Tuning In-Sensor Data Filtering to Reduce Energy Consumption in Wireless Sensor Networks

I. Kadayif
Dept. of Computer Engineering
Canakkale Onsekiz Mart University, TR
kadayif@comu.edu.tr

M. Kandemir
Dept. of Computer Science & Engineering
The Pennsylvania State University, USA
kandemir@cse.psu.edu

Abstract

In recent years, research on wireless sensor networks has been undergoing a revolution, promising to have significant impact on a broad range of applications from military to health care to food safety. An important problem in many sensor network applications is to decide the amount of computation (or filtering) that needs to be done in the sensor nodes before the data are shifted to a central base station. Right amount of data filtering in the sensor nodes can lead to large savings in network-wide energy consumption. The main goal of this paper is to develop an automated strategy for data filtering in wireless sensor nodes. Assuming that one needs to reduce the overall energy consumption (as opposed to reducing just computation energy or communication energy), the proposed strategy attempts to strike a balance between computation energy consumption and communication energy consumption. Our experimental results clearly indicate that the proposed data filtering strategy generates substantial energy savings in practice.

1. Introduction

Technological advances in low power wireless communication protocols and sensor devices made construction of large-scale sensor networks possible [9, 5, 11, 17, 12, 15, 20, 4, 8]. Sensor nodes in these networks play two major roles. First, they read data from their environment and communicate it to other nodes and/or to a base station. Second, they take part in high-level decision-making process that requires participation of multiple sensors. An important question that needs to be addressed is how much of the data they collect should be communicated to the other sensors in the network and/or to a central base station, and how much of it should be filtered out in the sensor nodes themselves. While it is possible in theory to communicate the entire data they collect (without operating on it), this will not be very efficient in practice because of at least two reasons. First, such an approach does not make use of the existing processing capability in sensor nodes; and second, it may lead to excessive energy consumption during communication.

Since energy efficiency is crucial to achieving satisfactory network lifetime, it is of utmost importance that both computation and communication should be performed in an energy-efficient manner. In particular, prior research [17] indicates that, in a wireless sensor network environment, communication energy can be orders of magnitude higher than computation energy. Therefore, in most cases, communicating all the data read (sensed) from the environment as it is (that is, without any filtering) may not be acceptable. Instead, data should be filtered in the sensor node before it is being passed to the other nodes in the wireless network or to the base station.

The main goal of this paper is to develop an *automated strategy* for data filtering in wireless sensor nodes. Assuming that one needs to reduce the overall energy consumption (as opposed to reducing just computation energy or communication energy), the proposed strategy attempts to strike a balance between computation energy consumption and communication energy consumption. In other words, the main idea in this work is to perform the right amount of data filtering in the sensor node. Our strategy is designed for wireless sensor network environments where an end-user wants to remotely monitor/control the environment. In such a situation, the data from the individual sensor nodes must be sent to a central base station, often located far from the sensor network, through which the end-user can access the data.

In addition, we want the sensor network to be easily *reprogrammable* when desired. In other words, we want to automate our data filtering strategy within an *optimizing compiler* so that it can be reused across different applications mapped to the wireless network. To do this, we make use of available optimizing compiler technology [13]. More specifically, the compiler analyzes the application code to be mapped to the sensor network and decides, for each sensor node, the type and amount of data filtering that needs to be accommodated.

In order to demonstrate the viability of our approach, we implemented the necessary compiler algorithms within an experimental compiler [1], and performed extensive experiments with several applications suitable for sensor environments using a custom energy simulator. Our experimental results clearly show that the proposed data filtering strategy generates substantial energy savings in practice. This, in turn, helps prolong lifetime of a wireless sensor network. Moreover, our results also reveal that working with a less aggressive or a more aggressive filtering strategy (than the one determined by our compiler-based approach) generates worse results across all applications tested.

The rest of this paper is organized as follows. Section 2 presents our data filtering strategy in detail. Section 3 introduces our experimental platform (the simulation setup and the benchmarks) and discusses experimental results. Section 4 concludes the paper with a summary of our major findings.

