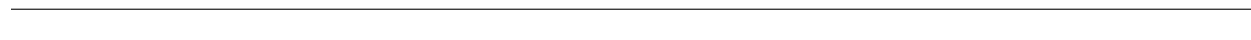
Compiler-Assisted Strategic Redundancy Elimination across Multiple Applications for Sensor Networks

Vikram Gupta[c,d,**], Josè Marinho[c], Eduardo Tovar[c], Ragunathan (Raj) Rajkumar[d]

[a] *CISTER Research Center (ISEP),*
Polytechnic Institute of Porto, Portugal
[b] *Real-Time Multimedia Systems Laboratory, Electrical and Computer Engineering,*
Carnegie Mellon University, Pittsburgh, USA



*Corresponding author
Email addresses: vikramg@ece.cmu.edu
(Vikram Gupta), jmsm@isep.ipp.pt (Josè Marinho),
emt@isep.ipp.pt (Eduardo Tovar), raj@ece.cmu.edu
(Ragunathan (Raj) Rajkumar)

Preprint submitted to Elsevier *April 24, 2013*

Compiler-Assisted Strategic Redundancy Elimination across Multiple Applications for Sensor Networks

Vikram Gupta[c,d,**], Josè Marinho[c], Eduardo Tovar[c], Ragunathan (Raj) Rajkumar[d]

[c] *CISTER Research Center (ISEP), Polytechnic Institute of Porto, Portugal*
[d] *Real-Time Multimedia Systems Laboratory, Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, USA*

Abstract

Wireless Sensor Networks are an important application of networked-embedded systems, and they have a key role to play in the development and materialization of the concept of *Internet-of-Things*. To promote the role of wireless sensor networks as an infrastructure technology in modern day-to-day life, support for multiple independent applications is essential, such that different users can concurrently submit their applications to accomplish diverse goals. With this perspective, several frameworks have been designed in the past to allow deployment and execution of concurrent applications on sensor networks. In this paper, we extend the advantages of a holistic over-the-air programming scheme by designing a novel compiler-assisted scheduling approach (called REIS) able to identify and eliminate redundancies *across* applications. Current generation sensor nodes can have various sensors of different types and it is quite probable that different applications may collect independent samples from the same sensors, leading to energy wastage because of redundant sampling across applications. To remove this redundancy, 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 the scheduling information. Furthermore, we propose a hierarchical assignment scheme where the applications may be merged into multiple intermediate blocks, rather than one large monolithic block. Our evaluation shows that significant energy savings can be obtained by removing redundancies in sensor sampling, while meeting the resource constraints on the sensor nodes.

Keywords: Wireless Sensor Networks; Energy Optimization; Scheduling; Compilers;

1. Introduction

Wireless Sensor Networks are one of the fast-growing and practical examples of networked-embedded systems. Research in the domain of sensor networks has an important role to play in the wide-spread adoption of the concept of *Internet-of-Things*. Wireless Sensor Networks consist of a network of small embedded platforms used to sense the ambient environment for various physical quantities such as temperature, pressure, humidity and light. The devices are typically called *motes* or *nodes*, and have a radio-interface to communicate with other devices in their neighborhood. The motes are either battery-powered or depend on energy harvesting, which limits the availability of resources (like energy-source, computational power, memory, radio etc.). Several challenges arise from the resource-constrained nature of these devices, and the applications deployed on sensor networks need to work within these resources. Originating from the concept of Smart-Dust [16], where many cheap-and-tiny embedded devices can be spread on an area, sensor nodes have evolved into more complex platforms with elaborate features like TelosB [26], Mi-

**Corresponding author
Email addresses: vikramg@ece.cmu.edu (Vikram Gupta), jmsm@isep.ipp.pt (Josè Marinho), emt@isep.ipp.pt (Eduardo Tovar), raj@ece.cmu.edu (Ragunathan (Raj) Rajkumar)

Preprint submitted to Elsevier *April 24, 2013*

caZ [34] and Firefly [29]. Similarly, the software for these devices has also evolved and several operating systems, programming abstractions and applications have been designed for modern hardware [2]. Furthermore, support for multiple applications is an important enabler for widespread adoption of sensor networks, such that independent users can submit their requirements to the sensor networking infrastructure.

Executing multiple applications on a sensor network brings new challenges for the already resource-constrained nodes, such as over-the-air programming [38], assignment of applications to nodes [4] and energy management [5]. There are various aspects in which there is scope for resource optimization on sensor networks, and the execution of multiple applications opens new dimensions in this regard. Typical sensor network platforms have several sensors of different kinds, allowing users with diverse requirements to deploy their applications. 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 as described in our previous work [13]. This work also provides further optimizations based on a hierarchical assignment scheme such that redundancy elimination is maximized within the resource constraints on sensor nodes.

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 the sensing requests among the 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. The challenges involved in such an approach 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 [17]. 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 identify-

ing 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)[15] and the Shortest Common Supersequence (SCS)[27] techniques. Our proposed solution creates a monolithic task-block resulting from the optimized merging of user applications with embedded scheduling information.

It might become impractical to combine all the applications into one large monolithic-block, as the size of the block might grow to be too large. As a further optimization, we present a hierarchical approach, where the tasks are merged into more than one intermediate task-blocks such that certain constraints regarding timeliness and memory usage are satisfied. The task-blocks are then executed as independent tasks on the sensor nodes. We show in this paper that this hierarchical scheduling problem can be modeled similar to that of a classical bin-packing problem. We provide certain approximations such that the problem can be reduced to that of a quadratic programming and hence, solvable within a reasonable time complexity.

The organization of the rest of this paper is as follows. First, we provide an overview of our approach in Section 2. Section 3 and Section 4 provide the details of the modeling of applications and the proposed redundancy elimination approach, respectively. In Section 5, we describe the hierarchical scheduling approach and model the problem as that of a quadratic program. We evaluate our approach in Section 6. The background research and related work is presented in Section 7. We then conclude the paper with a section on future work, the conclusions and the limitations of our approach.

