An Energy-Oriented Evaluation of Communication Optimizations for Microsensor Networks

I. Kadayif[1], M. Kandemir[1], A. Choudhary[2], and M. Karakoy[3]

[1] Pennsylvania State University, University Park, PA 16802, USA
[2] Northwestern University, Evanston, IL 60208, USA
[3] Imperial College, London SW7 2AZ, UK

Abstract. Wireless, microsensor networks have potential for enabling a myriad of applications for sensing and controlling the physical world. While architectural/circuit-level techniques are 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. In this paper, focusing on a sensor network where the sensors form a two-dimensional mesh, we experimentally evaluate a set of compiler-directed communication optimizations from the energy perspective. Our experimental results obtained using a set of array-intensive benchmarks show significant reductions in communication energy spent during execution.

1 Introduction and Motivation

A networked system of inexpensive and plentiful microsensors, with multiple sensor types, low-power embedded processors, and wireless communication and positioning ability offers a promising solution for many military and civil applications. As noted in [7], technological progresses in integrated, low-power CMOS communication devices and sensors make a rich design space of networked sensors viable. Recent years have witnessed several efforts at the architectural and circuit level for designing and implementing microsensor based networks [2]. While architectural/circuit-level techniques are 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.

Optimizing energy consumption of a wireless microsensor network is important not only because it is not possible (in some environments) to re-charge batteries on nodes, but also because software running on sensor nodes may consume a significant amount of energy. In broad terms, we can divide the energy expended during operation into two parts: computation energy and communication energy. To minimize the overall energy consumption, we need to minimize the energy spent in both computation and communication. While power-efficient customized wireless protocols [5] are part of the big picture as far as minimizing

H. Kosch, L. Böszörményi, H. Hellwagner (Eds.): Euro-Par 2003, LNCS 2790, pp. 279–286, 2003.

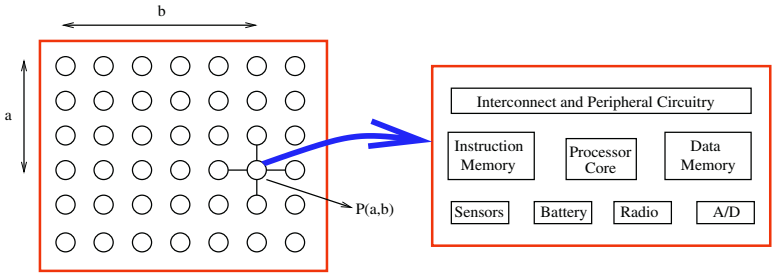

Fig. 1. Assumed sensor network architecture and blocks in a sensor node.

communication energy is concerned, we can also employ application level and compiler level optimizations.

In this paper, we focus on a wireless microsensor network environment that processes array-intensive codes. Array-intensive codes are very common in many image and signal processing applications [3]. Focusing on a sensor network where the nodes form a two-dimensional mesh, we present a set of source code level communication optimization techniques and evaluate them from the energy perspective.

2 Architecture and Language Support

The left part of Figure 1 shows the sensor network architecture assumed in this work. Basically, we assume that the sensor nodes are distributed over a two-dimensional space (area), and the distance between neighboring sensors are close to uniform throughout the network. In our study, each sensor node is assumed to have the capability of communicating with its four neighbors. Each node in this network can be identified using coordinates `a` (in the vertical dimension) and `b` (in the horizontal dimension), and can be denoted using `P(a,b)`. Consequently, its four neighbors can be identified as `P(a-1,b)`, `P(a+1,b)`, `P(a,b-1)`, and `P(a,b+1)`. Note that this nearest-neighbor communication style matches very well with many real-world uses of microsensors, as the neighboring nodes are typically expected to share data to carry out a given task. It should also be noted, however, that, although not discussed here, our software framework is able to handle general node-to-node or broadcast types of communications as well.

The right part of Figure 1 illustrates the major components of a given sensor node. Each node in our network contains sensor(s), A/D converter, battery, processor core, instruction and data memories, radio, and peripheral circuitry. Since sensor networks can be deployed in time-critical applications, we do not consider a cache architecture. Instead, each node is equipped with fast (SRAM) instruction and data memories. The software support for such an architecture is critical. Our compiler takes an input code written in C and parallelizes it across the nodes in the network. After parallelization, each sensor node executes the

same code (parameterized using parameters `a` and `b`) but works on a different portion of the dataset (e.g., a rectilinear segment of a multidimensional array of signals). In other words, the sensor nodes effectively exploit data parallelism. Our compiler parallelizes each loop nest in the code using the data decomposition supplied by the programmer. For a given array, a data decomposition specifies how the elements of the array are decomposed (distributed) across the memories of sensor nodes.