1530-1591/04 $20.00 (c) 2004 IEEE

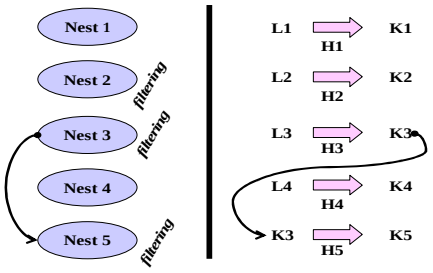

Figure 1. An example scenario (Li denotes input data and Ki denotes resulting data).

2. Data filtering

2.1. Problem description

The problem that we attack in this paper can be described as follows. Let us assume that, at processing step *k*, sensor node S_i reads (senses) $N_{i,k}$ bytes of data from the environment, and wants to perform some computation on it and subsequently pass the result to the central base station. Assume further that the node (after operating on the data) filters the data and reduces it to $N'_{i,k}$ bytes and spends an $Ecomp_{i,k}$ amount of computation energy in doing so. After that, it passes the data to the base station by expending an $Ecomm_{i,k}$ amount of communication energy. Our objective then is to determine the right amount of data filtering such that the total energy consumption (considering all sensors and all processing steps),

$$Eall = \Sigma_i \Sigma_k Ecomp_{i,k} + Ecomm_{i,k},$$

is minimized. It should be noted that there are several factors that make this problem very difficult to solve in practice. First, depending on the application mapping employed, different sensor nodes in the network can execute different code fragments. In the worst case, each sensor may need to employ a different filtering strategy (customized to its portion of the code) to obtain the best energy consumption behavior network-wide. Second, even one considers a single sensor node, it may be difficult to establish a relationship between the amount of data filtered and the overall energy consumed. This is because filtering a given amount of data may require different amount of computation energy depending on where it takes place during the course of execution. For example, filtering the first 10 bytes of a dataset may be fast (and energy efficient), whereas the second 10 bytes can be very costly as far as filtering is concerned (e.g., due to variances in the control flow of the application in question). Third, the sensor nodes might be different from each other in terms of their physical properties. That is, some of them may handle computation very efficiently (conserving computation energy), whereas some others may have low power mechanisms to reduce communication energy. Combined, these three factors can make it very hard from the perspective of the application programmer to decide an ideal data filtering strategy.

Our compiler-based approach to this problem works as follows. First, the compiler analyzes the code mapped to each sensor node, and for each statement in the code, identifies the amount of maximum beneficial filtering possible. Note that, if a statement does not perform any data filtering, there is no point in executing it in the sensor node since doing so will not reduce the data to be communicated (and will eat up battery power unnecessarily). Instead, such a statement is a perfect candidate to be executed on the central base station, where there is no energy constraint. On the other hand, if a statement exhibits some data filtering, then the compiler should determine whether to execute it on the sensor node or on the base station side. In our approach, the compiler makes this decision by analyzing how the result of this statement is later used in the application (i.e., how it will be consumed). The next subsection gives the details of our compiler-based approach.

2.2. Compiler analysis for data filtering and our algorithm

We define our problem as a *computation-mapping problem*; that is, for each sensor node in the wireless network, given a fixed amount computation to be performed (whose result needs to be communicated to the base station), we *divide* the computation between the sensor node and the base station. In other words, we decide what parts of the computation should be mapped to the base station and what parts of it to the sensor node. Informally, in order for a computation to be mapped to the sensor node, it should contain some amount of data filtering (or data reduction) type of computation. A code fragment can be considered as *filtering* if the size of the output data generated by it is much lower than the size of the input data. As an example, consider the following code fragment that consists of two separate loop nests (written in a C-like language):

```
for(i=2;i<n;i++)
    for(j=2;j<n;j++)
      L[i][j] = α•M[i-1][j+1] + β•M[i+1][j-1];
for(i=2;i<n;i++)
    for(j=2;j<n;j++)
      for(k=1;k≤n;k++)
        K[i][j] = θ•N[i-1][j+1][k] + σ•N[i+1][j-1][k];
```

In this code fragment, M, L, N, and K are arrays and α, β, θ, and σ are scalar variables. Assuming that arrays M and N represent the data sensed (read) from the environment, the first loop nest above does not have any filtering type of activity since

Table 1. Computations and communications due to three different execution strategies for the scenario in Figure 1.