2. Overview and Motivation

We assume that the users develop network-level sensing applications using a higher-level program-

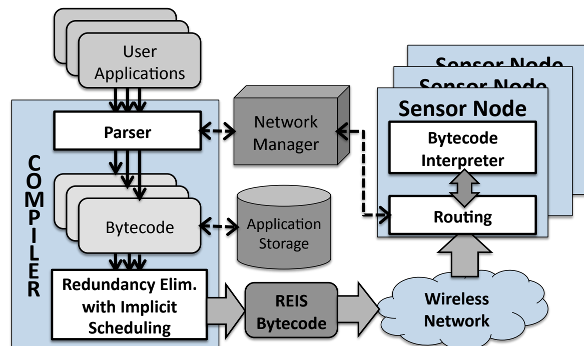


Figure 1: Overview of the approach for redundancy elimination

ming framework. The application code written by the users can either be at an abstract network-level, using a macro-programming language like Regiment [25], or it can use node-specific virtual-machines (for example Matè [21]). 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 function for accessing the hardware (including sensing, GPIO access or packet transmission). Bytecode corresponding to all the 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. The 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 the 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 is obtained. Bytecode from different applications share non-overlapping variable 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 cycles, which correspond to 375 nanoseconds. A majority of the time consumed in the case of Trace 2 is due to the pin toggling. Hence, a sensor sampling instruction consumes up to $\frac{(500-10)\times10^{-6}}{375\times10^{-9}} = 1306$ times more processor cycles. 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 [3] support 2 Mbps data rate, compared to 250 Kbps from the commonly used CC2420 [6] 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.

If the relative sequence of the sensor sampling requests is not important, then a caching-based solution can also be a possibility, where the sensor

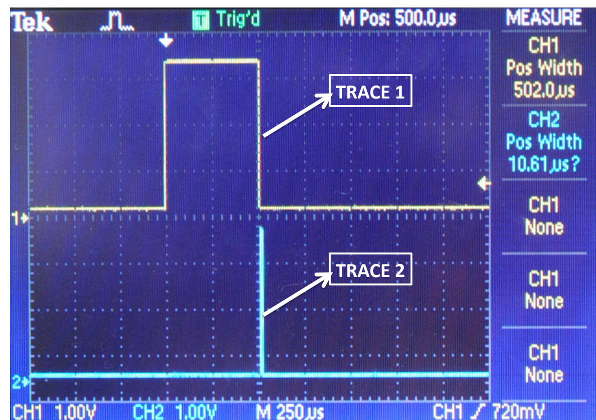


Figure 2: Oscilloscope screenshot showing the comparison of the time taken for reading a sensor (Trace 1) against a memory-based operation (Trace 2)

readings are cached in memory along with a time-stamp. Whenever an application requests a new sample, the cached value is checked for its *freshness* and if it newer than a threshold, the value is used as it is. Otherwise a new sensor sample is taken and provided to the application. Such a solution, devised using a wrapper function shown in Figure 3, is disadvantageous in a few major ways compared to our compile-time approach. Firstly, caching may not be practical in applications with fast-sampling rate. Secondly, it may not be directly applied in the cases where the relative order of samples from different sensors is important. As an example, an application may want to know when the light turned is on in a room by reading a light sensor, and then compare it with the reading from a motion sensor. Cached values in such a case may jeopardize the application semantics. If caching still needs to be used, the application behavior should be modified to be able to compare the time-stamps corresponding to different sensor samples requiring additional state maintenance. Thirdly, caching requires comparing of time-stamps which are typically 32-bit values; through simple experiments we found that it can take approximately $120\mu s$ to return the cached value (using the *else* path in Figure 3). This is significantly more computationally expensive as compared to reading a 16-bit value stored in memory with the Nano-RK [1] operating system running on a Firefly [29] sensor node that takes about $10\mu s$, as explained earlier and shown in Figure 2.

```
1 typedef struct {
2   int16_t value;
3   time_t  curr_time;
4   } sensor_t;
5 sensor_t sensor;
6 //wrapper function definition
7 sensor_t get_sensor_val(SENSOR){
8   time_t ct;
9   ct = get_curr_time();
10   if ( ct-sensor.curr_time > THRESH){
11     // collect new sensor sample using
12     // the original function
13     sensor.value = get_sensor(SENSOR);
14     sensor.curr_time = get_curr_time();
15   }
16   else {
17     //return the cached value
18     return sensor;
19   }
20   return sensor;
21 };
```

Figure 3: Pseudo-code showing the wrapper function to collect sensor readings from a cache-based solution

3. Application Modeling

Our proposed optimizations are aimed at applications whose main goal is to sample the sensors, process the sensor output 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 opcodes and corresponding operand values, 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, some example formats of some relevant bytecodes are provided in Table 1. Specific implementations can vary based on the design of the Parser and the bytecode Interpreter.

Even though bytecode resembles an assembly written program, it has some differences. In the bytecode, the operands are variables corresponding to the code written by the application programmer, whereas in assembly the operands are either registers or immediate values. This removes the problem of ensuring consistency with respect to each application context present in the set of hardware registers when performing the merge. It is the responsibility of the bytecode interpreter at the end-node to resolve the variables' addresses and load them into registers whenever these are referred to in the bytecode. The bytecode we consider in this work is a one-level higher abstraction than executable binaries, which is delivered to each node through the network-level programming framework.

Table 1: Example bytecode structure for some relevant subexpression instructions

Type	**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 V1 V2`	Transmit `V1` & `V2` to `DEST` node
Compute	`C VAR1 VAR2 VAR3`	`VAR1:=VAR2 'C' VAR3`

3.1. 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 the 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 4. S_t represents a sensing request for sensor type t, where t can either be temperature, light, accelerometer 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 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-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.

