Reducing Energy Dissipation of Wireless Sensor Processors Using Silent-Store-Filtering MoteCache

Gurhan Kucuk, Can Basaran

Department of Computer Engineering, Yeditepe University,
34755 Istanbul, Turkey
{gkucuk, cbasaran}@cse.yeditepe.edu.tr

Abstract. Wireless sensor networks (WSNs) gained increasing interests in recent years; since, they allow wide range of applications from environmental monitoring, to military and medical applications. As most of the sensor nodes (a.k.a. motes) are battery operated, they have limited lifetime, and user intervention is not feasible for most of the WSN applications. This study proposes a technique to reduce the energy dissipation of the processor component of the sensor nodes. We utilize a tiny cache-like structure called MoteCache between the CPU and the SRAM to cache the most recently used data values as well as to filter silent-store instructions which write values that exactly match the values that are already stored at the memory address that is being written. A typical WSN application may sense and work on constant data values for long durations, when the environmental conditions are not changing rapidly. This common behavior of WSN applications considerably improves our energy savings. The optimal configuration of MoteCache reduces the total node energy by 24.7% on the average across a variety of simulated sensor benchmarks. The average lifetime of the nodes is also improved by 46% on the average for processor-intensive applications. Using the proposed technique, the lifetime of the nodes that run communication-intensive applications, such as TinyDB and Surge, is also improved as much as 14%.

1 Introduction

Wireless sensor networks (WSNs) became a leading research and development area arose from the field of ad hoc networking research. Increased research activity is due to the potential for significant monitoring applications on various properties of matter such as heat, humidity, pressure, acceleration, smoke and light. This new technology combines sensing, computation, and communication into a tiny device (a.k.a. mote) which is typically provided with an embedded microprocessor and a small amount of memory. Consequently, the most difficult resource constraint to meet becomes energy consumption of these devices, since the energy is directly related to many critical parameters such as lifetime, scalability, response time and effective sampling frequency of the network.

The introduction of a small-scale operating system (TinyOS [1]) and more complex applications considerably increased the energy consumption of the processor and the memory system of these tiny devices. In [2], the authors show that the proces-

sor/memory component constitutes 35% of the total energy budget of the MICA2 platform while running *Surge*, a TinyOS monitoring application. In [3], the authors estimate similar energy values when running a TinyDB query reporting light and accelerometer readings once every minute. In [4], the researchers study the data compression algorithms for saving energy in sensor nodes, and find that the energy consumption of the processor/memory component for running the compression algorithms is so high that sending raw data without any compression instead is a much more feasible solution. Similarly, today most of the applications available in the sensor networks domain avoid extensive computations and choose to transfer raw data to server machines to increase the lifetime of the sensor nodes.

Moreover, the applications that are executed by sensor nodes have periodic behavior. Especially, monitoring applications, such as *Surge*, may sense and work on constant data for long durations. Similarly, a continuous TinyDB query may initiate sensor nodes to work on data with temporal and spatial locality for a long time. In this paper, we propose *MoteCache* architecture to exploit these types of characteristic behavior. By caching commonly used data in a small number of latches (*MoteCache*), we show that we can considerably reduce the energy dissipation of the WSN processors. In this study, we mainly focus on MICA2 platform; however, the proposed technique is platform-independent, and can be easily applied to other sensor platforms such as TelosB, Tmote Sky and EYES sensor nodes.

We begin by presenting an overview of current processor/memory architecture of MICA2 sensor nodes in Section 2. We present the details of MoteCache design in Section 3. The simulation methodology is given in Section 4 followed by the presentation and the discussion of the results in Section 5. Finally, we discuss the related work in Section 6, and conclude our study in Section 7.

2 Related Work

In the literature, there are various proposals that target energy-efficient microprocessors for wireless sensor nodes [5, 6, 7]. However, to our best knowledge, this is a unique study that proposes architecture-independent extensions to the existing commercially available, general-purpose microcontrollers that are used by sensor nodes.

