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

Vikram Gupta

Eduardo Tovar

Karthik Lakshmanan

Raj (Ragunathan) Rajkumar

VG-CAS-11

Version:

Date: 06-16-2011

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

Vikram Gupta, Eduardo Tovar, Karthik Lakshmanan, Raj (Ragunathan) 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:

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

Abstract

Wireless sensor network nodes supporting multi-tasking and multiple concurrent applications are becomingincreasingly common. These nodes are typically equipped withmultiple sensors of various types. This trend has been fosteringthe design of wireless sensor networks allowing several concur-rent users to deploy applications with dissimilar requirements.At the same time, the practical burden of programmingindividual sensor nodes has led researchers to design macro-programming schemes able to program the network as a whole.In this paper, we extend the advantages of a holistic program-ming scheme by designing a novel compiler-assisted schedulingapproach (dubbed REIS) able to identify and eliminate redun-dancies across applications. To achieve this useful high-leveloptimization, we propose to model each user application as alinear sequence of executable instructions; we show how it isthen possible to exploit well-known string-matching algorithmssuch as the Longest Common Subsequence (LCS) and theShortest Common Super-sequence (SCS) to produce an optimalmerged sequence of the multiple applications that takes intoaccount embedded scheduling information. We demonstratehow this novel approach leads to significant network-wideresource savings, including energy.

Inter-application Redundancy Elimination in 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*—Wireless sensor network nodes supporting multi-tasking and multiple concurrent applications are becoming increasingly common. These nodes are typically equipped with multiple sensors of various types. This trend has been fostering the design of wireless sensor networks allowing several concurrent users to deploy applications with dissimilar requirements. At the same time, the practical burden of programming individual sensor nodes has led researchers to design macro-programming schemes able to program the network as a whole. In this paper, we extend the advantages of a holistic programming scheme by designing a novel compiler-assisted scheduling approach (dubbed REIS) able to identify and eliminate redundancies across applications. To achieve this useful high-level optimization, we propose to model each user application as a linear sequence of executable instructions; we show how it is then possible to exploit well-known string-matching algorithms such as the Longest Common Subsequence (LCS) and the Shortest Common Super-sequence (SCS) to produce an optimal merged sequence of the multiple applications that takes into account embedded scheduling information. We demonstrate how this novel approach leads to significant network-wide resource savings, including energy.**

***Keywords*-Wireless Sensor Networks; Scheduling; Optimization; Programming**

I. Introduction

Recent advances in hardware and operating systems ([1], [2], [3]) 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 use a given sensor networking infrastructure concurrently. 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)*. There is a high possibility of overlap in this paradigm, as different applications may contain several requests for sampling the same sensors.

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 might be interested in collecting temperature from the sensors for fine-grained temperature control, and a civil engineer wants to find 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. Reading a sensor value typically involves accessing the Analog-to-Digital Converter (ADC) module on the microprocessor, for converting the 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-network programming is a key technology for promoting a widespread adoption of wireless sensor networks. In turn, network-level programming requires elaborate optimization for application deployment as the nodes are highly resource-constrained. Several macro-programming and in-network programming approaches have been proposed in the past (e.g. [4], [5], [6], [7]). In spite of the convenience of such programming support, network-level resource-usage can go up significantly with the increase in number of applications deployed on a WSN infrastructure. In this paper, we propose a compiler-assisted scheduling approach to identify and eliminate redundancies across applications that enables reducing the processor and radio usage in the network. Our proposed solution creates a monolithic task-block resulting from the optimized merging of user applications with embedded scheduling information.

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 [8]. Eliminating redundancy across applications, however, is challenging in many respects. As most sensing applications are periodic in nature with low duty-cycles, eliminating redundant sections in case of mismatching periods can be difficult, and may not provide significant gains if elimination is carried 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 aforementioned problem of finding overlapping sensing requests issued by large 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 shall use the Longest Common Subsequence (LCS)[9] and the Shortest Common Super-sequence (SCS)[10] techniques.

The organization of the paper is as follows: First, we describe an overview of our approach in Section 2. We then present the background research and related work in Section 3. Section 4 and Section 5 respectively provide details of the modeling of applications and the proposed redundancy elimination approach. We provide evaluation of our approach in Section 6. We then conclude the paper with discussion sections describing the future work and the conclusions and limitations of our approach.

