System-Level Synthesis for Wireless Sensor Node Controllers: A Complete Design Flow

MUHAMMAD ADEEL PASHA, SSE, LUMS, Lahore, Pakistan
STEVEN DERRIEN and OLIVIER SENTIEYS, IRISA-INRIA, University of Rennes 1

2

Wireless sensor networks (WSN) is a new and very challenging research field for embedded system design automation. Engineering a WSN node hardware platform is known to be a tough challenge, as the design must enforce many severe constraints, among which energy dissipation is by far the most important one. WSN node devices have until now been designed using off-the-shelf low-power microcontroller units (MCUs), even if their power dissipation is still an issue and hinders the widespread use of this new technology. In this work, we propose a complete system-level flow for an alternative approach based on the concept of hardware microtasks, which relies on hardware specialization and power gating to drastically improve the energy efficiency of the computational/control part of the node. Our case study shows that power savings between one to two orders of magnitude are possible w.r.t. MCU-based implementations.

Categories and Subject Descriptors: B.1.5 [**Control Structures and Microprogramming**]: Microcode Applications—*Special-purpose*

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: System-level design-flow, design space exploration, domain-specific language, hardware specialization, low-power design, microcoded FSM, power gating, WSN node controller

ACM Reference Format:
Pasha, M. A., Derrien, S., and Sentieys, O. 2012. System-level synthesis for wireless sensor node controllers: A complete design flow. ACM Trans. Des. Autom. Electron. Syst. 17, 1, Article 2 (January 2012), 24 pages.
DOI = 10.1145/2071356.2071358 http://doi.acm.org/10.1145/2071356.2071358

1. INTRODUCTION

Wireless sensor network (WSN) is a fast-evolving technology with a number of potential applications in various domains of daily-life, such as structural health and environmental monitoring, medicine, military surveillance, robotic explorations, and so on. A WSN is composed of a large number of sensor nodes that are usually deployed either inside a region of interest or very close to it. WSN nodes are low-power embedded devices consisting of processing and storage components (a processor connected to a RAM and/or flash memory) combined with wireless communication capabilities (RF transceiver) and some sensors/actuators.

Designing a WSN node is a daunting task, since the designers must deal with many stringent design constraints. For example, as nodes must have small form-factors and limited production cost, they cannot benefit from large power sources [UC Berkeley 1999]. In most cases they must rely on nonreplenishing (e.g., battery) or self-sufficient (e.g., solar cells) sources of energy. As WSN nodes may have to work unattended for

Author's address: M. A. Pasha, EE Department, SSE, LUMS, Lahore, Pakistan; email: adeelpasha@gmail.com.

DOI 10.1145/2071356.2071358 http://doi.acm.org/10.1145/2071356.2071358

long durations (months if not years), their energy consumption is often the most critical design parameter (the power consumption of a power-autonomous WSN node is expected to be lower than 1 mW [Roundy et al. 2004]).

As far as their design is concerned, WSN nodes are still based on low-power MCUs such as MSP430 [Texas Instruments 2009] and ATmega128L [Atmel Corporation 2009]. These embedded processors offer a reasonable processing power with low power consumption at a very affordable cost. However, they are designed for low-power operation across a wide range of embedded system applications. Consequently, they are not necessarily well-suited to WSN nodes, as they are based on a general-purpose processor. At the software end, WSN nodes generally rely on a lightweight operating system (OS) layer to provide concurrency and resource management. Power dissipation of current low-power MCUs still remains orders of magnitude too high for many potential applications of WSN.

In this work we propose a new architectural model in which the control subsystem of a WSN node is built out of *microtasks* that are activated on an event-driven basis. Each *microtask* is synthesized as an application-specific architecture customized to the task at hand. Contrary to traditional approaches, which improve energy efficiency by focusing on compute-intensive kernel, our interest is in control-dominated tasks (device drivers, MAC protocols, routing), which form the bulk of the WSN workload. In addition, we combine this use of specialization with *power gating*, a static power reduction technique, so as to drastically reduce both dynamic and static power. The contributions of this work are the following.

— We introduce a novel architectural model based on the notion of concurrent powergated microtasks, and specifically targeted toward ultra low-power WSN node systems.
— We create an original design-flow for this architectural model, borrowing concepts from both ASIP and high-level synthesis tools. This flow allows programmers to define the behavior of each microtask in C, and to specify the system-level interactions within a WSN node using a domain-specific language to finally generate a synthesizable HDL description of the whole system.
— We present a quantitative evaluation of the approach on typical WSN workloads which demonstrates that, compared to low-power MCUs such as the MSP430, energy savings by one to two orders of magnitude are possible thanks to combined savings on static and dynamic power.

The rest of the article is organized as follows. We start by presenting the related work in Section 2, and describe our proposed system model and notion of microtask in Section 3. Section 4 covers the complete design flow, and in Section 5 we present experimental results for design space exploration and corresponding power benefits. We then conclude, and draw future research directions in Section 6.

2. RELATED WORK

As mentioned in the introduction, power efficiency is one of the most important design parameter for WSN nodes, and consequently a lot of effort has been made to propose energy-efficient implementations of such devices.

To design an energy-aware WSN node, it is mandatory to analyze its power dissipation characteristics. A generic WSN node consists of four subsystems: (i) a control/computation subsystem having an MCU with its program/data memory (and some potential hardware accelerators); (ii) a communication subsystem having RF transceiver; (iii) a sensing subsystem having the sensor/actuator interfaces; and iv) a power supply subsystem. Figure 1 presents the block diagram of a generic WSN node.

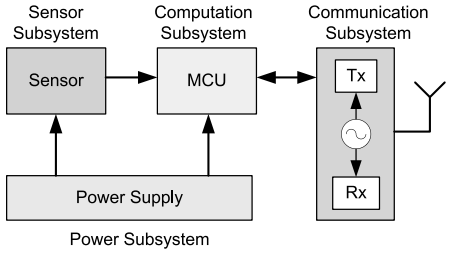

Fig. 1. Block diagram of a generic WSN node.

Among all the subsystems, communication and computation subsystems consume bulk of the power budget [Dutta et al. 2005; Raghunathan et al. 2002], and efforts toward energy reduction should target both of them in particular.

It is also widely accepted that communication energy cost dominates the overall energy budget, and that the focus should be on reducing this communication energy cost. As an example, the use of advanced digital communication techniques (efficient error correction, cooperative MIMO [Nguyen et al. 2007]) and network protocols (energy-efficient routing [Schurgers and Srivastava 2001] and/or MAC schemes such as B-MAC [Polastre et al. 2004] and RICER [Lin et al. 2004]) have shown to help in improving the energy efficiency for communication (see Akyildiz et al. [2002] for a survey).

However, these techniques may significantly increase the workload on the control/computation system and may require processing horsepower that would be above the power/energy budget allocated to typical WSN node MCUs (if custom hardware is not used).

As a result, improving the *control and computation* energy efficiency of WSN nodes is an important issue, which we propose to address through the joint use of hardware specialization and power gating. Indeed, we believe that energy savings obtained through our approach open possibilities for more computationally demanding protocols or modulations which, as a result, would provide better quality-of-service (QoS), lower transmission energy, and higher network efficiency.

In the following, we provide an overview of existing approaches to improve power and energy efficiency at various levels.

2.1 Power Gating

Power dissipation in VLSI devices can be divided into two categories: *dynamic power* caused by the capacitance switching that occurs while the circuit is operating and *static power* caused by leakage current between power supply and ground. The total power dissipation of a CMOS gate i is the sum of dynamic power and static power, and can be expressed as $P_{total} = \frac{1}{2} C_i f_{clk} \alpha_i V_{dd}^2 + V_{dd} I_{leak_i}$, where V_{dd} is the supply voltage, C_i the output capacitance, α_i the activity, f_{clk} the switching frequency, and I_{leak_i} the leakage current of gate i. When the device is active, power is usually largely dominated by dynamic power, and becomes roughly proportional to clock frequency. However, power alone is not a good metric, and especially when considering battery-operated devices, the energy cost is also considered.

In the context of WSN, things are sightly different, as the node remains inactive for long periods (MCU duty cycle is often lower than 1%), hence the contribution of static power also becomes significant and can not be ignored. Among the various techniques proposed to counter leakage power, *power gating* [Babighian et al. 2004; Long and He 2003] has emerged as a promising technique. *Power gating* (PG) consists in turning off

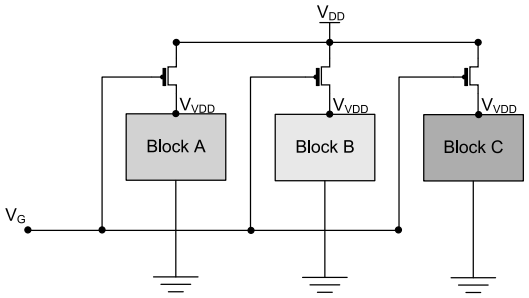

Fig. 2. An example of power gating.

the power supply of inactive circuit components. Therefore, it helps in reducing static power, and is thus very efficient for devices in which components remain idle for long time periods.