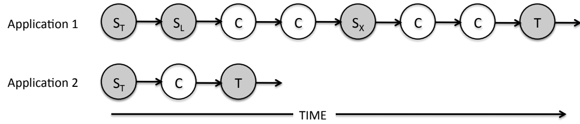


Figure 4: An example showing a linearized execution sequence for one instance of two applications. Application 1 samples three different sensors, and Application 2 samples the temperature and transmits its scaled-down value.

3.2. 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 directly possible to create a ρ-code (REIS-bytecode) from the input bytecodes based on a linear application model as described in Section 3.1. 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. We provide 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. An example scenario is shown in Figure 5, 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.

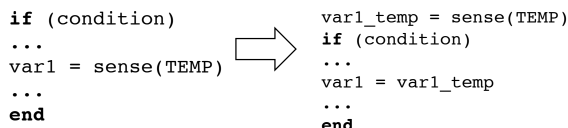


Figure 5: An example showing the modeling of an if-condition

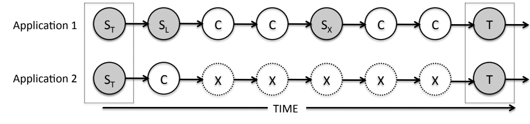


Figure 6: Application 2 modified to be aligned with Application 1 for sharing sensing requests and packet transmission (based on example in Figure 4)

4. Redundancy Elimination with Implicit Scheduling

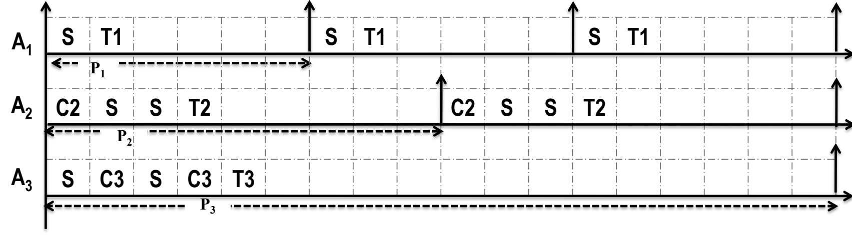
4.1. String-Matching Algorithms

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 [15] that are guaranteed to return *a* longest ordered subsequence between a set of input strings.

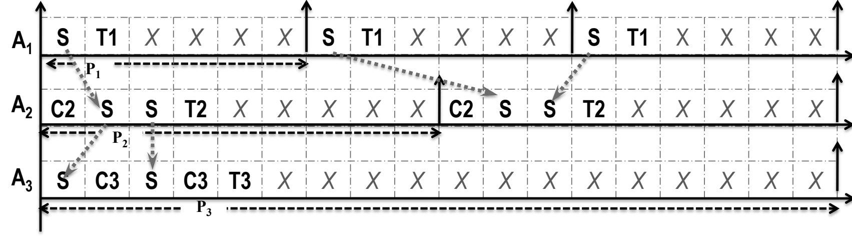
LCSS can help in finding redundant anchor nodes that appear consecutively in the input sequences. As an improvement over LCSS, LCS finds a subsequence with maximum overlap such that the relative order of nodes is not sacrificed. One or more of the input applications may be 'stretched' at various points, as illustrated in Figure 6 after applying LCS to the applications shown in Figure 4. An optimal merger of input sequences can be obtained by using an approach related to LCS called *Shortest Common Super-sequence (SCS)*[27].

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



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



(b) Process of locating overlapping sensing requests. The pattern repeats every hyperperiod

ρ-code	C2	S	C3	S	T1	T2	C3	T3	X	C2	S	T1	S	T1	T2	X	X	X
δ		3		2							2		2					

(c) One possible output of finding SCS, 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

than two sequences, finding a SCS is not a direct application of the LCS solution.

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.

4.2. 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 nodes as described in Section 3. The resultant strings contain 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 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 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 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 7(b) shows a possible alignment of the S nodes, and Figure 7(c) shows a merged sequence with the overlapping S nodes omitted. The degree of overlap δ for each merged node is also shown.

4.3. 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.

5. Hierarchical Assignment

In the approach described so far, all the user applications are merged into one monolithic task-block, which may have some disadvantages as listed below.

- Different periods of the deployed-applications may result in a large hyper-period, causing the size of the task-block to be prohibitively large. This not only increases the memory footprint, but also increases the number of packets required to transmit merged applications to the end-nodes.

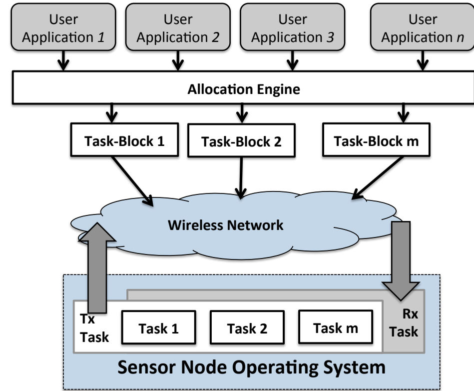


Figure 8: Hierarchical scheduling

- The applications may lose some timeliness because different parts of the applications may be executed with varying delays in the process of finding a better overlap.

- Some tasks may suffer significant delays in capturing the sensor sample and then using it, which may negatively impact the application semantics.

- Data from some critical tasks (e.g., monitoring fire) may not be delivered with the required responsiveness.

- Including less sensing-intensive applications in the task-block may not provide considerable energy savings.

We now present a hierarchical approach where we merge the applications into one or more intermediate task blocks instead of a large monolithic block. Let us consider a sensor node Operating System (OS) with support for multiple concurrent periodic tasks. The problem of executing the user-applications in such an OS can be represented as a hierarchical assignment one as shown in Figure 8. The goal is to strategically combine user-application tasks into multiple task-blocks such that several application requirements can be met. For example, merging applications with differing periods can cause the task-block to have a large memory footprint, and it may be beneficial to create more than one task-blocks such that tasks with similar periods are together. Similarly, creating task blocks from input tasks which share the same sensors may help in saving more energy.