II. Overview of the Approach

We assume that the users develop network-level sensing applications using a programming framework such as Nano-CF [11]. The application code written by the users can either be abstract network-level using a macro-programming language or node-specific virtual-machines (for example Matè [7]). In both the cases, the underlying framework creates node-level intermediate code based on the application logic specified by the user. In this paper, we describe our approach 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 function for accessing the hardware (including sensing, GPIO access or packet transmission).

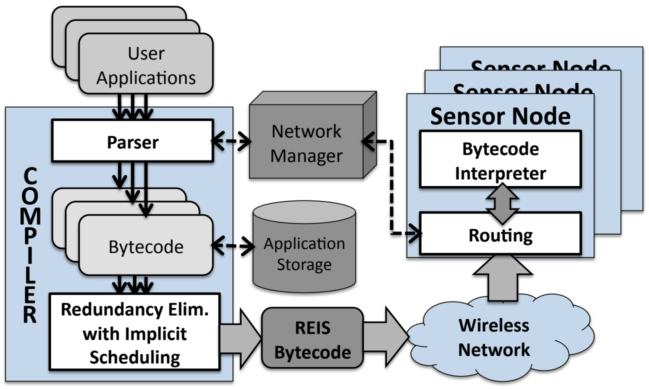


Figure 1. Overview of the approach for redundancy elimination among independent applications along with compiler-assisted scheduling

Bytecode corresponding to each application is 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 but the redundancies are eliminated according to temporal overlap of 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.

The approach assumes that a data link-layer and a suitable routing layer is 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 application bytecode and merge them using the REIS module whenever a new application is submitted. The logic of the user applications is interleaved inside the REIS-bytecode to provide maximum sharing of sensing requests and radio transmissions. Bytecode from different applications share non-overlapping variable and address space, which removes any need for context switching between applications, and the interleaving of bytecode provides an implicit schedule of execution.

The key motivation behind sharing sensing requests can be gleaned from the comparison of time taken for reading a sensor sample into memory with a simple memory-based instruction. Figure 2 shows the oscilloscope capture of this comparison on the Firefly [12] platform with the Nano-RK [1] operating system. This comparison is obtained by toggling a GPIO pin just before and after the execution of a sensor sampling instruction (shown by the Trace 1) and memory based loading of a 16 bit value into a register

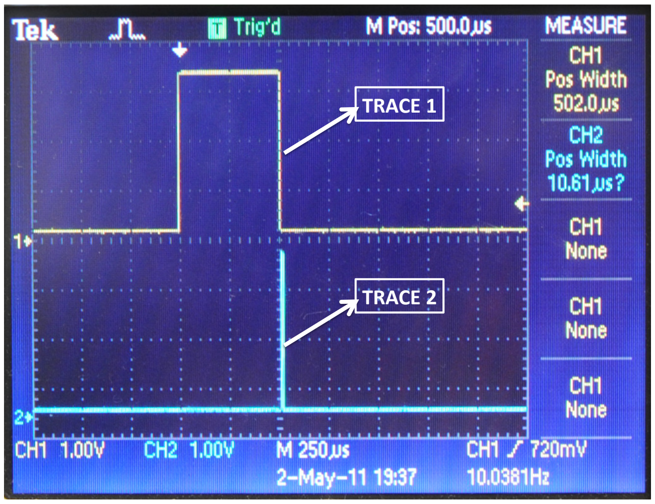


Figure 2. Oscilloscope screenshot showing two traces. Trace 1 (yellow trace) shows the time taken to acquire one sample reading of the light sensor on the Firefly sensor platform running Nano-RK, and Trace 2 (blue) shows the time taken for executing a simple variable assignment instruction

(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 Atmel ATMEGA1281 ($8MHz$) processor on the sensor node has on-chip memory, a load instruction takes a maximum of 3 cycles that corresponds 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 over most of the common sensing systems.

