnitude lower than just the overhead of a light-weight MAC protocol. Even if the size of each application is bigger than one packet, the power consumed by both the online and the compile-time approaches for sufficiently low reprogramming rates will be insignificant compared to the normal operation of the network.

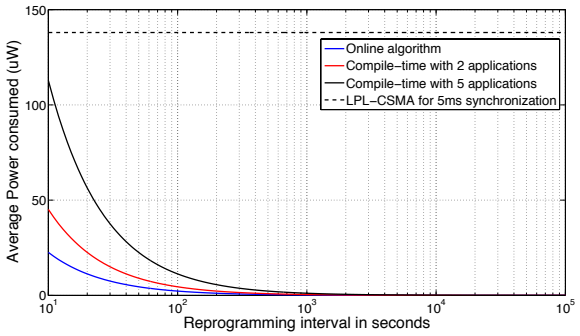


Fig. 7. Comparison of average power consumed by the radio of a sensor node with respect to the rate of reprogramming

B. Relative Energy Savings in Processor Usage

Energy savings in the processor usage after eliminating the redundancy in sensing requests can be estimated based on the degree of overlap δ by subtracting (3) from (2) and multiplying by the active power consumption of the processor. For the example scenario shown in Figure 6, the energy savings when the merged ρ-code is executed on the Firefly sensor platform [11] can be calculated as: $\Delta E = (2 + 1 + 1 + 1) * (490 * 10^{-3}) * (8.4 * 10^{-3})$. Hence, $\Delta E = 20.6\mu J$. On the other hand, the energy consumed by all applications running independently is approximately equal to $E_{orig} = 37.0\mu J$ if we ignore the negligible power consumed by other computation instructions. This corresponds to a significant 55% energy savings in processor usage for the particular example presented in Figure 6.

In addition to the above analysis, we conducted experiments to estimate the percentage power savings achievable from our approach in various cases. The results from these experiments are provided in Figures 8, 9 and 10. Each data point is collected by averaging across 50 iterations, and the error bars show the spread from the minimum to the maximum values over these iterations. These figures show percentage energy savings from our approach compared to the normal execution without any redundancy elimination. In this section, we consider only the processor usage because of the sensing requests, and we also assume that the energy consumption from other computations conducted on the processor is negligible in comparison. In Figure 8, energy savings are plotted in

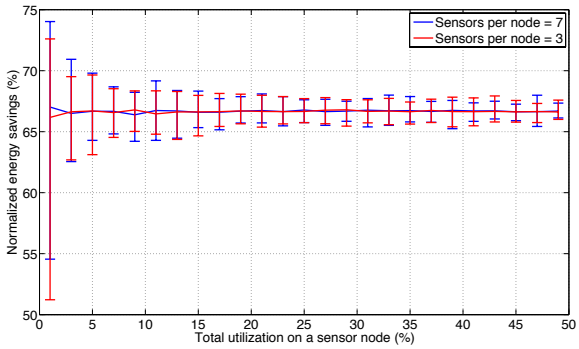


Fig. 8. Energy savings with respect to increase in utilization of processor with different number of sensors

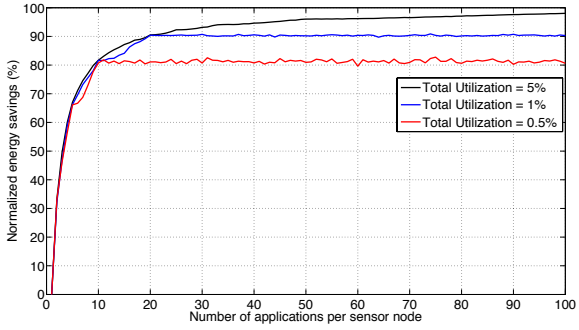


Fig. 9. Savings in average energy with increase in number of applications

the case of the execution of 5 randomly generated application strings on a sensor node and the total processor utilization is increased from 1% to 50%. Higher total utilization in our experiments arise from more sampling requests in the same ratio for each application. In this case, the average power savings remains more or less constant around 66%, but, with low utilization the error spread is quite high. This is because at low utilizations the number of sensing requests per application is low, and hence the possibility of finding redundancy is highly dependent on the type of the applications. As the utilization increases, the dependence of overall energy savings on application pattern reduces, as the chances of overlap are high anyway. When the number of applications deployed on a sensor node is increased, and the number of sensors per node is fixed to be 5, the energy savings increase as shown in Figure 9. This is because more applications can provide a higher degree of overlap, and hence more energy savings. The plot contains up to 100 applications just to illustrate the diminishing gains after a certain point. Such a large number of applications may, however, be impractical for most sensor nodes today. Figure 10 shows the reduction in energy consumption with respect to increasing the number of sensors on a node, and the average relative savings remains constant around 50% for 3 applications and 67% for 5 applications.

