Computers and Electrical Engineering 100 (2022) 107935

Contents lists available at ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng

# Remote reconfiguration of FPGA-based wireless sensor nodes for flexible Internet of Things☆☆

Syed Mahfuzul Aziz[a,*], Dylan H. Hoskin[a], Duc Minh Pham[a], Joarder Kamruzzaman[b]

[a] UniSA STEM, University of South Australia, Mawson Lakes Campus, SA 5095, Australia
[b] School of Engineering, Information Technology and Physical Sciences, Federation University, Gippsland Campus, VIC 3842, Australia

ARTICLE INFO

*Keywords:*
Wireless sensor networks (WSN)
Field programmable gate array (FPGA)
Remote reconfiguration
Wireless *bitstream* upload
Internet of things (IoT)
Data communication
Wireless communication

ABSTRACT

Recently, sensor nodes in Wireless Sensor Networks (WSNs) have been using Field Programmable Gate Arrays (FPGA) for high-speed, low-power processing and reconfigurability. Reconfigurability enables adaptation of functionality and performance to changing requirements. This paper presents an efficient architecture for full remote reconfiguration of FPGA-based wireless sensors. The novelty of the work includes the ability to wirelessly upload new configuration *bitstreams* to remote sensor nodes using a protocol developed to provide full remote access to the flash memory of the sensor nodes. Results show that the FPGA can be remotely reconfigured in 1.35 s using a *bitstream* stored in the flash memory. The proposed scheme uses negligible amount of FPGA logic and does not require a dedicated microcontroller or softcore processor. It can help develop truly flexible IoT, where the FPGAs on thousands of sensor nodes can be reprogrammed or new configuration *bitstreams* uploaded without requiring physical access to the nodes.

## 1. Introduction

The Internet of Things (IoT) is based on the vision of connecting physical things to the Internet to enable access to distributed sensor data and to control the physical world from a distance [1]. Wireless sensor networks (WSN) are a class of distributed systems [2] that provide a bridge between the cyber/electronic and the physical worlds [3], and hence are playing a key role in the realization of the IoT. At the heart of WSNs are a class of smart wireless sensor nodes, which can sense and control the IoT environment. A recent survey of the IoT has summarized a range of applications involving the sensing of chemical, biological, physical or mechanical parameters [1]. With continuous improvement in sensing technology, sensors are becoming more intelligent and energy-efficient, thereby paving the way for ever increasing number of IoT applications in the areas of environmental monitoring, disaster management, remote healthcare, security and surveillance [1]. Since energy is one of the most constrained resources in wireless sensor nodes, high level of energy conservation for WSN-based applications remains a significant research challenge.

It is envisaged by Cisco that around 500 billion sensor devices (nodes) would be interconnected via the IoT by 2030 [4], providing a tremendous capability to monitor and control the physical phenomena within the environment. A new class of wireless sensor nodes has started emerging based around Field Programmable Gate Arrays (FPGA) [5] to provide flexibility in implementing the hardware

☆ This paper is for special section VSI-fpga3. Reviews were processed by Area Editor Dr. E. Cabal-Yepez and recommended for publication.

* Corresponding author.
*E-mail address:* mahfuz.aziz@unisa.edu.au (S.M. Aziz).

https://doi.org/10.1016/j.compeleceng.2022.107935
Received 22 June 2021; Received in revised form 14 March 2022; Accepted 16 March 2022
Available online 31 March 2022

functions performed by the nodes. Hardware reconfigurability of the sensor nodes provides flexibility by enabling the nodes to adapt to varying functional and performance needs of the applications. Furthermore, the ability to remotely reconfigure the sensor nodes will extend the flexibility to a new level, because physically accessing hundreds or even thousands of nodes to reconfigure the FPGAs will be a very laborious, time consuming and costly exercise. While some works have been carried out on the reconfiguration of FPGA-based sensor nodes in general, we outline below the challenges associated with remote reconfiguration, which have not yet been fully tackled.

Existing remote reconfiguration schemes traditionally perform partial reconfiguration of the FPGA using a *bitstream* stored locally in the memory of the sensor node [6]. To the best of our knowledge, the existing schemes neither support reconfiguration of the entire FPGA nor support wirelessly uploading new configuration *bitstreams* to the memory of the remote sensor node. This paper seeks to address these limitations. However, full FPGA reconfiguration is generally considered to be costly because the entire FPGA needs to be reconfigured even for minor changes required in the FPGA configuration. This leads to high power consumption. As sensor nodes are energy constrained and are normally distributed over relatively large geographical areas, the power requirement for full reconfiguration may adversely affect the node lifetime. Fortunately, in contrast to the older generation of FPGAs, where full FPGA reconfiguration can cause a significant current draw, modern FPGAs such as the Xilinx 7 Series FPGAs do not suffer from this downfall [7]. Full reconfiguration can also be advantageous because it provides greater flexibility, allows the entire FPGA logic to be utilized and avoids complex circuitry required to manage partial reconfiguration.

In partial reconfiguration (PR) only a portion of the FPGA logic is modified (reprogrammed). PR has been promoted to be advantageous over full reconfiguration because it reduces the *bitstream* size as well as the reconfiguration time. PR also allows other sections of the FPGA to remain operational, therefore timers and communication links can remain active. The work presented in [8] claims that PR can reduce power consumption. However, a disadvantage of PR is that it can cause performance degradation (e.g., reduction in maximum clock frequency) compared to full reconfiguration [6]. A review of the literature reveals that all existing PR techniques use an external microcontroller or a softcore processor to control the reconfiguration process. Each of these components introduces significant hardware overhead and high power consumption. For example, each of the works reported in [9] requires a softcore processor to manage partial reconfiguration. The softcore processor uses up a large amount of FPGA logic, leaving much less logic resources within the sensor node's FPGA to implement user applications. This constraint often leads to the need for an external microcontroller or a second FPGA containing a softcore processor to manage the overall PR process. Often additional circuitry is required, for example, flash controllers to read and store the partial *bitstreams* and multiple peripherals to manage external communication [8].

This paper presents a technique to remotely reconfigure the entire FPGA on a wireless sensor node, and secondly, implement this without requiring an external microcontroller or a hardcore or softcore processor inside the FPGA. The goal is to not only reduce the amount of extra hardware or logic used for reconfiguration, but also to provide flexibility and reduce power consumption leading to longer lifetime for reconfigurable sensor nodes. In summary, this paper makes the following contributions:

- Efficient hardware architecture for remote reconfiguration of FPGA-based sensor nodes which eliminates the need for a dedicated microcontroller or a hardcore or softcore processor.
- Efficient protocol to remotely access the sensor node for uploading new configuration *bitstreams* into the senor node's memory. This *wireless upload* of new configuration *bitstream* is a novel feature not reported in existing literature on remote reconfiguration.
- Unlike the previously reported remote reconfiguration schemes which reconfigure the FPGA only partially, the proposed scheme reconfigures the entire FPGA on the sensor node using a very small amount of FPGA logic dedicated to reconfiguration.

Section 2 presents a brief review of the existing FPGA reconfiguration techniques. An overview of the sensor node used in this work is given in Section 3 along with a description of the wireless communication transceiver. The proposed remote reconfiguration scheme and the corresponding Finite State Machine (FSM) are presented in Section 4 along with the remote access protocol developed. Experimental results and comparison with existing work are presented in Sections 5 and 6 respectively. Section 7 concludes the paper.

## 2. Existing reconfiguration techniques

The existing reconfiguration techniques can be broadly divided into the following two categories.

### 2.1. Microcontroller-based approach

A common approach adopted for remote reconfiguration has been to utilize an additional microcontroller on the sensor node to receive the configuration *bitstream* and configure the FPGA. The Spartan-6 FPGA Configuration User Guide from Xilinx [10] provides the details of three modes in which a microcontroller can be used to configure a Spartan-6 FPGA, namely, JTAG, SPI or SelectMAP. In [11], the microcontroller is connected to the FPGA via a JTAG interface to allow configuration. Additionally, the microcontroller is used to assist with computation and to disable the FPGA when the node is idle to conserve power. The sensor node reported in [12] utilizes both a microcontroller and an FPGA to manage reconfiguration, however the details of the method used to reconfigure the FPGA are not provided.