The PG technique consists in adding a *sleep transistor* between the actual V_{dd} (power supply) rail and the component's V_{dd}, thus creating a *virtual supply voltage*, called V_{vdd}, as illustrated in Figure 2. This sleep transistor allows the supply voltage of the block to be cut off to dramatically reduce leakage currents. A substantial amount of work is also being done on the sleep transistor sizing to further reduce the leakage power caused by the sleep transistor insertion [Tan and Allen 1994]. *Power gating* has already been used in the context of high-performance CPUs [Hu et al. 2004], and FSM (finite state machine) implementations [Pradhan et al. 2008] where parts of the design are switched on/off according to their activity. The approach helps in reducing the static power dissipation for FPUs of a high-end CPU by up to 28% at the price of a performance loss of 2%, for FSMs the average reported power reduction was also 28%. In the context of WSN, where nodes remain idle most of the time, such a technique has obvious advantages, and is therefore intensively used for implementing the low-power modes of typical nodes MCUs, which we describe in the following section.

2.2 Low-Power Microcontrollers for WSN

As far as low-power microcontrollers (e.g., MSP430 and ATmega128L) are concerned, they share many characteristics: a simple datapath (8/16-bit wide), a reduced number of instructions (only 27 instructions for the MSP430), and several power-saving modes which allow the system to select at runtime the best compromise between power saving and reactivity (i.e., wake-up time). These processors are designed for low-power operation across a range of embedded system application settings, but are not necessarily well-suited to the event-driven behavior of WSN nodes, as they are based on a general-purpose, monolithic compute engine.

Most of the current WSN nodes are built on these commercial MCUs. For example, Mica2 mote [Hill et al. 2000] has been widely used by the research community, and is based on ATmega128L. The same MCU has also been used by the designers of the eXtreme Scale Mote (XSM) [Dutta et al. 2005]. The Hydrowatch platform is built on the MSP430F1611; whereas TI has launched a series of MCUs, MSP430F21x2, which is specialized for WSN nodes [Texas Instruments 2009]. Similarly, NXP came up with a series of low-power microcontrollers, LPC111x, which is based on an ARM Cortex-M0 core [NXP Semiconductors 2010]. Table I summarizes power consumptions of these MCUs at a normalized frequency of 16 MHz along with the actual consumptions presented in the literature.

These figures may seem extremely small in the context of typical embedded devices, however when looking at energy efficiency metrics such as *Joules per instruction*, it

Table I. Actual and Normalized Power Consumption for Various Low-Power MCUs

WSN MCU	Normalized Power	Actual Power
ATmega103L [Atmel Corporation 2007]	66 mW (@ 16 MHz)	5.5 mA (@ 4 MHz, 3.0V)
ATmega128L [Dutta et al. 2005]	48 mW (@ 16 MHz)	8 mA (@ 8 MHz, 3.0V)
MSP430F1611 [Fonseca et al. 2008]	24 mW (@ 16 MHz)	500 μA (@ 1 MHz, 3.0V)
MSP430F21x2 [Texas Instruments 2009]	8.8 mW (@ 16 MHz)	250 μA (@ 1 MHz, 2.2V)
LPC111x [NXP Semiconductors 2010]	13.2 mW (@ 16 MHz)	3.0 mA (@ 12 MHz, 3.0V)

Table II. Actual and Normalized Energy Efficiencies for Various Ultra Low-Power WSN-Specific Controllers

Processor	Operating Voltage (V)	Operating Frequency (MHz)	Actual Energy (pJ/inst)	Normalized Energy (pJ/inst) *(@ 0.50 V)*	Process Technology
SNAP/LE [Ekanayake et al. 2004]	0.60	23	75	52	180
Phoenix [Seok, Mingoo et al. 2008]	0.50	0.1	2.8	2.8	180
Charm [Sheets et al. 2006]	1.03	8	96	23	130
BlueDot [Raval et al. 2010]	NA	8	26	NA	130
MSP-like core [Kwong et al. 2009]	0.50	0.4	27	27	65

appears that the proposed architectures such as MSP430 still offer room for improvement. In particular, it is clear that the combination of specialization and parallelism would significantly help in improving energy efficiency. However, such architectural improvements usually come at the price of a significant increase in silicon area, which leads to unacceptable levels of static power dissipation for WSN.

It is also an acknowledged fact that the power budget of a WSN node that would rely only on energy harvesting techniques is estimated to be approximately 100 μW [Roundy et al. 2004]. Comparing this constraint with that of current MCUs power consumption profiles, clearly drives us toward alternative architectural solutions (e.g., hardware specialization of the system).

Apart from the general-purpose COTS processors, there are several WSN-specific controller implementations that have been proposed by the research community. These controllers try to exploit the WSN-specific characteristics such as *event-centric behavior* and *asynchronous communication* to achieve a lower energy per instruction. Moreover, some of them exploit power gating as well, to reduce the static power consumption. Table II summarizes their energy efficiencies at a normalized operating voltage of 0.5 V along with the actual values presented in the literature.

Although all of these WSN-specific processors show impressive energy efficiency, they also suffer from their inherent weaknesses. For instance, subthreshold logic is highly susceptible to temperature and process variations. In addition, due to the low voltage swing, noise arising from other on-chip components could be an issue. As far as asynchronous processors are concerned, it can be difficult to integrate asynchronous logic into conventional synchronous design flows. The operating frequency range of some of these controllers is also quite limited (e.g., a Phoenix processor can operate at around 100 kHz). Moreover, all of these processors are manually designed and optimized, and no automatic design tool exists for them.

2.3 Power Efficient Hardware Synthesis

High-level synthesis (HLS) and retargetable compilation for ASIPs have been a very active research domain for the last 15 years. Even if there are still many open research

issues, there now exist several mature academic/commercial tools (e.g. XPilot [Cong et al. 2007]; Spark [Gupta et al. 2003]; NISC [Gorjiara and Gajski 2008; Gorjiara et al. 2006]; Catapult-C from Mentor Graphics, Impulse-C, etc.) that are capable of producing specialized hardware descriptions from software specifications in C/C++.

Interestingly, all these tools share a feature: they generally see hardware specialization as a mean to improve performance over a standard software implementation. This performance improvement, however, often comes at the price of an increased area cost (coprocessor or instruction-set extension requires additional area).

Such hardware customization is well known for being a very efficient technique for improving energy efficiency for dataflow-oriented and compute-intensive kernels (digital filters, image processing kernels, FFT, etc.). Hence there has been a lot of research effort on the design of power-aware HLS tools (see Mohanty et al. [2008] for a survey on the topic).

However, the application domain targeted by these tools exhibits specific characteristics: applications have a relatively simple control flow and consist in repetitive computation patterns which exhibit important instruction-level parallelism. It is therefore important to understand that the energy savings mainly come from the ability to take advantage of the extensive amount of (fine grain) parallelism available in the kernel. The performance improvements resulting from this parallelism then enable the use of lower clock frequencies, which in turn permit the use of lower supply voltage, hence improving the energy efficiency of the system.

However, there is very little (if any) quantitative evidence of the benefit of customization for applications that exhibit highly irregular and complex control flows, and in which there are very few opportunities for taking advantage of such instruction-level parallelism. It turns out that most of the programs that are executed by WSN node exhibit such characteristics (low-level device drivers for controlling sensors and peripherals, MAC and routing protocols, etc.).

Besides, most research effort in ASIP and HLS has focused on improving performance over software implementations, even when it implies large area overhead. Indeed, apart from Fin et al. [2001] and L'Hours [2005], very little research has addressed the problem of using processor specialization as a mean of area reduction. In the context of WSN node architecture, where silicon area and ultra low-power are two main design issues, such criteria deserve attention.

It is also important to keep in mind that even though hardware specialization can help in reducing the energy budget of a WSN node, the way concurrency, event and shared resources are managed by the operating system is also crucial. Hence this topic is addressed in the following section.

2.4 WSN Operating Systems

Most embedded system rely on so-called real-time embedded OS to provide adequate constructs to the system designer (preemptive task scheduling, mutex, etc.). However, such full featured OSs cannot be implemented in WSN nodes due to the strong constraints on the memory footprint. In this section, we outline some of the characteristics of the most common OS infrastructures targeted at WSN.

TinyOS [Levis et al. 2005] is one of the earliest and the most commonly used OS in WSN platforms. TinyOS provides a component-based event-driven concurrency model, which focuses on optimizing power usage. TinyOS executes tasks without possibilities for preemption and follows a run to completion semantic. This forces the programmer to split each application functionality in many distinct tasks so as to accommodate the absence of blocking statements, and makes the program more difficult to write and to debug compared to standard *threads like* constructs.

Contiki [Dunkels et al. 2004] is another WSN-specific OS, proposed by Dunkels et al. Contiki uses a notion of *protothreads*, a programming abstraction providing a conditional blocking statement to simplify the event-driven programming for memory-constrained systems. In Contiki, protothreads are stackless, that is, local variable contents are lost whenever the scheduler switches from one protothread to another. Hence the programmer must be cautious while using local variables in his or her program.

We have observed that programmers using Contiki in their WSN systems usually avoid local variables to reduce complications and because they are not familiar with the concept. For instance, PowWow [INRIA 2010a] is an open source WSN platform where programmers have completely avoided the use of local variables.

The MANTIS operating system (MOS) [Bhatti et al. 2005] uses a traditional multithreaded approach. It offers a time-sliced approach in which an interleaved concurrency of multithreading is used to prevent one long-lived task from blocking execution of a second time-sensitive task. However, a larger memory resource is needed for thread-management, as task preemption requires that the complete stack of the preempted thread is to be saved. The average stack size of TinyOS is approximately 16 KB, and it does not support multithreading, while MOS dedicates 128 bytes/thread with a multithreading-based kernel.