The intuition behind this behavior is the following. Even though the average degree of overlap, δ, may be lower for a larger number of sensors per node, equivalent energy savings are obtained. This is because, there is a proportional increase in the types of sensing requests (anchor nodes) that leads to

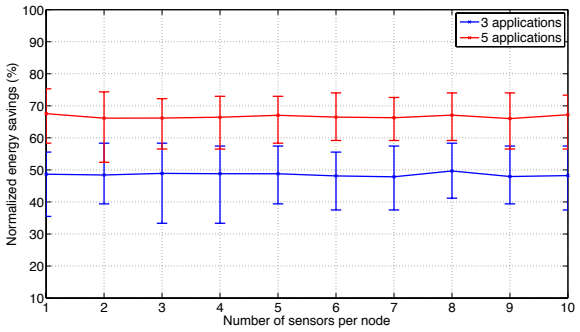


Fig. 10. Percentage energy reduced with an increase in the number of sensors

lesser overlap, since the total utilization is kept constant.

Overall, the achievable energy savings from the proposed approach is highly case-specific, but there is a high potential of energy savings if there are more applications or the utilization is high because of sensing intensive workload.

VI. Related Work

Redundancy elimination is a common optimization strategy in compilers, but it is mostly limited to the case of a single program. Several compiler optimizations have also been designed for multi-processor architectures for enhancing parallelism in sequential code [12], [13]. The direct application of such compiler techniques, however, is not possible in the case of sensor networks, because of the distributed nature of the network and the correlation of data to the physical environment and, hence, the physical location. A compiler for network-level programming of sensor networks should take into account the node characteristics including the hardware limitations and sensor peripherals, and the network interactions.

A scheme for sharing sensed data among multiple applications has been proposed in [14] by aligning sensing requests according to the periods, and sensing at time-instants providing the maximum overlap. The solution proposed by the authors is a runtime algorithm that can significantly increase the scheduler and timing complexity on a sensor node. Moreover, that work is limited to finding overlap in case of one sensor per node, and efficiently extending it for multiple sensors is not trivial. Integrating concurrency control at the device-driver level in sensor nodes (ICEM), proposed in [15], supports energy management by providing explicit interfaces which applications can leverage. In addition, ICEM also provides power locks that turn off the device if the lock is idle. In contrast, our approach not only shares the data from external devices, but also simplifies the execution on a node by eliminating the complexities arising from a scheduler.

Techniques for optimizing applications in sensor networks can find inspiration from the field of database research, as several optimizations have been developed over previous decades. Common expression detection proposed in [16] creates intermediate requests that assist reuse of intermediate data to save redundant accesses to overlapping sections of a relational database. Query optimization for detecting common data, as described in [17], also provides an improved solution based on interleaving smaller chunks of query execution.

These schemes are limited to parallel or temporally close queries, and optimized for large data-sets. A window-based solution is proposed in [18] to share data among independent dynamically-issued queries. Similar schemes may be applied to reduce redundancies across multiple queries in database-based approaches for sensor networks (like [19], [20]) allowing temporal reuse of data-subsets. However, a node-level mechanism is still required to eliminate redundant sensing requests from different network-level applications or queries.

VII. Conclusions and Future Work

In this paper, we have proposed and discussed a novel compiler-assisted scheduling approach that is able to identify and eliminate redundancies across applications in wireless sensor network infrastructures. Our approach models applications as linear sequences of executable instructions and we propose suitable algorithms for accomplishing such a model. We then show how it is possible to exploit and adapt well-known string-matching algorithms such as the Longest Common Subsequence (LCS) and the Shortest Common Supersequence (SCS) techniques to produce an optimal merged sequence of the multiple applications taking into account implicit scheduling information.

As modern radio designs support higher data-rates for the same amount of power, the optimizations on processor power consumption become more relevant for energy-saving and increasing the lifetimes of sensor networks. On the other hand, with the increase in the number of applications deployed on a sensor network, the overhead because of sampling the sensors can increase dramatically. However, by sharing sensing requests among applications, a significant percentage of resource-usage and energy can be saved on a sensor node. We demonstrate how our approach of using high-level optimization leads to significant network-wide resource savings, importantly energy. Our approach outperforms many other known techniques in the case of sensor node platforms supporting multiple sensors of multiple types. Our approach is highly predictable and its runtime is fairly simple: execution of bytecode with implicit scheduling. We show, based on experiments, that our proposed compile-time redundancy elimination approach can provide on an average about 60% energy savings on the processor with several simultaneous applications.