III. Related Work

Redundancy elimination is a common optimization strategy in compilers, but it is mostly limited to the case of a single program. Interprocedural redundancy elimination was proposed in [13] with focus towards communication optimization in distributed memory systems. Several compiler optimizations have also been designed for multi-processor architectures for enhancing parallelism in sequential code [14], [15]. 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.

The significance of supporting multiple applications on a sensor infrastructure has been stressed in the past; a detailed survey of sensor network programming approaches is provided in [16]. A layered architecture for a middleware to enable multiple independent applications on sensor networks has been proposed in [17]. Melete [18] builds on the Matè [7] virtual-machine concept for sensor networks to allow multiple concurrent applications. Other works such as [19], [20] focus on optimizing the operation and deployment of applications, respectively, on a geographically distributed wireless sensor network. Deployment of applications on a sensor network requires strategic mapping of applications to nodes. This problem is addressed in [21], where the authors develop a strategy for assigning nodes to applications based on a proposed 'Quality of Monitoring' metric. The approach is further improved in [22]. A scheme for sharing sensed data among multiple applications has been proposed in [19] 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, this work is limited to finding overlap in case of one sensor per node, and efficiently extending it for multiple sensors is not trivial.

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 [23] 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 [24], 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 [25] 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 [26], [27], [28]) 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.

IV. Application Modeling

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

Definition 1. *An application once parsed and converted into* *bytecode β consists of a list of subexpressions, which can be represented as a sequence of nodes. This string of nodes is called* ***Application Bytecode Sequence*** *and is referred to as β_η.*

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 `OP<K>` can have specific usage based on the bytecode. For the sake of clarity, example formats of some relevant bytecodes is 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`

A. Conversion to a sequence of nodes

Most data-collection 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. As algebraic computations are generally data-dependent, finding overlap for 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*.

Definition 2. *A node η in an application string is called an* ***Anchor Node*** *if it participates in detecting the overlap between applications. In this paper, a node η is an Anchor Node, if $\eta = S_t$ or $\eta = T$.*

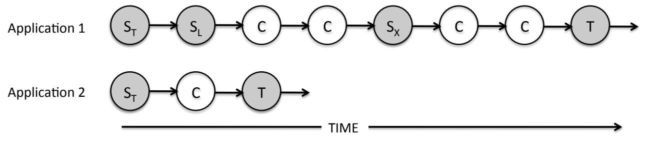


Figure 3. A simple example showing linearized execution sequence for one instance of two applications. Application 1 samples three different sensors at different points in time, and Application 2 just samples the temperature sensor and transmits its scaled-down value.

Conditional statements in an application logic may not allow application logic 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

Conditional statements have a general form as shown in the pseudo-code in Figure 4. As it cannot be known at the 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 IV.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 $\beta_{\eta a}$ re purely computational. Algorithm 2 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 sensed instruction, as shown in lines 19-21 in the algorithm. Algorithm 2 uses a subroutine called `create_node()` to create an instruction from the input parameters (shown in Algorithm 1). An example scenario for application of Algorithm 2 is shown in Figure 5, where the original S type node inside an if-condition is replaced by a S' node, representing a variable loading instruction.

```
  if (Condition1)
    Execute Section1
  else if (Condition2 )
    Execute Section2
  else if(ConditionM)
    Execute SectionM
  else
    Execute Section(M+1)
end
```

Figure 4. General form of an *if* conditional statement

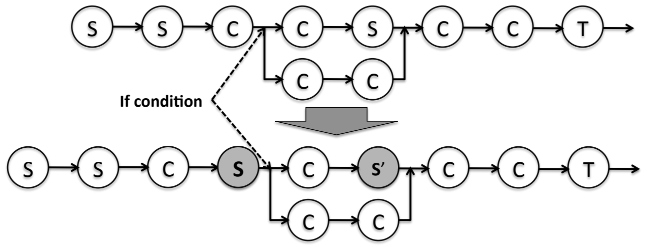


Figure 5. An example showing the modeling of an if-condition using Algorithm 2

Algorithm 1: create_node($type, argv[]$): for creating a bytecode subexpression node

```
Input  : $type, argv[]$: type of node, arguments for subexpression
Output: $\eta$: A subxpression node with a valid bytecode instruction
1 A node in the bytecode is of the form:
2 <TYPE OP1 OP2 OP3 . . . >
3 INITIALIZATIONS: $\eta := \emptyset$;
4 $\eta \cdot append(opcode(type))$;
5 foreach $arg \in argv$ do
6     $\eta \cdot append(arg)$;
7 end
```

