A UNIFYING INTERFACE ABSTRACTION FOR ACCELERATED COMPUTING IN SENSOR NODES

Srikrishna Iyer, Jingyao Zhang, Yaling Yang and Patrick Schaumont

Virginia Polytechnic Institute and State University
Department of Electrical and Computer Engineering
Blacksburg, Virginia 24060
Email: {skr, jingyao, yyang8, schaum}@vt.edu

ABSTRACT

Hardware-software co-design techniques are very suitable to develop the next generation of sensornet applications, which have high computational demands. By making use of a low-power FPGA, the peak computational performance of a sensor node can be improved without significant degradation of the standby power dissipation. In this contribution, we present a methodology and tool to enable hardware/software codesign for sensor node application development. We present the integration of nesC, a sensornet programming language, with GEZEL, an easy-to-use hardware description language. We describe the hardware/software interface at different levels of abstraction: at the level of the design language, at the level of the co-simulator, and in the hardware implementation. We use a layered, uniform approach that is particularly suited to deal with the heterogeneous interfaces typically found on small embedded processors. We illustrate the strengths of our approach by means of a prototype application: the integration of a hardware-accelerated crypto-application in a nesC application.

1. INTRODUCTION

Sensor nodes, the building blocks of sensornets, consist of wireless embedded platforms fitted with sensors. The flexible part of a sensor node application is typically executed as software on a small embedded microcontroller unit (MCU). During execution, the sensor node software has to juggle a combination of sensor readings, data-packet transmissions, and other concurrency-intensive operations. To help the development of sensor node software, systematic support in the form of tiny, lightweight operating system was developed over a decade ago [1]. Such resource control software remains very popular to this day to support concurrency and, more recently, post-deployment re-programming [2, 3].

Most of the time, of course, a sensor node does nothing and is in sleep mode to conserve battery energy. When it wakes up, it completes its job as quickly as possible, and goes back to sleep. The time required to complete a job is limited by the peak performance that can be delivered by the sensor node components. This applies to computational limits of the embedded processor, as well as to communication-bandwidth limits between peripheral components and the processor.

Our research is motivated by the observation that, under similar resource constraints, parallel implementations in hardware deliver better peak performance than sequential implementations on a small micro-processor [4]. Thus, by adding a Field Programmable Gate Array (FPGA) to the sensor node, and by re-mapping part of the sensor node application onto the FPGA, the sensor node application has a higher peak computational performance.

A good example of the increased computational needs in future sensornet applications may be found in healthcare [5]. Sensor nodes can continuously monitor vital metrics such as blood pressure level, glucose level, heart rate, and so on. This data can be logged or transmitted, but it obviously represents a privacy issue that may call for encryption. Furthermore, a healthcare sensor may need authentication to support the security policy of their environment. All of this results in strong cryptographic requirements, which in turn imply higher computational needs [6].

A possible downside of adding FPGAs to a sensor node is that the resulting node has more resources, all of which increase latent energy consumption. Thus, the sensor node may become less efficient at doing nothing. We believe that this issue is less critical then it appears to be. Recent generations of flash-based FPGA have dedicated support for ultra-low-power standby operation. For example, an Actel IGLOO AGL250 FPGA (250K system gates) has a standby current of 24 μA; a standard alkaline type AAA battery has sufficient energy to power the FPGA in standby for 6.7 years. Thus, although FPGAs may not be suitable for all types of sensornet applications, we believe they are an appropriate choice for those applications whose peak computational load is beyond the capabilities of current MCU-based sensor nodes.

Unfortunately, the current generation of sensor node development environments, such as TinyOS [1] or Contiki [2] do not support multiple programmable components. These

environments support sensor nodes that have a single, central MCU. The sensor application is expressed with a limited form of concurrency, and the hardware abstraction layer is oriented towards fixed components with limited configuration capabilities, such as timers, A/D converters and UARTs.

To implement our vision of an FPGA-enabled sensor node, we need to expand the capabilities of current sensor node design environments. In this paper, we focus on the lack of design support for hardware/software interfaces. This is the first step towards an approach that integrates hardware/-software co-design into the sensor node application design flow.

Our solution integrates nesC, a well-known programming language for sensor node application development [7], with GEZEL, a cycle-based hardware description language [8]. We demonstrate a layered interface between application software and application hardware. It captures the interaction between parallel entities in hardware and software using `send` and `receive` operations. Our communication protocol provides a systematic refinement of communications in MCU software or FPGA hardware, onto typical inter-chip communication interfaces such as SPI and UART. Hence, the sensor node developer can use convenient high-level semantics to express hardware/software communications.