In [5], the authors present SNAP/LE, an event-driven microprocessor for sensor networks. They avoid the TinyOS overhead by their asynchronous, event-driven approach. In [6], the authors seek to fully leverage the event-driven nature of applications. They study the application domain of sensor networks and they seek to replace the basic functionality of a general-purpose microcontroller with a modularized, event-driven system.

In [8], the authors reveal that significant benefits can be gained by detecting and removing silent store instructions. They propose free silent store squashing and successfully increase the processor performance by 11%, on the average. In this study, on the other hand, we choose to move to a completely different direction, and utilize a similar mechanism to reduce the energy consumption of the microcontrollers that are used in wireless sensor devices.

3 Current State of the Art

In this paper, we study the MICA2 platform manufactured by Crossbow Inc. [9]. The platform includes an AVR-RISC processor, ATmega128L. The AVR architecture has two main memory spaces, 4Kx8 bytes of SRAM data memory and 64Kx16 bytes of flash program memory space. Additionally, the Atmega128 features an EEPROM memory for data storage. All three memory spaces are linear and regular [10]. Up until now, there are only a few applications making use of EEPROM memory. Therefore, in this paper, we mainly focus on the energy reduction of SRAM storage component.

Figure 1 shows the timing diagram of memory read/write operations for the SRAM structure of the AVR. Each memory read or memory write operation takes two cycles to complete. In the first clock cycle, T1, the memory address is computed, and in the second clock cycle, T2, the actual memory access takes place.

The AVR architecture does not contain a cache structure. The reason behind this cacheless architecture becomes very clear, when we examine the timing diagram in Figure 1 in detail. The main idea to include a cache structure in any design is to increase the processor performance, and it is obvious that no cache structure may improve upon a 1-cycle-access SRAM. On the other hand, a cache structure may be beneficial from the energy/power point of view. A tiny cache structure may reduce the memory access energy by filtering most of the accesses to the main memory. In this study, we propose such cache structure, called MoteCache, for the same purpose. Moreover, we show that with a very small, additional complexity we can also use this cache structure to filter silent-store instructions which write values that exactly match the values that are already stored at the memory address that is being written.

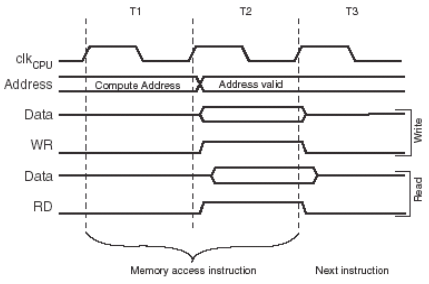

Fig. 1. On-chip Data SRAM access cycles

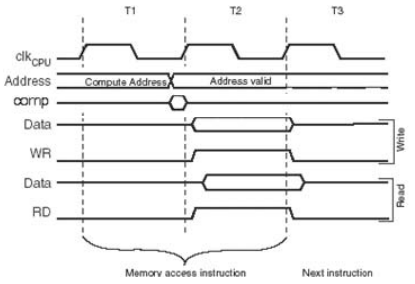

Fig. 2. Modified On-chip Data SRAM and MC access cycles

4 Memory with MoteCache

We now briefly review the design of memory with MoteCache (hereafter MC). In a memory with MC, each row consists of data buffers and an associative CAM (content-addressable memory) for holding the addresses or the tags (parts of the addresses) corresponding to the contents of these buffers. The idea is to cache data values of the N most recently accessed addresses. The design presented here is a

generalization of the designs of [11, 12]. By using independent CAMs for more than one way, we can improve the hit ratio of the MC. In this paper, we study three types of MC configuration:

1) Direct-Mapped MoteCache (DMMC): This configuration mimics the behavior of an ordinary direct-mapped cache. Since, the data SRAM of AVR microcontroller contains a single byte at each line, the direct-mapped cache is organized to contain a single byte at each of its sets. In this configuration, we do not need to have a CAM structure, since we can locate the required line using a small line decoder. Consequently, the corresponding tag comparator of that line may be activated to find if it is a cache hit (MC-hit) or a miss (MC-miss).