In the next section we propose a new architectural model (along with a complete system-level down to the register-transfer level (RTL) design-flow) for WSN node design, based on the notion of concurrent power-gated hardware microtasks.

3. A NEW ARCHITECTURAL MODEL FOR ULTRA-LOW POWER WSN NODES

We propose a new kind of sensor node architecture where the control and computation subsystems consist of several hardware microtasks that are activated on an event-driven basis, each of them being specialized for a specific task of the system.

This original architectural model, combining power gating and specialized hardware, is supported by a complete design flow that enables the specification of a node platform using a combination of C and a domain-specific language. Our design flow is able to produce RTL-level VHDL descriptions of the whole system, but also to generate directives for transistor-level power-gating insertion.

The benefit of our approach w.r.t. to WSN-specific controllers such as BlueDot [Raval et al. 2010] or SNAP/LE [Ekanayake et al. 2004] is that our flow is fully automated, and can be adapted to a broad family of process technologies without too much design effort.

3.1 An Illustrative Case Study

WSN applications are good candidates for being modeled as tasks flow graphs (TFG). In this model, a task execution is triggered by *events*, such events being external or produced by another task.

Figure 3 shows the TFGs of a lamp-switching application, where a transmitting node demands a receiving node to switch on/off its lamp when a button is pressed at the transmitter end. The tasks given in TFGs are implemented as specialized hardware *Microtasks* (the notion of a hardware microtask is explained in the next section).

Figure 3(a) shows the TFG for a receive mode when a node waits for a signal from the transmitter and switches the lamp, whereas Figure 3(b) presents the TFG for transmit node where a node waits for a push-button event and sends a signal to the receiver to switch the lamp. This application involves several tasks: data transmission, data reception, wait for acknowledgment, and the timer, push-button and lamp switching APIs, and so on.

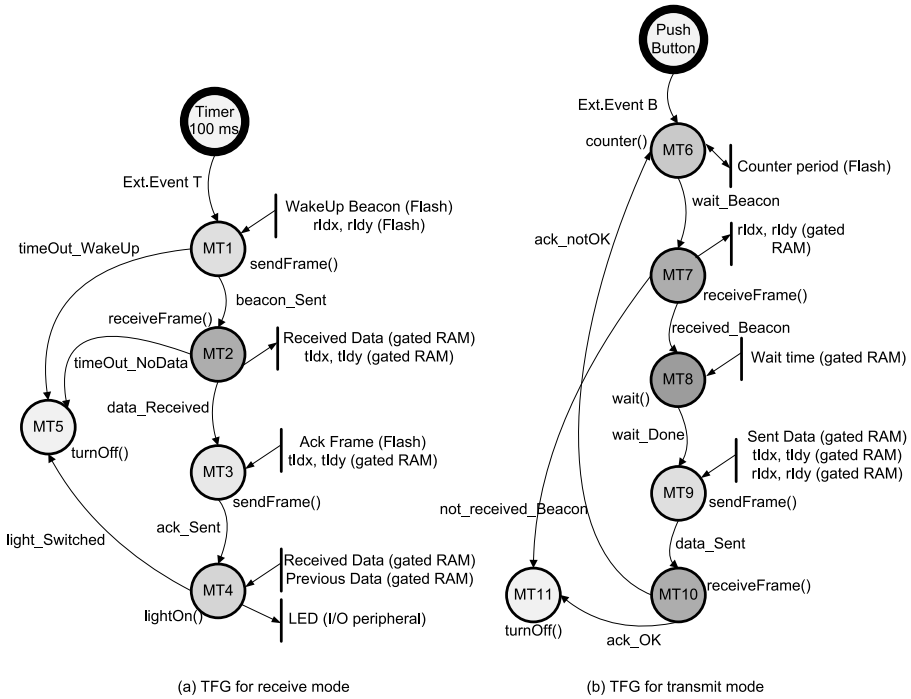

Fig. 3. TFGs presenting the microtasks running during a lamp-switching application.

All these control-oriented tasks are spread across different layers of the communication stack and involve further subtasks associated with them. For instance, beacon and data packet transmission and reception involve physical-layer functions that exchange data between I/O peripherals of the MCU and the RF transceiver using the SPI-protocol. The control flow itself follows a simplified version of RICER, a low-power MAC protocol [Lin et al. 2004].

In a typical WSN node, such tasks are handled by an MCU and corresponding OS that provides support for multitasking features.

3.2 Notion of Hardware Microtask

Our approach relies on the notion of a specialized hardware structure called a *microtask*, which executes parts of the WSN node code. In contrast to an instruction-set processor, the program of a microtask is hardwired into an FSM that directly controls a semi-custom datapath. This makes the architecture much more compact (no need for an instruction decoder, nor instruction memory) and allows the size of storage devices (register file and RAM/ROM) as well as the arithmetic logic unit (ALU) functions to be customized to the target application/functionality. Each of these microtasks can access a local and/or shared data memory, and can also access peripheral I/O ports (e.g., SPI link to an RF transceiver such as the CC2420).

Figure 4 shows the microarchitecture for such a microtask (here with an 8-bit datapath), dotted lines represent control signals generated by the control FSM, whereas solid lines represent dataflow connections between datapath components.

Such a drastically simplified architecture allows for obvious dynamic power savings compared to a standard MCU architecture, with a negligible loss in performance, since the datapath is tailored to the application at hand. However, the approach is more

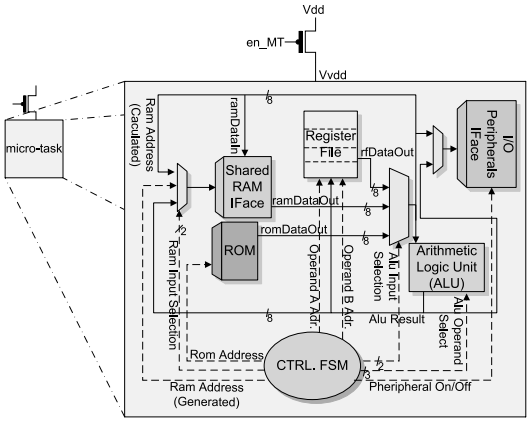

Fig. 4. Architecture of a generic microtask.

efficient for fine-grain tasks, as the specialization can be less effective if more and more functionalities are handled by the task.

To take advantage of this approach, we therefore propose to distribute the entire WSN node software framework into a set of hardware microtasks, so as to maintain a high degree of specialization within each task. For example, a complete WSN communication stack uses approximately 3500 instructions by distributing the stack functionality onto 7 microtasks, we can reach an average task size of 500 instructions, which is a granularity level at which we can expect significant energy improvements. This of course comes at the price of a (limited) increase in area and, more importantly in static power dissipation.

To tackle this *static power*, we provide microtasks with a power-gating mechanism, which allows us to turn-off the microtask power supply and drastically reduce the leakage current. We have shown in a previous work that this approach is very promising, and that computational energy efficiency improvements of around two orders of magnitude were possible compared to state-of-the-art MCUs, even for control-dominated code, where architectural optimizations are known to have a lesser impact on power than in pure dataflow [Pasha et al. 2009].

At first glance, our approach may seem similar to the processor specialization of Gorjiara and Gajski [2008], however, there are two significant differences. First, our main goal is to minimize the silicon footprint of the resulting microtask, improving performance is only a secondary objective. Second, our microtasks should be seen as a temporary computational resource, which can be completely powered off when not needed (thanks to power gating).

3.2.1 The Issue of Reprogrammability. Of course, our solution has a drawback: it assumes that the microtasks are hardwired into silicon as ASIC (application-specific integrated circuit) blocks. This means that the behavior of each micro-task is fixed, making postproduction upgrade or bug-fixing very costly. This may look like a show stopper, as flexibility is often of a great concern for WSN system designers. However, when looking more carefully to actual design practices, we can see that the need for flexibility and reprogrammability is essentially geared toward the user application layer, which happens to represent only an extremely small fraction of the WSN node processing workload, this latter one being almost entirely dedicated to the communication stack.

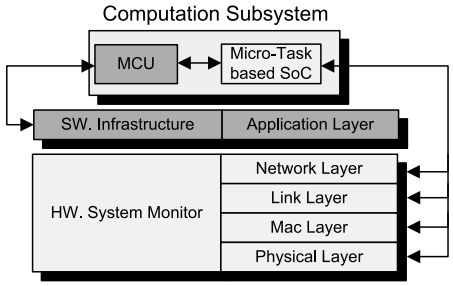

Fig. 5. Proposed solution to tackle the issue of loss of reprogrammability.

Besides, in practice, designing a new WSN application usually means adapting an existing WSN software framework to a new user application. In other words, the communication stack is generally reused "as is" and routing algorithms, MAC protocols, and device driver layers remain the same (even if their behavior is parametrized). We therefore propose to combine the best of both worlds, that is:

— use a very small silicon footprint instruction-set processor (8-bit datapath, minimalistic RISC instruction set) with a power-gating feature to implement the application layer user software;
— use a distributed system of microtasks to handle the OS-level services of the WSN node (mostly the communication stack).

Such an approach preserves most of the energy savings provided by specialization, while providing flexibility at the application level. Figure 5 shows the block diagram of a computation/control subsystem based on the proposed approach above.

However, managing interactions between these tasks raises many other issues which we propose to address in the following section by proposing an associated system-level execution model (and its associated automated design flow).

3.3 System-Level Execution Model

In this section, we describe an execution model for our proposed approach and detail its particularities and limitations w.r.t to the features of standard WSN OS.