Naïve Mapping			Step 1			Step 1 + Step 2		
Computation		Communication	Computation		Communication	Computation		Communication
Base	Sensor		Base	Sensor		Base	Sensor	
H1	-	L1	H1		L1	H1		L1
H2	-	L2		H2	K2		H2	K2
H3	-	L3		H3	K3		H3	-
H4	-	L4	H4		L4	H4		L4
H5	-	-		H5	K5		H5	K5

it takes a two-dimensional array (M) as input, and generates a two-dimensional array (L) as output. In contrast, the second loop nest exhibits filtering; that is, it takes a three-dimensional array (N) and generates a two-dimensional array (K). Therefore, it is a better candidate to be executed in the sensor node (as opposed to the central base station). This is because if we do not execute it in the sensor node, we need to execute it in the base station. But, to do this, we need to transfer an entire array N (a total of n^3 elements) from the sensor node to the base station, resulting in tremendous network traffic, thereby consuming potentially intolerable amount of communication energy. Instead, if we can execute the nest in the sensor node, we need to transfer only the resulting array (K) to the base station (a total of n^2 array elements). In this way, the sensor node filters data before it is shifted to the base station. As a simple rule, those computations that result in filtering and data reduction, while requiring small amount of computation (or energy) per unit of data, are most suitable for being mapped to the sensor node; because, they would exploit relatively less powerful processors in the sensor nodes, while tremendously reducing the data communication volume between the wireless sensor network and the base station.

Our compiler algorithm for detecting the computations to be performed in the sensor nodes (and those to be performed in the base station) operates on loop nest granularity, and consists of two steps. In the first step, the compiler considers each nest in turn and marks the nests that should be executed in the sensor node since they contain some sort of data filtering. In the second step, we try to reduce the data that needs to be communicated to the base station further by checking whether the resulting data sets (from the loop nests selected in the first step) are actually needed in the base station. If not, these data sets are not transmitted to the base station, thereby further reducing communication volume. Therefore, in the second step of our algorithm, the compiler performs a data-flow analysis at the loop nest granularity. In the following paragraphs, we discuss these two steps in more detail, and make a case that in order to obtain the best energy consumption behavior, both of these steps are necessary.

To explain the first step of our algorithm, let us consider the following generic loop nest and the assignment statement shown, assuming that this loop nest is to be executed by a sensor node:

```
for (i1=L1; i1≤U1; i1++)
    for (i2=L2; i2≤U2; i2++)
            ……..
        for (is=Ls; is≤Us; is++)
            K[f1][f2]...[fτ] = ... L[g1][g2]...[gχ] ...
```

We assume that f_1, f_2, ..., f_τ, g_1, g_2, ..., g_χ are the subscript expressions (array index functions), and each f_i ($1 \le i \le \tau$) and g_j ($1 \le j \le \chi$) is an affine function of loop indices i_1, i_2, ..., i_s and loop-independent variables. We also assume that arrays K (τ-dimensional) and L (χ-dimensional) are declared as *type* $K[N_1][N_2]...[N_\tau]$, $L[M_1][M_2]...[M_\chi]$, where *type* can be any (data) type of interest such as integer or float. The compiler decides that the assignment statement in this loop nest exhibits a filtering if and only if:

$$c \bullet G\{K[f_1][f_2]...[f_\tau]\} < G\{L[g_1][g_2]...[g_\chi]\}$$

Here, c is a constant to make sure that the difference between the two sides is large enough so that executing the computation (the statement) in the sensor node will be really beneficial. Also, G{E} gives the number of distinct array elements accessed by array reference E (which is either $K[f_1][f_2]...[f_\tau]$ or $L[g_1][g_2]...[g_\chi]$ for the assignment statement above). In other words, this constraint checks whether the size of the output data generated (K) is sufficiently smaller than the input data (L). One problem with this constraint is that determining the number of elements accessed by an affine expression is in general a costly operation [16][14]. In checking this constraint, we represent the set to be counted using the Presburger formulas and use the technique proposed in [6].