5.1. Problem formulation

In order to find the assignment of tasks to intermediate blocks, we consider only the string model of the tasks. Other properties of the tasks, such as the semantics and timing behavior, are maintained as described earlier in Section 3. Let us assume that we have a set τ of n tasks deployed on the sensor nodes, and the corresponding bytecode string set is $\mathbb{B}$. The ith string is represented by β_i, where $i = 1, 2, \ldots, n$. The tasks can be mapped to m task-blocks denoted by ρ_j, where $1 \leq j \leq m$ and $m \leq n$. The task-block is denoted by the symbol ρ because each task-block corresponds to a merged sequence of application strings, which we termed as REIS-bytecode or ρ-code as in Section 4. The challenge is to find an optimal mapping of n bytecode strings to m blocks such that the total energy consumption of all the applications is minimized within certain constraints. We model the problem as that of quadratic integer programming, which is an NP-hard problem with a solution space growing exponentially with respect to n. We also derive suitable approximations that provide a solution within a reasonable time-complexity, but at the expense of optimality.

The total energy consumed by the system can be considered as the sum of the energy consumption of every application as follows:

$$E_{total} = \sum_{\forall i} E_{\beta_i} \qquad (2)$$

Also, if the tasks are combined into task blocks, the total energy consumed is the sum of the energy of all blocks. Therefore,

$$E'_{total} = \sum_{\forall j} E_{\rho_j} \qquad (3)$$

The energy consumption of the jth block can be obtained by subtracting the energy corresponding to the total degree of overlap across the tasks in the block from the total energy consumed by the tasks in the block, as follows.

$$E_{\rho j} = \sum_{\forall i \in \rho_j} E_{\beta_i} - \Delta_j \cdot e_s \qquad (4)$$

where, Δ_j is the total degree of overlap in the jth task-block, and e_s is the energy consumption of each sensing request. It results from (2), (3) and (4) that the energy savings after mapping tasks to task blocks can be estimated as:

$$E_s = E_{total} - E'_{total} = \sum_{\forall j} \Delta_j \cdot e_s \qquad (5)$$

From (5) it is clear that, to maximize the energy savings, the total degree of overlap across all task-blocks should be maximum. The estimation of Δ_j is challenging as it involves finding the Shortest-Common Supersequence (SCS) for application strings with respect to each block. The degree of overlap for a given set $B \subseteq \mathbb{B}$ of bytecode strings is given as:

$$\Delta_j = \sum_{\forall \beta_i \in B} (\frac{P_H}{P_i}) L_i - L_\rho \qquad (6)$$

where, P_H is the hyper-period of all the n tasks as defined in (1). P_i is the period of the ith task. L_i denotes the length of β_i in number of bytecode instructions or nodes (Figure 4), and L_ρ is the length of the task-block obtained by finding the SCS of the input tasks strings. The challenge, however, lies in calculating the SCS for all combinations of β_i's so that the optimum combinations can be chosen for m blocks where $\sum_{j=1}^{m} \Delta_j$ can be maximized. There can be a prohibitively large number of such combinations that can make the optimization problem intractable. The optimization problem can consist of terms of arbitrary polynomial degree up to n. The problem will require partitioning of set $\mathbb{B}$ into m number of disjoint subsets, and thus, there can be an exponential number of such partitions that are typically calculated by the means of Bell number and Bell polynomials [7]. In order to be able to find an optimum assignment of tasks to task-blocks, the SCS's of all possible partitions may need to be found along with solving the optimization problem. Hence, a major challenge lies in decoupling the problem from the calculation of SCS, so that it can be solved as a typical optimization problem with an objective function to be maximized while meeting some constraints.

Let us assume that the jth block has h applications and $\lambda_{k,l}$ denotes the longest-common subsequence of the kth and the lth string in the set. The following theorem helps to approximate the optimization problem to that of Quadratic Integer Programming (QIP) with linear constraints. Such problems can still be NP-hard, but commercial solvers [8, 14] are available that and be used to solve them. Subsequently, we further approximate the model to a Quadratic Continuous Program that

may be solved in polynomial time, but with loss of optimality.

Theorem 1. *The total degree of overlap over a set S of bytecode strings is less than or equal to the sum of the lengths of the Longest Common Subsequences (LCS's) of all the pairs in this set.*

$$\Delta_j \leq \frac{1}{2} \sum_{\forall k,l \in B} \lambda_{k,l} \quad s.t.\ k,l \leq n\ \&\ k \neq l \tag{7}$$

The equation is multiplied by a factor of half because $\lambda_{k,l} = \lambda_{l,k}$.

Proof. According to its definition, the Shortest Common Supersequence (SCS) of a set S of k strings contains all the elements of all the input strings but without any redundant elements. The SCS ρ of the strings in S is the shortest possible string such that all the strings in the set are subsequences of ρ. The shortest possible string can be found by removing redundant elements across all pairs of strings in the set S. The number of removed elements is equal to the sum of the lengths of the LCS's of all the pairs.

However, in this process, an element that participates in all possible pairs of LCS's of p $(2 < p < k)$ strings, gets removed $\binom{p}{2}$ times rather than its actual redundancy degree of $p-1$. As $\binom{p}{2}$ is greater than $p-1$, the total degree of overlap is less than the sum of the lengths of the LCS's. The equality prevails when no same elements occur in more than two LCS's.

□

As evident from the proof above, the exact value of the total degree of overlap is dependent on the elements of the constituent strings. Hence, it is not possible to isolate the problem formulation from finding the SCS, if an exact solution is to be found. The sum of the lengths of the LCS's of all the pairs provides an upper-bound on the degree of overlap across all strings, and is used as an approximation to strategically combine tasks into task-blocks. Based on Theorem 1, we can now reformulate the objective function so that the problem space can be reduced.