### 2.2. Softcore processor-based approach

Valverde et al. [13] reported a sensor node architecture based on partial reconfiguration *(PR)* of an FPGA. In this work, a Virtual Architecture was developed to define the size and position of the reconfigurable and the static components of the FPGA configuration. Included in this was a MicroBlaze softcore processor to work as the system controller and a Processor Local Bus (PLB) based on a System on Chip (SoC) approach to allow communication between the modules. A compression algorithm to reduce the configuration *bitstream* size, and hence programming time and power consumption, was also presented. In another work [14], a detailed explanation of the PR process was presented. Other architectures based around PR also make use of softcore processors to control the reconfiguration process [15] and to implement secure remote reconfiguration.

## 3. The sensor node

Fig. 1 shows the block diagram of the sensor node used in this work. The details of the sensor node have been reported in one of our recent publications [16]. It was custom designed using a Xilinx Spartan-6 FPGA. An FPGA was selected over a microcontroller for the central processing unit due to the flexibility offered by reconfigurability. The Spartan-6 family of FPGAs was selected due to its low power design features [17]. From this FPGA family, the XC6SLX9 device was selected due to its relatively lower cost and availability in the TQG144 quad-flat-package, which could be assembled using in-house facilities. This provided 102 user I/O pins and 1430 slices (each slice contains four 6-input LUTs and eight flip-flops) [17].

A 100 MHz oscillator is used to provide a system clock. A 64-Mbit flash memory module is connected to the FPGA via SPI to store the *bitstreams* required to configure the FPGA. A 64-Mbit SDRAM is also provided to enable more complex applications such as image processing. User input is achieved via four DIP switches and a reset button is connected to the FPGA to allow manual reconfiguration from the flash memory. There are also headers to allow the connection of a camera and PMOD devices. PMOD is an interface standard defined by Digilent [18]. This interface is included because over 60 I/O interface boards are available from Digilent ranging from GPS receivers and Wi-Fi modules to motor drivers and temperature sensors [18].

USB connectivity is provided using the FTDI FT2232D dual *USB-to-serial* converter chip [19]. The first channel is dedicated to programming the FPGA via JTAG, while the second channel is available as a UART interface connected to the FPGA that can be used in user designs. The FT2232D is selected as it is a simple device that provides the performance needed for JTAG programming of the FPGA. Other more advanced devices are available which utilize *USB 2.0 Hi-Speed*, but require more complex circuitry. Since physical size of the sensor node is a constraint, the simpler *USB 2.0 Full Speed* FT2232D is selected.

A switch-mode buck regulator is used to regulate the input voltage from the battery or from the USB connection to the levels required by the devices on the sensor node. All components on the node are supplied with 3.3 V including the headers. The FPGA is also supplied with 1.2 V for the internal circuitry.

### 3.1. Wireless communication

The sensor node is designed with two separate header modules to allow the connection of two different wireless communication transceivers. The first one is an XBee Pro S2B module to implement a ZigBee network [20]. This facilitates RF communication in the ISM 2.4 GHz band and supports RF date rates up to 250 kbps. The module implements a high-level application layer and can be operated in two modes: (1) the API mode where the user interacts with the raw packets, or (2) the AT mode where the modules act as a

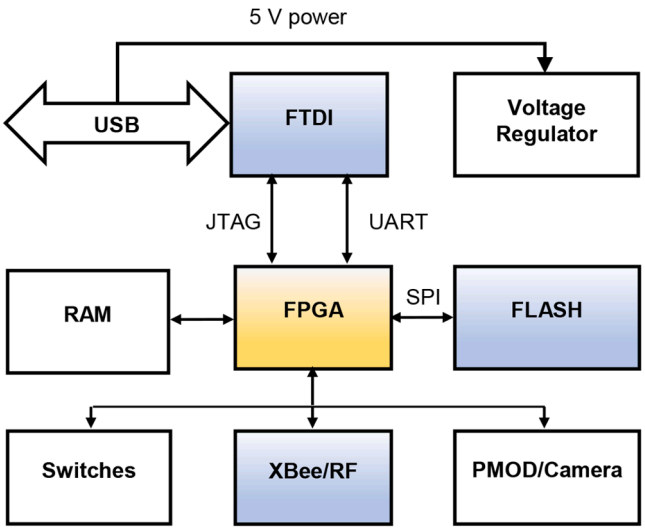


**Fig. 1.** Block diagram showing key components of the sensor node [16].

transparent UART serial connection. Broadcast transmission as well as direct addressing of modules on the network is supported. The network is automatically configured with the ability to support both star and mesh layouts. The XBee modules also feature a *sleep* mode whereby they enter a *low-power state* to conserve energy.

The second header module included on the sensor node supports a standard RF module. This is provided to allow for the future migration from a ZigBee network to an 802.15.4 protocol stack developed in-house. This will allow for more flexibility and control over the communications within the sensor network and hopefully give the ability to reduce power requirements involved in the wireless transmission of data within the WSN. For the work presented in this paper, the off-the-shelf XBee Pro S2B modules were utilized in the *AT mode*. This allowed for quick development without the intricacies of handling the low-level networking of the sensor nodes. The XBee transceiver modules were used to set up a wireless network using the ZigBee protocol. A photograph of the base sensor node (all modules removed) is shown in Fig. 2 [16]. It was developed on a custom six-layer PCB with dimensions 7cm × 8.5 cm. Fig. 2 shows the empty camera header on the far left, RF transceiver header on the bottom left, PMOD header on the bottom right and XBee transceiver header on the far right. The USB input, voltage regulator, DIP switches, reset button, FTDI USB converter and FPGA can all be seen on the top of the sensor node. The SDRAM and flash memory are mounted on the bottom of the 6-layer PCB to conserve space.

## 4. FPGA configuration process

### 4.1. Traditional configuration

Traditional configuration of a Spartan-6 FPGA is achieved via JTAG. The sensor node is connected to a PC using a USB cable and the FTDI USB converter produces the JTAG signals required to interact with the FPGA. In this way, once a configuration has been synthesized to a *bitstream*, a computer program such as *xc3sprog* [21] is able to load the *bitstream* onto the FPGA. However, the configuration memory of the Spartan-6 FPGA is volatile, meaning that once power to the FPGA is lost the memory is erased. Because of this, it is standard practice to install external, non-volatile memory alongside the FPGA. Upon receiving power, the FPGA automatically loads a configuration *bitstream* stored on the non-volatile memory. As discussed in the previous section, the sensor node used in this paper makes use of an SPI flash module for the non-volatile memory. The Spartan-6 FPGA family supports the following configuration modes [10]:

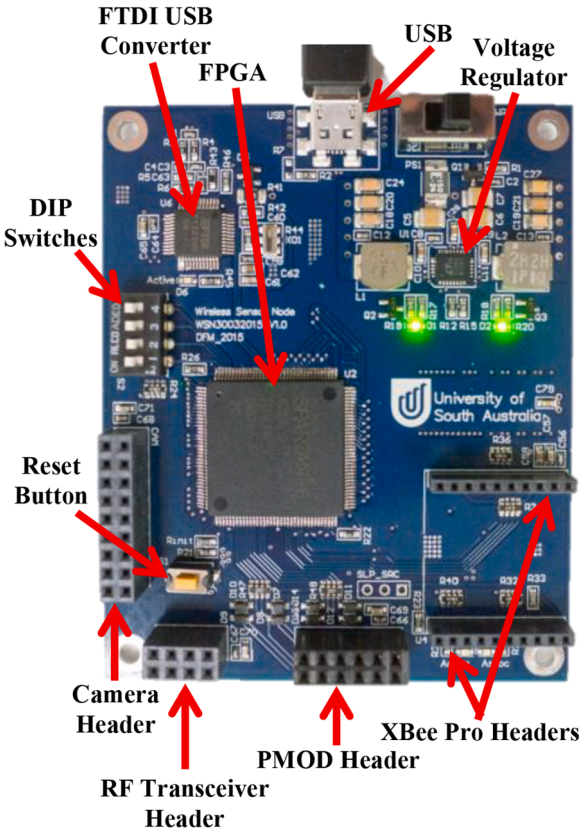


**Fig. 2.** FPGA-based wireless sensor node on a custom-built PCB [16].

- JTAG
- Master Serial/SPI
- Master Select MAP/BPI
- Slave Serial
- Slave Select MAP

In the sensor node used in this work, the Master Serial/SPI mode [10] is utilized to load a *bitstream* from the flash memory to the FPGA. As the flash memory is not connected to the FTDI USB converter (see Fig. 1), during standard programming over USB the FPGA is first configured with a special *bitstream* that allows the FPGA to act as a *JTAG to SPI converter*. In this way, the computer program running on the PC can interact with the flash memory module to write configuration *bitstreams*.