It should be observed that in most of the cases with array-dominated applications encountered in practice, it is possible to check the above condition statically (at compile-time) using a polyhedral tool such as the Omega Library [10]. In cases where this is not possible, we have two options. First, we can collect profile data (e.g., by instrumenting the code) to see whether the condition above holds for typical input data. Second, we can insert a conditional statement (if-statement) into the code that chooses between performing computation in the base station and performing it in the sensor node, depending on the outcome of the condition. It should be noted that selecting a suitable c value is critical. This is because a small c value can force aggressive computation mapping to the sensor node. This, in turn, can result in some unsuitable computation being mapped to the embedded processor in the sensor node (which is typically much less powerful than the one in the central base station), thereby reducing overall performance (i.e., increasing application execution time) and impacting energy consumption behavior negatively. On the other hand, a very large c value can be overly conservative and can result in a code mapping that does not make use of the processor in the sensor node at all. If, in a given loop nest, there exists at least one statement that exhibits filtering, our current implementation marks the entire loop to be executed in the sensor node. As an example, let us consider the example code fragment given at the beginning of Section 2.2 (which consists of two separate nests). Assuming that all array dimensions are of the same size (extent), using the approach summarized in the previous paragraph, one can easily see that only the second nest is identified to be executed in the sensor node (assuming c = 1). While it might be possible to have more elaborate strategies for identifying the loop nests that need to be executed in the sensor node (instead of the base station), as the experimental results presented later show, our approach performs very well in practice.

We now discuss the second step of our compiler-based approach. It is to be noted that mapping large code fragments to the sensor node is preferred to mapping smaller ones as the former implies less communication between the base station and the sensor node. In the second step of our algorithm, in order to minimize communication between the sensor nodes and the base station further, we use data-flow analysis. Data-flow analysis is a program analysis technique that is mainly used to collect information about how data flows through program statements/blocks [13]. To explain our approach, let us assume that in the first step we have decided that two loop nests, named ξ_1 and ξ_2, have been decided to be executed in the sensor node. If the output of ξ_1 is used only by ξ_2 (as input), then the mentioned output does not need to be communicated to the base station (following the execution of ξ_1). Instead, we can communicate the output of ξ_2 after the execution of both ξ_1 and ξ_2. In this way, we

Table 2. Benchmark codes used in the experiments. The last column shows the energy consumption (computation plus communication) when no data filtering is performed in the sensor nodes.

Benchmark	Input	Arrays	Energy
3-step-log	295.08KB	3	230.4mJ
adi	271.09KB	6	408.6mJ
full-search	98.77KB	3	316.6mJ
hier	97.77KB	7	228.5mJ
mxm	464.84KB	3	838.7mJ
parallel-hier	295.08KB	3	220.6mJ
tomcatv	174.22KB	9	910.0mJ
jacobi	312.00KB	4	661.9mJ
red-black SOR	156.00KB	4	892.2mJ

can reduce communication volume beyond what could be achieved by using only step 1.

To illustrate the importance of these two steps of our optimization algorithm, we now consider the example scenario depicted in Figure 1. In this figure, we have five separate loop nests (shown on the left side of the figure) and the function performed by each nest is shown on the right hand side. The arrow in the figure indicates that the output of the third loop nest is used as input to the fifth loop nest. As can be seen from the first three columns of Table 1, in a naïve execution, all data sensed from the environment are communicated to the central base station, which in turn executes all functions. The middle part of Table 2 shows the computations and communications when only the first step of our approach is used, under the assumption that the second, third, and fifth nests contain filtering. In comparison, the last portion of the table shows the situation if both of the steps of our algorithm are applied. Comparing this with the middle part of the same table, one can see the advantage of employing both the steps. If we use only step 1, at the end of the third nest, the sensor node communicates K3 to the base station. On the other hand, if we use both the steps, the sensor node does not perform this communication. Instead, since K3 itself is not required by the base station, the sensor node keeps it and uses it in the fifth nest. In other words, using both the steps of our approach reduces communication beyond what could be possible had we used only the first step of the algorithm.

