Hardware-Software Co-design for Heterogeneous Multiprocessor Sensor Nodes

Jingyao Zhang, Srikrishna Iyer, Xiangwei Zheng, Patrick Schaumont, and Yaling Yang
Department of Electrical and Computer Engineering
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24060
Email: {jingyao, skr, xiangwei, schaum, yyang8}@vt.edu

***Abstract*—To meet the needs of innovative sensor network applications, sensor nodes have long evolved from underpowered single microcontroller designs into complex architectures that accommodate multiple processors and Field Programmable Gate Arrays (FPGAs). We address the problem of conceiving and implementing programs for such sensor node architectures. We rely on a universal, layered hardware/software interface that provides seamless interconnection between tasks running on a micro-controller and tasks running on a FPGA. Resource sharing is handled transparently through a shared-bus communication architecture. We demonstrate our methodology, through a heterogeneous sensor node simulator called SUNSHINE [1] for an application running the sensor nodes. We validate SUNSHINE by demonstrating the applications on a multiprocessor sensor node's testbed, which consists of a FPGA, a microcontroller and a radio front-end.**

I. Introduction

Wireless sensor network applications have gained attraction in many fields, such as health care, environment monitoring, industrial measurements [2]. Most of these applications require sensor nodes to sense the environment and to relay the sensing data to gateways via other sensor nodes. To avoid packets congestion in communication channel and save network bandwidth in transmission, it is often desirable for sensor nodes to preprocess the sensing information before transmission. In addition, sensor nodes may need to execute additional complex communication tasks, such as maintaining and calculating routing table, encrypting/decrypting packets, and compressing packets. All these computation-intensive tasks may happen concurrently and, hence, place a heavy burden on the processing unit of a sensor node. Currently, the processing unit is usually like a microcontroller (MCU), such as Atmega128 (on MICA series motes [3]), MSP430 (on telosB [4]), and ARM (on IMote2 [5]). When processing concurrent computation-intensive tasks in a busy network, a MCU often becomes a bottleneck for the execution speed due to its sequential execution nature. Such inadequacy in processing capability would degrade sensor networks' performance in many aspects, such as increasing network's packet loss rate and time delay for task processing. Therefore, increasing execution capability of sensor nodes is a key factor in enhancing performance of sensor networks.

One approach to increase the execution capability is to add a coprocessor to a single processor sensor node. Several work [6] [7] [8] have shown that adding a coprocessor is beneficial for sensor nodes' performance. However, implementing applications for these nodes from scratch is non-trivial because writing applications from application level down to the lower hardware driver level takes many efforts and is prone to developmental bugs.

In this paper, a methodology is proposed to drastically reduce the efforts of programming applications for multiprocessor sensor nodes. The major contributions are summarized as follows.

1) We provide a design methodology to facilitate application programming for multiprocessor sensor nodes handling computation-intensive tasks in wireless networks. The methodology includes a three-layered architecture, and application interfaces for nodes' processing units. The methodology can support different processing units, such as MCUs, and FPGAs, to serve as either processors or coprocessors.
2) We use a sensor network emulator SUNSHINE [1] to simulate multiprocessor sensor nodes' behaviors in wireless networks to demonstrate the effectiveness of the design methodology.
3) We setup a testbed to demonstrate our methodology.

The rest of the paper is organized as follows. Section II reviews related work. Section III presents problem statements of our work. Section IV describes methodology's framework for multiprocessors wireless sensor nodes. Section V presents application interfaces for processing units according to the methodology for multiprocessor sensor nodes. Section VI introduces resource sharing technique among communication entities. Section VII shows simulation and testbed results. Section VIII concludes the paper.

II. Related Work

So far, no methodologies have been developed for designing wireless sensor nodes with multiprocessors. In [9], a reusable hardware/software interface between a processor (MCU) and a coprocessor (FPGA) is demonstrated. However, [9] has several limitations. First, [9] focuses on the simulation for the processor (MCU) with coprocessor (FPGA). However, our design flow for multiprocessor nodes' applications between simulation and actual hardware development is different. The

details of the design flow are introduced in Section V. As a result, more development efforts and actual hardware evaluations are needed. In this paper, we run and evaluate applications on both single processor and multiprocessor sensor nodes using our design methodology. In addition, [9] does not consider wireless sensor network environment.