We demonstrate the effect of migrating the compute-intensive encryption part of a sensor node application in nesC into a parallel hardware implementation in GEZEL. Furthermore, we also demonstrate a prototype implementation that combines a micro-controller board and an FPGA board.

The remainder of this paper is structured as follows. In the next section, we discuss background material: a brief review of nesC and GEZEL, and a discussion of related work in the area of hardware abstraction for sensor node applications. Section III presents a layered interface model for the connection of sensor node applications in nesC to sensor node hardware modules in GEZEL. Section IV describes our proof-of-concept implementation with an example, while Section V concludes the paper.

2. BACKGROUND

In this section, we provide a brief description of the language constructs used in our co-design environment for developing sensor node applications. We further discuss the design flow that includes co-simulation, as well as refinement onto sensor node hardware. The integration of the GEZEL kernel with an instruction-set simulator will be the starting point of our layered hardware/software interface, presented in the next section.

2.1. NesC

NesC is a C-like programming language which reflects TinyOS's event-based concurrency model [7]. A nesC application is composed of `components` which are tied together using bi-directional `interfaces`. The design of a component requires a specification, which enlists the interfaces it `provides` and `uses`, as well as an `implementation`, which incorporates the logic tied to those interfaces.

```
ipblock avr1 {
   iptype "atm128core";
   ipparm "exec=spitest";      // AVR binary
}

ipblock avr_spi_port (
   out ss      : ns(1);        // SPI pins
   out sck     : ns(1);
   out mosi    : ns(1);
   in  miso    : ns(1)) {

   iptype "atm128port";        // component type
   ipparm "core=avr1";         // attached to
   ipparm "port=B";            // avr1, port B
   ipparm "pindir=xxxx0111";   // pin configuration
}
```

Listing 1. AVR with SPI interface modeled in GEZEL

NesC distinguishes a `command` from an `event`. A `command` is a synchronous invocation (`call`) of one component's interface by another component. An `event` is an asynchronous call-back (`signal`) from one component to the other. The event interface is used to implement a split-phase mechanism when interacting with slow components. In a split-phase approach, commands are used to initiate an operation (e.g. send one byte through a UART), and events are used to mark their completion (e.g. the UART peripheral indicates completion).

2.2. GEZEL

GEZEL is a hardware description language that captures hardware behavior at the cycle-accurate level, using FSMD (Finite State Machine with Datapath) semantics [8]. GEZEL provides an extensible simulation kernel that simulates the descriptions, as well as a code generator to convert GEZEL descriptions into synthesizable VHDL.

The extensibility of GEZEL makes it particularly suited for our application. The GEZEL kernel has a mechanism to include foreign simulation engines, such as instruction-set simulators. At the language-level, a foreign simulator is captured using an `ipblock` construct, a black-box module with a pre-defined behavior. When simulating a black-box module, the GEZEL kernel invokes the foreign simulator. The co-simulation proceeds in lock-step and ensures that the hardware-software co-simulation remains causal.

For our application, we have integrated the SimulAVR simulator, an open-source, cycle-accurate simulator for the Atmel series of AVR processors. GEZEL instantiates an AVR core as an `ipblock`. The core module only specifies the type

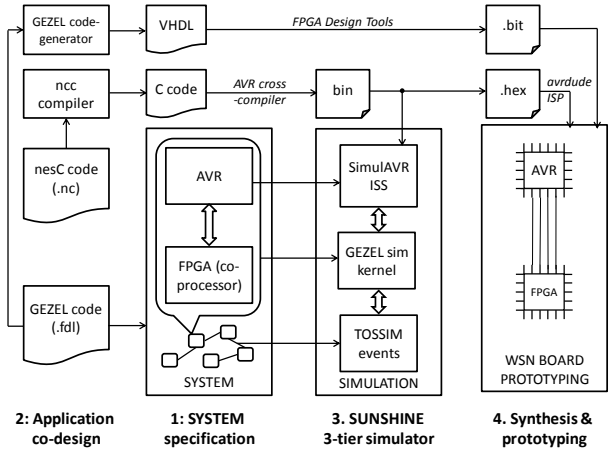

Fig. 1. Co-design environment with GEZEL and nesC code

of MCU and the software application it executes. I/O interfaces, such as SPI, UART, and parallel ports, can be included using additional `ipblock`. Listing 1 shows an example of an AVR core running a program `spitest`, together with an SPI interface. The `avr_spi_port` module shows the actual I/O pins available from the AVR MCU; GEZEL FSMD modules of the hardware can be connected to these pins to obtain an integrated hardware-software model.