C. Merging Packet Transmission

Based on Algorithm 2, 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 because in a previous work [11], we have already proposed a solution to harmonize packet transmissions from different applications. Instead of transmitting whenever 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 novel compiler-assisted scheduling approach. Many other solutions (such as [29]) have been proposed that optimize the network-wide scheduling of packets. For brevity purposes, we skip further details of packet transmission optimization in this paper.

D. Period Alignment

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

Algorithm 2: convert_app(A_i): for converting an application to bytecode sequence β_η

```
Input  : $A_i$: A user created application
Output: ($\beta_\eta$, $N_{an}$): Bytecode node sequence, Number of moved anchor nodes
1 Parse $A_i$ to bytecode $\beta$ using the parser
2 INITIALIZATIONS:
3 $\beta_{\eta:} = \emptyset$; $if\_index := \emptyset$; $node := \emptyset$
4 $if\_depth := 0$; $num\_T := 0$;
5 $N_{an} = 0$;
6 foreach sub-expression $\eta \in \beta$ do
7     $i = IndexOf(\eta)$
8     if $\eta$ is an if-clause then
9         $if\_depth++$;
10        $if\_index \cdot append(i)$;
11        $\beta_\eta \cdot append(\eta)$;
12    else if $\eta = S$ then
13        if if_index·isEmpty() then
14            $index = i$;
15        else
16            $index = if\_index(i)$;
17            $N_{an}++$;
18        end
19        $node := create\_node(\texttt{type}(\eta), var)$;
20        $\beta_\eta \cdot insert(index, node)$;
          // move $S$ node before the beginning
             of outermost if-condition
21        $node := create\_node(\texttt{assign}, var, \texttt{op2}(node))$;
          $\beta_\eta \cdot append(node)$;
22    else if $\eta$ 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 should remain at the
             same relative location
28    end
29 end
```

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 as:

$$P_H = LCM(P_1, P_2, \ldots P_n) \quad (1)$$

where LCM stands for Least Common Multiple.

Proposition 1. *The minimum repeating pattern of a task set, $\Gamma =< A_1, A_2, \ldots A_n >$ is of duration equal to the harmonizing period P_H.*

Proof: The proof is trivial because it follows directly from the definition of LCM. As applications are assumed to

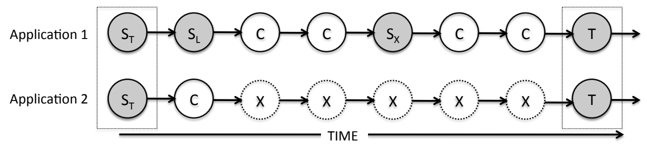


Figure 6. Application 2 modified to be aligned with application 1 for sharing sensing requests and packet transmission (based on example in Figure 3)

be strictly periodic and repeat at the start of every period, any time-duration less than the LCM of the periods cannot include integral multiples of all P_i, where $i = 1, 2, \dots n$.

V. Redundancy Elimination with Implicit Scheduling

A. String-Matching Algorithms

Once an application is modeled as a sequence of nodes as shown in the previous sections, the problem of finding overlapping sections among two or more applications can be reduced to finding a common subsequence between a pair of applications. Longest Common Subsequence (LCS) is a technique commonly used to find overlap between a pair of strings of symbols such that the relative order of common symbols is the same as in both 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 ordered 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 vice-versa may not be true. There are some commonly available solutions [9] that are guaranteed to return *a* longest ordered subsequence between a set of input strings.

LCSS can help find 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 it is exemplified in Figure 6 concerning 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 Supersequence (SCS)*[10].