Figure 6 represents a system-level view of a WSN node platform designed according to our proposed approach. Such a system consists of the following:

— an application-level programmable instruction set processor that is used to implement the application level code;
— a set of power-gated application microtasks accessing peripherals (RF, LED, sensor) and shared memory blocks;
— a hardware *system monitor (SM)* that controls the execution of all the micro-tasks along with the application processor;
— event-triggering peripherals (e.g., wake-up timer) that send events to the *SM*.

3.3.1 Events and Commands. In our approach, we use *command* and *event* message structures between monitor and microtasks similar to those of TinyOS. A *command* is an enable signal generated by the *SM* toward a microtask signaling the start of its operation. On the other hand, an *event* is a control signal generated by a microtask to the *SM*.

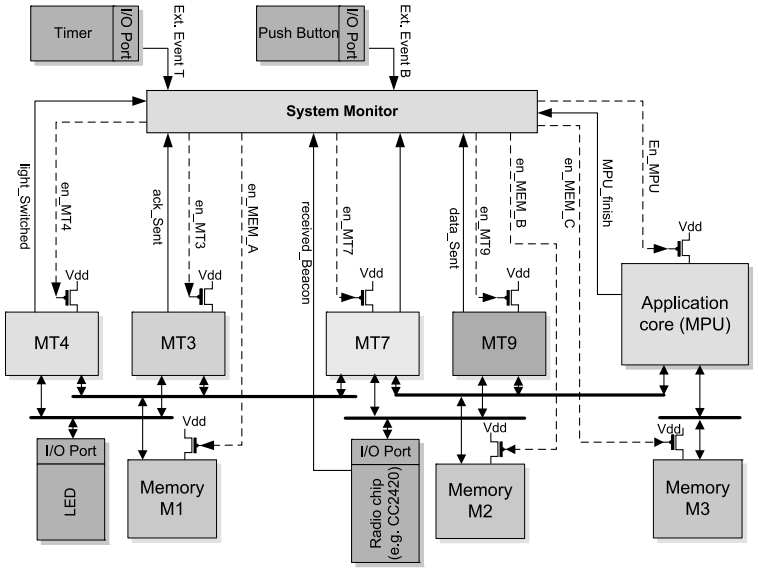

Fig. 6. System-level view of a microtask based WSN node control/computation subsystem.

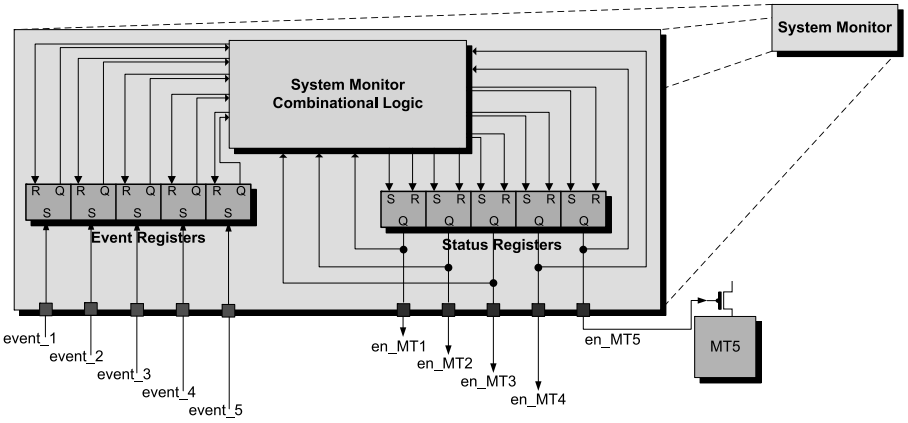

Fig. 7. Block diagram of the *system monitor* designed for the lamp-switching example of Figure 3.

Dotted lines in Figure 6 represent command signals driven by the *SM* to the microtasks and shared memories. Solid lines represent external and internal events sent back to the *SM*. *Events* can be of two types.

— an *internal event* is generated by a microtask indicating its termination or preemption (in case of a subroutine call).
— an *external event* is generated by a peripheral that can serve as wake-up call from shut-down mode.

3.3.2 System Monitor. The *SM*, which is responsible for powering on/off microtasks/memory blocks upon reception of an (or a combination) event, is itself implemented as a simple combinational logic block, to evaluate the guard conditions for microtask activation, and a set of 1-bit status registers carrying the state of events

and command signals until they are used by microtasks. Figure 7 provides a block diagram of the *SM* components for the case study example described in Section 3.1.

3.3.3 Microtask Activation/Deactivation. In our execution model, we restrict ourselves to microtasks following a *run-to-completion* semantic, as in the case of TinyOS tasks. This ensures that a given microtask will never reach a state in which it is activated (i.e., it is powered-on) while not executing useful computation (i.e., blocked waiting for some event).

In addition, we make sure that at a given time instant there are not two active tasks sharing an access to the same shared resource. This property is ensured (in a conservative way) by the *SM*, which makes sure that prior to activating a candidate task, there are no other active tasks that may need access to a resource that *may* also be used by the candidate microtask.

In the following paragraphs, we present some noticeable features of our system model and compare them whenever possible with those of existing WSN OSs.

3.3.4 Concurrency Management. Both TinyOS and Contiki allow the programmer to express concurrency in their application through the use of specialized constructs such as *Protothreads* for Contiki and *Tasks* for TinyOS. One of the goal of such nonstandard constructs is to help in reducing context switching and task-scheduling overhead.

In our approach, as we rely on several physically distinct hardware microtasks, we provide a natural support for concurrency and task-level parallelism, and do not have to pay for any context-switching overhead. Similarly, scheduling does not have any execution time overhead, even if taking into account the extra silicon area required by the *SM*.

Another advantage of the approach lies in the fact that shared resources such as memory or I/O ports are much easier to handle than in a standard multiprocessor architecture, thanks to power gating. Indeed, since no two tasks accessing a shared resource can be active (powered-on) at the same time, we can save the typical extra tristate (or MUX) logic used to share data/address bus lines, which results in savings in terms of power and area.

3.3.5 Task Hierarchy and Granularity. As mentioned in Section 1 and illustrated in Section 5, the power savings obtained through specialization are sensitive to the task computation granularity, the size of which should remain in the order of a few hundreds of assembly-level instructions to enable significant savings.

A natural way to control the granularity of a task in an imperative language is through the use of subprograms. In our execution model, we offer two ways for handling subroutine calls made within a hardware task specification.

The first one is straightforward and consists in inlining the subroutine calls, this increases the task granularity and is acceptable for small subprogram calls. The second one is more complex and consists in generating a new microtask dedicated to the subprogram execution. In the latter case, the parent (i.e., caller) microtask invokes (i.e., activates) this child microtask. As the child microtask is also power-gated, it contributes to the power budget only marginally, while helping in maintaining a high level of specialization within the parent task.

3.3.6 Memory Management. There are small locally shared memories used by microtasks that can be power-gated once their corresponding microtasks are shut down. We must emphasize that a system-level model (see Section 4.2) is used to specify, after a given task termination, which memory was being used by the task, and this information is used to turn the shared memories on/off. This notion of small

power-gated locally shared memories, instead of a large global one, will also contribute to the overall reduction in power consumption.

There is also a very small global memory (based on nonvolatile flash technology) that is used to store the global data such as the node-ID, node-address, neighborhood table, and if there is some potential data to be saved by the microtasks in case of local memory shut-down. Since an *always-ON* memory can be critical from the point of view of static power dissipation, we use a nonvolatile flash memory to store the needed data that can be turned off in case of total node shutdown.

4. SOFTWARE DESIGN FLOW

Of course, even the soundest execution model is useless without a supporting solid design flow, which allows the programmer to proceed directly from a specification written in a high-level language to an executable specification, which in our case consists of an RTL description of the entire hardware system.

This section starts with an overview of our hardware microtask synthesis flow, we then introduce the domain specific language (DSL) which we use to describe the system-level organization of the platform.

4.1 Microtask Synthesis Flow

As mentioned in Section 2.3, even if several industrial strength ASIP and HLS tools exist, they are still not well suited to the WSN application domain.

— They focus on performance improvement and do not leverage specialization as a way to reduce area/cost, which is an important criterion in WSN.
— They focus on compute-intensive kernels, whereas WSN workloads mostly consist of firmware/device driver tasks, with irregular execution flow dominated by complex bitlevel I/O operations.

Hence, this application domain is a niche that has not received attention either from the ASIP or the HLS community. Because of this, and to address the shortcomings of existing tools, we designed a custom hardware synthesis flow by extending the GeCoS compiler infrastructure [L'Hours 2005], an open-source retargetable compiler framework for ASIP (Application Specific Instruction Set Processors).

4.1.1 Custom Pattern Extraction. GeCoS uses a simple *intermediate representation (IR)* in the form of a CDFG (Control and Data Flow Graph), which is used as an input to a highly flexible instruction selection framework. We used this ability to efficiently map the target task program on each microtask.

In the GeCoS *IR*, each basic operation (e.g., memory fetch or store, addition or subtraction, conditional jumps, etc.) is represented by a tree node, whereas a real machine instruction can often perform several such operations within a single execution. Finding the appropriate mapping of machine instructions to a given *IR*-tree is done through the so-called instruction selection phase.