2) Set-Associative MoteCache (SAMC): In order to reduce the possible conflict misses of DMMC configuration; we also decided to try the set associative cache configuration. This configuration is similar to a standard set-associative cache configuration, and requires the activation of more comparators in a single access. Since, this configuration requires less number of lines compared to the DMMC configuration for the same buffer size, it enables us to utilize a smaller decoder.

3) Fully-Associative MoteCache (FAMC): This configuration is the extreme cache configuration that activates all the tag/address comparators for each memory access. In this configuration, a MC-miss results in no-match situation in all of these tag comparators. Incase of a MC-hit, only one of the comparators will have a full-match situation. After careful observation of this phenomena, for this configuration, we prefered to use Dissipate-on-Match Comparators (DMCs) proposed in [13]. Basically, DMCs dissipate power when there is a full-match situation, whereas traditional comparators dissipate power when there is a mismatch situation. The rest of the match cases dissipate negligible amount of power in both types of comparators.

4.1 Access to MoteCache

A read access in the MC structure proceeds as follows:

Step 1. MoteCache Tag Comparison: After the address computation, the address number issued by the CPU is compared associatively with the address numbers associated with the contents of the MoteCache. If an associative match is detected ("MC-hit"), the scheduled readout of the memory, for the next clock cycle, is cancelled, potentially saving energy wasted in reading out the row (again) from the memory. Again, for the DMMC configuration, there is a single comparison at the selected MC set.

When the associative match using the CAM fails, the memory access continue in the next cycle and the data is read out into a MC entry selected as a victim, based on some replacement policy. In our simulations, we studied least recently used (LRU) replacement policy, since it is feasible to implement it for small number of entries.

Step 2. Data Steering: On a tag match, the desired data is steered out from the corresponding MC entry. On a mismatch, a row from a victim way is eventually replaced with the tag and data value corresponding to the target. Of course, if we use DMMC configuration, there is no victim selection (i.e., the conflicted line automatically becomes the victim line).

Writing to a MC entry has analogous steps, followed by a step that installs the update into the tag and data part of the corresponding MC entry. Here, we only study the writeback policy, since our initial experiments showed that write-through policy is not suitable for an energy-aware design.

Figure 2 shows the modified timing diagram for the proposed architecture. Notice that, at the end of the first clock cycle, T1, there is a comparison latency that compares all the tags of the MC set with the computed address[1]. According to the outcome of the tag comparison process, either MC or SRAM is accessed.

Energy savings result in the MC as locality of reference guarantees that there is a good chance of producing a MC-hit. As our result section shows later, using just 8 entries for a 4-way SAMC, the hit rate is in excess of 82% (see Section 5) on the average.

4.2 Silent-Store Filtering

A recent study notes that many store instructions write values that actually match the values that are already stored at the memory address that is being written [8]. These store instructions are called *silent stores*, since they have no effect on machine state. Detecting and filtering silent stores may considerably improve the lifetime of WSNs. Our experiments show that WSN applications have a high potential for introducing silent store instructions, since they generally work on sensed data that may stay constant for long durations. Figure 3 shows the percentages of silent store instructions in our simulated benchmarks. Across all simulated WSN benchmarks, the average percentage of silent stores is 80.5%.