Definition 3. *Given input sequences X and Y, the shortest common super-sequence, $Z = SCS(X, Y)$, is the shortest possible sequence such that both X and Y are subsequences of Z.*

In case of two input sequences, it is trivial to find SCS, if LCS is known. In case of more than two sequences, finding a

Algorithm 3: REIS(Γ): Generate a monolithic ρ-code with implicit scheduling from an input set of applications

```
Input  : Γ: a set of $n$ applications $< A_1, A_2, \dots A_n >$
         each with period $P_i$ for $i^{th}$ application
Output: ρ-code: a monolithic bytecode sequence
// From Equation 1
1  $P_H := LCM(P_1, P_2 \dots P_n)$
2  INITIALIZE:
3  for $i = 1 : n$ do
4      $\beta_{new,i} := \emptyset$;
5  end
6  foreach application $A_i \in \Gamma$ do
7      $(\beta_{\eta,i}, N_{an,i}) = convert\_app(A_i)$;
8      for $j = 1 : \frac{P_H}{P_i}$ do
9          $\beta_{new,i} := concatenate(\beta_{new,i}, \beta_{\eta,i})$;
               // create new strings with
               concatenated $\beta_\eta$
10     end
11 end
   // compute a shortest common supersequence
      (SCS)
12 ρ-code $= SCS(\beta_{new,1}, \beta_{new,2}, \dots \beta_{new,n})$;
```

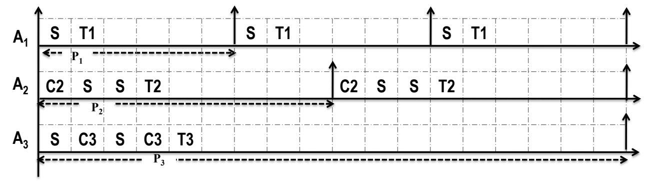
Shortest Common Supersequence is not a direct application of LCS.

Proposition 2. *For a given pair of bytecode sequences $< \beta_{\eta,j}, \beta_{\eta,k} >$ as an input, LCS finds the maximum possible overlap of anchor nodes.*

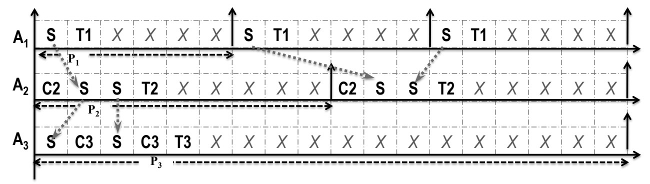
Proof: LCS has an optimal substructure, as justified by its recursive implementation. If the relative order of nodes in an application has to be maintained, the sequence generated by LCS is the longest subsequence in terms of the number of nodes.

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

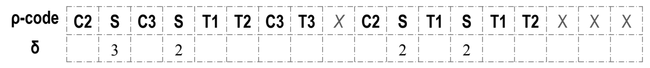
Let us consider that there is a set Γ of n independent applications, where each application is denoted by A_i and $i = 1, 2, \dots n$. The period of an application A_i is P_i. First of all, each application is converted into a sequence of bytecodes as described in Algorithm 2. The output of Algorithm 2 contains nodes within each periodic execution. As the periods can mismatch, the minimum length of time for which the overlap among 2 or more applicatins 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. After this operation all the sequences are of equal length of P_H. Thereafter, Shortest Common Supersequence (SCS) solution is applied to find a merged sequence ρ-code from the concatenated input bytecode. This approach is expressed by Algorithm 3.



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


A1
S T1 X X X X S T1 X X X X S T1 X X X X
P1
A2
C2 S S T2 X X X X X C2 S S T2 X X X X X
P2
A3
S C3 S C3 T3 X X X X X X X X X X X X X
P3

(b) Process of locating overlapping sensing requests. The output of SCS will aligned with respect to the location of nodes. The pattern repeats every hyper-period



(c) One possible output of the Algorithm 3, along with the degree of overlap of each shared sensing request

Figure 7. Identifying overlap in sensing instructions in three different applications and creating a merged ρ-code using Algorithm 3

An example for demonstrating the merging of bytecode is shown in Figure 7. There are three input application bytecodes as shown in Figure 7(a). Please note that all applications only sample one type of sensor for the sake of simplicity. The periods of 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 period of 9 units, and A_3 is $< S, C, S, T >$. It should be noted that 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 output length of the merged sequence is the shortest possible. Figure 7(b) shows a possible alignment of S nodes, and Figure 7(c) shows a merged sequence with overlapping S nodes omitted. The degree of overlap δ for each merged node is also shown.