V. Handziski et al. [10] present TinyOS [11] three-layered hardware-abstraction architecture for wireless sensor network design. The architecture separates sensor nodes' drivers to three distinct layers: Hardware Interface Layer (HIL), Hardware Adaption Layer (HAL), and Hardware Presentation Layer (HPL). HIL is the topmost layer that provides hardware-independent interfaces for programming sensor nodes. HAL is the second layer that represents "platform-specific" driver. As the intermediate layer between HIL and HPL, HAL provides general platform interfaces for HIL while using the interfaces of device drivers provided by HPL. HAL serves as a bridge between actual hardware driver and general purpose (hardware-independent) programming interfaces. It translates the upper layer's commands to hardware driver at compile time. Meanwhile, it responds to hardware requests (interrupts for example) at run time. HPL, which is responsible for device drivers of specific components, deals directly with hardware components. As mentioned above, HPL encapsulates hardware drivers and provides general components' interfaces to its upper layer HAL. Using three-layered architecture framework prevents programmers to deal directly with hardware drivers.

Even though [10] provides a practical architecture for designing sensor network applications, it only considers single processor (MCU) sensor nodes. Our work provides a methodology for application designs on multiprocessor sensor nodes.

CoMOS [8], an operating system for programming sensor nodes equipped with multiple and heterogeneous processors, is implemented to support programming the coexistence of ARM processor, MSP430 processor and wireless transceivers on a platform. However, CoMOS has several limitations. First, it only supports programming ARM7 and MSP430 processors. It cannot fit in a general multiprocessor platform with different processing types. Furthermore, CoMOS does not support methods for programming FPGA processors. Since both ARM and MSP430 processors run applications in serial, their programming schemes are similar. Both of them can use C language to program. However, FPGA, an integrated circuit, runs tasks in parallel and is configured via logic blocks to execute relevant applications. Hardware programming language such as VHDL, Verilog or GEZEL [12] is needed to program FPGAs. Hence, the programming scheme on FPGA is totally different from programming scheme on software related processors such as ARM, and MSP430.

Our methodology, which supports programming both software related and hardware related processors on a platform, is provided to solve this limitation.

III. Problem Statements

To have an intuitive illustration for multiprocessor sensor nodes, an example of a multiprocessor sensor node's functional blocks is provided in Fig. 1. To easily control radio and other peripherals, the processor is usually a MCU. The coprocessor can be either a MCU or a FPGA according to the requirements of different network applications. A communication bus is connected between processor and coprocessor to carry out their mutual communications. Since both processor and coprocessor have their own clock systems, the two units run independently at different clock frequency domains. Consequently, a handshake communication protocol should be provided to synchronize the two processing units before exchanging packets between each other. As shown in the figure, the radio on the sensor node is also connected and controlled by the processor via the communication bus. Therefore, the processor needs to make resource arbitration between the radio and the coprocessor. In addition, both processing units have their own program interfaces so that different software binaries can be loaded on the corresponding processors. The binaries can be stored in their own memories (RAM or flash). Each processing unit also has I/O ports to connect to its peripherals, such as LEDs, and sensors.

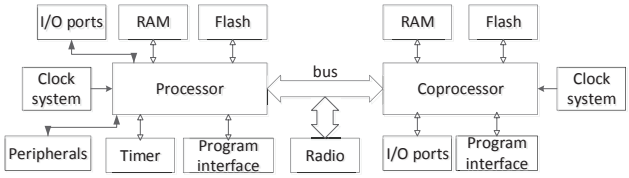

Fig. 1. An Example of A Multiprocessor Sensor Node's Functional Blocks

Based on the discussions above, programming such multiprocessor nodes' applications is non-trivial. To solve this problem, we propose our methodology to reduce efforts for programming multiprocessor nodes' applications.

IV. Methodology's Framework

In this section, we discuss the three-layered architecture of our methodology's framework for multiprocessor sensor nodes. The objective of designing the layered architecture is to provide flexibility and modularity for multiprocessor nodes' software drivers.