In order to formulate the optimization problem, we make use of *inclusion variables* denoted by X_i^j, where

$$X_i^j = \begin{cases} 1 & \text{if } \beta_i \text{ is assigned to block } j \\ 0 & \textit{otherwise} \end{cases} \tag{8}$$

Each task can only be assigned to one task-block to avoid repetition, which implies the following:

$$\text{for each } i, \sum_{\forall j} X_i^j = 1 \tag{9}$$

To find an optimal assignment of tasks to task-blocks, we quantify the degree of overlap by summing the pairwise degree of overlap multiplied by the inclusion variables, and the constraint in (9) accomplishes exclusion by making sure that each task is assigned to only one block. The goal of the optimization problem now is to find the values of the inclusion variables such that the total degree of overlap is maximized. Please note that the degree of overlap across two bytecode strings is the same as the LCS of the two, as described in Section 4. Hence, we can say:

$$\sum_{\forall \text{pairs} \in B} \lambda_{\text{each pair}} = \sum_{\forall a,b \in \{1,2,\ldots n\}, a \neq b} X_a^j X_b^j \delta_{a,b} \tag{10}$$

From Theorem 1, the objective function for the jth task-block can now be written in an expanded form as:

$$\begin{aligned} \Delta_j \leq\ & X_1^j X_2^j \delta_{1,2} + X_1^j X_3^j \delta_{1,3} + \cdots + X_1^n X_n^j \delta_{1,n} + \\ & X_2^j X_3^j \delta_{2,3} + \cdots + X_2^j X_n^j \delta_{2,n} + \cdots + \\ & X_{n-1}^j X_n^j \delta_{(n-1),n} \end{aligned} \tag{11}$$

We can now create a general form so that the total degree of overlap D_{total} across all task-blocks can be maximized. Let $\mathbf{D}$ represent the $n \times n$ matrix where the $(k,l)^{th}$ element is $\delta_{k,l}$. The degree of overlap of a string with itself is assumed to be zero, hence $\delta_{k,k} = 0, \forall k = \{1, 2, \ldots, n\}$.

$$\begin{aligned} D_{total} &= \sum_{j=1}^{m} \Delta_j \\ &\leq \frac{1}{2} \mathbf{X}_{n\times m}^T \mathbf{D}_{n\times n} \mathbf{X}_{n\times m} \end{aligned} \tag{12}$$

Most commercial optimization problem solvers, however, need the variables to be in a vector form. So, in order to vectorize $\mathbf{X}$, we need to replicate the matrix $\mathbf{D}$ into a matrix $\hat{\mathbf{D}}$ of size $M \times M$, where $M = m \times n$.

$$\hat{\mathbf{D}} = \begin{pmatrix} \mathbf{D}_{n\times n} & \mathbf{0}_{n\times n} & \cdots & \mathbf{0}_{n\times n} \\ \mathbf{0}_{n\times n} & \mathbf{D}_{n\times n} & \cdots & \mathbf{0}_{n\times n} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0}_{n\times n} & \mathbf{0}_{n\times n} & \cdots & \mathbf{D}_{n\times n} \end{pmatrix}_{M\times M} \quad (13)$$

Similarly, the vector form of $\mathbf{X}$ denoted by $\hat{\mathbf{X}}$ is:

$$\hat{\mathbf{X}} = \left[X_1^1, X_2^1, \ldots X_n^1, X_1^2, \ldots, X_n^2, X_1^m, \ldots X_n^m\right]^T \quad (14)$$

Now, the total degree of overlap from (12) can be rewritten as:

$$D_{total} \leq \frac{1}{2}\hat{\mathbf{X}}^T_{M\times 1}\hat{\mathbf{D}}_{M\times M}\hat{\mathbf{X}}_{M\times 1} \quad (15)$$

As the degree of overlap for any two strings is always positive, the total degree of overlap is maximized when all the tasks are assigned to one task-block. The problem, however, has other solutions if some other constraints come into play. As an example, if the maximum number of tasks in a task-block is fixed, then solving the above equation optimally assigns the tasks to specific task-blocks.

5.2. Constraints

The motivation behind the hierarchical assignment of tasks in this case is based on the fact that some constraints may not allow all the tasks to be merged into one monolithic block. These constraints may arise either because of some limitations on the sensor networking platform, or to satisfy timeliness/criticality requirements of the applications.

5.2.1. Memory Constraint

As mentioned earlier, the length of a task-block corresponding to the hyper-period of all input applications may result in too large task-blocks for the amount of memory available on a typical sensor node. Also, the length of reprogramming packets may become prohibitive. Therefore, the overall memory requirements of each block may be specified as a constraint as follows, where μ represents the maximum amount of memory that can be allocated to each task-block:

$$\text{for all } j, \sum_{i=1}^{n} \frac{P_H}{P_i} L_i X_{i,j} \leq \mu \quad (16)$$

where, P_H is the hyperperiod, and is equal to the Least Common Multiple of the periods of all the tasks. L_i denotes the length of the ith task in number of bytecode instructions or nodes.

5.2.2. Number Constraint

The size of the task-block is also dependent on the number of tasks allocated to it. In order to simplify the constraints, an upper-bound, U, on number of tasks-per-block can be set. The constraint in this case can be written as:

$$\text{for all } j, \sum_{i=1}^{n} X_{i,j} \leq U \quad (17)$$

5.3. Objective Function

?? Based on (15),(8),(9),(16) and (17), we can formulate the optimization problem as follows:

$$\begin{aligned} \underset{\hat{\mathbf{X}}}{\text{maximize}} \quad & \frac{1}{2}\hat{\mathbf{X}}^T\hat{\mathbf{D}}\hat{\mathbf{X}} \\ \text{subject to:} \quad & \\ 1. \quad & X_i^j = 0 \text{ or } 1 \\ 2. \quad & \text{for each } i, \sum_{\forall j} X_i^j = 1 \\ 3. \quad & \text{for all } j, \sum_{i=1}^{n} \frac{P_H}{P_i} L_i X_i^j \leq \mu \\ 4. \quad & \text{for all } j, \sum_{i=1}^{n} X_i^j \leq U \end{aligned}$$

The objective function is an upper-bound on the total degree of overlap as explained with Theorem 1, but can serve as a suitable approximation. This solution to this objective function provides an assignment where n tasks are allocated to m blocks such that the total degree of overlap across all the blocks can be maximized. Please note that the constraints 3 and 4 may or may not be simultaneously applied. Using any one of them makes sure that all the tasks are not merged into one task-block.