Proposition 3. *For n applications to be executed on a sensor node, each with Worst Case Execution Times (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 the harmonizing period, and is also the period of ρ-code.

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

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

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 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 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.

VI. 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 8. It is intuitive that more frequent reprogramming will consume more power. We compare the power consumed for following three scenarios.

1) The network is programmed using an online approach where a single application can be dynamically added to the system.
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 monolithic ρ-code is equal to 2 applications.
3) The size of monolithic ρ-code is equal to 5 applications.

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 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. If the network is reprogrammed every 100 secs, the online approach will consume about $2\mu W$ on an average, whereas our approach

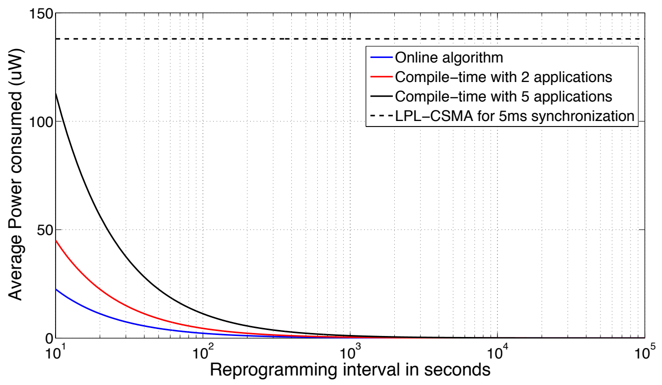


Figure 8. Comparison of average power consumed by the radio of a sensor node with respect to the rate of reprogramming of the network for online and compile-time approaches

consumes about $11\mu W$ for a monolithic block of 5 applications. This can be compared to the average power consumed by a basic LPL-CSMA (Low Power Listen - Carrier Sense Multiple Access) medium access protocol (MAC), which is about $138\mu W$ for a background operation of maintaining time synchronization within 5ms accuracy [30]. 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 online and compile-time approach will be insignificant compared to normal operation of the network.

B. Power savings

Energy savings in the processor usage (excluding the radio) available because of the redundancy elimination in sensing requests can be estimated based on the degree of overlap δ as follows:

$$\Delta E = E_{orig} - E_{merged} = \sum_{i=1}^{t} ((\delta_i - 1) \times E_i) \times P_w \quad (4)$$

where t is the number of overlapping sensing requests in a set of input applications, E_i is the execution duration of each instruction, and P_w is the power consumption of the processor. For the example scenario shown in Figure 7, the energy savings when the merged ρ-code is executed on the Firefly sensor platform 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, 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 7.

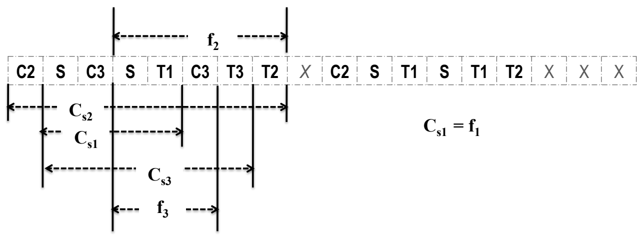


Figure 9. Various parameters for assessment of the proposed redundancy elimination approach shown with the merged ρ-code sequence generated in Figure 7

Table II
VALUES OF ASSESSMENT PARAMETERS FOR THE RESULTING ρ-CODE IN FIGURE 7

Application	WCET Spread C_s	Freshness F
A_1	4	4
A_2	8	5
A_3	6	3

C. Criteria for Assessment of REIS Compiler

WCET spread, C_s: C_s is defined as the total spread in execution time of an application instance in a merged application ρ-code.

Freshness, f: Freshness is the measure of maximum delay caused by early execution of an anchor node, because the sensed value may not be obtained close enough to the point in time where it is used. Freshness is the distance between an anchor node and next occurrence of a non-anchor node belonging to the same application.

The above parameters are shown in Figure 9 for the application scenario given in Figure 7. Please note that all the parameters are in time units, and measured in nanoseconds, but in the Table II we provide their values in number of units based on the divisions shown in Figure 9.