Each component, such as processor, radio, LEDs and other peripherals, on the sensor node has its corresponding three-layered architecture. For multiprocessor sensor nodes, the drivers for radio and processor's peripherals follow TinyOS' three-layered architecture [10]: Hardware Presentation Layer (HPL), Hardware Adaption Layer (HAL), and Hardware Interface Layer (HIL). The communication between processor and coprocessor of sensor node should follow our architecture design which also includes three layers: Channel Presentation Layer (CPL), Channel Abstraction Layer (CAL) and Channel Interface Layer(CIL). The architecture is shown in Fig. 2.

The bottom layer CPL directly interacts with the actual sensor node's communication bus, as well as provides software interfaces to its upper layer, CAL. Specifically, CPL provides physical-level drivers of standard communication protocols, such as SPI, UART, and parallel. CPL takes care of hardware

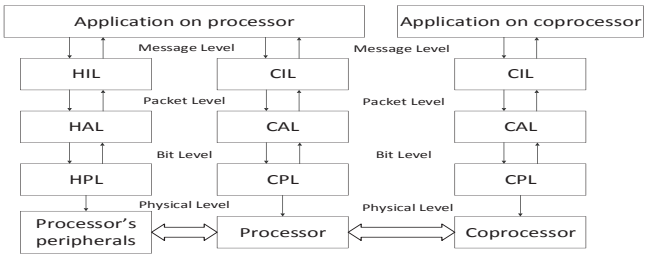

Fig. 2. Three-layered Architecture for Multiprocessor Sensor Nodes

pins' connections among one communication master and one/-multiple communication slaves so that processor, coprocessor, and radio can interact with each other. CPL layer passes all the packets received from other entities via the communication bus up to CAL layer. CPL layer can also send data passed from CAL layer to other entities via the communication bus.

The middle layer CAL is in charge of initiating and terminating communications between processor and coprocessor based on a two-way handshake protocol. The two-way handshake scheme is implemented in CAL layer. To start communicating with the other processing unit (either processor or coprocessor), one processing unit (unit A) sends out a request message through the communication bus. After getting the request message, if the other processing unit (unit B) is ready to start communication, it sends back an acknowledgement packet. Otherwise, unit B keeps executing its own task and ignores the request. Upon sending out the request message, unit A starts a timeout timer and waits for the acknowledgement packet from unit B. If unit A gets the acknowledgement packet within the timeout, the communication handshake succeeds. Unit A then starts exchanging packets with unit B. If no acknowledgement packet is received within the timeout, unit A retransmits the request message to unit B. After packets exchanging between the two processing units, unit A sends a finish message to unit B to release the processing unit from executing the communication tasks. Once the packet exchanging process starts, CAL layer passes all the received packets to CIL layer.

The upmost layer CIL provides interfaces for network applications running on processors/coprocessors. CILs of both processors and coprocessors provide platform independent interfaces. The interfaces provided by HIL for different network applications can be used for different hardware platforms. To be specific, once handshake succeeds, CIL layer gets packets from CAL layer, and relays the packets up to network applications.

Based on the three-layered architecture, interactions between processor and coprocessor are hidden to application programmers so that programmers only need to consider the design of the application itself. Programmers do not need to consider the nature of processors/coprocessors when executing interactions.

In addition, from the hardware drivers' development perspective, for sensor nodes using the same hardware configurations, the implementations of the three layers do not vary for different applications. For sensor nodes using different communication protocols, only CPL layer needs to be modified. This reuse of code consequently enhances the reliability of software drivers for multiprocessor sensor nodes. Also, the distinct layered architecture makes the software drivers flexible.

V. Application Interfaces

The architecture of the methodology's framework introduced in Section IV is implemented as layered functional blocks. The implementation includes interfaces for applications over FPGA coprocessors and interfaces for applications over MCU processors and coprocessors. In the following, we discuss the design details of these application interfaces.

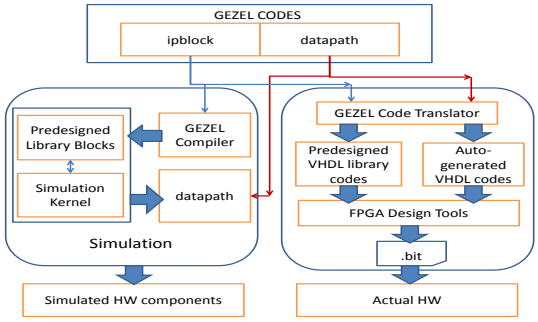

Fig. 3. GEZEL's Design Flow

A. GEZEL-based Application Interfaces for FPGA Coprocessors

Based on the layered architecture, we design application interfaces for FPGA coprocessors using GEZEL [12].