### 4.2. Remote reconfiguration

As opposed to traditional configuration using a PC connected to the FPGA board, two requirements were identified to allow remote reconfiguration of the FPGA over a wireless network:

1 Wireless access to the SPI flash memory on the sensor node to upload new configuration *bitstreams*.
2 Mechanism to remotely trigger the FPGA to reload the configuration *bitstream* from the flash memory.

To provide wireless access to the sensor node, the XBee module was used in the AT mode to act as a transparent UART connection. This required the development of logic blocks within the FPGA to control the reconfiguration process, and logic blocks to act as interfaces between the control logic and external modules, namely, XBee module and flash memory. Fig. 3 presents the various logic blocks implemented within the FPGA for this purpose. The *Top-Level Finite State Machine (Top-Level FSM)* shown in Fig. 3 is the logic block that controls the overall reconfiguration process. The *UART controller* and *SPI controller* are the logic blocks for providing connections to the XBee module and flash memory respectively. The Spartan-6 FPGA has a dedicated configuration logic built within the FPGA. It includes four registers named GENERAL1 – GENERAL4 and a primitive called *internal configuration access port (ICAP)* that allows control of the built-in configuration logic [10]. In Fig. 3, the block called *Reconfiguration via ICAP* is a small finite state machine (FSM) that utilizes the ICAP [10] to set the above registers to the required SPI flash memory addresses. This FSM is also used to trigger the FPGA to reload a configuration *bitstream* from the specified memory location on the SPI flash. One of our recently published papers [16] has described how the ICAP can be used to control the FPGA's built-in configuration logic.

Fig. 4 shows a simplified block diagram of the *Top-Level FSM* indicating the features it supports to control the reconfiguration process. This FSM is controlled by sending one of the four command characters (see Fig. 4) from the *gateway* to the sensor node over the ZigBee network. If the sensor node receives a "T" command, it responds by transmitting a lower case "t" to test if a proper connection has been established and to ensure that it is operating as expected. If the sensor node receives an "R" command, the FPGA will use the ICAP to trigger a reboot of the FPGA and load the default configuration *bitstream*. The "L" and "S" commands are for remotely

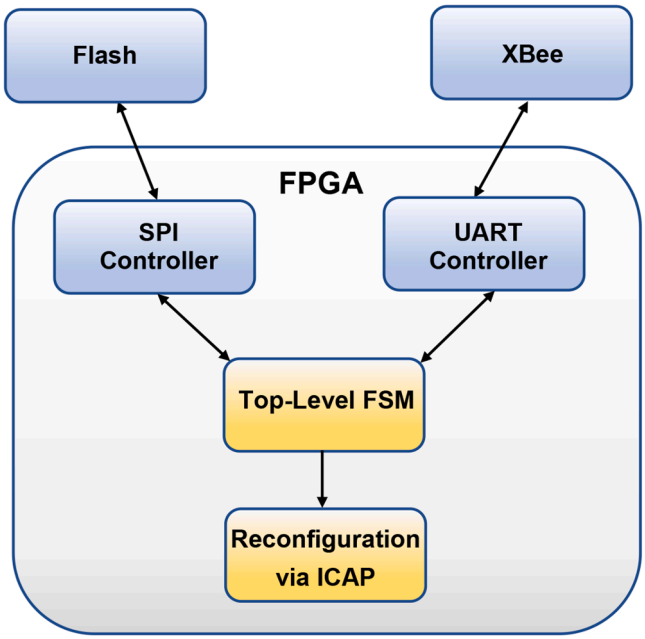


**Fig. 3.** Block diagram of the FPGA logic for remote reconfiguration.

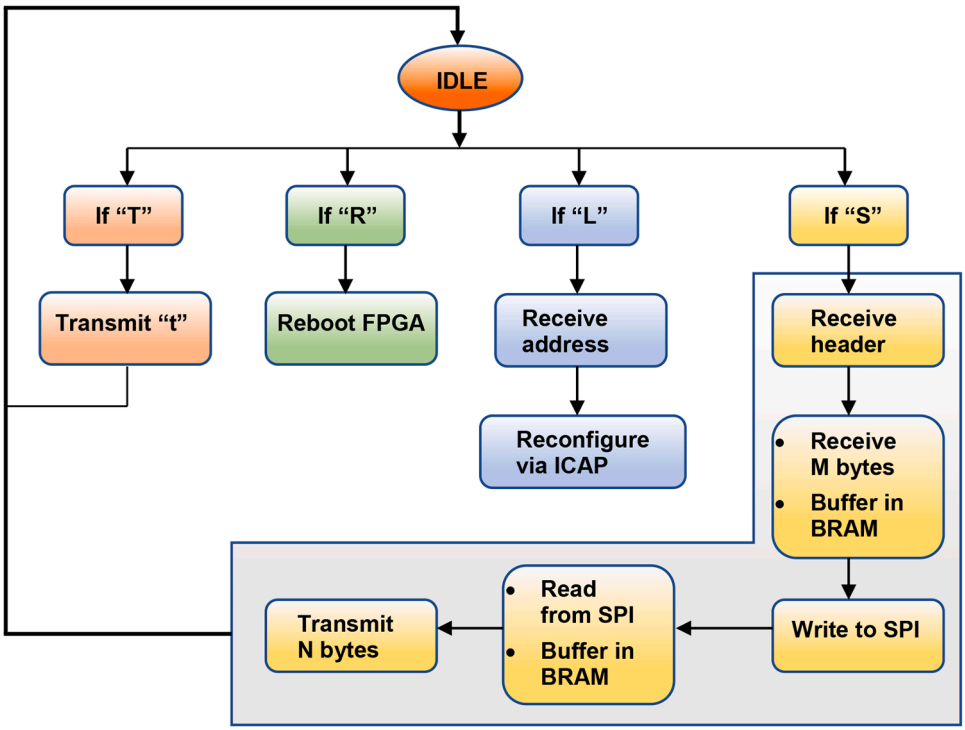


**Fig. 4.** States and transitions in the Top-Level FSM.

reconfiguring the FPGA and writing a new configuration *bitstream* to the flash memory respectively. Included within the *Top-Level FSM* is a watchdog timer to return to the IDLE state, thus preventing the FSM from getting stuck in an unknown state if an error occurs.

As stated previously, the block *Reconfiguration via ICAP* (see Fig. 3) triggers the FPGA to reload a configuration *bitstream* from a specific memory location on the SPI flash. The FSM achieves this by interacting with the ICAP. The reconfiguration process is initiated when the sensor node receives an "L" command over the network. With the "L" command, the top-level FSM receives three additional bytes specifying the address to load the configuration *bitstream* from. This address is passed on to the *Reconfiguration via ICAP* logic (see Fig. 3), which writes the address into the GENERAL1 and GENERAL2 registers. Finally, the ICAP is used to issue an IPROG command causing the FPGA to reconfigure from the specified flash memory address. The IPROG command allows the contents of the above registers to be used to set the flash memory address to a location from where to load the configuration *bitstream*. This way, full reconfiguration of the FPGA can be performed using any *bitstream* stored in the flash memory. More details about Spartan-6 FPGA configuration process can be found in [10].

Storing a copy of this reconfiguration architecture in a protected area of the flash known as the Golden *bitstream* allows for easy recovery of the sensor node if a configuration error is encountered. If an incorrect address is provided or the *bitstream* at the specified address is corrupted then the FPGA will fall back to the Golden *bitstream*, loading the safe version of the reconfiguration architecture. A new address can then be provided, or a new *bitstream* uploaded and written to the flash as described in the following sections.

### *4.3. Remote access protocol*

The previous section has described how the FPGA is reconfigured using a *bitstream* stored in a specific location of the flash memory. This section presents an application layer protocol developed to provide full remote access over the ZigBee network to the SPI bus on the sensor node so that it is possible to wirelessly upload a new *bitstream* to the flash memory. The advantage of this is that the full range of commands and operations offered by the flash memory module can be utilized remotely, as opposed to hardcoding specific commands into the top-level FSM logic. This allows for greater flexibility and possibilities for future development.