2.3. NesC-GEZEL Integration

Figure 1 illustrates the resulting integration of the NesC environment with GEZEL. The overall simulation environment, called SUNSHINE, integrates GEZEL, SimulAVR, and TOSSIM (a sensor network simulator) [9]. In this contribution, however, we focus on the GEZEL-SimulAVR interface and do not consider TOSSIM further. Our target system architecture combines an FPGA and an AVR MCU. The FPGA-AVR interconnect is configurable and can use different interfaces of the AVR (SPI, UART, I^2C).

An application for this architecture is created as a combination of a software program in nesC with a platform description in GEZEL. The nesC program is compiled for AVR using the `ncc` compiler. The system simulator parses the GEZEL description, instantiates an AVR instruction-set simulator, and loads the NesC binary. A cycle-accurate simulation then enables the designer to predict the performance of the resulting system. When the application development is complete, the GEZEL code can be converted into VHDL and synthesized for the FPGA. The resulting bitstream, as well as the nesC binary, can then be downloaded onto an actual sensor node prototype.

The SUNSHINE simulation environment along with the work presented in this paper is available for download at [10].

In the next section, we focus on the design and implementation of a structured communication channel between the MCU software and the FPGA hardware.

3. COMMUNICATION INTERFACE ARCHITECTURE

In this section, we first describe the challenges in building an efficient hardware-software interface for a constrained, microcontroller-oriented environment. We decompose the transmission of a *message* (an arbitrary-length data packet) between software and hardware into three abstraction layers. We then present the implementation of each layer in software as well as hardware.

3.1. Challenges

Our objective is to use the FPGA to accelerate compute-intensive tasks. Such a task, mapped in hardware, will be called an Accelerating Computation Unit (ACU). The ACU assists the MCU at completing the overall system activity in a shorter time.

MCUs offer many different interfaces which have a wide range of capabilities. Besides general purpose input/output pins (GPIO), this includes for example UART (up to 250 Kbps), I^2C (400 Kbps), SPI (1 Mbps), as well as more advanced interfaces including CAN (250 Kbps), Ethernet (10 Mbps), and/or USB (1.5 Mbps). Serial interfaces are most common because they reduce the pin-count of the MCU package. A complete sensornode architecture can be integrated using these interfaces: all chips of a sensornode board (RAMs, ROMs, radio, sensors) are interconnected through a heterogeneous collection of MCU interfaces.

Within this environment, we need to integrate an ACU in such a way as to provide the most flexible platform for the application developer. A first challenge is the heterogeneity of interfaces, which may hamper the portability of applications written for the ACU. Indeed, each time a different hardware interface is selected, the ACU application must be potentially rewritten. A second challenge is that serial hardware interfaces are too low-level, and represent a design-productivity issue when integrating hardware and software.

These two challenges led us to the layered communication architecture, which is described next.

3.2. Layered Channel Model

The objective of our layered communication interface is to simplify the development of applications, for the hardware as well as for the software. Ideally, we aim to unify all possible communication schemes between an MCU and an ACU into a single higher level abstraction, called a `Channel`. We will first explain two interface design decisions before introducing the `Channel` abstraction.

The first design decision is to configure ACU and MCU as Slave and Master devices, respectively across the `Channel`.

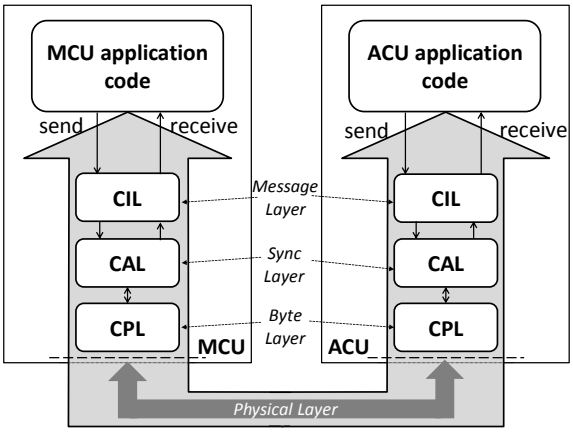

Fig. 2. The unified Channel abstraction and three layers of our communication interface architecture

This implies that communication primitives, regardless of their direction (send/receive), are always initiated by the MCU. When the ACU needs to return a result to the MCU, it will remain idle until instructed by the MCU to return the result.