1) GEZEL Introduction: GEZEL is a language that can be used to program FPGAs. It includes a simulation kernel and a cycle-accurate hardware description language. GEZEL's design flow is shown in Fig. 3. GEZEL supports two ways to describe functional modules: ipblock and datapath. An ipblock is a blackbox where the detailed functions of a module are implemented via predesigned library blocks written in other languages, such as VHDL. The datapath, on the other hand, describes the detailed internal activities of a module down to register transfer level using the native GEZEL language. In simulation, the simulation kernel links ipblocks used in the codes to their corresponding library blocks through GEZEL compiler. When running simulation, the simulation kernel together with the library blocks interprets datapath at cycle level. Based on this scheme, the hardware components' behaviors can be accurately emulated. For implementation on actual hardware, the GEZEL code translator can translate GEZEL codes to VHDL codes. Specifically, via GEZEL code translator, different ipblocks are linked to corresponding predesigned VHDL codes, while datapths are translated to auto-generated VHDL codes. Using corresponding FPGA design tools, the generated VHDL codes are then compiled to binaries that can be loaded onto actual FPGAs.

One advantage of writing applications in GEZEL is that the applications can be simulated in network environment using SUNSHINE, a cycle-level accurate simulator for sensor networks. Applications written in GEZEL, hence, can be quickly and accurately evaluated even without actual hardware platforms. In addition, the VHDL codes generated by GEZEL code translator can be synthesized to binary images which can be loaded onto actual hardware. Thus, to minimize the time and cost for design and deployment for wireless sensor network applications, it is desirable to implement multiprocessor sensor nodes' applications in GEZEL. Therefore, we provide an interface for developing coprocessor's applications using GEZEL language.

2) Application Interfaces for FPGA processors: While using GEZEL to program FPGA coprocessors saves development time, GEZEL-generated VHDL codes may not be as efficient as directly designed VHDL codes. Due to the restricted resources of sensor nodes, this efficiency issue cannot be ignored. To solve this challenge and balance the tradeoff between design efforts and code efficiency, we leverage the following features of GEZEL to implement our layered architecture framework.

To generate efficient implementation codes for FPGA coprocessor, we let applications be written as datapaths using GEZEL's native language, while we build our three-layered architecture framework using GEZEL ipblocks that are linked to efficient VHDL libraries provided by us. When compiling applications, GEZEL code translator translates the application itself, which is written in datapath, into VHDL codes and then links the ipblock-based three-layered architecture referenced by the application to the corresponding VHDL programs predesigned by us. Based on this mechanism, application design efforts are minimized, while the application efficiency for FPGA coprocessors is improved.

Fig. 4(a) shows the application interfaces for a FPGA-based coprocessor. The application uses ipblocks of our three-layered architecture, a.k.a., CPL, CAL and CIL. The application itself is programmed as a datapath inside the HW_APP component. Interactions between each layer are achieved via each layer's corresponding input/output signals, such as "valid", "din", and "ack", as shown in the figure. Based on these application interfaces, developers only need to focus on implementing the computation-intensive tasks of network applications, because the communication bus functionalities are already implemented inside CPL, CAL and CIL ipblocks. This separation of implementation methods of application interfaces ensures a good balance between easy-development and code efficiency.

B. Application Interfaces for MCUs

The CIL interface contains four commands, init(), send(), recv() and release(). Command init() is used to initialize packet transmission protocol. Commands send() and recv() are in charge of sending and receiving a packet via the communication bus between processor and coprocessor. After packets exchange, command release() should be called to release the communication process. This CIL interface can be

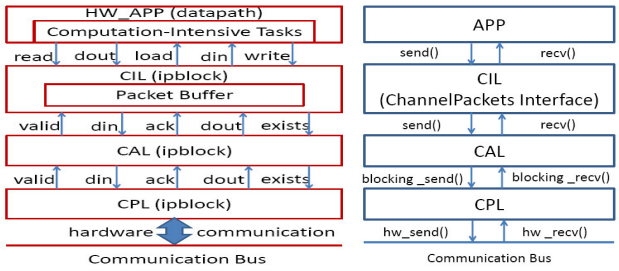

Fig. 4. (a) Application Interfaces for FPGA Coprocessors. (b) Examples of Application Interfaces for MCUs

combined with other TinyOS interfaces to implement sensor network applications.