5.4. Continuous approximation

The objective function described in Section **??** can be solved with the constraint that the inclusion variables are binary, and hence the problem becomes that of a Quadratic Integer Programming. To reduce the time-complexity, the problem can be relaxed to a Quadratic Continous Program (QCP) where the inclusion variables can take continous values from 0 to 1 ($0 \leq X_i^j \leq 1$). The problem can now be solved in polynomial-time, but the solution fractionally assigns tasks to task-blocks. Simple heuristics can provide an integral solution, which may not be optimal, but is computationally inexpensive. We propose one as described below:

1. A solution is obtained by solving the QCP, where the tasks may be fractionally assigned to task-blocks.
2. The tasks in each block are sorted in a descending order of the values of inclusion variables.
3. Starting with the first block, the tasks are now assigned to blocks in a first-fit manner, until both the number constraint (Constraint 3) and the memory constraint (Constraint 4) are satisfied.
4. The same process is continued in the next block, until all the blocks are considered.
5. If all the tasks can not be assigned to blocks, while meeting the contraints 3 and 4, the algorithm returns with a failure

6. Evaluation

6.1. 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 9. 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

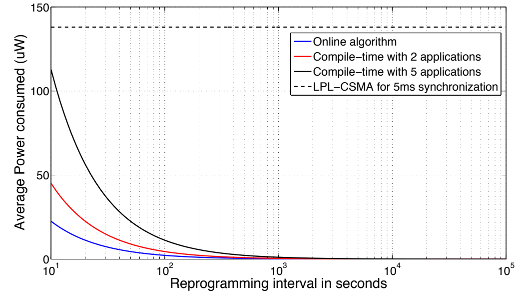


Figure 9: Comparison of average power consumed by the radio of a sensor node with respect to the rate of reprogramming

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 [28]. 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 as compared to the normal operation of the network.

6.2. 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 and multiplying by the active 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 [29] 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% of energy savings

in processor usage for the particular example presented in Figure 7.

In addition to the above analysis, we conducted experiments to estimate the percentage of power savings achievable with our approach in various cases. The results from these experiments are provided in Figures 10, 11 and 12. 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 as 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 10, energy savings are plotted in the case of the execution of 5 randomly generated application strings on a sensor node and with the total processor utilization varying 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 a low utilization the error spread gets 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 11. 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 that the gains do not increase much after a certain point. Such a large number of applications may, however, be impractical for most sensor nodes today. Figure 12 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

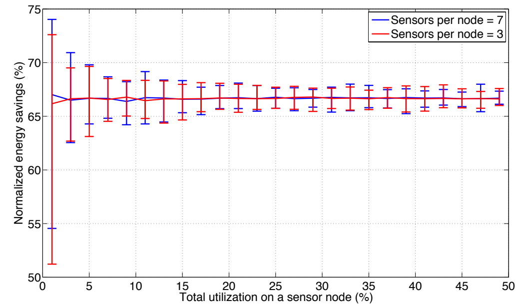


Figure 10: Energy savings with respect to increase in utilization of processor with different number of sensors

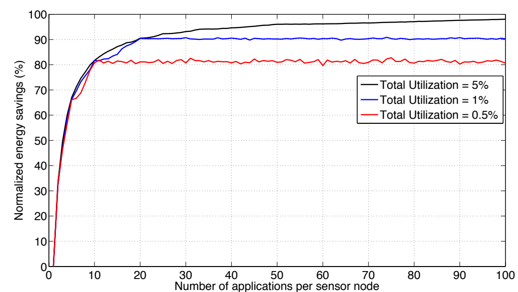


Figure 11: Savings in average energy with increase in number of applications

types of sensing requests (anchor nodes) that leads to 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 the sensing intensive workload.

6.3. Gains with Hierarchical Scheduling

The hierarchical assignment selectively merges tasks such that the degree of overlap is maximized within the given constraints of memory consumption or the maximum number of tasks allowed in each block. As shown in Section 5, Quadratic Integer Programming (QIP) can be used to compute an optimal assignment of tasks to task-blocks. We use the Gurobi optimizer [14] to solve the QIP. One example result with a maximum of 3 blocks is shown in Figure 13. We vary the number of tasks to be allocated from 4 to 13. Each block can have up to $(\lceil \frac{N_{tasks}}{N_{blocks}} \rceil + 1)$ number of tasks. For comparison, we also found an assignment using Quadratic Continuous Programming (QCP) where the inclusion variables $X_{i,j}$ can have real values rather than being integers. This reduces the computation time