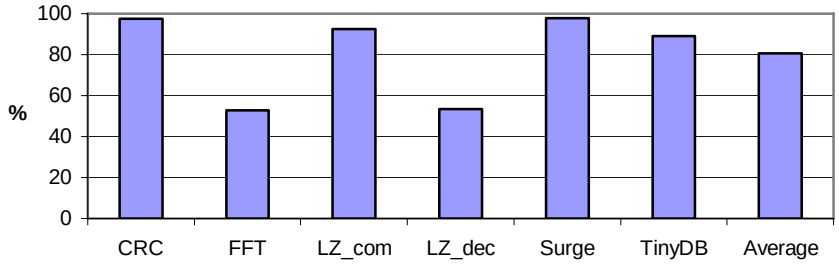

Fig. 3. Percentages of silent stores in WSN benchmarks

[1] Notice that, we assume that the address computation process may be completed a little earlier than the original design. This is possible, since the microprocessors used by the sensor nodes have very low frequency (approx. 7 MHz). Therefore, we assume large clock cycle periods provided by these processors make small tasks such as tag comparison possible in the same clock cycle. However, this is not a strict requirement for the success of our solution. Actually, WSNs may easily tolerate a small increase in processor cycle time for serializing address computation and MoteCache tag comparison in cycle T1, since performance is not their major concern, anyways.

We propose to add another functionality to the *dirty-bit* in our writeback MoteCache for detecting and filtering silent stores. Therefore, the *dirty-bit* is renamed to *dirty&noisy-bit* (or DN-bit in short.) Hereafter, we writeback data value to SRAM when the data is *dirty* (i.e. it is modified) and also *noisy* (i.e. if the new data value is different than the one that was stored before.) To set the DN-bit, we need additional 8-bit data value comparators at each line of MoteCache structure.

A write access in the silent-store filtering MC structure proceeds as one of the two ways described below:

1. MC-hit and Silent Store Detection: After an MC-hit (i.e. when the MC has a copy of the data stored in the requested address), the data value to be written is compared with the data value of the corresponding MC line. If there is no match, we set the DN-bit of that MC line to indicate that this store instruction is not silent but *noisy*. On the other hand, if there is a match situation, then the corresponding DN-bit stays clean.

2. MC-miss and MoteCache Writeback: After a MC-miss, we select a victim line. Before replacing the victim line, we check its DN-bit to see if it is set. If it is so, we continue with the standard MC writeback procedure. Otherwise, we just replace the MC line and cancel the rest of the writeback process. This prevents us from executing an energy-hungry SRAM write operation, since we already know that MC and SRAM data are in sync.

5 Simulation Methodology

We chose Avrora, The AVR Simulation and Analysis Framework, [14] as our simulation platform, since it accurately models ATmega128L processor of MICA2 platform, and also simulates sensor networks in various topologies. For network-based benchmarks (Surge and TinyDB), a three node network topology is used. Table 1 gives the details of the benchmarks used for this study:

Table 1. WSN benchmarks details

Benchmark	Size in ROM (bytes)	Size in RAM (bytes)	Execution Time (mins)
CRC	1072	1492	5
FFT	3120	1332	5
LZ77 Compression	2700	486	5
LZ77 Decompression	1228	356	5
Surge	17120	1928	50
TinyDB	65050	3036	50

- CRC benchmark is a simple CRC32 algorithm continuously ran on 448 byte data. It contains send/receive operations triggered every 2 seconds to imitate typical WSN applications.
- FFT benchmark is a discrete Fourier Transform on 256 bytes of data. This transformation is executed within an infinite loop [15]. This benchmark also contains periodic send/receive calls.
- LZ77 compression is again enclosed in an infinite loop and works on 448 bytes of data taken from the header file of an *excel* file.

- LZ77 decompression is the decompression of the compressed data obtained from the LZ77 compression. This data is 330 bytes long. LZ77 applications do not use radio communication, and, therefore, represent CPU-intensive applications.
- Surge application ran on three nodes using Avrora sensor network simulation and the average of the result values is taken.
- TinyDB application is used to execute the query “select light from sensors”. Two sampling periods were used: 128 and 1024 milliseconds.

5.1 Calculation of MC Energy Dissipation

The simulator records transitions at the level of bits for the processor/memory components. Coupled with energy per transition measurements obtained from the SPICE simulations of the actual component layouts for 0.35u CMOS technology (SRAM and MC, this allows us to estimate the overall energy dissipated within the processor. We modified the energy model of Avrora to accurately model very fine-grain, instruction-level energy dissipation of the processor, and we test our benchmarks against MC configurations implemented using Dissipate-on-Match (DMC) comparators [13]. For accurate energy savings estimations, we consider detailed energy dissipations from such additional comparators (both for tag and data values), line decoders and LRU replacement logic.