VII. HIERARCHICAL SCHEDULING AND FUTURE WORK

Let us consider a sensor network Operating System (OS) with multitasking support with real-time characteristics, where the tasks are scheduled by a scheduling policy Π. The problem for assigning user-applications in such an OS can be represented as a hierarchical scheduling problem as shown in Figure 10. The monolithic REIS-Bytecode (or ρ-code) can be assigned to one of the tasks running on the OS. Each of these tasks implement the bytecode interpreters.

The problem of assigning a given set of n applications to a set of k intermediate ρ-code blocks can be thought of as a classical application of the bin-packing problem in a similar fashion as in multiprocessor scheduling. The applications can be merged to intermediate ρ-code based on various criteria such as priority or memory requirements. We plan to explore this further as future work.

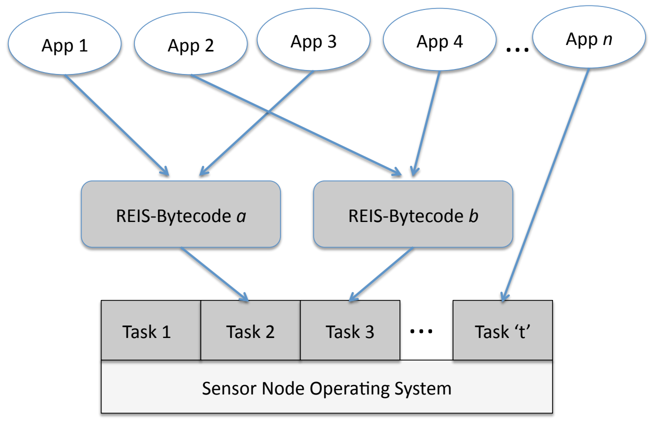


Figure 10. Hierarchical scheduling and merging of application on a multitasking sensor node operating system

VIII. Discussion and Conclusions

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 in which sensor node platforms allow multi-tasking and concurrent applications and the network is programmed as a whole following a macro-programming paradigm. Our approach proposes to cleverly model applications as linear sequences of executable instructions (we propose suitable algorithms for achieving that). We then show how it is then possible to exploit and adapt well-known string-matching algorithms such as the Longest Common Subsequence (LCS) and the Shortest Common Super-sequence (SCS) to produce an optimal merged sequence of the multiple applications taking into account implicit scheduling information.

With the increase in number of applications deployed on a sensor network, the overhead because of sampling the sensors can also 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 novel approach, which materializes this high-level optimization, leads to significant network-wide resource savings, importantly energy. No other related approach could achieve this 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.

It can be argued that our application model is simplistic. It is, however, practically well-applicable 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 higher if loop unroll is implemented, something that we will assess in future work. Our approach is a compile-time approach, 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, a pre-profiling of application may be required add both memory and processor overhead. Moreover, a compile-time approach is still be beneficial if the rate of reprogramming of network is low.

References

[1] A. Eswaran, A. Rowe and R. Rajkumar, "Nano-RK: an Energy-aware Resource-centric RTOS for Sensor Networks," *IEEE Real-Time Systems Symposium*, 2005.

[2] T. T. 2.x Working Group, "Tinyos 2.0," in *Proceedings of the 3rd international conference on Embedded networked sensor systems*, ser. SenSys '05. New York, NY, USA: ACM, 2005, pp. 320–320. [Online]. Available: http://doi.acm.org/10.1145/1098918.1098985

[3] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, ser. LCN '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 455–462. [Online]. Available: http://dx.doi.org/10.1109/LCN.2004.38

[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, Massachusetts, USA: ACM, 2007, pp. 489–498. [Online]. Available: http://doi.acm.org/10.1145/1236360.1236422

[5] R. Gummadi, N. Kothari, R. Govindan, and T. Millstein, "Kairos: a macro-programming system for wireless sensor networks," in *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*. Brighton, United Kingdom: ACM, 2005, pp. 1–2.

[6] M. S. Hossain, A. B. M. A. A. Islam, M. Kulkarni, and V. Raghunathan, "uSETL: A Set Based Programming Abstraction for Wireless Sensor Networks," in *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*. ACM, 2010, pp. 350–361.