The second design decision is to use a synchronous model of communication for the `Channel`. This means that communication activities are executed in-line with other computational tasks. Considering the accelerated-computing model, hardware-software communication in polled mode makes more sense than an asynchronous interrupt-driven mode. First, the interrupt-driven model suffers from the overhead of frequent context switching. Next, hardware interrupts are a scarce resource on an MCU, and we choose to reserve them for truly asynchronous events such as messages received from the radio and sensor readings. Finally, for many applications, the computation time of the ACU is known, or it can be accurately measured using our co-simulator. Therefore, the MCU can select an optimal polling rate, and at the same time still benefit from sleep mode while the ACU is performing long computations.

Figure 2 illustrates the `Channel` abstraction between MCU and ACU. The layered abstraction ensures that the messages are correctly delivered at the other side for use, irrespective of the interface being used. Each abstraction layer of the `Channel` is implemented inside the MCU as well as in the ACU. The topmost layer is independent of the communication interface used to connect MCU and ACU. The top layer offers simple send and receive primitives that support transmission of variable-length messages.

As we move towards the bottom of the layered interface, the intricacies of the underlying communication peripheral become increasingly visible. The lowest layer of the `Channel` corresponds to the physical wiring between the MCU and ACU.

In this perspective, our communication stack is not limited to the interconnection of hardware and software; the ACU may as well be another MCU connected as a slave device. In the next subsections, we briefly summarize the objective of each layer. After that, we describe their implementation in software (nesC) as well as in hardware (GEZEL).

3.2.1. Communication interface-independent layer (CIL)

The **CIL** provides interface-independent functionality to `send` and `receive` messages to the main application code; hence it is also known as the *message layer*. The application code on the MCU and ACU will remain the same regardless of the underlying implementation of the `Channel`. This greatly simplifies application design.

The main function of this layer is to perform *message segmentation* in the `send` direction and *message aggregation* in the `receive` direction. Messages sent by the application code are split into equal-length packets, before being forwarded to the next lower layer. Similarly, the packets received from the middle layer are aggregated into complete meaningful messages which can be used by the application code above.

3.2.2. Communication adaptation layer (CAL)

CAL is also called *packet-level synchronization and multiplexing layer*. It has two functions.

The first function is to provide logical synchronization. The MCU and ACU operate in parallel and are asynchronous with respect to each other. Hence, this layer synchronizes the MCU and ACU at the packet level, so that correctness of the flow of data is maintained.

The transfer of arbitrarily sized messages is unpredictable and prone to dropped bytes. This can potentially lead to a deadlock. Hence, we designed CAL to handle fixed length packets along with a handshake mechanism. Before any packet transfer, the MCU sends a request byte to the ACU. The request indicates the intended direction of communication. The ACU sends back a response byte indicating whether it can be completed. After the handshake, the entire packet transfer is done in burst. This in-band approach to handshaking comes at the cost of two extra byte transfers per packet. The alternative is to perform out-of-band handshaking using dedicated GPIO pins on the MCU. This would lessen the delay overhead, but it increases the signaling complexity and consumes a precious MCU resource. Our solution finds motivation from simplicity of implementation and minimal GPIO resource usage.

The second function is to distinguish between upstream and downstream data. Some, though not all, serial interfaces for MCUs can only operate in a bi-directional mode: for each bit sent, another bit is received. Since the upstream and downstream communications can be logically independent, additional de-multiplexing must be provided.

3.2.3. Communication presentation/peripheral layer (CPL)

CPL forms the lowest *byte layer* in our abstraction, positioned right above the communication hardware peripheral. This abstraction 'presents' the hardware and provides easy to use functionality to access the hardware via a byte interface. The communication peripheral does the task of converting data stream from the physical layer to byte-stream onto the byte interface and vice versa. In case this interface would be absent, then the CPL layer offers the possibility to emulate this functionality.