Software codes for CAL layer implement the communication handshake protocol described in Section IV. Codes for CPL layer implement communication drivers for the specified hardware. Different from TinyOS HPL communication bus drivers that only contain one communication slave, software codes in CPL layer consider multiple communication slaves because both the coprocessor and the radio are communication slaves for the processor. Codes for CAL and CPL layers are hidden to network applications. It is the compiler's job in TinyOS to compile the network applications together with the three-layered codes to software binaries that can be loaded to actual MCUs. Based on this framework, different MCUs can be served as processors/coprocessors with ease.

To provide an intuitive illustration for MCUs' application interfaces, two interfaces: "send()" and "recv()" are shown in Fig. 4(b) as examples. If a network application (APP) needs to send out packets to other communication entities via the communication bus, it only needs to issue a "send()" command via our designed "ChannelPackets" interface in CIL layer. The command is translated to "blocking_send()" in CAL layer which takes care of the handshake mechanism between communication entities. Then, the command is passed to CPL layer as "hw_send()" that directly interacts with the actual communication bus. The "recv()" command follows the same procedure and layered architecture. The application adopts "recv()" command in "ChannelPackets" interface. When receiving packets from the communication bus, the received packets pass through interfaces of the three layers to topmost network applications so that the application can read the data without concerning lower levels' working mechanisms.

VI. Resource Sharing

Upon designing application interfaces for different processing units, resource arbitration is proposed to facilitate interactions among processor, coprocessor and radio. We leverage the resource arbiter of TinyOS to make processor, coprocessor and radio work coordinately via communication bus. Since radio and coprocessor of a multiprocessor sensor node share the same communication bus with the processor, the processor needs to make arbitrations between the two components when

they need to use the communication bus. We provide an arbitration scheme as shown in Fig. 5 to control resource assignments between different units.

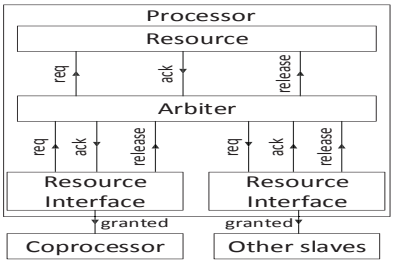

Fig. 5. Resource Arbitration

For each component that wants to access a shared resource of a processor, such as SPI communication bus, the processor needs to instance a resource interface. Before using the shared resource, a component's resource interface sends a request command to the arbiter. The arbiter tracks whether the resource is in use. If the resource is available to use, the arbiter issues an acknowledgment command to the requested resource interface. The resource interface then allows the component to access the resource. Once getting the granted information, the component occupies the resource. Otherwise, the resource interface needs to wait some time and then sends the request command out again to the arbiter. After using the resource, the resource interface should send a release command to the arbiter to release the resource so that other components can access the resource.

This scheme helps the processor arbitrate the shared resource to different hardware components so that the resource can be efficiently used. This scheme is especially suitable for resource-constrained sensor nodes.

VII. Evaluation

Experiments for evaluating our multiprocessor nodes' design methodology are provided through the network simulator SUNSHINE and a multiprocessor sensor node testbed.

A. Development Efforts

We first evaluate a multiprocessor node's application which consists of a pure three-layered framework. In the application, MCU first sends a 16 bytes' packet to FPGA. Once receiving the whole packet, FPGA sends the packet back to MCU. The communication process is achieved by our designed three-layered framework.

Using our framework, around 180 lines' codes are needed to program MCU processor. However, around 400 lines are needed if developers directly write applications for MCU processor because the developers need to write both specific hardware drivers and applications running on the sensor nodes. Table I compares development efforts between developing the application for FPGA coprocessor using our methodology and directly writing FPGA codes without using our methodology. Using our methodology, around 18 lines' codes for CPL layer, 20 lines' codes for CAL layer, 44 lines' codes for CIL layer, and 28 line's codes for FIFOs in CIL layer are needed. As a result, only 110 lines' codes are needed to use our methodology's interface at FPGA side. However, around 800 lines' codes must be provided if developers prefer directly programming FPGA applications. In addition, developers do not need to worry much about the low level hardware components' interactions when programming applications for multiprocessor sensor nodes using our framework.