[7] P. Levis and D. Culler, "Matè: a tiny virtual machine for sensor networks," in *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS-X. San Jose, California: ACM, 2002, pp. 85–95. [Online]. Available: http://doi.acm.org/10.1145/605397.605407

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

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

[10] 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.

[11] 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 *to appear: In Proceedings of the 8th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON), 2011*, ser. SECON-VIII. IEEE, 2011.

[12] 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.

[13] J. Knoop and B. Steffen, "The interprocedural coincidence theorem," in *Proceedings of the 4th International Conference on Compiler Construction*, ser. CC '92. London, UK: Springer-Verlag, 1992, pp. 125–140. [Online]. Available: http://portal.acm.org/citation.cfm?id=647471.727286

[14] 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 international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS-VIII. New York, NY, USA: ACM, 1998, pp. 46–57. [Online]. Available: http://doi.acm.org/10.1145/291069.291018

[15] 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 *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS-IV. New York, NY, USA: ACM, 1991, pp. 164–175. [Online]. Available: http://doi.acm.org/10.1145/106972.106990

[16] L. Mottola and G. P. Picco, "Programming wireless sensor networks: Fundamental concepts and state-of-the-art," *ACM Computing Surveys*, 2010.

[17] Y. Yu, B. Krishnamachari, and V. Prasanna, "Issues in designing middleware for wireless sensor networks," *Network, IEEE*, vol. 18, no. 1, pp. 15 – 21, 2004.

[18] Y. Yu, L. J. Rittle, V. Bhandari, and J. B. LeBrun, "Supporting concurrent applications in wireless sensor networks," in *Proceedings of the 4th international conference on Embedded networked sensor systems*, ser. SenSys '06. New York, NY, USA: ACM, 2006, pp. 139–152. [Online]. Available: http://doi.acm.org/10.1145/1182807.1182822

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

[20] J. Steffan, L. Fiege, M. Cilia, and A. Buchmann, "Towards multi-purpose wireless sensor networks," in *Systems Communications, 2005. Proceedings*, aug. 2005, pp. 336 – 341.

[21] S. Bhattacharya, A. Saifullah, C. Lu, and G.-C. Roman, "Multi-application deployment in shared sensor networks based on quality of monitoring," *Real-Time and Embedded Technology and Applications Symposium, IEEE*, vol. 0, pp. 259–268, 2010.

[22] Y. Xu, A. Saifullah, Y. Chen, C. Lu, and S. Bhattacharya, "Near optimal multi-application allocation in shared sensor networks," in *Proceedings of the eleventh ACM international symposium on Mobile ad hoc networking and computing*, ser. MobiHoc '10. New York, NY, USA: ACM, 2010, pp. 181–190. [Online]. Available: http://doi.acm.org/10.1145/1860093.1860118

[23] S. Finkelstein, "Common expression analysis in database applications," in *Proceedings of the 1982 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '82. New York, NY, USA: ACM, 1982, pp. 235–245. [Online]. Available: http://doi.acm.org/10.1145/582353.582400

[24] T. K. Sellis, "Multiple-query optimization," *ACM Trans. Database Syst.*, vol. 13, pp. 23–52, March 1988. [Online]. Available: http://doi.acm.org/10.1145/42201.42203

[25] S. Krishnamurthy, C. Wu, and M. Franklin, "On-the-fly sharing for streamed aggregation," in *to appear: Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks*.

[26] Y. Yao and J. Gehrke, "The cougar approach to in-network query processing in sensor networks," *SIGMOD Rec.*, vol. 31, no. 3, pp. 9–18, 2002.

[27] 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.

[28] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "Tag: a tiny aggregation service for ad-hoc sensor networks," in *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*. Boston, Massachusetts: ACM, 2002, pp. 131–146.

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

[30] A. Rowe, V. Gupta, and R. R. Rajkumar, "Low-power clock synchronization using electromagnetic energy radiating from ac power lines," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys '09. New York, NY, USA: ACM, 2009, pp. 211–224. [Online]. Available: http://doi.acm.org/10.1145/1644038.1644060