There exist very sophisticated approaches for instruction selection (e.g., instruction selection of DAG (Directed Acyclic Graph) [Liao et al. 1995; Martin et al. 2009]). However, since our target programs offer little opportunity for benefiting from such algorithms, our current implementation uses a simple BURG-based tree-covering technique [Fraser et al. 1992]. Instead, the originality of our approach comes from the fact that our microtasks are not constrained by a predefined processor instruction set architecture (ISA), and we can therefore leverage a relatively large number of operation patterns to obtain an efficient covering.

For example, our microtask uses several patterns involving complex memory operands that are common in device driver code. It also uses wordlength-specific

Pattern	Action	Comments
SET(INDIR(INT), AND(INDIR(INT),INT))	andIO #ioPort, #const	**Performs an AND operation of an I/O port with a constant value**
SET(INDIR(INT), INT)	movIO #ioPort, #const	**Moves a constant value to an I/O port**
SET(INDIR(INT), OR(INDIR(INT), reg8))	orIOB #ioPort, rByte_2	**Performs an OR operation of an I/O port and an 8-bit variable**
ADD(mem, reg8)	addGB @(rByte_3), rByte_2	**Adds and stores an 8-bit variable to a memory location**
SET(GLOBAL, AND(GLOBAL, INT)	andiG @(symbol), #const	**Performs an AND operation of the memory contents pointed by a symbol to a constant value**

Fig. 8. Some of the rules followed in our instruction selection phase, where rByte_2 and rByte_3 are 8-bit temporaries.

instructions (byte, word, or long operand) so as to efficiently determine the required bitwidth for the micro-architecture datapath. Needless to say, our microtask ISA is highly customizable with minimum development effort.

As an illustration, Figure 8 shows some of the complex patterns that we try to match in the IR. While complex in terms of operations, such patterns can be easily mapped to hardware, and hence are good candidates for being executed on custom functional units.

4.1.2 Microtask HDL Generation. The machine-specific *IR* obtained by instruction selection is transformed into an FSM, in which each instruction is mapped to a microcoded sequence (FSM states) that controls the microtask datapath.

This transformation stage also involves a wordlength conversion step in which instructions operating on 16-bit or 32-bit operands may be transformed into sequential byte-level microcode in order to match the characteristics of the underlying microtask datapath.

From the set of instruction patterns used in the selection phase, we also derive a template of the microtask datapath, which is trimmed down to provide the minimum-required functionality (types of operators and number of registers) required to execute the task at hand.

Our microtask generation flow is build on top of an *Eclipse Modeling Framework* (EMF), a Model-Driven Engineering (MDE) framework. More precisely, we defined a metamodel enabling the specification of complex custom microarchitectures as assembly of an FSM and datapath components (e.g., register file, ALU operators, ROM, I/O ports). We also leveraged the code generation facilities provided by the framework to provide a synthesizable VHDL backend for the microtask description.

4.2 System-Level Synthesis

Because most power-gating task activation/deactivation policies would be somewhat difficult to express in a language such as C, we designed a domain specific language that is used to specify the system-level execution model and the internal organization (e.g., microtasks, shared memories, peripherals, etc.) of a WSN node.

In this DSL, each microtask of the system is linked to a specific C function which specifies the behavior for the task. The DSL also specifies the event configuration which can trigger the task activation and register the events that can be produced by the task. Similarly, we indicate for each task which global variables are live (still used) or dead (e.g., their content can be lost without harm) at the end of the task execution. Finally, the DSL is also used to describe the global/shared variable memory allocation by mapping them to specific memory blocks.

As an example, a micro-task T_N can only be activated when the following conditions are met.

— All internal/external event signals present in T_N's event configuration are true.

```
system send_receive_data {
	include "send_receive.gecos"
	events { extPB, extET, beacon_Sent, data_Received,
		ack_Sent, timeOut0, timeOut1, timeOut2, receiver_OFF, transmitter_OFF,
		counter_Start, beacon_Received, data_Sent, ack_OK, ack_NOK, radio_OFF}

	memory memB [gated] {
		contains globals {neigh_IdX,neigh_IdY, receiveFrame, sentFrame, pushButtonStatus}
	};

	memory memC [permanent] {
		contains globals {my_IdX, my_IdY}
	};

	ioModule led {
		contains ports { port LED 8}
	};

	ioModule pushButton {
		contains ports { port PUSHBUTTON 7}
	};

	ioModule cc2420 {
		contains ports { port P2IN 0, port P5OUT 1, port U1TCTL 2, port U1RXBUF 3,
					port U1TXBUF 4, port URXIFG1 5, port IFG 6}
	};

	microTask receiveData {
		activates With { beacon_Sent }
		produces { data_Received }
		reads ioModule { cc2420 }
		writes memory { memB }
	};

	microTask sendBeacon {
		activates With { extET }
		produces { beacon_Sent }
		writes ioModule { cc2420 }
		reads memory { memC }
	};
}
```

Fig. 9. A snapshot of the system DSL.

— Event signals present in the event configuration of a task T_M are false where T_M is such a task that is sharing a write-access with T_N to a memory or I/O resource.

Using the above-mentioned conditions, we derive the following guard condition expression for a micro-task T_N: $C_{G_N} = E_{A_N} \& \forall_{T_M} not(E_{A_M}) \& \forall_{T_I} not(E_{A_I})$

where E_{A_N} is the event configuration for T_N activation, T_M is a micro-task sharing a write-access with T_N to a memory resource and T_I is a micro-task sharing a write-access with T_N to an I/O resource. The command signals, generated by combinational logic, that are used to control the status registers (Figure 7) are evaluated by the guard expression derived above.

The DSL was developed using Xtext (another MDE framework) and generates a VHDL description for the monitor. Figure 9 shows a sample system description written in our developed DSL.

Figure 10 shows the complete design-flow for microtask-based node generation: it starts from the application description modeled as a TFG using the DSL, each task written in C, and goes till the VHDL code generation for the node.

5. EXPERIMENTAL VALIDATION

This section discusses the WSN benchmarks and OS tasks used for design space exploration and microtask generation and, later on, analyzes the experimental results and corresponding power savings.

5.1 WSN Benchmarks and OS Tasks

Several attempts have been proposed to profile the workload of a generic WSN node. Two of the recent application benchmarks for WSN are SenseBench [Nazhandali et al.

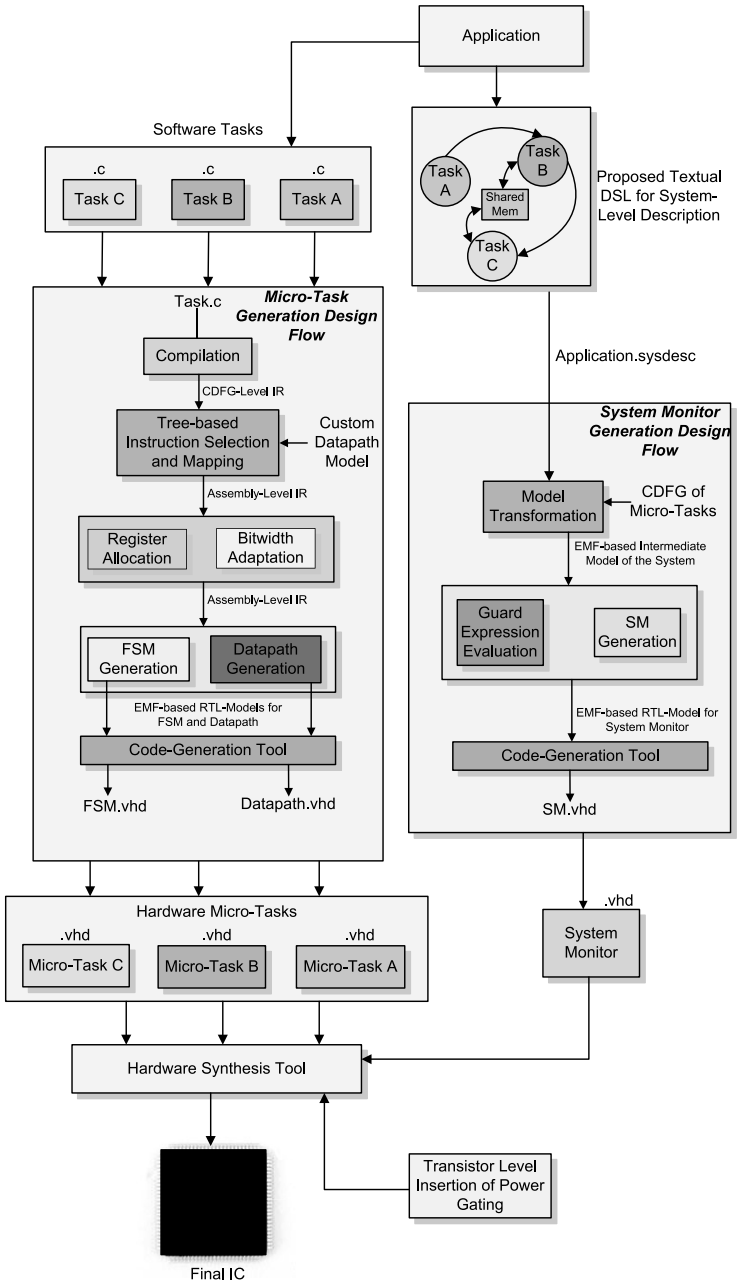

Fig. 10. Software design flow for the IC generation.

2005] and WiSeNBench [Mysore et al. 2008], from which we extracted most of our examples.