3. Experimental setup and evaluation

3.1. Benchmark codes

We use a set of array-intensive benchmark programs in our experiments. The salient characteristics of the benchmark codes in our experimental suite are summarized in Table 2. The first, third, fourth, and sixth benchmarks are motion estimation codes. The second one is an alternate direction integral code. mxm and tomcatv are an integer matrix multiplication code and a mesh generation code, respectively. The last two codes, Jacobi relaxation and red-black successive over-relaxation (SOR), contain stencil-like computations and reductions. Each array element is assumed to be 4-bit wide. The third column in this table gives the number of arrays (including the temporary ones) in each benchmark code. The last column shows the energy consumption (computation plus communication) when *no* data filtering is performed in the sensor nodes. The energy consumption values reported in the rest of this section are normalized with respect to this last column of Table 2 (our energy modeling will be discussed shortly).

3.2. Application parallelization over sensor nodes

In this work, we focus on applications where arrays (which represent the sensed data) are processed by multiple sensor nodes in parallel. In such applications, given an array, typically, each processor is responsible from processing a portion of it. Note that this operation style matches directly to an environment where each sensor node is collecting some data from the portion of an area covered by it, and processing the collected data [17].

Our parallelization strategy works on a single loop nest at a time; that is, each loop nest in the application code being optimized is parallelized independently of the other loop nests. In order to parallelize a given loop nest over sensor nodes, we need to perform two main tasks: (1) decomposing arrays of signals across the memories of the sensor nodes, and (2) distributing the loop iterations across the sensor nodes. Note that array decomposition and loop iteration distribution, together, achieve some sort of *data parallelism* across sensor nodes. In this work, we adopt an array decomposition oriented parallelization strategy based on the owner-computes rule used by state-of-the-art optimizing compilers [13]. In this strategy, an array element is updated (written) by only the node that owns it.

3.3. Energy modeling

In this study, we focus only on dynamic energy. Dynamic energy consumption is due to switching of hardware components is dependent strongly on how different components of a sensor node are exercised by a given application [2]. We separate the overall energy consumption into two components: *computation energy* and *communication energy*. Computation energy is the energy consumed in processor core (datapath), instruction memory, data memory, and clock network. In this work, we focus on a simple, single-issue, five-stage pipelined embedded (and low-power) processor core that is suitable to be employed in a sensor node. This core has instruction fetch (IF), instruction decode/operand fetch (ID), execution/address calculation (EXE), memory access (MEM), and write-back (WB) stages. We use SimplePower [18], a publicly-available cycle-accurate energy simulator, to model the energy consumption in this processor core. The modeling approach used in SimplePower has been validated to be accurate (with an average error rate of 8.98%)

Table 3. The default parameters used in our base configuration. Some of these parameters are later modified to conduct a sensitivity analysis.

Parameter	Value
|P|	120
Instruction Memory	8 KB
Data Memory	16 KB
P_{tx}	80 mW
P_{rx}	200 mW
T_{st}	450 msec
P_{out}	1 mW
l	250 bits
b	1 Mb/sec

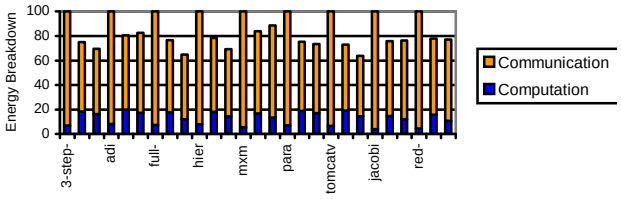

Figure 2. Energy breakdown between communication and computation. For each benchmark, the first bar corresponds to the default case (without any filtering) and the second one corresponds to the optimized case when both steps of our algorithm are used.