B. Simulation Experiment

We set up a tree network in a simulation as shown in Fig. 6. In this setting, we use TDMA scheme to assign each leaf node (node 5 to node 10) a time slot to process tasks and send one packet to their parents (node 2, 3, 4) respectively. After receiving packets from their children, the parent nodes forward the packets to the root node 1. In the experiment, we let the leaf nodes process the AES-128 encryption task before sending the encrypted packet out. The time slots were properly set to avoid packet collision as well as to maximize the throughput. We set the leaf nodes (5 to 10) as multiprocessor nodes. The root node 1 receives the leaf nodes' packets in 31.65ms.

As can be inferred from the results, the design methodology works well in heterogeneous network environments.

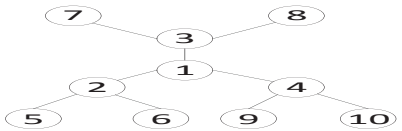

Fig. 6. Tree network topology

To validate the design methodology, CubeHash-512 [15], a sensor node's computation-intensive application, is used in this experiment. The process for the node to execute the application is described as follows. The processor first sends the data to the coprocessor. After executing the CubeHash function on the received data, the coprocessor sends the results back to the processor. We let the multiprocessor node execute the task before sending out packets to wireless channel. Since the time for sending the same size packets out is fixed, we only consider sensor nodes' execution time for computation-intensive tasks.

In SUNSHINE simulator, CubeHash-512 is configured to run on a multiprocessor sensor node. The simulation process is summarized as follows. We first write network applications for multiprocessor sensor nodes and then generated three-layered software codes for MCUs using TinyOS compiler, as well as codes for FPGAs using GEZEL code translator. Then, the codes are compiled to binary images. Those binary images are simulated in SUNSHINE. The simulation time of executing CubeHash-512 is $562.9\mu s$, which is close to the actual execution result on real multi-processor sensor platform as shown in the next section.

C. Testbed Evaluation

Since designing and validating new PCB boards takes time, to minimize the development period and to save cost, it is

TABLE I
COMPARISON OF DEVELOPMENT EFFORTS BETWEEN OUR METHODOLOGY AND DIRECT DEVELOPMENT

Number of Lines' Codes for a FPGA coprocessor	Our Methodology	Direct Development
CPL layer	18	171
CAL layer	20	226
CIL layer	44	136
2 FIFOs in CIL layer	14 * 2 = 28	156 * 2 = 312
Knowledge Required From Programmers	High level specification of node's architecture	FPGA, MCU and radio's driver experience

common to first use demonstration boards to evaluate the software codes and hardware architecture. The PCB boards should be designed and implemented after extensive experimental evaluations. Therefore, in this paper, we connected two demonstration boards (TI CC2420DBK [13], and Xilinx Spartan-3E FPGA boards [14]) to serve as a mulitprocessor sensor node. While real multiprocessor sensor nodes will have a much compact size and lower energy consumption than our demonstration-board-based prototypes, the prototypes have the same architecture and functionality as real multiprocessor sensor nodes. Therefore, these boards can be applied to validate our framework design. Fig. 7 shows the sensor network testbed. The network is composed of a multiprocessor sensor node and a single processor sensor node (MICAz).

On actual sensor node's testbed, MCU first sends a packet to FPGA. After encrypting the packet using CubeHash-512, the FPGA sends the encrypted packet back to MCU. After receiving the packet, MCU sends the packet to the other sensor node (MICAz) via CC2420 radio. Once receiving the packet, MICAz turns on one LED.

we use the same CubeHash-512 algorithm ran in simulator. The testbed's evaluation process is similar to the process in simulation. The only difference is that the compiled binary images are downloaded to actual hardware instead of being simulated in the simulator. We use oscilloscope to record the execution time of executing the computation-intensive task.