In order to express communication at the source language level, we assume that two communication primitives are available. The first primitive is of the form:

```
send DATA to P(c,d).
```

When executed by a sensor node `P(a,b)`, this primitive sends data (denoted by `DATA`) to the sensor node `P(c,d)`. The data communicated can be a single array element, an entire array, or (in many cases) an array region. We assume that this data is received by the node `P(c,d)` when it executes our other primitive:

```
receive DATA from P(a,b).
```

In order for a communication to occur, each send primitive should be matched with a corresponding receive primitive. All communication protocol-related activities are assumed to be captured within these primitives. If the size of the message indicated in these calls is larger than the packet size, the message is divided into several packets. This activity occurs within the `send` and `receive` routines.

3 Communication Optimizations

Our focus is on array-based applications that can benefit from parallel execution in a microsensor-based environment. In such applications, given an array of signals, typically, each processor is responsible from processing a portion of the array. 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. In this paper, we use the communication optimizations summarized in Table 1. Details of these optimizations can be found elsewhere [10,8]. In contrast, in a naive communication strategy, each data item is communicated only when it is needed and no two communication messages are combined to improve bandwidth.

4 Experimental Setup

4.1 Benchmark Codes

We use a set of array-intensive benchmark programs in our experiments. The salient characteristics of the codes in our experimental suite are summarized in Table 2. Each array element is assumed to be 4-bit wide. All code modifications have been performed using the SUIF compiler [1].

Table 1. Communication optimizations evaluated in this study.

Optimization	Brief Explanation
Message Vectorization	Hoisting communication that occurs due to a single reference to outer loop position, and combining them.
Message Coalescing	Combining communications that occur due to the different references to the same array
Message Aggregation	Combining communications that occur due to references to the different arrays
Inter-Nest Optimization	Eliminating communication required in a nest if it has already been performed in a previous nest
Computation/Communication Overlap	Eliminating the time spent in communication by doing useful computation while waiting for data to arrive

Table 2. Benchmarks used in the experiments. The second, and third columns give, respectively, the total input size, and the number of arrays in the code.

Benchmark	Input Size	Number of Arrays	Brief Description
`3-step-log`	295.08 KB	3	Motion Estimation
`adi`	271.09 KB	6	Alternate Direction Integral
`full-search`	98.77 KB	3	Motion Estimation
`hier`	97.77 KB	7	Motion Estimation
`mxm`	464.84 KB	3	Matrix Multiply
`parallel-hier`	295.08 KB	3	Motion Estimation
`tomcatv`	174.22 KB	9	Mesh Generation
`jacobi`	312.00 KB	2	Stencil-like Computation
`red-black SOR`	156.00 KB	1	Stencil-like Computation

4.2 Modeling Energy Consumption

We separate the system energy into two parts: 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 processor core which is suitable for employing in a sensor node. We use SimplePower [11], a publicly-available, cycle-accurate energy simulator, to model the energy consumption in this processor core and its clock generatin unit (PLL). We assume that each 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 [6] to capture the number of references to instruction and data memories and used the CACTI tool [12] 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.

As our communication energy component, we consider the energy expended for sending and 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

Table 3. Computation and communication energy breakdown for our applications. All energy values are in microjoules.