3.3. NesC layers

NesC implements the hardware abstraction model of TinyOS, after which our layering is closely modeled [11]. NesC distinguishes three layers of hardware abstraction: a hardware-independent layer (HIL), a hardware-abstraction layer (HAL), and a hardware-presentation layer (HPL). The main goal of these layers is to expose the functionality of a fixed hardware which is independent of the sensor node platform. Our implementation however is aimed at providing a method to communicate with an external, re-configurable ACU and an interface-independent way of accessing it.

It is straightforward to express the layers of out communication abstraction for each interface in NesC. However. we found that the highest layer, HIL, still exposes information about the kind of communication peripheral used. In order to turn HIL interfaces into a completely generic CIL, we wrap the HIL implementation of the communication peripherals with a new component called `ChannelC`.

The `ChannelC` component provides an interface to the main application called `MessageStream` as shown in Listing 2. The main application uses this interface for message transfers. `ChannelC` component simply passes messages between the internal HIL and the `MessageStream`.

The only purpose of `ChannelC` is to isolate the main application from the device-oriented HIL semantics. Applications thus developed will not only be platform-independent, but also, the actual hardware peripheral used for communication will be transparent leading to a truly generic message-passing scheme.

3.4. GEZEL layers

The application part mapped onto the ACU has a similar layered interface. Figure 3 represents the interfaces provided to the GEZEL application.

The uppermost **CIL** layer is the only layer that stores data. It uses a FIFO in each of the transmit and receive directions to hold one packet. The application hardware in the ACU will access these FIFO's through a streaming interface, shown on the left side of Figure 3. The FIFO's need to be able to hold at least one packet, but they may be over-dimensioned depending on the application.

```
module ChannelM {
  provides interface MessageStream;
  uses interface MessageStream as ChannelStream;
  ...
}

configuration ChannelC {
  provides interface MessageStream;

implementation {
  components ChannelM, HilUartC as UART; // UART's
       HIL component
  MessageStream = ChannelM.MessageStream;
  UART.MessageStream -> ChannelM.ChannelStream;
  ...
}

interface MessageStream {
  command result_t send(uint8_t *msg, uint16_t
       length);
  event void sendDone(uint8_t *msg, uint16_t
       length, error_t error);
  command result_t receive(uint8_t *msg, uint16_t
       length);
  event void receiveDone(uint8_t *msg, uint16_t
       length, error_t error);
}
```

Listing 2. CIL interfaces to the main application

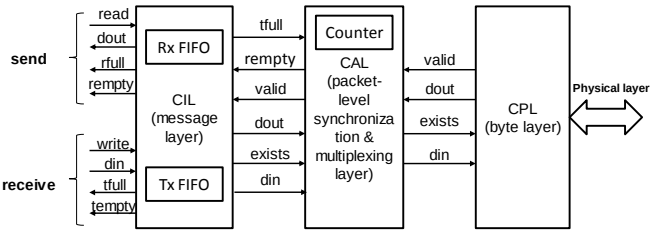

Fig. 3. Layers in hardware as GEZEL modules and their interfacing

The **CAL** layer in hardware inherits the same streaming interface as the CIL layer. CAL maintains synchronization with the help of a finite state machine (FSM) and a byte counter. The FSM intercepts the request byte from the MCU. In response, it sends back a response byte by checking the FIFO signals as shown (symbolically) in Figure 4. Based on the request, the upstream, downstream or both are enabled. The byte counter tracks the correct number of bytes per packet. As soon as the set packet size is reached, the CAL resets to the idle state.

The reconfigurable nature of FPGA obviates the need for simple dedicated hardware such as communication interfaces. Hence, in hardware, we implement an RTL description of the communication peripheral itself in the CPL layer. It converts the raw data stream on the physical layer into a byte stream for the CAL layer and vice versa. A standard set of streaming interfaces connects CPL with CAL as shown in Figure 3.

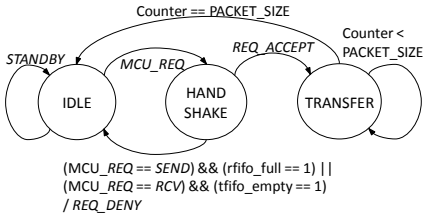

Fig. 4. FSM in CAL layer which performs synchronization

All abstraction layers in hardware (FPGA) are instantiated in the GEZEL application using the `ipblock` mechanism. This allows a user to quickly assemble the required communication stack in between a user-defined hardware application and a microprocessor interface (SPI, UART, I^2C).

In the next section, we will illustrate the layered mechanism by means of a prototype application.