To cover the OS-task aspect, we used several OS-related control tasks such as a simple next-node calculation function used in multihop geographical routing algorithm (similar to that used by PowWow [INRIA 2010a]), and the drivers used to exchange

Table III. Power Consumption for Datapaths With Different Design Parameters (@ 16 MHz)

Bit Width	Register File Depth	RAM Depth	ROM Depth	ALU Fns.	Power (μW)		Area (μm^2)	
					130 nm	65 nm	130 nm	65 nm
8	4	4	4	8	57	16.76	7635	2159
8	8	0	8	6	49.7	14.17	7038	2093
8	16	0	2	4	69	20.2	11163	3387
8	4	2	2	2	42	12.23	6160	1617
8	16	2	4	6	93	26.8	13098	4034
16	8	0	4	6	99	27.8	13184	4013
16	4	4	4	8	116	34.05	14423	4199
16	16	0	2	4	139.5	40.73	21590	6555
16	4	2	2	2	88	25.62	11715	3251

data with the SPI-interface of RF transceiver such as CC2420. All of the application/OS tasks above provide a comprehensive database from real-life WSN applications used most by WSN programmers.

5.2 Power and Energy Benefits of the Proposed Approach

We used the proposed software designflow to generate microtasks having both 8- and 16-bit datapaths to monitor the power savings as compared to commercial MCUs such as the MSP430.

We synthesized the datapaths that have optimized design parameters for different microtasks extracted from the above benchmarks. The synthesis was performed for both 130 nm and 65 nm CMOS technologies using *design compiler* from Synopsys; the power and area estimations are given in Table III.

5.2.1 Power Savings w.r.t. Conventional MCU. The complete microtask (FSM + datapath) VHDL designs have also been synthesized for both process technologies. We used these synthesis results to extract gate-level static and dynamic power estimations (@ 16 MHz). These results were compared to the power dissipated by two different versions of MSP430 MCU:

— *tiMSP*, a TI MSP430F21x2 using the datasheet information (8.8 mW @16 MHz in active mode) which includes memory and peripherals;
— *openMSP*, an open-source MSP430 processor core without accounting for program and data memory. We synthesized it and did the statistical power estimation for 130 nm technology and found it to be 0.96 mW @ 16 MHz.

We expect the actual power dissipation of the MSP430 core, including memory, to lie somewhere between the two values, and compare our results to both of them.

The results are given in Tables IV, V, and VI, where Table IV shows the instruction and cycle count, time taken, power and energy consumption for both *tiMSP* and *openMSP* (for software implementation). On the other hand, Table V shows the power and energy benefits for 8-bit microtasks over both the MSP430 implementations. Similarly, Table VI summarizes the results for 16-bit microtasks. It can be observed that, for different benchmark applications and OS tasks, our approach gains between one to two orders of magnitude in power and energy consumption.

For the sake of completeness, we also synthesized the VHDL code generated for the *system monitor* of a lamp-switching WSN node. The result shows that it consumes only 12 μW (@ 16 MHz).

Table IV. Power/Energy Consumption of MSP430 for Different Application Tasks (@ 16 MHz)

Task Name	MSP430						
	Instr. Count	Clk Cycles	time (μs)	Power (mW)		Energy (nJ)	
				tiMSP	openMSP	tiMSP	openMSP
crc8	30	81	5.1	8.8	0.96	44.9	4.9
crc16	27	77	4.8	8.8	0.96	42.2	4.6
tea-decipher	152	441	27.5	8.8	0.96	242	26.4
tea-encipher	149	433	27.0	8.8	0.96	237.6	26
fir	58	175	10.9	8.8	0.96	96	10.4
calcNeigh	110	324	20.2	8.8	0.96	177.7	19.4
snd2SPI	132	506	31.6	8.8	0.96	278	30.3
rcvFromSPI	66	255	15.9	8.8	0.96	139.9	15.2

Table V. Power and Energy Gain of 8-bit Microtasks over MSP430 (@ 16 MHz). (Here, P1 and E1 are the Power and Energy Gains w.r.t. tiMSP, whereas P2 and E2 are the Power and Energy Gains w.r.t. openMSP)

Task Name	8-bit Micro-task											
	No. States	time (μs)	Power (μW)		Energy (pJ)		P. Gain (x) P1/P2		E. Gain (x) E1/E2		Area (μm^2)	
			130 nm	65 nm	130 nm	65 nm	130 nm	65 nm	130 nm	65 nm	130 nm	65 nm
crc8	71	4.4	30.09	8.0	132.4	35.3	292/32	1095/32.0	339/37	1272/37.3	5831.7	1762
crc16	103	6.4	46.92	12.4	300.3	79.2	187/20.4	710/21	140.5/15.3	532.8/15.5	8732.5	2678
tea-decipher	586	36.6	84.5	22.6	3090	827	104/11.4	389/11.3	78/8.55	292.6/8.6	19950	6138
tea-encipher	580	36.2	87.3	23.3	3160	845	101/11	377/11	75/8.2	281/8.3	20248	6230
fir	165	10.3	75.3	20.4	775.6	209.7	116/12.8	432/12.5	123.8/13.4	458/13.3	13323.7	4124
calcNeigh	269	16.8	74.3	20.1	1248.2	337.8	118/12.9	437/12.7	142.4/15.5	526/15.4	14239.4	4454
snd2SPI	672	42	33.3	8.84	1400.3	371.3	264/28.8	995/29	198.5/21.7	748/21.7	10578	3434
rcvFromSPI	332	20.7	27.3	7.4	565	153.2	322/35	1189/34.6	247.6/26.7	913/26.8	5075.3	1561

5.2.2 Approximate Energy Efficiency. As the microtasks generated through our design-flow are also not instruction-set processors, their exact energy efficiency in terms of *Joules/instruction* cannot be measured. Hence, we used a notion of *Joules/task* as used by Hempstead et al. [2009] in their work.

Since these microtasks are comparable to the MSP430 in terms of their execution time, we used the instruction count for MSP430 implementation from Table IV and the actual energy consumption of the hardware microtask (for each application and control task) to calculate equivalent energy efficiency. The average energy efficiency of a microtask is 3.2 pJ/instruction for 1.2 V, 16 MHz in 65 nm. In addition, if we scale the results down to 0.5 V, the normalized energy efficiency of the approach is 1.3 pJ/instruction. This value is much better than most of the WSN-specific controllers mentioned in related work (c.f., Table II).

5.2.3 Optimum Bitwidth for Microtasks. We generated the microtasks for application codes having a variety of wordlength operations. For example, the application *crc16* mostly uses 16-bit wordlength operations, while operations in *tea-decipher* and *tea-encipher* mostly involve 32-bit wordlength data. The rest of the applications under-test use 8-bit wordlength operations.

As expected, for application codes having wordlengths greater than 8-bits, an 8-bit microtask has twice the number of FSM states than a 16-bit microtask. However, interestingly, the FSM of a microtask consumes much less power than the datapath,

Table VI. Power and Energy Gain of 16-bit Microtasks over MSP430 (@ 16 MHz). (here Again, P1 and E1 are the Power and Energy Gains w.r.t. tiMSP, whereas P2 and E2 are the Power and Energy Gains w.r.t. openMSP)

Task Name	16-bit Micro-task											
	No. States	time (μs)	Power (μW)		Energy (pJ)		P. Gain (x) P1/P2		E. Gain (x) E1/E2		Area (μm^2)	
			130 nm	65 nm	130 nm	65 nm	130 nm	65 nm	130 nm	65 nm	130 nm	65 nm
crc8	71	4.4	55.3	14.71	242.6	64.72	159.6/17.4	598.2/17.4	185.1/20.2	693.7/20.3	10348	3097
crc16	73	4.56	55.0	14.69	251.0	66.98	159.8/17.4	599/17.4	168.1/18.3	630/18.4	10280	3102
tea-decipher	308	19.2	152.8	40.85	2940	784.3	57.6/6.2	215.4/6.3	82/9	308.5/9.04	27236	8380
tea-encipher	306	19.1	152.3	40.61	2910	776.0	57.8/6.3	216.7/6.3	81/8.93	306.2/9	27069	6211
fir	168	10.5	144.2	39.03	1514	409.8	61.02/6.7	225.5/6.56	63.4/6.9	234.3/6.8	23547	7164
calcNeigh	269	16.8	142.4	38.58	2392	648.1	61.8/6.7	228/6.4	74.3/8.1	274/8	24745	7613
snd2SPI	672	42	58.1	15.53	2440	652.2	151.5/16.5	566.6/16.5	114/12.4	426/12.52	14863	4771
rcvFromSPI	332	20.7	50.0	13.67	1036	283.0	175.8/19.2	643.7/18.72	135/14.7	494/14.42	9485	2858

and power consumption of even very large FSMs increases in a sub-linear fashion with the number of states.

As a result, an 8-bit microtask consumes nearly half the power and silicon area as a 16-bit microtask, Figure 11 (a) and (b). As far as the energy consumption is concerned, for codes with wordlength greater than 8-bits, total energy consumption of an 8-bit and 16-bit microtask is nearly the same. On the other hand, for application codes with 8-bit wordlength, an 8-bit microtask consumes half of that of a 16-bit microtask, Figure 11(c).

Hence, since the datapath power dominates the FSM power in our examples, an 8-bit microtask is a better solution. Nevertheless, for cases where FSMs could be comparatively much larger and consume more power than the datapath, microtasks having larger bitwidth would become more suitable.