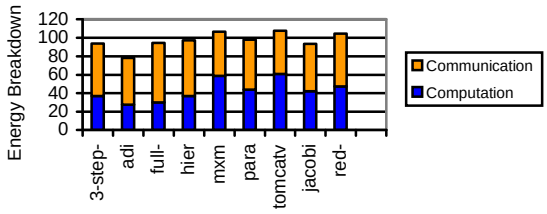

Figure 3. Energy breakdown between communication and computation. For each benchmark, the first bar corresponds to the default case (without any filtering) and the second one corresponds to the optimized case when only the first step of our algorithm is used.

using actual current measurements of a commercial architecture [3].

We assume that each sensor node has an instruction memory and a data memory (both are SRAM). The energy consumed in these memories is dependent primarily on the number of accesses and memory configuration (e.g., capacity, the number of read/write ports, and whether it is banked or not). We modified the Shade simulation environment [7] to capture the number of references to instruction and data memories, and used the CACTI tool [19] to calculate the per access energy cost. The data collected from Shade and CACTI are then combined to compute the overall energy consumption due to memory accesses. The clock generation circuit (PLL), the clock distribution buffers and wires, and the clock-load on the clock network presented by the clocked components are the main energy consumers for the clock network in our sensor node. We enhanced SimplePower to estimate the clock network energy consumption in each cycle by determining which parts of clock network are active, and using the corresponding energy models for active components.

As our communication energy component, we consider the energy expended for sending/receiving data. The radio in the sensor nodes is capable of both sending data and, at the same time, sensing incoming data. We assume that if the radio is not sending any data, it does not spend any energy (omitting the energy expended due to sensing). After packing data, the processor sends the data to the other processors via radio. The radio needs a specific startup time to start sending/receiving a message. In this study, we used the radio energy model presented by [17] to account for communication energy. In this model, the power equation of the radio is expressed as:

$$P_{radio} = N_{tx}[P_{tx}(T_{on\text{-}tx}+T_{st})+P_{out}T_{on\text{-}tx}] + N_{rx}[P_{rx}(T_{on\text{-}rx}+T_{st})],$$

where $N_{tx/rx}$ is the average number of times per second that the transmitter/receiver is used; $P_{tx/rx}$ is the power consumption of transmitter/receiver; P_{out} is the output transmit power that drives the antenna; $T_{on\text{-}tx/on\text{-}rx}$ is the time interval required to send/receive data; and T_{st} is the startup time of the transceiver. Also, note that $T_{on\text{-}tx/on\text{-}rx}=l/b$, where l is packet size (message length in bits), and b is the data transmit/receive rate in bits per second.

Our base configuration uses the values given in Table 3. The power values in this table are similar to those used in [17, 12]. In all our experiments we maintain that P_{rx} is equal to $2.5P_{tx}$ (since the receiver has more circuitry than transmitter). That is, whenever P_{tx} is modified, P_{rx} is also modified accordingly to satisfy $P_{rx}=2.5P_{tx}$. In Table 3, |P| denotes the total number of sensor nodes that participate in the execution of the application.

3.4. Results

We start our discussion of experimental results by giving energy breakdown for each application in our benchmark suite. In Figure 2, for each benchmark, the first bar corresponds to the energy breakdown between computation and communication when no filtering is performed. The second bar, on the other hand, shows the breakdown when our filtering based strategy is employed. We see from these results that, on the average, when no filtering is used communication energy constitutes 93.62% of the overall energy budget. When we use filtering, however, the contribution drops to 59.68% (at the expense of some increase in computation energy). In other words, filtering is very effective in practice, giving 31.44% saving in overall energy consumption. In order to illustrate the contribution of the second step of our algorithm, in Figure 2, the last bar for each application gives the energy when only the first step of our approach is used. We observe that the average energy savings is around 26.04%, much lower than the case when we used both the steps of the algorithm. These results emphasize the importance of using both the steps of our algorithm. In the remainder of this section, we always use both the steps of our algorithm.