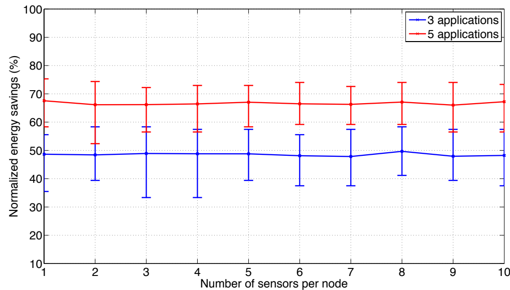


Figure 12: Percentage energy reduced with an increase in the number of sensors

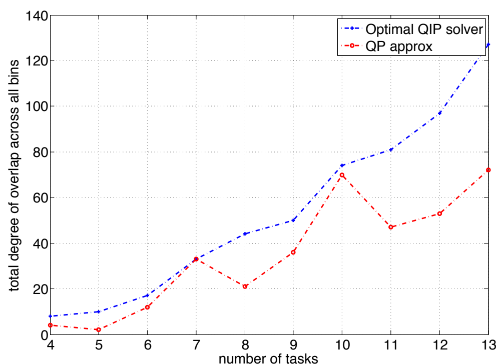


Figure 13: Relative energy savings in the case of hierarchical assignment. Better results are obtained using the optimal Quadratic Integer Programming (QIP) compared to an approximation obtained using Quadratic Continuous Programming (QCP or QP)

significantly, but the solution found may not be optimal. The values calculated for inclusion variables are rounded off, while making sure that number of tasks per block does not exceed a *maximum* threshold. The QIP computation time becomes unpractical as the number of tasks increase. Even for 20 tasks the computation time was in excess of 4 hours on a dual-core, 2.7 GHz machine.

7. Related Work

Several approaches in the past have stressed on the importance of supporting multiple concurrent applications on Wireless Sensor Networks [32]. A multi-application over-the-air programming system, Melete, was proposed in [38]. Melete combines Mate [21] with an information dissemination protocol, Trickle [22], to allow applications to be deployed on-the-fly. Earlier, the importance of middleware support for allowing multiple application was emphasized in [37]. A system for enabling sensor network as a shared infrastructure with independent application deployment was recently demonstrated in [11]. In addition to supporting multi-application deployment, the approach proposed in [4] describes strategic application deployment on sensors that achieves high Quality-of-Monitoring (QoM). The best-suited sensor nodes are automatically chosen for a given application. An optimal solution for this strategy was provided in an extension [35] of that work.

7.1. Sharing Sensors

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 multiprocessor architectures for enhancing parallelism in sequential code [20, 9]. 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.

With multiple applications executing on a sensor network, the overhead of sensing and radio-usage can be significant, and sometimes prohibitive, for the limited resources on sensor networks. Tavakoli *et.al* propose an optimization to share sensing across applications in [33] by finding temporal overlap. The redundancies are removed by using a joint data-flow graph, where each application specifies a sampling window rather than an instant in typical applications. Such a windowing approach may not be practical in the case of several types of sensors coexisting on a node, as extending the proposed approach is not trivial. In our approach, however, we find the maximum possible overlap in the case when applications sample different sensors at different time-instants, while maintaining the relative sequencing within the application logic. Integrating concurrency control at the device-driver level in sensor nodes (ICEM), proposed in [18], 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. The approach of detecting common expressions proposed in [12] creates intermediate requests that assist the reuse of intermediate data to save redundant accesses to overlapping sections of a relational database. Query optimization for detecting common data, as described in [30], also provides an improved solution based on interleaving smaller chunks of query execution. These schemes are limited to parallel or temporally close queries, and are optimized for large data-sets. A window-based solution is proposed in [19] 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 [36, 24]) allowing temporal reuse of data-subsets. However, a node-level mechanism is still required to eliminate redundant sensing requests from different applications or queries.

7.2. *Hierarchical Scheduling*

In the domain of real-time scheduling on uniprocessor and multiprocessor systems, hierarchical approaches have been employed in the past for providing isolation, scalability and improve the task-allocation. A two-level scheduling approach for uniprocessor systems was proposed in [10] and there have been several similar works (e.g., [23]) that allow the allocation of tasks on processors via intermediate *servers*, where servers represent abstract resources with pre-allocated budgets. In the case of multiprocessors, the task assignment is facilitated by virtually clustering the input tasks and allocating clusters as one entity [31]. Our proposed scheme is conceptually similar to such hierarchical systems, where tasks are merged into intermediate task-blocks to exploit energy savings while ensuring other important requirements like timeliness or memory footprint are fulfilled. The major difference between our approach and the above mentioned schemes is in that while hierarchical schemes are focussed mainly towards providing hard timeliness guarantees, we focus primarily on reducing energy consumption by eliminating redundancy of sensing with soft real-time considerations.

8. 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. 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 Super-sequence (SCS) to produce an optimal merged sequence of the multiple applications with implicit scheduling. We also propose a hierarchical system, where the redundancies are removed across multiple subsets of applications, rather than all applications at once. Under this approach, tasks are merged into intermediate task-blocks, rather than one monolith. The task-blocks execute as independent applications on the sensor node such that the resource constraints are met, and redundant sensing requests are eliminated within a task-block.

As modern radio designs support higher data-rates for the same amount of power, the optimizations on processor power consumption becomes more relevant in energy-savings and therefore in 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 considerable energy savings on the processor executing 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. In-

deed, 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.

Acknowledgements

This work was supported by FCT (Portuguese Foundation for Science and Technology) and by European Regional Development Fund (ERDF) through COMPETE (Operational Programme 'Thematic Factors of Competitiveness') under project ref. FCOMP-01-0124-FEDER-022701; also by the FCT under the Carnegie Mellon Portugal Program, with grant ref. FCT-CMU-PT/0012/2006; and by the European Commission - CONET NoE, grant ref. FP7-ICT-224053.

References

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

[2] Akyildiz, I., Su, W., Sankarasubramaniam, Y., Cayirci, E., 2002. Wireless sensor networks: a survey. Computer Networks 38 (4), 393 – 422. URL http://www.sciencedirect.com/science/article/pii/S1389128601003024

[3] Atmel, 2011. Atmega 128rfa1 data sheet.

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

[5] Boulis, A., Srivastava, M., Nov. 2004. Node-level energy management for sensor networks in the presence of multiple applications. Wirel. Netw. 10 (6), 737–746. URL http://dx.doi.org/10.1023/B:WINE.0000044032.41234.d7

[6] Chipcon, 2003. Chipcon cc2420 data sheet.