		Communication Energy			Computation Energy				
Benchmark		**Col.**	**Row**	**Block**	**IMem.**	**DMem.**	**Clock**	**Dpath**	**Total**
3-step-log	v	6867	5922	12064	266503	58376	27411	69894	422185
	v+c	5449	5449	5449					
	v+c+a	3559	3559	3559					
adi	v	173196	173196	82886	38204	7734	8037	19541	73516
	v+c	173196	173196	82886					
	v+c+a	165636	165636	36894					
full-search	v	127615	123835	189985	343946	84058	31132	84654	543788
	v+c	110605	110605	110605					
	v+c+a	108715	108715	108715					
hier	v	191422	185752	284977	24961	5945	1902	5373	38179
	v+c	165907	165907	165907					
	v+c+a	164017	164017	164017					
mxm	v	431700	431700	286080	16609	2653	2576	5734	27572
	v+c	367440	367440	146976					
	v+c+a	367440	367440	146976					
parallel-hier	v	63807	61917	94992	168742	36775	15326	40626	261468
	v+c	55302	55302	55302					
	v+c+a	55302	55302	55302					
tomcatv	v	153780	61512	58934	18877	3788	3195	7844	33706
	v+c	153780	61512	42494					
	v+c+a	142440	57732	33623					
jacobi	v	72839	72839	33671	75264	12016	11858	27568	126706
	v+c	72839	72839	33671					
	v+c+a	72839	72839	33671					
red-black-SOR	v	76619	76619	39719	73732	11472	12288	28160	125652
	v+c	76619	76619	39719					
	v+c+a	76619	76619	39719					

the other processor via radio. The radio needs a specific startup time to start sending/receiving message. We used the radio energy model presented by [9] to account for communication energy. We assumed 160 sensor nodes participating in parallel execution of the application.

5 Results

5.1 Energy Breakdown

Table 3 gives the energy breakdown for our benchmarks for three different data decompositions and communication optimizations. Since the computation energies for different decompositions and different communication optimizations are almost the same we report only one computation energy value for each benchmark. The column decomposition (in Table 3) refers to a decomposition, whereby each sensor node owns a column-block of each array; the other decompositions correspond to cases where the arrays involved in the computation are decomposed in row-block and block-block manner across sensor nodes. Also, we consider three different communication optimizations: message vectorization (denoted `v`), message vectorization + message coalescing (denoted `v+c`), and message vectorization + message coalescing + message aggregation (denoted `v+c+a`). Unless stated otherwise, all energy numbers given are for dynamic energy only (i.e., they do not include the leakage energy consumption).

Table 4. Communication energy with naive communication. All energy values are in microjoules.

Benchmark	Column	Row	Block
3-step-log	173200144	173200144	173200144
adi	2429856	2429856	971942
full-search	360833632	360833632	360833632
hier	22719154	22719154	22719154
mxm	15819376	15819376	25311000
parallel-hier	28866690	28866690	28866690
tomcatv	2024880	809952	570206
jacobi	1036739	1036739	414695
red black SOR	1036739	1036739	414695

We can make several observations from the numbers reported in Table 3. First, we see that in some benchmarks the computation energy dominates the communication energy, whereas in some benchmarks it is the opposite. This is largely a characteristic of the access pattern exhibited by the application under a given (data decomposition, communication optimization) pair. Based on these results, we can conclude that communication energy constitutes a significant portion of the overall energy budget. The second observation is that the communication optimizations save significant amount of energy. For example,`v+c` improves the communication energy of the `hier` benchmark by 13.7% (over the only message-vectorized version) when column decomposition is used. As another example, the `v+c+a` version in `tomcatv` saves 7.2% communication energy over the `v+c` version. In fact, as can be seen from our results, in some cases, an optimization can even shift the energy bottleneck from communication to computation. For example, in the `adi` benchmark, when block decomposition is used, the communication energy of the `v+c` version is larger than the computation energy. However, when we use the `v+c+a` version, the communication energy becomes less than the computation energy. Third, we can observe that data decomposition in some benchmarks makes difference in communication energy. For example, the block decomposition performs much better than column and row decompositions in `adi`. In fact, for this benchmark code, while communication energy dominates computation energy in column and row decompositions, communication energy is approximately half of the computation energy if the message optimizations are applied in conjunction with block decomposition. It should be stressed that working with the naive communication strategy described earlier (that is, not using any communication optimization) may result in intolerable communication energies. To illustrate this, Table 4 shows the communication energy consumption when naive communication is employed. Comparing these values with those in Table 3 emphasizes the difference between optimizing and not optimizing communication energy.