5.2.4 Overall Energy Gain of a Hardware Microtask. Tables V and VI show the energy gains of the proposed technique during the active period only, whereas the overall energy gain of a hardware microtask can be obtained if the complete time period of a task activation and the static energy dissipation during stand-by mode are considered. Figure 12 shows the simplified version of time distribution for *snd2SPI* task activation having a time period of 100ms which can be considered highly reactive w.r.t. WSN applications. Here, we can clearly see that the node has a pretty low duty cycle (even less than 1% of the whole time period). The switching delays of the MSP430 are taken from its datasheet, whereas that of the microtask block is taken from the previous work [Pasha et al. 2009].

The overall energy gain of our approach over the MSP430 is presented by the following expression:

$$Gain_{tot} = \frac{E_{act_{msp}} + (P_{stby_{msp}} \times T_{stby_{msp}}) + (\frac{1}{2} P_{act_{msp}} \times (T_{on_{msp}} + T_{off_{msp}}))}{E_{act_{mt}} + (P_{slp_{mt}} \times T_{slp_{mt}}) + (\frac{1}{2} P_{act_{mt}} \times (T_{on_{mt}} + T_{off_{mt}}))}$$

— $E_{act_{msp}}$ is the dynamic energy of the MSP430 (given in Table IV),
— $P_{stby_{msp}}$ is the static power consumption of the MSP430,
— $T_{stby_{msp}}$ is the time spent in standby mode by the MSP430,
— $P_{act_{msp}}$ is the dynamic power of the MSP430 (given in Table IV),
— $T_{on_{msp}}$ is the turn-on delay of the MSP430 (given in [Texas Instruments 2009]),
— $T_{off_{msp}}$ is the turn-off delay of the MSP430 (given in [Texas Instruments 2009]),

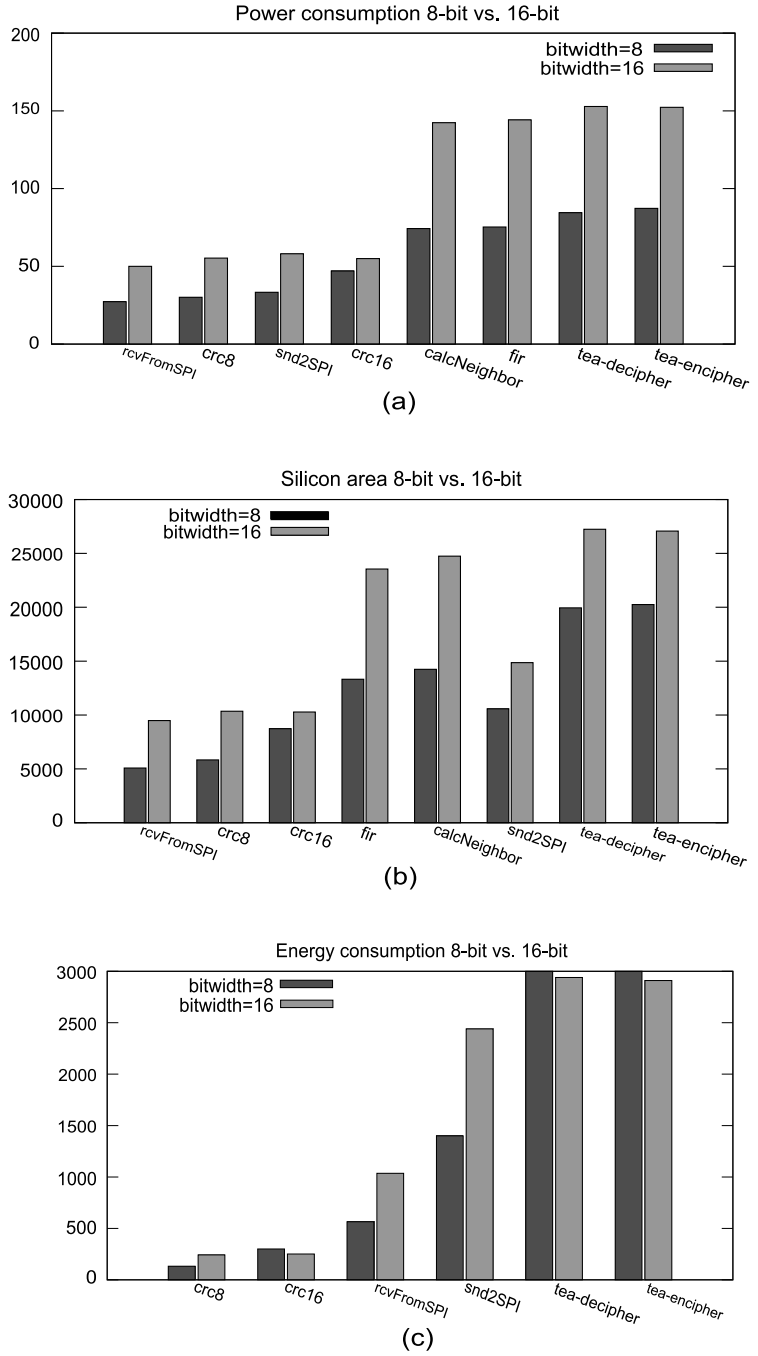

Fig. 11. Comparison of power, area, and energy consumption for 8- and 16-bit microtasks.

— $E_{act_{mt}}$ is the dynamic energy of the micro-task (given in Table V)
— $P_{slp_{mt}}$ is the static power consumption of the micro-task,
— $T_{slp_{mt}}$ is the time spent in sleep mode by the micro-task,

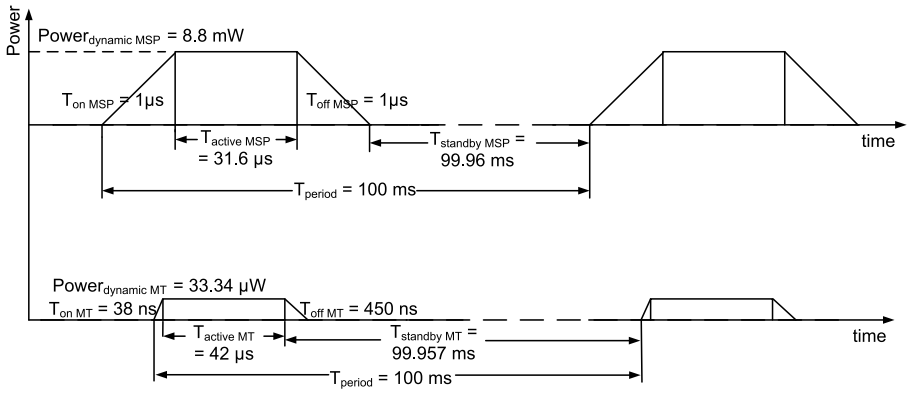

Fig. 12. Time distribution of *snd2SPI* task duty cycle.

— $P_{act_{mt}}$ is the dynamic power of the micro-task (given in Table V),
— $T_{on_{mt}}$ is the turn-on delay of the micro-task ([Pasha et al. 2009]),
— $T_{off_{mt}}$ is the turn-off delay of the micro-task ([Pasha et al. 2009]).

The MSP430F21x2 consumes approximately 1.54 μW in stand-by mode with 512 bytes of RAM. The static power of a microtask in power-gated mode depends on the size of its RAM. We considered the same static power consumption as that of the MSP430 for a microtask and just scaled it down, since a microtask needs much less global memory (6 bytes on average, c.f., Table III). Hence, average static power consumption of a microtask is around 18.04 nW.

Considering this static power, a time period of 100 ms and using the above expression, we calculated the overall energy saving for the *snd2SPI* microtask over a complete period of task-activation to be 138 x over the MSP430.

5.3 Area Consumption w.r.t. Conventional MCU

We synthesized the *openMSP*-core for 130 nm process technology and estimated the silicon area for the core. The results show that it consumes around 75000 μm^2 whereas the total area of all the microtasks in our case study is around 48000 μm^2 for the same technology. Similarly, the area of the hardware *SM* is only 1710 μm^2. There are 12 microtasks that implement the SPI communication protocol, a low-power MAC protocol, and a part of geographical routing protocol. Since these tasks normally remain fixed after deployment, we do not expect the number and area of these microtasks to blow up. These estimates show that our approach is comparable to the low-power general-purpose MCUs in terms of area consumption, and much more efficient in terms of power and energy consumption.

6. CONCLUSION

In this article, we proposed an original architectural model for optimizing the energy efficiency of WSN node platforms. Our approach, based on the use of concurrent power-gated hardware tasks, is a promising alternative to programmable MCU for WSN nodes with very low power constraints.

The approach was validated experimentally on a set of representative applications, and we show that energy improvements up to two orders of magnitude can be obtained. We also show that it is possible to provide a design flow enabling the automatic synthesis of a complete hardware system from high-level specifications (C+DSL) which

make the approach practical for an embedded system designer with no VLSI design expertise.

This ability has significant benefits compared to WSN-specific controllers such as BlueDot [Raval et al. 2010] or SNAP/LE [Ekanayake et al. 2004], as our flow can be adapted with little effort to new families of process technologies. Another original contribution stems from the combined use of techniques borrowed from the ASIP and high-level synthesis communities, and in the use of a DSL which eases the description and synthesis of the system-level model. Interestingly, our results show that hardware customization can be beneficial in applications which were not considered good targets for hardware acceleration.

In the future, we would like to explore the trade-offs between power efficiency and reprogrammabilty, thanks to the use of reconfigurable hardware within the WSN node. We are also working on bridging our flow to the existing network-level WSN simulator (e.g., WSim and WSNet [INRIA 2010b]) so as to be able to run large-scale experiments involving several WSN nodes.