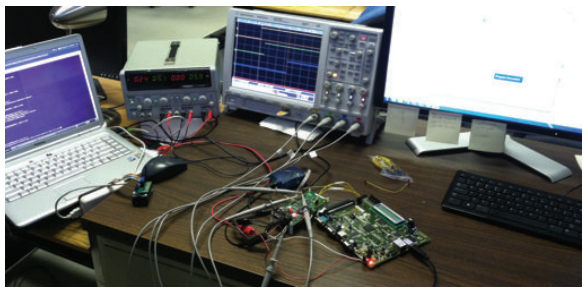

Fig. 7. Testbed for Multiprocessor Node with a MCU as Processor and a FPGA as Coprocessor

The multiprocessor sensor node takes $530.72\mu s$ to execute the CubeHash-512 application. The execution time is close to the simulation results. The capability of being able to evaluate applications in simulation eliminates the need for actual hardware platforms. This makes developers possible to quickly evaluate various application designs over different potential hardware platforms using a pure simulation-based method.

VIII. CONCLUSION

A design methodology for programming applications for multiprocessor sensor nodes to deal with computation-intensive tasks is provided. In detail, we first provide three-layered architecture for multiprocessor sensor nodes. We then implement application interfaces according to the methodology for programming multiprocessor sensor nodes with ease. We use simulation and testbed to demonstrate the effectiveness of our methodology.

ACKNOWLEDGMENT

This work is supported by National Science Foundation award CCF-0916763.

REFERENCES

[1] J. Zhang, Y. Tang, S. Hirve, S. Iyer, P. Schaumont, and Y. Yang: A Software-Hardware Emulator for Sensor Networks. In *IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON)*, 2011.

[2] C. Y. Chong, and S. P. Kumar: Sensor Networks: Evolution, Opportunities, and Challenges. In emphProceedings of the IEEE, Volume: 91, Issue: 8, pp. 1247-1256, 2003.

[3] J. L. Hill, and D. E. Culler: MICA: A Wireless Platform For Deeply Embedded Networks. *Micro, IEEE*, Volume: 22, Issue: 6, pp. 12-24, 2002.

[4] TelosB: http://openwsn.berkeley.edu/wiki/TelosB

[5] L. Nachman, J. Huang, J. Shahabdeen, R. Adler, R. Kling: Imote2: Serious Computation at the Edge. In *Wireless Communications and Mobile Computing Conference, IWCMC '08. International*, 2008.

[6] U. Roedig, S. Rutlidge, J. Brown and A. Scott: Towards multiprocessor sensor nodes. In: *Proceedings of the 6th Workshop on Hot Topics in Embedded Networked Sensors (HotEmNets)*, 2010.

[7] V. Raghunathan, S. Ganeriwal, and M. Srivastava: Emerging Techniques for Long Lived Wireless Sensor Networks. *Communications Magazine, IEEE*, Volume: 44, Issue: 4, pp. 108-114, 2006.

[8] C. Han, M. Goraczko , J. Helander, J. Liu, N. B. Priyantha , F. Zhao: CoMOS: An Operating System for Heterogeneous Multi-Processor Sensor Devices. Res. tech. rep. MSR-TR-2006-177. Microsoft Research, Redmond, WA.

[9] S. Iyer, J. Zhang, Y. Yang, and P. Schaumont: A Unifying Interface Abstraction for Accelerated Computing in Sensor Nodes. In: *Electronic System Level Synthesis Conference*, pp. 1-6, 2011.

[10] V. Handziski, J. Polastre, J. H. Hauer, C. Sharp, A. Wolisz and D. Culler: Flexible Hardware Abstraction for Wireless Sensor Networks. *2nd European Workshop on Wireless Sensor Networks (EWSN)*, 2005.

[11] TinyOS homepage. http://www.tinyos.net/

[12] GEZEL: Hardware/Software Codesign Environment. http://rijndael.ece.vt.edu/gezel2/.

[13] User Manual: CC2420DBK Demonstration Board Kit. http://www.ti.com/lit/ug/swru043/swru043.pdf

[14] Spartan-3E Starter Kit http://www.xilinx.com/products/boards-and-kits/HW-SPAR3E-SK-US-G.htm.

[15] CubeHash. http://en.wikipedia.org/wiki/CubeHash.