In this protocol, the structure of the packet sent from the *gateway* to the sensor node is shown in the top row of Fig. 5. The packet is designed with six sections. The first section is a byte-long *header*. If the *header* contains the "S" command, then it is an instruction to the top-level FSM to enter the *Remote Access* mode. The actions that take place in this mode were indicated in Fig. 4 under the "S" command. The *header* is followed by a single byte sequence number ranging from 0 to 254, with 255 reserved for NACK. The next three bytes are split into two parts, with 12 bits allocated to the number of bytes being written to the SPI bus (*num_write*) and the remaining 12 bits specifying the number of bytes being read from the SPI bus (*num_read*) in case of a *read* operation. The fifth section of the packet consists of the payload data (*num_write* bytes long), i.e., all the data bytes that are to be written to the SPI bus. The last section contains a single byte *checksum* calculated by XORing all the previous bytes in the packet with 0xFF.

Upon entering the *Remote Access* mode, the next step is to analyze the sequence number. If the sequence number is either equal to

| | | | | | |
|---|---|---|---|---|---|
| | Header "S" (1 byte) | Sequence number (1 byte) | Number of bytes to write (1.5 bytes) | Number of bytes to read (1.5 bytes) | Payload data (num_write bytes) | Checksum (1 byte) |
| ACK | Header "s" (1 byte) | Sequence number (1 byte) | Payload data (num_read bytes) | Checksum (1 byte) | |
| NACK | Header "s" (1 byte) | NACK 0xFF (1 byte) | Sequence number (1 byte) | | |

**Fig. 5.** Structure of the three types of Remote Access packets.

one more than the last sequence number received or a *zero*, then the operation will continue. If neither of these conditions is met then the sensor node will respond with a three-byte NACK packet consisting of the response header "s", the NACK identifier 0xFF and then the expected sequence number as shown in the bottom row of Fig. 5.

If the sequence number received is an expected one then the number of bytes to write and read are stored within the block RAM of the FPGA, before *num_write* bytes of the payload are received and buffered into the block RAM. If less than *num_write* bytes are received, then the top-level FSM will wait until the watchdog timer times out before sending a NACK with the sequence number. If the correct number of bytes is received, a *checksum* of the received data is computed locally on the sensor node and compared with the received *checksum*. If they do not match, then a NACK is sent. The NACK itself does not include a *checksum*, because this message does not have any payload data and therefore there is no need to check for errors in this case.

If the received data has passed all the checks, then it is read from the block RAM and transmitted over the SPI bus. Once *num_write* bytes are written to the SPI bus, one of two things can happen depending on the value of *num_read* in the original packet received by the sensor node: (1) If *num_read* is zero, an ACK is transmitted to notify the *gateway* that the packet was correctly received. (2) If *num_read* is greater than zero, *num_read* bytes are read from the SPI bus and once again buffered in the block RAM. The read data is then transmitted back to the *gateway* after a one-byte *response header* "s" and the one-byte sequence number, and then followed by a *checksum*, as shown in the second row of Fig. 5. Note that lower case "s" is used to designate the *response header* so that it is distinguished from the header "S" received by the sensor node from the *gateway* as depicted in the first row of Fig. 5.

In the second situation, the bytes read could include the contents of a status register or the data requested from a read operation. The first situation, when *num_read* is zero, would occur when no response is required from the SPI bus, such as when only writing to the flash memory. To simplify the protocol, the ACK packet is simply a three-byte response packet (shown in the second row of Fig. 5) with an empty payload. If the *gateway application* does not receive a response after transmitting a packet or if it receives a NACK, then it can assume that the packet is lost and retransmit it.

### *4.4. Wireless upload algorithm*

With full wireless access to the SPI bus now available, a *software application* has been written in C++ that runs on the *gateway* to wirelessly upload configuration *bitstream* to a sensor node. It reads the configuration *bitstream* file generated using the Xilinx ISE tool and transmits this to the wireless sensor node for storing on the SPI flash memory. The flow chart of this application is shown in Fig. 6. This program is based on the SPI programming algorithm used in the JTAG programming tool *xc3sprog* [21]. The program reads the configuration file and splits it into appropriately sized chunks based on the page size of the flash memory, then erases the flash and writes the data into it. It also includes the ability to *test* the connection using the "T" command (as per Fig. 4), provide a flash memory address to boot from using the "L" command (as per Fig. 4), *erase* the entire flash, and *verify* that the configuration *bitstream* was written correctly.

As illustrated in Fig. 6, during the *wireless upload* process, the *gateway application* first queries the device ID of the flash module, then determines the page and sector size based on the device ID returned. The flash memory module used on the sensor node has a *page size* of 256 bytes, hence the configuration file is uploaded in 256-byte chunks. Also, the minimum erasable area for the flash memory is 4 kb, and this is used as the *sector size*. This clearly means that an *erase* needs to occur each time sixteen pages have been written.

After receiving the device ID, a sector erase is performed to prepare the sector to be written to. The data is then written page by page, with a sector erase occurring each time a new sector is reached. Finally, the data is verified at the application level by reading back the data stored on the flash memory and comparing it with the file on the *gateway* that was originally sent. As the erasing of sectors and writing of pages do not occur instantaneously, a method is needed to determine when the next *write* or *erase* command can be issued. This is accomplished by polling the status register of the flash module at 5 ms intervals. Once the status register on the flash indicates a *write/erase* was complete, the program can continue with the next *page write* or *sector erase*.

In the ideal case, to perform *write* and *verification* of a typical default *bitstream* file for reconfiguration of the entire FPGA, which is 342,816 bytes in size for our selected device XC6SLX9 [10], a minimum of 5614 *Remote Access* packets needs to be sent to the sensor node. As each packet sent will receive a response (even an empty ACK), at least 5614 packets are expected to be received. This leads to

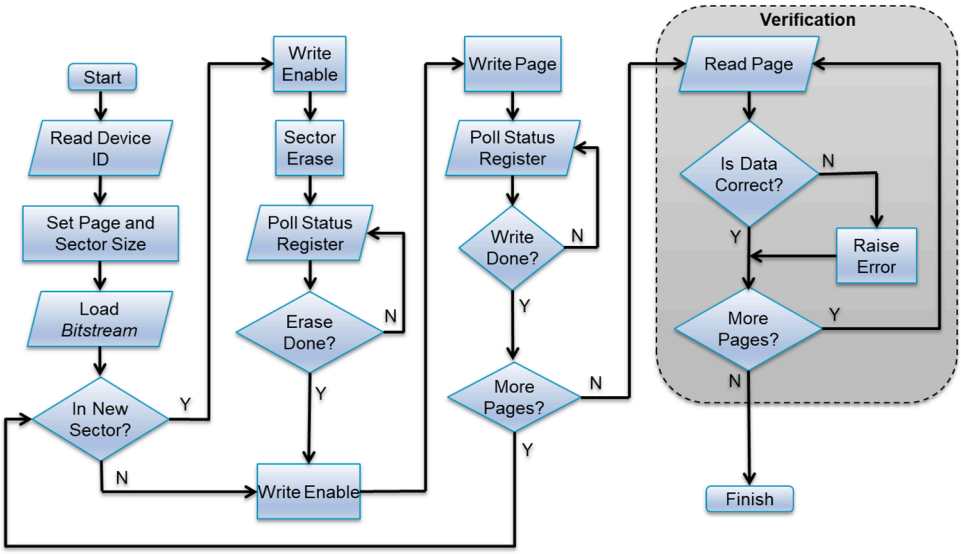


**Fig. 6.** Gateway application to wirelessly upload configuration *bitstream* to the sensor node.

a total of 390,630 bytes sent to and 361,330 bytes received from the sensor node. More bytes are transmitted to the sensor node than received from it due to the larger packet size (i.e., the inclusion of *num_write* and *num_read*) and the commands being sent. This minimum data transmission assumes that the *write* and the *erase* commands sent to the flash memory module are completed instantly, i.e., the first polling of the status register indicates that the *write/erase* has completed. In practice, this is not likely to happen, so the number of packets will increase. However, as each additional poll of the status register only introduces 7 extra bytes sent to the sensor node and 4 extra bytes received from it, the impact on the total amount of data sent/received is almost negligible.