REFERENCES

AKYILDIZ, I., SU, W., SANKARASUBRAMANIAM, Y., AND CAYIRCI, E. 2002. Wireless sensor networks: A survey. *Comput. Netw. 38*, 4, 393–422.

ATMEL CORPORATION. 2007. ATmega 103L 8-bit AVR low-power microcontroller. Tech. rep.

ATMEL CORPORATION. 2009. ATmega 128L 8-bit AVR low-power MCU. Tech. rep.

BABIGHIAN, P., BENINI, L., MACII, A., AND MACII, E. 2004. Post-layout leakage power minimization based on distributed sleep transistor insertion. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'04)*. ACM, New York, 138–143.

BHATTI, S., CARLSON, J., DAI, H., DENG, J., ROSE, J., SHETH, A., SHUCKER, B., GRUENWALD, C., TORGERSON, A., AND HAN, R. 2005. MANTIS OS: An embedded multithreaded operating system for wireless micro sensor platforms. *Mobile Netw. Appl. 10*, 4.

CONG, J., HAN, G., AND JIANG, W. 2007. Synthesis of an application specific soft multiprocessor system. In *Proceedings of the 15th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '07)*. 99–107.

DUNKELS, A., GRONVALL, B., AND VOIGT, T. 2004. Contiki: A lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN'04)*. IEEE, Washington, D.C., 455–462.

DUTTA, P., GRIMMER, M., ARORA, A., BIBYK, S., AND CULLER, D. 2005. Design of a wireless sensor network platform for detecting rare, random, and ephemeral events. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks (IPSN'05)*. IEEE, Washington, D.C., 497–502.

EKANAYAKE, V., KELLY, IV, C., AND MANOHAR, R. 2004. Anultra low-power processor for sensor networks. *SIGOPS Oper. Syst. Rev. 38*, 5, 27–36.

FIN, A., FUMMI, F., AND PERBELLINI, G. 2001. Soft-cores generation by instruction set analysis. In *Proceedings of the 14th International Symposium on System Synthesis (ISSS'01)*. ACM, New York, 227–232.

FONSECA, R., DUTTA, P., LEVIS, P., AND STOICA, I. 2008. Quanto: Tracking energy in networked embedded systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*. 323–338.

FRASER, C. W., HENRY, R. R., AND PROEBSTING, T. A. 1992. BURG: Fast optimal instruction selection and tree parsing. *SIGPLAN Not. 27*, 4, 68–76.

GORJIARA, B. AND GAJSKI, D. 2008. Automatic architecture refinement techniques for customizing processing elements. In *Proceedings of the 45th ACM/IEEE Conference on Design Automation (DAC'08)*. ACM, New York, 379–384.

GORJIARA, B., RESHADI, M., AND GAJSKI, D. 2006. Generic architecture description for retargetable compilation and synthesis of application-specific pipelined IPs. In *Proceedings of the IEEE International Conference on Computer Design (ICCD'06)*. 356–361.

GUPTA, S., DUTT, N., GUPTA, R., AND NICOLAU, A. 2003. SPARK: A high-level synthesis framework for applying parallelizing compiler transformations. In *Proceedings of the 16th International Conference on VLSI Design (VLSI'03)*. 461–466.

HEMPSTEAD, M., WEI, G.-Y., AND BROOKS, D. 2009. An accelerator-based wireless sensor network processor in 130nm CMOS. In *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'09)*. ACM, New York, 215–222.

HILL, J., SZEWCZYK, R., WOO, A., HOLLAR, S., CULLER, D., AND PISTER, K. 2000. System architecture directions for networked sensors. *SIGPLAN Not. 35*, 11, 93–104.

HU, Z., BUYUKTOSUNOGLU, A., SRINIVASAN, V., ZYUBAN, V., JACOBSON, H., AND BOSE, P. 2004. Microarchitectural techniques for power gating of execution units. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'04)*. 32–37.

INRIA. 2010a. PowWow. Protocol for low power wireless sensor network. Tech. Project.

INRIA. 2010b. WSNet. An event-driven simulator for large scale wireless sensor networks. Tech. Project.

KWONG, J., RAMADASS, Y., VERMA, N., AND CHANDRAKASAN, A. 2009. A 65 nm Sub-Vt Microcontroller with Integrated SRAM and switched capacitor DC-DC converter. *IEEE J. Solid-State Circuits 44*, 1, 115–126.

L'HOURS, L. 2005. Generating efficient custom FPGA soft-cores for control-dominated applications. In *Proceedings of the IEEE 16th International Conference on Application-Specific Systems, Architectures and Processors*. IEEE, Washington, D.C., 127–133.

LEVIS, P. ET AL. 2005. TinyOS: An operating system for sensor networks . In *Ambient Intelligence*. Springer, Berlin.

LIAO, S., DEVADAS, S., KEUTZER, K., AND TJIANG, S. 1995. Instruction selection using binate covering for code size optimization. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'95)*. ACM, New York, 393–399.

LIN, E.-Y., RABAEY, J., AND WOLISZ, A. 2004. Power-efficient rendezvous schemes for dense wireless sensor networks. In *Proceedings of IEEE International Conference on Communications (ICC'04)*. vol. 7., IEEE, Washington, D.C., 3769–3776.

LONG, C. AND HE, L. 2003. Distributed sleep transistor network for power reduction. In *Proceedings of the 40th ACM/IEEE Design Automation Conference (DAC'03)*. ACM, New York, 181–186.

MARTIN, K., WOLINSKI, C., KUCHCINSKI, K., FLOCH, A., AND CHAROT, F. 2009. Constraint-driven instructions selection and application scheduling in the DURASE system. In *Proceedings of the 21st IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP'09)*. IEEE, Washington, D.C., 145–152.

MOHANTY, S. P., RANGANATHAN, N., KOUGIANOS, E., AND PATRA, P. 2008. *Low-Power High-Level Synthesis for Nanoscale CMOS Circuits*. Springer, Berlin.

MYSORE, S., AGRAWAL, B., CHONG, F., AND SHERWOOD, T. 2008. Exploring the processor and ISA design for wireless sensor network applications. In *Proceedings of the 21st IEEE International Conference on VLSI Design (VLSI'08)*. IEEE, Washington, D.C., 59–64.

NAZHANDALI, L., MINUTH, M., AND AUSTIN, T. 2005. SenseBench: Toward an accurate evaluation of sensor network processors. In *Proceedings of the IEEE International Workload Characterization Symposium (IISWC'05)*. IEEE, Washington, D.C., 197–203.

NGUYEN, T.-D., BERDER, O., AND SENTIEYS, O. 2007. Cooperative MIMO schemes optimal selection for wireless sensor networks. In *Proceedings of the 65th IEEE Vehicular Technology Conference (VTC2007-Spring)*. IEEE, Washington, D.C., 85–89.

LPC1111/12/13/14, 32-bit ARM Cortex-M0 Microcontroller. Product Data Sheet.

PASHA, M. A., DERRIEN, S., AND SENTIEYS, O. 2009. Ultra low-power FSM for control oriented applications. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS'09)*. IEEE, Washington, D.C., 1577–1580.

POLASTRE, J., HILL, J., AND CULLER, D. 2004. Versatile low power media access for wireless sensor networks. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys'04)*. ACM, New York, 95–107.

PRADHAN, S. N., KUMAR, M. T., AND CHATTOPADHYAY, S. 2008. Integrated power-gating and state assignment for low power FSM synthesis. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (VLSI'08)*, vol. 0, IEEE, Washington, D.C., 269–274.

RAGHUNATHAN, V., SCHURGERS, C., PARK, S., AND SRIVASTAVA, M. 2002. Energy-aware wireless microsensor networks. *IEEE Signal Process. Mag. 19*, 2, 40–50.

RAVAL, R. K. ET AL. 2010. Low-power TinyOS tuned processor platform for wireless sensor network motes. *ACM Trans. Des. Autom. Electron. Syst. 15*, 3, 1–17.

ROUNDY, S., WRIGHT, P., AND RABAEY, J. 2004. *Energy Scavenging for Wireless Sensor Networks: With Special Focus on Vibrations*. Springer, Berlin.

SCHURGERS, C. AND SRIVASTAVA, M. 2001. Energy efficient routing in wireless sensor networks. In *Proceedings of the IEEE Military Communications Conference (MILCOM'01)*. IEEE, Washington, D.C., 357–361.

SEOK, MINGOO ET AL. 2008. The Phoenix processor: A 30pW platform for sensor applications. In *Proceedings of the IEEE Symposium on VLSI Circuits (VLSI'08)*. IEEE, Washington, D.C., 188–189.

SHEETS, M., BURGHARDT, F., KARALAR, T., AMMER, J., CHEE, Y., AND RABAEY, J. 2006. A power-managed protocol processor for wireless sensor networks. In *Symposium on VLSI Circuits. Digest of Technical Papers*. 212–213.

TAN, C. H. AND ALLEN, J. 1994. Minimization of power in VLSI circuits using transistor sizing, input ordering, and statistical power estimation. In *Proceedings of the International Workshop on Low- Power Design*. 75–80.

TEXAS INSTRUMENTS. 2009. MSP430 User Guide. Tech. rep.

UC BERKELEY. 1999. Tech. Project: Smart dust.

Received July 2010; revised May 2011; accepted July 2011