4. PROTOTYPE AND RESULTS

In this section, we present a prototype application which performs 128-bit AES encryption. We first show how it is co-designed in nesC and GEZEL using our layered communication architecture. Then, we implement it on our prototype sensor board which connects an MCU with an FPGA.

The MCU and the FPGA are connected using an SPI interface. The AES encryption algorithm is mapped entirely in hardware. The nesC code on the MCU prepares 16 bytes of text and 16 bytes of key. Using the `MessageStream.send` command, the text and the key are transferred. The text and the key packets travel through the layers finally filling up the rxFIFO within CIL in the FPGA. The AES co-processor is connected to the `send` and `receive` interfaces of CIL. It reads out the text and the key from rxFIFO, performs encryption and fills the txFIFO with the encryption result. The MCU requests for the result by calling `MessageStream.receive` command. The txFIFO is read out and the encrypted text flows from the CIL in the FPGA through the layers reaching the application code on the MCU.

Table 1. AES computational costs

Implementation	Cycles	Speedup
MCU (reference software)	19364	1
FPGA (w/o comm overhead)	10	1936.4
FPGA (with comm overhead)	5804	**3.3**

Table 1 provides performance numbers for the AES encryption task in comparison to a reference software implementation, obtained from the simulation stage. Despite the serial communication overhead, we observe that the AES encryption algorithm as a task ported in hardware performs 3.3 times faster than software.

After simulation, we map the application in our prototype platform. Our prototype platform consists of a BDMicro board with AVR ATMega128 MCU and a Spartan-3E FPGA board. We use the same nesC binary used in our previous simulation step to configure the AVR MCU. The VHDL generated from the GEZEL code is synthesized into a bitstream using Xilinx ISE tool.

5. CONCLUSIONS

We presented a three layered abstraction model to integrate hardware and software on constrained sensor node platforms. In contrast with the existing environments we support a flexible sensor node topology thanks to the use of hardware description. We are currently extending our communication library to support other MCU interfaces and we plan to develop a sensor node board for our design environment which will pair an MCU with a low-power FPGA. Our future work will focus on performing an accurate assessment of energy savings on our sensor board.

6. REFERENCES

[1] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister, "System architecture directions for networked sensors," *SIGPLAN Not.*, vol. 35, pp. 93–104, November 2000.

[2] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, Washington, DC, USA, 2004, LCN '04, pp. 455–462, IEEE Computer Society.

[3] Athanassios Boulis, Chih-Chieh Han, and Mani B. Srivastava, "Design and implementation of a framework for efficient and programmable sensor networks," in *Proceedings of the 1st international conference on Mobile systems, applications and services*, New York, NY, USA, 2003, MobiSys '03, pp. 187–200, ACM.

[4] J. Rabaey, *Low Power Design Essentials*, Springer, 2009.

[5] JeongGil Ko, Chenyang Lu, M.B. Srivastava, J.A. Stankovic, A. Terzis, and M. Welsh, "Wireless sensor networks for healthcare," *Proceedings of the IEEE*, vol. 98, no. 11, pp. 1947 –1960, nov. 2010.

[6] Srivaths Ravi, Anand Raghunathan, Paul Kocher, and Sunil Hattangady, "Security in embedded systems: Design challenges," *ACM Trans. Embed. Comput. Syst.*, vol. 3, pp. 461–491, August 2004.

[7] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler, "The nesc language: A holistic approach to networked embedded systems," *SIGPLAN Not.*, vol. 38, pp. 1–11, May 2003.

[8] P. Schaumont, *A Practical Introduction to Hardware-Software Codesign*, Springer, 2010.

[9] J. Zhang, Y. Tang, S. Hirve, P. Schaumont, and Y. Yang, "A software-hardware emulator for sensor networks," in *Proc. 8th IEEE Conference on Sensor, Mesh and Ad Hoc Communications and Networks*, 2011.

[10] J. Zhang, S. Iyer, S. Hirve, Y. Yang, and P. Schaumont, "Sunshine: A software-hardware emulator for sensor networks," Website, `http://rijndael.ece.vt.edu/sunshine/index.html`.

[11] V. Handziski, J. Polastre, J.-H. Hauer, C. Sharp, A. Wolisz, and D. Culler, "Flexible hardware abstraction for wireless sensor networks," in *Wireless Sensor Networks, 2005. Proceeedings of the Second European Workshop on*, jan.-2 feb. 2005, pp. 145 – 157.