## 5. Experiments and results

Wireless sensor nodes are expected to incur minimal power consumption to survive extended periods on limited battery capacity, hence ensuring that the reconfiguration process does not consume a large amount of power is extremely important. Also, data transmission is the most energy intensive operation for the sensor node. As the WSN is set up in a mesh network, retransmissions will adversely affect the battery life of the intermediate nodes, which means minimizing the amount of data requiring retransmission is imperative. Therefore, in this work, the key metrics that will be used to assess the performance of the proposed reconfiguration architecture are the power consumed by the node and the amount of data requiring retransmission.

The time taken to wirelessly upload a new configuration *bitstream* to the flash memory and the time taken to reconfigure the FPGA using a *bitstream* stored on the flash memory will also be taken into consideration. These are the high-power states of the sensor node, therefore reducing the time spent in these states is advantageous for extending battery life. However, as the frequency of these processes is likely to be extremely low, the upload and reconfiguration times are of lesser concern than the overall power consumed by the sensor node.

### 5.1. Testing procedure

Xilinx Synthesis Tool was used to find the amount of FPGA logic required by the proposed remote reconfiguration architecture. Power consumption was determined by practically measuring the current through the 6 V power rail using a 1 Ω shunt resistor and a digital storage oscilloscope (DSO). To assist with accurate power measurements, one of the pins in the PMOD header was used as a trigger for the DSO. When the sensor node leaves the IDLE state, this pin is set high, causing the DSO to trigger and record power consumption during the *receive, write, read* and *send* states. To measure the amount of data retransmissions, a feature was added to the *gateway application* for *wireless upload* (covered in Section 4.4) to record all UART transmissions to a log file. By analyzing the log files, the amount of excess data retransmission can be determined.

The system was first tested using the USB-to-UART interface available on the second channel of the FTDI chip. Once the reconfiguration process and the *Remote Access* protocol were validated via the USB connection, the communication was switched to the wireless network using the ZigBee protocol. As stated in Section 3.1, the XBee Pro module is used in the AT mode. As such, it acts as a transparent UART connection. Hence, changing from USB to ZigBee communication requires no changes in the logic on the FPGA other than changing the pins for the UART connection from the FTDI chip to the XBee module. This function is linked to one of the user input DIP switches shown in Fig. 2, so that either USB or ZigBee communication method can be selected by simply changing the position of switch 1.

### 5.2. FPGA resource utilization

The first row in Table 1 shows the total amount of logic available on the Spartan-6 FPGA and the third row shows the logic required for the proposed remote reconfiguration architecture. Clearly, the amount of logic required for remote reconfiguration is quite small, consuming only 8.9% of the LUTs and 2.7% of the Flip-Flops. This includes the digital logic required by the two finite state machines to handle the reconfiguration and the modules to handle UART and SPI communications as shown in Fig. 3. It does not include any user applications. The small amount of logic required for remote reconfiguration means that plenty of logic resources remain available on the FPGA to implement user applications. To incorporate the remote reconfiguration feature into a senor node, the proposed reconfiguration architecture must be integrated with the logic implemented on the FPGA for user applications, i.e., the reconfiguration architecture must be part of the full *bitstream* containing user applications. The communication logic blocks can be reused by the user applications, thus reducing the total FPGA logic required by these applications. To measure the power consumed by the XBee module in the *sleep* mode, a small configuration logic was developed just to assert the *sleep* pin on the XBee module. This required virtually no logic, resulting in a "minimal configuration" as shown in the second row of Table 1.

Table 1 includes a comparison of resources with the partial reconfiguration (PR) architecture presented in [8] called the VAPRES architecture, which is a virtual architecture developed to provide a communication layer among the different PR Regions (PRR) implemented on a FPGA. It used a Xilinx Virtex-4 FPGA, which contains 4-input LUTs as opposed to 6-input LUTs on the Spartan-6 FPGA used in our work. So, for a fair comparison, the number of 4-input LUTs required in [8] was adjusted to equivalent number of 6-input LUTs using a conversion ratio of 2:1 according to the procedure recommended in [22]. The number of 6-input LUTs required by the design in [8] would be approximately 9596, which would not fit in the Saprtan-6 XC6SLX9 device we have used. Clearly, our proposed reconfiguration architecture uses significantly less resources than that used by [8], although it needs to be pointed out that the latter included five filter modules along with the VAPRES architecture. However, the partial reconfiguration architecture in [8] requires a softcore processor as the central control unit, a flash controller core to read and store the partial *bitstreams*, and numerous peripherals for external communication. These components occupy significant FPGA resources and aren't included in the LUT count shown in Table 1. As pointed out in [22], comparison of different architectures, particularly those implemented on devices from different FPGA families, isn't straightforward. However, such comparison gives some indication of relative resource requirements.

### 5.3. Power consumption

Several power measurements were made on the sensor node with the FPGA programmed with *bitstreams* for various configurations, with the XBee Pro module attached and removed, and the top-level FSM in various states. These configurations are listed in Table 2. For each case, Table 2 shows the average power consumption (Avg.) obtained from 10 measurements with 95% confidence interval (CI).

The first two measurements in Table 2 were taken on the *unconfigured* sensor node. They show that the addition of the XBee Pro module increases the power consumption of the sensor node by 182.1 mW. Similarly, comparing the fifth and the sixth entries shows an increase of 192.9 mW when the XBee module is attached to the sensor node *configured* with the reconfiguration architecture. Clearly, in the IDLE state, the XBee module consumes over one quarter (25.9%) of the total power consumed by the sensor node when configured with the reconfiguration architecture.

By comparing the third and fourth entries in Table 2 it is observed that when the XBee module is put in the *sleep* mode then the increase in power due to having the XBee module attached reduces to 38.5 mW. This is around 80% decrease in power consumed by the XBee module compared to when it is not in *sleep* mode, i.e., in the IDLE state of the reconfiguration architecture. It is also interesting to note that when the sensor node is configured with the minimal configuration (i.e., the image used to put the XBee to *sleep* mode), the power consumption is less than the power consumed by the sensor node when the FPGA remains *unconfigured*.

Compared to the IDLE state (sixth entry in Table 2), when the sensor node is receiving data over the network and writing it to the flash memory as shown in the seventh entry in Table 2, there is an increase in power consumption by approximately 21.9%. Similarly, the eighth entry in Table 2 shows the power consumed during reading from the flash memory and transmitting this over the network. In this case, the power consumption increases by 19.6% compared to the IDLE state. These increases are expected because, in these states, the XBee Pro module is highly active, i.e., transmitting and receiving data.

Fig. 7 shows the recorded instantaneous power consumed by the sensor node over a complete *wireless upload cycle*. The *Erase/Write, Verify* (read) and *Idle* states are marked in the figure. Clearly, the power consumed by the sensor node increases during the upload process, due to the extra power consumed by the XBee Pro module for data transmission. These brief power spikes reach approximately 1230 mW, however, as these spikes only occur for very short periods of time, the average power over the entire upload process is approximately 906 mW. This corresponds to item 7 in Table 2. As the frequency of uploading new *bitstreams* is expected to be low, this

**Table 1**
Amount of FPGA logic consumed by various architectures.

| Configuration | LUTs | | Flip-Flops | |
|---|---|---|---|---|
| | Number | % | Number | % |
| Full FPGA (Saprtan-6 XC6SLX9) | 5720 | 100 | 11,440 | 100 |
| Minimal configuration | 0 | 0 | 0 | 0 |
| Reconfiguration Architecture | 509 | 8.9 | 310 | 2.7 |
| VAPRES Comparison Architecture [8] | 9596 | 167.8 | 5,566 | 48.7 |

**Table 2**
Average power consumed in various states of the sensor node.

| Setup | Power (mW) Avg. | CI |
|---|---|---|
| 1. Unconfigured FPGA, no modules attached | 426.4 | ±2.2 |
| 2. Unconfigured FPGA, XBee Pro attached | 608.5 | ±5.8 |
| 3. Minimal configuration, no modules attached | 408.9 | ±6.8 |
| 4. Minimal configuration, XBee Pro attached in *sleep* mode | 447.4 | ±8.2 |
| 5. Reconfiguration architecture, no modules attached, IDLE state | 550.7 | ±5.2 |
| 6. Reconfiguration architecture, XBee module attached, IDLE state | 743.6 | ±3.6 |
| 7. Reconfiguration architecture, XBee module attached, uploading & writing to flash | 906.4 | ±3.5 |
| 8. Reconfiguration architecture, XBee module attached, reading & verifying from flash | 889.1 | ±2.3 |
| 9. Remote reconfiguration from the flash memory, XBee module attached | 652.2 | ±3.4 |