[7] Comtet, L., 1974. Advanced Combinatorics. Reidel, Dordrecht.

[8] CPLEX, 2013. Ibm ilog cplex optimizer. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/.

[9] Culler, D. E., Sah, A., Schauser, K. E., von Eicken, T., Wawrzynek, J., 1991. 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. ASPLOS-IV. ACM, Santa Clara, California, United States, pp. 164–175.

[10] Deng, Z., Liu, J.-S., dec 1997. Scheduling real-time applications in an open environment. In: Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE. pp. 308 –319.

[11] Efstratiou, C., Leontiadis, I., Mascolo, C., Crowcroft, J., 2010. Demo abstract: A shared sensor network infrastructure.

[12] Finkelstein, S., 1982. Common expression analysis in database applications. In: Proceedings of the 1982 ACM SIGMOD international conference on Management of data. SIGMOD '82. ACM, Orlando, Florida, pp. 235–245.

[13] Gupta, V., Tovar, E., Lakshmanan, K., Rajkumar, R., 2012. Inter-application redundancy elimination in wireless sensor networks with compiler-assisted scheduling. In: Industrial Embedded Systems (SIES), 2012 7th IEEE International Symposium on. IEEE, pp. 112–119.

[14] Gurobi, 2013. Gurobi optimizer. http://www.gurobi.com/products/gurobi-optimizer/gurobi-overview.

[15] Hirschberg, D. S., October 1977. Algorithms for the longest common subsequence problem. J. ACM 24, 664–675.

[16] Kahn, J. M., Katz, R. H., Pister, K. S. J., 1999. Next century challenges: mobile networking for smart dust. In: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking. MobiCom '99. ACM, New York, NY, USA, pp. 271–278. URL http://doi.acm.org/10.1145/313451.313558

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

[18] Klues, K., Handziski, V., Lu, C., Wolisz, A., Culler, D., Gay, D., Levis, P., 2007. Integrating concurrency control and energy management in device drivers. In: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles. SOSP '07. ACM, New York, NY, USA, pp. 251–264. URL http://doi.acm.org/10.1145/1294261.1294286

[19] Krishnamurthy, S., Wu, C., Franklin, M., 2006. On-the-fly sharing for streamed aggregation. In: Proceedings of the 2006 ACM SIGMOD international conference on Management of data. SIGMOD '06. ACM, New York, NY, USA, pp. 623–634. URL http://doi.acm.org/10.1145/1142473.1142543

[20] Lee, W., Barua, R., Frank, M., Srikrishna, D., Babb, J., Sarkar, V., Amarasinghe, S., 1998. 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. ASPLOS-VIII. ACM, San Jose, California, United States, pp. 46–57.

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

[22] Levis, P., Patel, N., Culler, D., Shenker, S., 2004. Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1. USENIX Association, Berkeley, CA, USA, pp. 2–2.
URL http://dl.acm.org/citation.cfm?id=1251175.1251177

[23] Lipari, G., Bini, E., April 2005. A methodology for designing hierarchical scheduling systems. J. Embedded Comput. 1, 257–269.
URL http://dl.acm.org/citation.cfm?id=1233760.1233768

[24] Madden, S. R., Franklin, M. J., Hellerstein, J. M., Hong, W., 2005. Tinydb: an acquisitional query processing system for sensor networks. ACM Trans. Database Syst. 30 (1), 122–173.

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

[26] Polastre, J., Szewczyk, R., Culler, D., 2005. Telos: enabling ultra-low power wireless research. In: Proceedings of the 4th international symposium on Information processing in sensor networks. IPSN '05. IEEE Press, Piscataway, NJ, USA.
URL http://dl.acm.org/citation.cfm?id=1147685.1147744

[27] Raiha, K.-J., Ukkonen, E., 1981. The shortest common supersequence problem over binary alphabet is np-complete. Theoretical Computer Science 16 (2), 187 – 198.

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

[29] Rowe A., Mangharam R., Rajkumar R., 2006. 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.

[30] Sellis, T. K., March 1988. Multiple-query optimization. ACM Trans. Database Syst. 13, 23–52.

[31] Shin, I., Easwaran, A., Lee, I., july 2008. Hierarchical scheduling framework for virtual clustering of multiprocessors. In: Real-Time Systems, 2008. ECRTS '08. Euromicro Conference on. pp. 181 –190.

[32] Steffan, J., Fiege, L., Cilia, M., Buchmann, A., aug. 2005. Towards multi-purpose wireless sensor networks. In: Systems Communications, 2005. Proceedings. pp. 336 – 341.

[33] Tavakoli, A., Kansal, A., Nath, S., 2010. 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. ACM, Stockholm, Sweden, pp. 47–57.

[34] Technology, C., 2013. Micaz sensor network platform datasheet. http://bullseye.xbow.com:81/Products/Product_pdf_files/Wireless_pdf/MICAz_Datasheet.pdf.

[35] Xu, Y., Saifullah, A., Chen, Y., Lu, C., Bhattacharya, S., 2010. Near optimal multi-application allocation in shared sensor networks. In: Proceedings of the eleventh ACM international symposium on Mobile ad hoc networking and computing. MobiHoc '10. ACM, Chicago, Illinois, USA, pp. 181–190.

[36] Yao, Y., Gehrke, J., 2002. The cougar approach to in-network query processing in sensor networks. SIGMOD Rec. 31 (3), 9–18.

[37] Yu, Y., Krishnamachari, B., Prasanna, V., 2004. Issues in designing middleware for wireless sensor networks. Network, IEEE 18 (1), 15 – 21.

[38] Yu, Y., Rittle, L. J., Bhandari, V., LeBrun, J. B., 2006. Supporting concurrent applications in wireless sensor networks. In: the 4th international conference on Embedded networked sensor systems. SenSys '06. ACM, Boulder, Colorado, USA, pp. 139–152.