It can be argued that our application model is simplistic. It is, however, practical and it increasingly covers more and more scenarios of applications of large-scale sensor network deployments. Indeed, it does not support variable for-loops, and memory requirements can get prohibitive if loop unrolling is implemented. We will assess these issues in our future work. Our approach is a compile-time technique, and therefore all applications are affected if one application changes or is added. On the other hand, a dynamic run-time approach can add significant overhead to the bytecode interpreter on the sensor node. In order for a run-time approach to efficiently eliminate redundancies across applications, pre-profiling of those may be required that can result in significant memory and processor overhead. Moreover, a compile-time approach is still beneficial if the rate of reprogramming of the network is low.

As future work, we also plan to design a hierarchical system, where a monolithic REIS-Bytecode can be assigned to one of the tasks running on the operating system, rather than being the only executing block. Such a system can be modeled as an application of the classical bin-packing problem, where tasks can be clubbed together based on properties such as priority and memory requirements, providing scope for Quality-of-Service support in addition to resource optimizations.

References

[1] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[2] D. S. Hirschberg, "Algorithms for the longest common subsequence problem," *J. ACM*, vol. 24, pp. 664–675, October 1977.

[3] K.-J. Raiha and E. Ukkonen, "The shortest common supersequence problem over binary alphabet is np-complete," *Theoretical Computer Science*, vol. 16, no. 2, pp. 187–198, 1981.

[4] R. Newton, G. Morrisett, and M. Welsh, "The regiment macroprogramming system," in *Proceedings of the 6th international conference on Information processing in sensor networks*, ser. IPSN '07. Cambridge, MA, USA: ACM, 2007, pp. 489–498.

[5] P. Levis and D. Culler, "Matè: A tiny virtual machine for sensor networks," in *10th conference on Architectural support for programming languages and operating systems*, ser. ASPLOS-X. San Jose, California: ACM, 2002, pp. 85–95.

[6] "Atmel corporation, atmega 128rfa1 data sheet," 2011.

[7] "Chipcon inc., chipcon cc2420 data sheet," 2003.

[8] V. Gupta, J. Kim, A. Pandya, K. Lakshamanan, R. Rajkumar, and E. Tovar, "Nano-cf: A coordination framework for macro-programming in wireless sensor networks," in *In 8th IEEE Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON)*, 2011.

[9] P. Dutta, D. Culler, and S. Shenker, "Procrastination might lead to a longer and more useful life," 2007.

[10] A. Rowe, V. Gupta, and R. R. Rajkumar, "Low-power clock synchronization using electromagnetic energy radiating from ac power lines," in *the 7th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys '09. Berkeley, California: ACM, 2009, pp. 211–224.

[11] Rowe A., Mangharam R., Rajkumar R., "FireFly: A Time Synchronized Real-Time Sensor Networking Platform," *Wireless Ad Hoc Networking: Personal-Area, Local-Area, and the Sensory-Area Networks, CRC Press Book Chapter*, 2006.

[12] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe, "Space-time scheduling of instruction-level parallelism on a raw machine," in *Proceedings of the eighth Intl. Conf. on Architectural support for programming languages and operating systems*, ser. ASPLOS, San Jose, California, 1998, pp. 46–57.

[13] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek, "Fine-grain parallelism with minimal hardware support: a compiler-controlled threaded abstract machine," in *ASPLOS-IV*. Santa Clara, United States: ACM, 1991, pp. 164–175.

[14] A. Tavakoli, A. Kansal, and S. Nath, "On-line sensing task optimization for shared sensors," in *IPSN '10: Proceedings of the 9th ACM/IEEE Intl. Conf. on Information Processing in Sensor Networks*. Stockholm, Sweden: ACM, 2010, pp. 47–57.

[15] K. Klues, V. Handziski, C. Lu, A. Wolisz, D. Culler, D. Gay, and P. Levis, "Integrating concurrency control and energy management in device drivers," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, ser. SOSP '07. Stevenson, Washington, USA: ACM, 2007, pp. 251–264.

[16] S. Finkelstein, "Common expression analysis in database applications," in *Proceedings of the 1982 ACM SIGMOD international conference on Management of data*. Orlando, Florida: ACM, 1982, pp. 235–245.

[17] T. K. Sellis, "Multiple-query optimization," *ACM Trans. Database Syst.*, vol. 13, pp. 23–52, March 1988.

[18] S. Krishnamurthy, C. Wu, and M. Franklin, "On-the-fly sharing for streamed aggregation," in *the 2006 SIGMOD international conference on Management of data*. New York, NY, USA: ACM, pp. 623–634.

[19] Y. Yao and J. Gehrke, "The cougar approach to in-network query processing in sensor networks," *SIGMOD Rec.*, pp. 9–18, 2002.

[20] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "Tinydb: an acquisitional query processing system for sensor networks," *ACM Trans. Database Syst.*, vol. 30, no. 1, pp. 122–173, 2005.