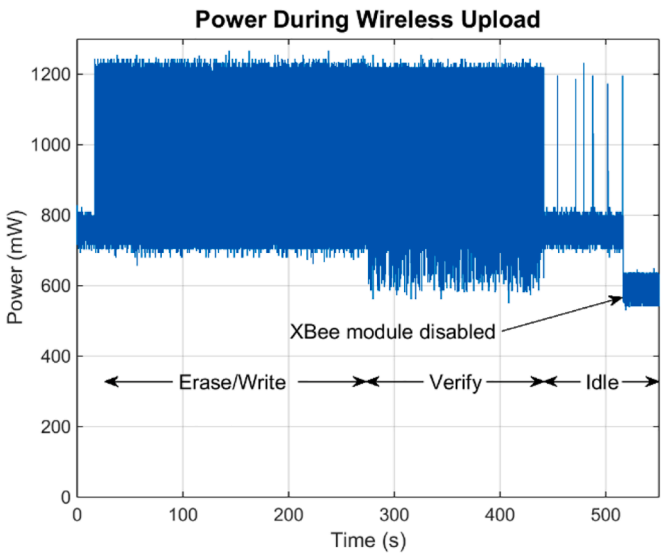


**Fig. 7.** Instantaneous power consumption of the sensor node during wireless upload of a new configuration *bitstream* over the ZigBee network.

small increase in power (~163 mW) compared to the IDLE state shown in Fig. 7 (also shown in item 6 of Table 2) will have negligible impact on the overall battery life of the sensor node. For continuous *wireless upload* operations at 906 mW, a fully charged 40,000 mAh battery will last for over 9 days for a supply voltage of 5 V, whereas a battery runtime of over 11 days can be achieved in the IDLE state of the XBee module. Based on the power consumption of 652.2 mW reported in item 9 of Table 2, a runtime of 13 days can be achieved for continuous remote reconfiguration from the flash memory. In practice, *wireless upload* of new *bitstream* or remote reconfiguration operations are not likely to occur frequently, therefore the battery runtime is expected to be higher than the above estimates. Using the *sleep* mode of the XBee module will further increase the battery runtime. Finally, by using smart power management strategies, if most of the components on the sensor node is put in the *power down* mode when the node isn't in use, then the node runtime will increase significantly.

Fig. 8 shows the recorded waveform of the instantaneous power consumed by the wireless sensor node during reconfiguration of the FPGA from a *bitstream* stored in the flash memory. By repeating the reconfiguration process 10 times, the average power consumed during reconfiguration was calculated to be 652.2±3.4 mW. This value is shown as the last entry in Table 2. This is less than the 715 mW power consumed by the FPGA-based sensor node reported in [23] in *active* mode. Considering that FPGA reconfiguration is typically a high power consuming operation, the power consumption of the proposed scheme is quite reasonable. Also, this power will be consumed only when FPGA reconfiguration is required, which would not be as frequent as the active sensing operations of IoT end devices. The power consumption of our sensor node in the *sleep* mode is 447 mW, which is comparable to the *standby* power of 435 mW consumed by the FPGA-based sensor node reported in [23].

### 5.4. Data transmission

As was stated in Section 3, the second channel of the FTDI chip provided a USB-to-UART interface to communicate with the FPGA. The reconfiguration algorithm was first tested using the USB connection. Then the reconfiguration algorithm was tested using UART over ZigBee wireless connection. During the testing of the *wireless upload* algorithm over USB, the number of packets and bytes sent to and received from the sensor node has been recorded for each of the 10 trials. It was observed that the average number of packets

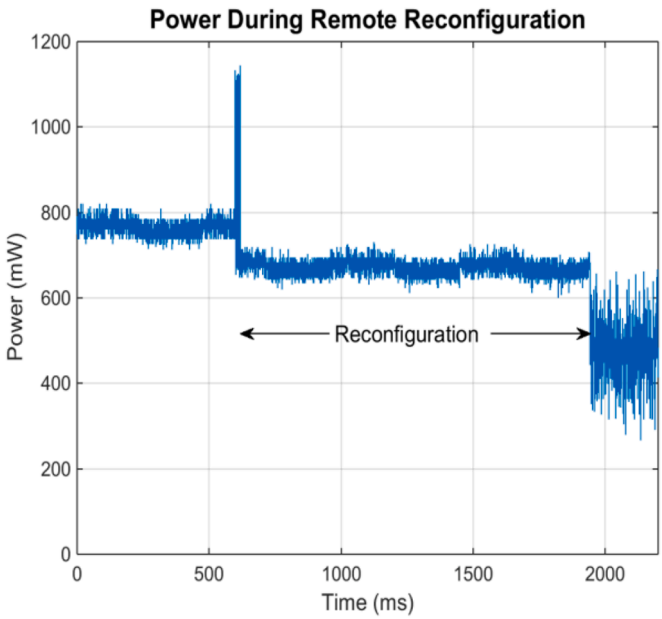


**Fig. 8.** Instantaneous power consumed by the sensor node when a remote reconfiguration is triggered.

transmitted to the sensor node (including *write enable* commands, *write* commands, *polling* the status register and *read* commands) as well as the number of packets received from the sensor node (such as ACKs, NACKs and the response to *reads*) both came to 7615±25. This represents 402,436±172 bytes being sent to the sensor node and 367,203±98 bytes being received from the sensor node. This shows that on average there is a 35.7% overhead in the number of packets sent or received when compared to 5614 packets sent/received for *writing* and *verification* of a typical default configuration *bitstream* file (referred to as the *ideal* case in Section 4.4). However, this equates to only 3.02% overhead in the number of bytes sent to and 1.63% overhead in the number of bytes received from the sensor node. When compared to the typical default configuration *bitstream* file size of 342,816 bytes, the entire *wireless upload* algorithm and *Remote Access* protocol adds 17.4% overhead to the amount of data sent to the sensor node; some of this is unavoidable, for example, the commands for the flash memory module.

For the 10 trials conducted, when transmitting via USB connection, the entire *upload* process takes an average of 104.0±1.6 s to complete, i.e., from the instant the *upload* program was started to when it had completed execution. In these 10 trials, there were no packet errors reported by the *Remote Access* protocol.

Migrating from using USB communication to using the ZigBee network proved to be more challenging. Although a mesh network consisting of a base station (coordinator) and 18 wireless nodes had been set up to conduct various tests, a simpler ZigBee network comprising two XBee Pro S2B modules was used to take measurements to ascertain the effectiveness of the *wireless upload* program. One XBee Pro module, connected to a sensor node, was configured as an "AT router" and the destination address was set to the coordinator. The second XBee Pro module was connected to the *gateway* via a USB to serial converter and configured as an "AT coordinator" with the destination address set to broadcast. However, with this configuration there was a packet error rate exceeding 28%, causing the *wireless upload* to be unsuccessful. Changing the destination address of the coordinator XBee Pro module to the specific address of the XBee Pro module of the router node rectified this issue, allowing the *wireless upload* to complete successfully.

For the 10 trials conducted over the ZigBee network, an average of 6179±6 packets were both sent to and received from the sensor node. This equated to 393,393±307 bytes sent to the sensor node and 361,455±24 bytes received from the sensor node. On average, the overhead compared to the *ideal* case is 10.1% in the number of packets sent or received, 0.71% in the number of bytes sent to the sensor node and 0.03% in the number of bytes received from the sensor node. The performances of the *wireless upload* algorithm for both USB and ZigBee communications are summarized in Table 3. The time required for the entire *upload* process was significantly longer over the ZigBee network than over USB, taking an average of 419.6±2.0 s. There was also an average of 3.9 packet errors detected by the *Remote Access* protocol, however the erroneous packets were successfully retransmitted to the sensor node.