To see what would happen if we perform all computations in the sensor nodes, we computed another set of experiments. The results are given in Figure 3 as fraction of the values represented by the first bar in Figure 2. One can clearly see from these results aggressive in-sensor computation is not a good idea. Specifically, it generates worse results that the base case in three of our benchmarks. Even for the remaining benchmarks, its energy behavior is very close to that of the base case. Therefore, careful tuning the aggressiveness of data filtering is critical.

3.5. Sensitivity analysis

In this section, we study the behavior of our algorithm when several experimental parameters are modified. The parameters modified here are T_{st} (startup time), P_{tx} (transmitter power), and b (data rate). Note that some of these variations also help us (indirectly) evaluate the impact of different communication protocols. For example, increasing the number of error-control bits added by a protocol can be thought of as increasing P_{tx}. In this subsection, using the base configuration, we also experimented with different network sizes. In most of the results presented in this section, we focus on two applications only (tomcatv and red-black-SOR) due to lack of space. However, our observations hold for other applications as well.

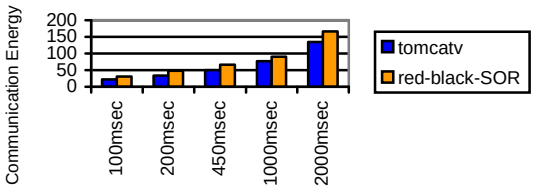

Figure 4. Communication energy consumption due to our approach with different startup latencies.

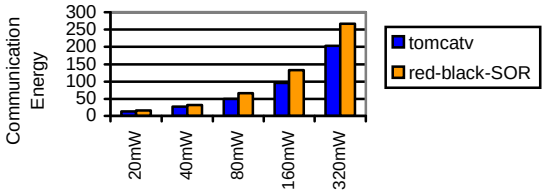

Figure 5. Communication energy consumption due to our approach with different transceiver power values.

First, in Figure 4, we show the influence of startup latency on communication energy consumption. We observe that startup latency has a great impact on energy behavior of these two applications. In our second experiment, we measure the effect of transceiver power on communication energy. As can be seen clearly from Figure 5, the communication energy increases almost linearly with the transceiver power (P_{tx}). This is because the transceiver power is very large as compared to the transmit power of the antenna (P_{out}), and is the main factor that determines the overall trend in communication energy. The effect of data transmit/receive rate on energy behavior of these two benchmarks is given in Figure 6. Since an increase in transmit/receive rate reduces transmit/receive time, the radio will need to be active for a smaller period of time to send/receive the message, and consequently, communication energy is reduced. Also, note that, for very high rates, the startup time balances or dominates the transmit/receive time, so energy overhead due to startup time plays a very critical role in total communication time. As a consequence, the number of messages (rather than total size of messages) determines the communication energy.

4. Concluding remarks

Wireless, microsensor networks have potential for enabling a myriad of applications for sensing and controlling the physical world. Recent years have witnessed several efforts at the architectural and circuit level for designing and implementing microsensor-based networks. While architectural/circuit-level techniques are extremely critical for the success of these networks, software optimizations are also expected to become instrumental in extracting the maximum benefits from the performance and energy behavior angles. The main goal of this paper is to develop an automated strategy for data filtering in wireless sensor nodes. Data filtering performs some select computation at the sensor nodes and shifts to the central base station only the results. Assuming that one needs to reduce the overall energy consumption (as opposed to reducing just computation energy or communication energy), the proposed strategy attempts to strike a balance between computation energy

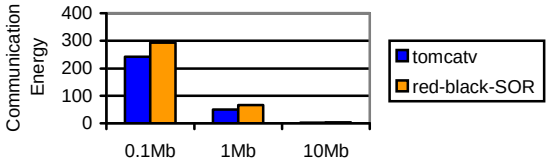

Figure 6. Communication energy consumption due to our approach with different transmit/receive rates.

consumption and communication energy consumption. Our experimental results clearly indicate that the proposed data filtering strategy generates substantial energy savings in practice.