6 Results and Discussions

Figure 4 shows the average energy savings on memory read/write operations in MC without the silent store filter. Note that, when we increase the MC size in both FAMC (first five configurations) and DMMC (configurations from 4x1 to 64x1), either comparator energy (in FAMC configurations) or decoder energy (in DMMC configurations) becomes the dominant factor and reduces our energy savings. In *LZ77_comp* benchmark, 1x64 configuration dissipates considerably more energy (31% more, to be exact) than the baseline case. Again, this is due to the activation of 64 comparators at each memory access.

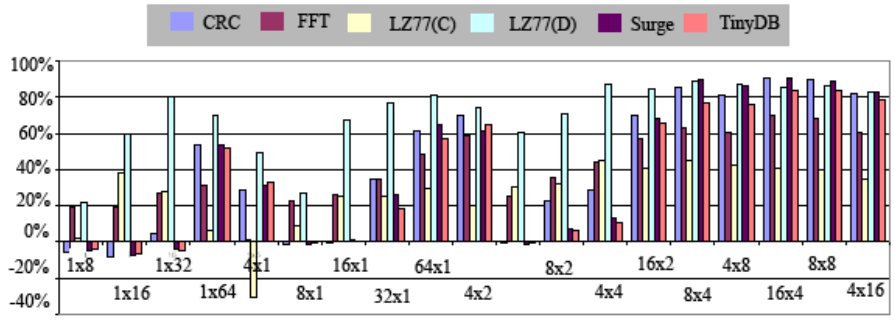

Fig. 4. Average savings on memory access energy dissipation for various MC configurations without the silent store filter (s x a, s: set no, a: associativity)

Figure 5 shows our energy savings for MC configurations with the silent store filter. Notice that, after enabling the silent store filter, even the smallest MC configurations (1x4 and 4x1) save more than 52% of the memory access energy on the average. The optimal configuration is found to be the 8x4 configuration, since it gives similar savings compared to the MC configurations with twice of its size (78.7% of 8x4 vs. 79.2% of 16x4.)

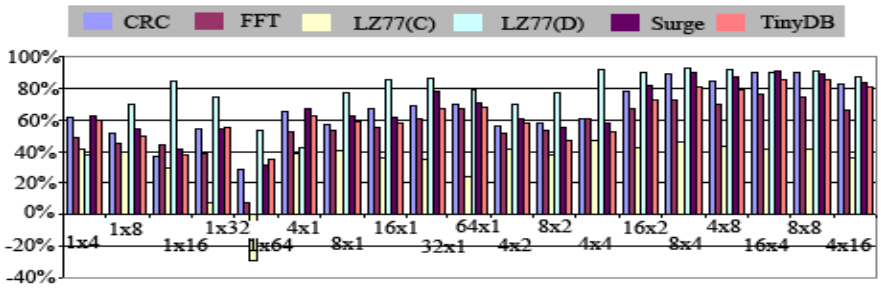

Fig. 5. Average savings on memory access energy dissipation for various MC configurations with the silent store filter (s x a, s: set no, a: associativity)

In Figure 6, we show the hit rates to each MC configurations. The lowest hit rate is observed in the smallest FAMC configuration (i.e. 1x4) as 5.6%, on the average. On the other hand, the highest hit rate is again observed in the FAMC configuration with maximum size (i.e. 1x64) as 95.9%, on the average. The hit rate for the optimal configuration chosen above (i.e. 8x4) is 81.7%, on the average.