While the 104 s required to upload a *bitstream* via USB is quite reasonable, almost seven minutes are required for the upload over ZigBee. This is because the extended transmission time not only clogs up the network, potentially preventing other sensor nodes from transmitting data, it also increases the power consumed by the sensor node for a longer period. This has a detrimental impact on the overall battery life of the sensor node. Future work to migrate from the XBee Pro module to a standard RF module using an 802.15.4 protocol stack developed in-house has the potential to reduce transmission time and power overhead that the XBee Pro module introduces.

The average reconfiguration time was found to be 1350.6±0.5ms, which equates to a throughput of 0.74 remote reconfiguration events per second, or 44 remote reconfiguration events per minute. This represents the duration from the instant when the command is received at the sensor node to the instant when the reconfiguration is completed. This time varies significantly depending on the

**Table 3**
Average packets and bytes transmitted during wireless upload.

| Communication | Ideal | USB | ZigBee |
|---|---|---|---|
| Number of packets | 5614 | 7615 | 6179 |
| Packet overhead | - | 35.7% | 10.1% |
| Bytes sent to sensor node | 390,630 | 402,436 | 393,393 |
| Bytes sent overhead | - | 3.02% | 0.71% |
| Bytes received from sensor node | 361,330 | 367,203 | 361,455 |
| Bytes received overhead | - | 1.63% | 0.03% |
| Total time for upload (seconds) | - | 104.0 | 419.6 |
| # of packet errors | - | 0 | 3.9 |

*bitstream* that is used to configure the FPGA. In some cases, the reconfiguration takes as little as 600ms, but the average presented above includes measurements when configuring the FPGA using the *bitstream* requiring the longest time (worst case). The reconfiguration time is longer than expected, however, as the frequency of reconfiguration is expected to be low it does not pose a significant issue.

## 6. Comparison and discussion

While remote reconfiguration of FPGA-based sensor nodes has been done before, this has always required a separate dedicated controller to oversee the process [12,15]. In some cases, this is dedicated logic on the FPGA in the form of a softcore processor [6,14]. Each of these schemes is limited to partial reconfiguration, which means the entire FPGA isn't reconfigured. Other existing methods require additional hardware in the form of a separate off-the-shelf microcontroller [11]. To the best of the authors' knowledge, the remote reconfiguration architecture presented in this paper is the only current method that allows full FPGA reconfiguration without requiring a dedicated softcore or hardcore processor or an external microcontroller. Existing schemes only allow reconfiguration of the FPGA remotely using *bitstreams* pre-stored in the sensor node's memory but can't remotely upload new configuration *bitstreams* to the sensor node. The scheme proposed in this paper has addressed this limitation with a new *Remote Access* protocol. In contrast with System-on-Chip devices, low power FGPA devices such as those from the 7 series families [7] or the Spartan-6 family [17] are preferred for wireless sensor nodes to keep power consumption to a low level. These are 'FPGA only' devices, i.e., there is no built-in hardcore processor available within the FPGA to help manage the reconfiguration process. Therefore, the small reconfiguration architecture presented in this paper is attractive for remote reconfiguration of sensor nodes that are implemented using 'FPGA only' devices from these low power families. The proposed scheme can be easily applied to other FPGA families that support reconfiguration with ICAP, for example, all the 7 series families [7].

Partial reconfiguration can be a desirable feature to add to the reconfiguration architecture, however, literature detailing how to implement partial reconfiguration without a dedicated softcore processor is lacking. Available literature indicates that a softcore processor is required to control the partial reconfiguration process, as it is a significantly complicated process. While partial reconfiguration has the advantages of shorter reconfiguration times and smaller *bitstreams* resulting in shorter transmission times and energy, using a softcore processor to achieve this not only offsets these advantages but also consumes a large part of the FPGA logic. The proposed full FPGA reconfiguration architecture requires less than 9% of the LUTs and less than 3% of the Flip-Flops on the XC6SLX9 device, leaving almost the entire FPGA for user designs/applications.

The power requirements of the wireless sensor node used in this work are quite competitive compared to other nodes. It is reported that the sensor node used in [11] consumed 946 mW in just the FPGA and the reconfiguration microcontroller, not including other factors such as the static power consumed by the board and the power consumed by the wireless communication module. This is 27% higher than the total power consumed by the sensor node used in this paper in the IDLE state (sixth row of Table 2). Including the power consumed by the transceiver would significantly increase the power consumption figures for the sensor node presented in [11]. To compare the power consumption of the proposed reconfiguration architecture with others, one would ideally require practical results for reconfiguration implemented using devices from the same FPGA family (Spartan-6). Such results are very scarce in existing literature. Hence, we are only able to compare with the few sensor nodes that have reported some results, although they might have used devices from different FPGA families or reported results for different tasks. Therefore, this is not a straightforward comparison. The power consumption reported in [24] is 89.53 mW for a RISC processor implemented on a partially reconfigured FPGA to perform AES encryption, whereas our reconfiguration architecture increases the sensor node power by 124.3 mW compared to the unconfigured node (based on comparing items 1 and 5 in Table 2). Note that the total power reported in [24] does not include the power overheads for other components on the board. Moreover, the IoT platform in [24] used an FPGA from the relatively newer Xilinx Artix-7 family (XC7A100T); this family of FPGAs consumes much less power than the older Spartan-6 FPGA family used in our work. Therefore, it would be sensible to assert that the proposed reconfiguration architecture fares well in terms of power consumption and is expected to consume even less power than reported in this paper if it is implemented on IoT platforms that use devices from recent low-power FPGA families.

For many practical applications, the frequencies of remote reconfiguration of the FPGA or *wireless upload* of new *bitstreams* are not likely to be very high. Therefore, with the level of power consumption reported in this paper, today's high-end IoT-suitable batteries are likely to last for significant periods of time. Nonetheless, further work to improve the power efficiency of the sensor node is envisaged. Further power savings can be achieved by deploying advanced power management strategies including the use of *sleep*

modes of the various components on the sensor node when they are not in use (such as the flash memory, SDRAM and XBee Pro). Other ways to improve energy efficiency include utilizing innovative *bitstream* compression techniques to reduce the amount of data requiring transmission, as was demonstrated by the FPGA-based image compression architecture presented in [25]. Also, the energy consumption of FPGA-based sensor nodes can be greatly reduced by using hardware optimized architecture for implementing the desired logic, for example, the object extraction architecture presented in [2].

It is important to ensure the security of the reconfiguration *bitstreams*. This aspect was outside the scope of this paper, because the primary focus was to develop an efficient architecture for remote reconfiguration and a protocol for wirelessly uploading new *bitstreams* to remote sensor nodes. Nonetheless, the ZigBee wireless network uses secure 128-bit AES-based encryption system, so it is unlikely that the reconfiguration *bitstream* could be stolen off air. However, the physical link between the XBee module and the sensor node (FPGA) is not encrypted; this could lead to security concerns. Further work may be undertaken to add some form of encryption on the communication link between the XBee module and the FPGA.

## 7. Conclusions

The proposed remote reconfiguration scheme is implemented using very small amount of logic within the sensor node's FPGA, less than 9% of the LUTs and less than 3% of the Flip-Flops on the Spartan-6 FPGA device. This is negligible compared to other remote reconfiguration architectures reported to date. A novel contribution of this work is that new FPGA configuration *bitstreams* can be wirelessly uploaded to the flash memory of remote sensor nodes using the proposed *Remote Access* protocol. On average, the power consumed by the sensor node during the *wireless upload* of a full configuration *bitstream* over ZigBee is approximately 906 mW, which is only around 163 mW higher than the power consumed by the node in the IDLE state. The average power consumed by the sensor node during the reconfiguration operation is approximately 652 mW. This is less than the power consumed in the *active* mode by a FPGA-based sensor node reported in literature. The above results demonstrate that the remote reconfiguration of the FPGA and the *wireless upload* of a new *bitstream* do not drastically increase the sensor node's overall power consumption. For uploading a full configuration *bitstream* to the sensor node over ZigBee, the *wireless upload algorithm* introduces only 0.71% overhead in the number of bytes sent to the sensor node and 0.03% overhead in the number of the bytes received by the *gateway* from the sensor node. These overheads are clearly negligible. The proposed remote reconfiguration scheme is attractive as it reconfigures the entire FPGA, uses very small amount of FPGA logic, incurs modest power consumption and allows completely *wireless upload* of new configuration bitstreams with very little data communication overhead.