5. References

[1] S. P. Amarasinghe et al. "Multiprocessors from a software perspective." IEEE Micro, June 1996, pages 52-61.

[2] A. Chandrakasan, W. J. Bowhill, and F. Fox. "Design of High-Performance Microprocessor Circuits." IEEE Press, 2001.

[3] R. Y. Chen, R. M. Owens, and M. J. Irwin. "Validation of an architectural level power analysis technique. " In Proc. the 35th Design Automation Conference, June 1998.

[4] B. Chen, K. Jamieson, R. Morris, and H. Balakrishnan. "Span: an energy-efficient coordination algorithm for topology maintenance in ad hoc wireless networks." In Proc. the ACM MOBICOM Conference, Rome, Italy, July 2001.

[5] S.-H. Cho and A. Chandrakasan. "Energy-efficient protocols for low duty cycle wireless microsensor networks." In Proc. ICASSP'2001, May 2001.

[6] P. Clauss. "Counting Solutions to Linear and Nonlinear Constraints through Ehrhart Polynomials: Applications to Analyze and Transform Scientific Programs". In Proc. the 10th International Conference on Supercomputing, pp. 278--285, May 25--28, 1996, PA.

[7] B. Cmelik and D. Keppel. "Shade: a fast instruction-set simulator for execution profiling." In Proc. the 1994 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems, May 1994, pp. 128—137.

[8] J. Elson and K. Römer. "Wireless sensor networks: a new regime for time synchronization." In Proc. the First Workshop on Hot Topics In Networks (HotNets-I), Princeton, New Jersey. October 28-29 2002.

[9] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar "Next century challenges: scalable coordination in sensor networks." In Proc. the Fifth Annual International Conference on Mobile Computing and Networks, August 1999, Seattle, Washington.

[10] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. "The Omega library interface guide." Technical Report CS-TR-3445, CS Dept., University of Maryland, College Park, MD, March 1995.

[11] A. Lim. "Distributed services for information dissemination in self-organizing sensor networks." Special Issue on Distributed Sensor Networks for Real-Time Systems with Adaptive Reconfiguration, Journal of Franklin Institute, Elsevier Science Publisher, Vol. 338, 2001, pp. 707--727.

[12] R. Min, M. Bhardwaj, S.-H. Cho, A. Sinha, E. Shih, A. Wang, and A. Chandrakasan. "Low-power wireless sensor networks." In Proc. VLSI Design'01, January 2001.

[13] S. S. Muchnick. "Advanced Compiler Design Implementation." Morgan Kaufmann Publishers, San Francisco, California, 1997.

[14] W. Pugh "Counting Solutions to Presburger Formulas: How and Why", In Proc. the ACM Conference on Programming Language Design and Implementation 1994, Orlando, Florida.

[15] J. Rabaey et al. "PicoRadio supports ad-hoc ultra-low power wireless networking." IEEE Computer Magazine, July 2000, pp. 42--48.

[16] A. Schrijver. "Theory of Linear and Integer Programming", John Wiley and Sons, Inc., New York, NY, 1986.

[17] E. Shih, S. H. Choo, N. Ickes, R. Min, A. Sinha, A. Wang, A. Chandrakasan. "Physical layer driven protocol and algorithm design for energy-efficient wireless sensor network." In Proc. the 7th Annual International Conference on Mobile Computing and Networking, July 16-22, 2001, Rome Italy.

[18] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. Y. Kim, and W. Ye. "Energy-driven integrated hardware-software optimizations using SimplePower." In Proc. the International Symposium on Computer Architecture, June 2000.

[19] S. Wilton and N. P. Jouppi. "CACTI: an enhanced cycle access and cycle time model." IEEE Journal of Solid-State Circuits, pp. 677--687, 1996.

[20] W. Ye, J. Heidemann, and D. Estrin. "An energy-efficient MAC protocol for wireless sensor networks." In Proc. the 21st International Annual Joint Conference of the IEEE Computer and Communications Societies, New York, NY, USA, June, 2002.