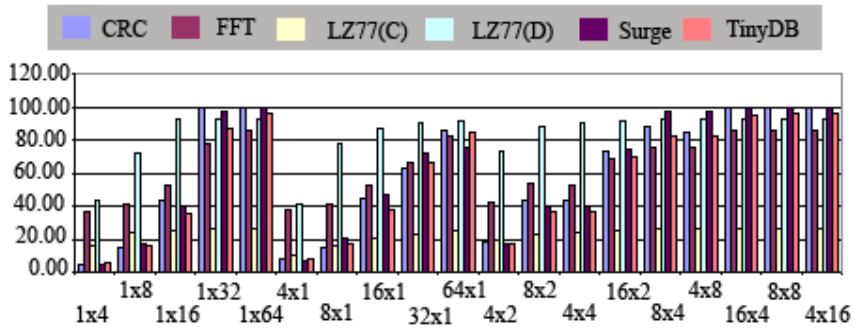

Fig. 6. MC hit rates to various MC configurations

As a result, SAMC (8x4) configuration yields to 78.7% energy savings (average) on memory access operations. However, although energy saving results of memory read/write operations provide means to identify the most effective MC configuration, they do not give the details about total processor energy savings. This consideration requires more figures such as percentage of memory read/write operations. Figure 7.a shows that, on the average, 36% of total instructions are memory read/write instruc-

tions. When we apply these figures to our optimal configuration, we found that the total processor energy savings turn out to be 49.7%, on the average (Figure 7.b)

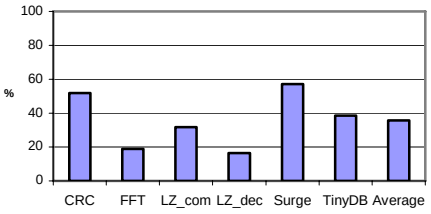

Fig. 7.a. Percentages of memory access instructions over total instructions

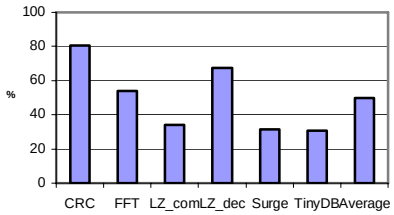

Fig. 7.b. Processor energy savings

Next, we identify the percentage of CPU energy dissipation over total energy dissipation of a sensor node to compute the possible lifetime increase with MC architecture. Figure 8 shows that, CPU energy constitutes 49.6% of the total energy dissipation across the simulated benchmarks, on the average. Using the percentages given in this graph, we computed the percentages of total energy savings of individual WSN nodes. We present the results in Figure 9. Notice that, for communication-intensive applications (*CRC*, *FFT*, *Surge* and *TinyDB*), the savings are 14.2%, on the average. Unfortunately, both *Surge* and *TinyDB* do not utilize low power listening mode; therefore, the results can be further improved through utilizing this mode in these applications. For computation-intensive applications (*LZ77_comp* and *LZ77_decomp*), the savings are 45.9%, on the average. The significant difference between LZ// compress and decompress applications is due to the significant hit rate difference between those applications. Notice that, these energy savings percentages are directly related to lifetime improvement of the node running that specific application.