5.2 Impact of Overlapping Communication with Computation

The results presented so far were obtained under the assumption that a processor sits idle during communication. In fact, we assumed that the processor, when

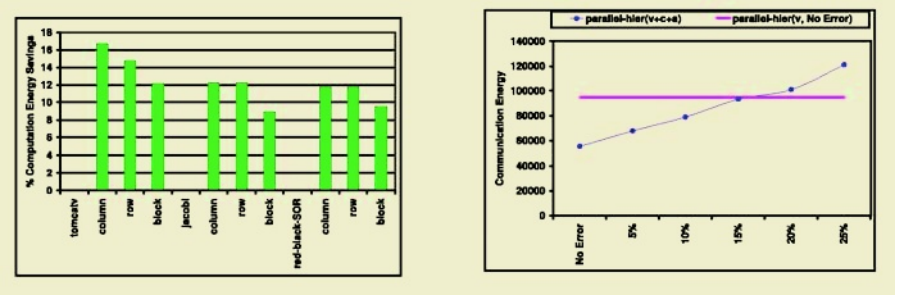

Fig. 2. Left: Percentage savings in computation energy due communication/computation overlapping. Right: Impact of communication error.

waiting for the data to be sent, places itself (or a tiny OS does this for it) into a low-power mode, where energy consumption is negligible. This is not very realistic as processor and memories typically consume some amount of leakage energy [4] during these waiting periods. Obviously, the longer the waiting period, the higher the leakage energy consumption. Consequently, we can reduce the leakage energy consumption by reducing the time that the processor waits idle for the communication to be completed. This can be achieved by overlapping the computation with communication. To evaluate the energy savings due to this optimization, we applied it to three benchmarks from our experimental suite, and measured the percentage computation energy savings. The results shown in the left graph in Figure 2 indicate that our approach can save around 20% of original computation energy of the `v+c+a` version. In these experiments, we have assumed that the leakage energy consumption per cycle of a component is 20% of the per access dynamic energy consumption of the same component. Since current trends indicate that leakage energy consumption will be more important in the future [4], this optimization can be expected to be more useful with upcoming process technologies.

5.3 Communication Error

Communication errors in a sensor network may cause a significant energy waste due to re-transmission. To see whether our optimizations better resist against this increase in energy consumption, we performed another set of experiments where we measured the energy consumption under a random error model. Since the results with most of the benchmarks are similar, here we focus only on `parallel-hier`. The graph given in the right part of Figure 2 shows the communication energy consumption of the `v+c+a` version of this application assuming different error rates. For comparison purposes, we also give the communication energy consumption (in microjoules) of the `v` version without error. The results shown indicate that the `v+c+a` version starts to consume more energy than the error-free `v` version only beyond a 15% error rate. In other words, an applica-

tion powered by our optimization can tolerate more errors (than the original application) under the same energy budget.

6 Conclusions and Future Work

Advances in CMOS technology and microsensors enabled construction of large, power-efficient networked sensors. The energy behavior of applications running on these networks is largely dictated by the software support employed such as operating systems and compilers. Our results presented in this paper indicate that high-level communication optimizations can be vital if one wants to keep the communication energy consumption of microsensor network under control.

References

1. S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C. W. Tseng. The SUIF compiler for scalable parallel machines. In *Proc. SIAM Conference on Parallel Processing for Scientific Computing,* Feb 1995.
2. G. Asada, M. Dong, T. S. Lin, F. Newberg, G. Pottie, and W. J. Kaiser. Wireless integrated network sensors: low power systems on a chip. In *Proc. ESSCIRC'98,* Hague, Netherlands, Sept 1998.
3. F. Catthoor et al. *Custom Memory Management Methodology – Exploration of Memory Organization for Embedded Multimedia System Design,* Kluwer Academic Publishers, June, 1998.
4. A. Chandrakasan et al. *Design of High-Performance Microprocessor Circuits.* IEEE Press, 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. 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.
7. J. Hill et al. System architecture directions for network sensors. In *Proc. ASPLOS,* 2000.
8. M. Kandemir et al. A global communication optimization technique based on data-flow analysis and linear algebra. *ACM Transactions on Programming Languages and Systems,* 21(6):1251–1297, November 1999.
9. E. Shih et al. 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.
10. C-W. Tseng. An optimizing Fortran D compiler for MIMD distributed-memory machines. *Ph.D. Thesis, Rice COMP TR93-199,* Dept. of Computer Science, Rice University, January 1993.
11. N. Vijaykrishnan et al. Energy-driven integrated hardware-software optimizations using SimplePower. In *Proc. the International Symposium on Computer Architecture,* June 2000.
12. 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.