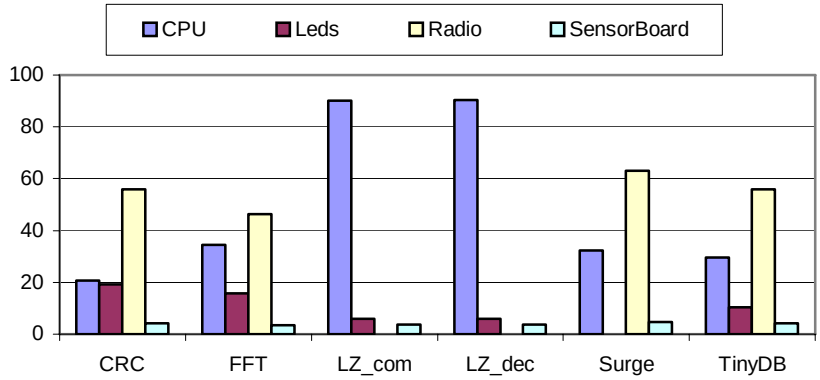

Fig. 8. Percentages of CPU, leds, radio and sensor board energy dissipation

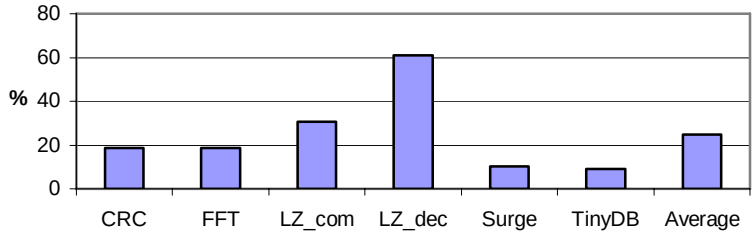

Fig. 9. Node-level energy savings and lifetime improvement percentages

7 Concluding Remarks

In this study, we proposed the architecture-independent MoteCache structure for reducing the energy dissipation of the processor/memory component of wireless sensor nodes. We studied various MoteCache configurations and found that 8-byte 4-way set-associative, silent-store-filtering configuration shows the best energy/lifetime characteristics. This optimal configuration reduces the node energy by 24.7% on the average across a variety of simulated sensor benchmarks. For the CPU-intensive applications, the lifetime of the nodes can be improved by 46%, on the average. The lifetime of the nodes that run communication-intensive applications, such as TinyDB and Surge, is also improved as much as 14%. We also believe that the savings will be greatly improved when low-power listening modes are utilized for these applications.

References

1. Hill J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K., "System Architecture Directions for Networked Sensors", in Proc. of the 9th Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (2000), ACM Press, pp. 93-104
2. Conner, W.S., Chhabra, J., Yarvis, M., Krishnamurthy, L., "Experimental Evaluation of Synchronization and Topology Control for In-Building Sensor Network Applications", in Proc. of WSNA'03, Sep. 2003
3. Madden, S., Franklin, M.J., Hellerstein, J.M., and Hong, W., "TinyDB: An Acquisitional Query Processing System for Sensor Networks", in ACM TODS, 2005
4. Polastre, J.R., "Design and Implementation of Wireless Sensor Networks for Habitat Monitoring", Research Project, University of California, Berkeley, 2003
5. .Ekanayake, V. et al, "An Ultra Low-Power Processor for Sensor Networks", in Proc. ASPLOS, Oct. 2004
6. Hempstead, M. et al, "An Ultra Low Power System Architecture for Sensor Network Applications", in the Proc. of 32nd ISCA'05, Wisconsin, USA, 2005
7. Warneke, B.A. and Pister, K.S., "An Ultra-Low Energy Microcontroller for Smart Dust Wireless Sensor Networks", in Proc. ISSCC, Jan. 2005
8. Lepak, K.M., Bell, G.B., and Lipasti, M.H., "Silent Stores and Store Value Locality", in IEEE Transactions on Computers, (50)11, Nov. 2001

9. Crossbow Technology, Inc., http://www.xbow.com
10. Atmel 8-bit AVR Microcontroller with 128K Bytes In-System Programmable Flash, http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf
11. Ghose, K. and Kamble, M., “Reducing Power in Superscalar Processor Caches Using Sub-banking, Multiple Line Buffers and Bit-Line Segmentation”, in ISLPED'99, 1999, pp.70-75
12. Kucuk, G. et al, “Energy-Efficient Register Renaming”, in PATMOS'03, Torino, Italy, September 2003. Published as Lecture Notes in Computer Science, LNCS 2799, pp.219-228
13. Ponomarev, D. et al, “Energy-Efficient Comparators for Superscalar Datapaths”, in IEEE Transactions on Computers, vol.53, No. 7, July 2004, pp.892-904
14. Titzer, B. et al, “Avrora: Scalable Sensor Network Simulation with Precise Timing”, In 4th International Conference on Information Processing in Sensor Networks, 2005
15. Numerical recipes in C, Cornell University, Online book, http://www.library.cornell.edu/nr/cbookcpdf.html