## CRediT authorship contribution statement

**Syed Mahfuzul Aziz:** Conceptualization, Methodology, Visualization, Supervision, Project administration, Writing – original draft, Writing – review & editing. **Dylan H. Hoskin:** Conceptualization, Methodology, Software, Investigation, Visualization, Writing – original draft, Writing – review & editing. **Duc Minh Pham:** Conceptualization, Methodology, Visualization, Supervision, Writing – review & editing. **Joarder Kamruzzaman:** Methodology, Writing – review & editing.

## Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Acknowledgments

The authors acknowledge Mathew Potaczek for assistance with implementing the first version of the PCB based on the concepts and initial design of Duc Minh Pham and Syed Mahfuzul Aziz. They are grateful to Daniel Forero Morales for assistance with fine tuning the revised PCB design and for liaising for manufacturing the PCB.

## References

[1] Kassab W, Darabkh KA. A–Z survey of internet of things: architectures, protocols, applications, recent advances, future directions and recommendations. J Netw Comput Appl 2020;163:1–49. https://doi.org/10.1016/j.jnca.2020.102663.

[2] Pham DM, Aziz SM. Object extraction scheme and protocol for energy efficient image communication over wireless sensor networks. Comput Netw 2013;57(15): 2949–60. https://doi.org/10.1016/j.comnet.2013.07.001.

[3] Elson J, Estrin D. Sensor networks: a bridge to the physical world. In: Raghavendra CS, Sivalingam KM, Znati T, editors. Wireless Sensor Networks. Boston: Springer; 2004. p. 3–20. https://doi.org/10.1007/978-1-4020-7884-2_1.

[4] CISCO. (2016). "Internet of Things," At-a-Glance, no. C45-731471-01, 2016, pp 1–3. Available: http://www.audentia-gestion.fr/cisco/pdf/at-a-glance-c45-731471.pdf [accessed 11 March 2022].

[5] Gomes T, Salgado F, Pinto S, Cabral J, Tavares A. A 6LoWPAN accelerator for internet of things endpoint devices. IEEE Internet Things J 2018;5(1):371–7. https://doi.org/10.1109/JIOT.2017.2785659.

[6] Hymel R, George AD, Lam H. Evaluating partial reconfiguration for embedded FPGA applications. In: Proceedings of the high-performance embedded computing workshop. Lexington, MA: MIT Lincoln Lab; 2007. p. 1–2. Available: https://archive.ll.mit.edu/HPEC/agendas/proc07/Day2/10_Hymel%20Conger_Abstract.pdf [accessed 29 August 2021].

[7] Xilinx. (2020). "7 Series FPGAs Data Sheet: Overview," Document no. DS180 (v2.6.1), 8 September 2020, pp. 1–19. Available: https://www.xilinx.com/content/dam/xilinx/support/documentation/data_sheets/ds180_7Series_Overview.pdf [accessed 11 March 2022].

[8] Garcia R, Gordon-Ross A, George AD. Exploiting partially reconfigurable FPGAs for situation-based reconfiguration in wireless sensor networks. In: Proceedings of the field programmable custom computing machines (FCCM), 17th IEEE symposium on. IEEE; 2009. p. 243–6. https://doi.org/10.1109/FCCM.2009.45.
[9] Xiao Z, Koch D, Lujan M. A partial reconfiguration controller for Altera Stratix V FPGAs. In: Proceedings of the field programmable logic and applications (FPL), 26th international conference on. IEEE; 2016. p. 1–4. https://doi.org/10.1109/FPL.2016.7577349.
[10] Xilinx. (2019). "Spartan-6 FPGA Configuration User Guide," Document no. UG380 (v2.11), 22 March 2019, pp. 1–174. Available: http://www.xilinx.com/support/documentation/user_guides/ug380.pdf [accessed 30 March 2021].
[11] Liu F, Jia Z, Li Y. A novel partial dynamic reconfiguration image sensor node for wireless multimedia sensor networks. In: Proceedings of the high performance computing and communication, 14th international conference on & embedded software and systems, 9th international conference on (HPCC-ICESS). IEEE; 2012. p. 1368–74. https://doi.org/10.1109/HPCC.2012.201.
[12] Portilla J, Riesgo T, de Castro A. A reconfigurable Fpga-based architecture for modular nodes in wireless sensor networks. In: Proceedings of the programmable logic (SPL '07), 3rd southern conference on; 2007. p. 203–6. https://doi.org/10.1109/SPL.2007.371750.
[13] Valverde J, Otero A, Lopez M, Portilla J, De La Torre E, Riesgo T. Using SRAM based FPGAs for power-aware high performance wireless sensor networks. Sensors 2012;12(3):2667–92. https://doi.org/10.3390/s120302667.
[14] Otero A, Llinas M, Lombardo ML, Portilla J, de la Torre E, Riesgo T. Cost and energy efficient reconfigurable embedded platform using Spartan-6 FPGAs. In: Proceedings of the SPIE Microtechnologies, Proceedings Volume 8067, VLSI Circuits and Systems V. International Society for Optics and Photonics; 2011. p. 1–13. https://doi.org/10.1117/12.887498. 806706.
[15] Braeken A, et al. Secure remote reconfiguration of an FPGA-based embedded system. In: Proceedings of the Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 6th International Workshop on; 2011. p. 1–6. https://doi.org/10.1109/ReCoSoC.2011.5981501.
[16] Pham DM, Aziz SM. FlexiS-a flexible sensor node platform for the internet of things. Sensors 2021;21(15). https://doi.org/10.3390/s21155154.
[17] Xilinx. (2011). "Spartan-6 Family Overview," Document no. DS160 (v2.0), 25 October 2011, pp. 1–11. Available: http://www.xilinx.com/support/documentation/data_sheets/ds160.pdf [accessed 11 March 2022].
[18] Digilent. (2022). "Pmods," Digilent's I/O interface boards. Available: https://digilent.com/shop/boards-and-components/system-board-expansion-modules/pmods/ [accessed 11 March 2022].
[19] Future Technology Devices International Limited. (2010). "FT2232D Dual USB to Serial UART/FIFO IC Datasheet," Document no. FT_000173 (v2.05), 18 April 2011, pp. 1-61. Available: http://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT2232D.pdf [accessed 11 March 2022].
[20] Digi International. (2018). "XBee/XBee-PRO ZigBee RF Modules User Guide," Document no. 90000976, 2018, pp. 1–195. Available: chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/viewer.html?pdfurl=https%3A%2F%2Fwww.digi.com%2Fresources%2Fdocumentation%2Fdigidocs%2Fpdfs%2F90000976.pdf&clen=2463502&chunk=true [accessed 24 March 2022].
[21] A. Rogers, U. Bonnes, et.al., "xc3sprog (rev. 778)," Utilities for programming Xilinx FPGAs using Xilinx Parallel Cable. Available: http://xc3sprog.sourceforge.net [accessed 11 March 2022].
[22] 1-Core Technologies. "FPGA logic cells comparison." Available: http://ee.sharif.edu/~asic/Docs/fpga-logic-cells_V4_V5.pdf [accessed 11 March 2022].
[23] Bengherbia B, Kara R, Toubal A, Zmirli MO, Chadli S, Wira P. FPGA implementation of a wireless sensor node with a built-in ADALINE neural network coprocessor for vibration analysis and fault diagnosis in machine condition monitoring. Measurement 2020;163:1–13. https://doi.org/10.1016/j.measurement.2020.107960.
[24] Kiat WP, Mok KM, Lee WK, Goh HG, Achar R. An energy efficient FPGA partial reconfiguration based micro-architectural technique for IoT applications. Microprocess Microsyst 2020;73:1–10. https://doi.org/10.1016/j.micpro.2019.102966.
[25] Pham DM, Aziz SM. An energy efficient image compression scheme for wireless sensor networks. In: Proceedings of the Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), IEEE Eighth International Conference on. IEEE; 2013. p. 260–4. https://doi.org/10.1109/ISSNIP.2013.6529799.

**Syed Mahfuzul Aziz** is a professor of Electrical & Electronic Engineering at the University of South Australia. He holds a PhD degree. His research interests include digital systems, reconfigurable processing, IC design and renewable energy. He has attracted significant funding from industry and government agencies. His achievements include the 2009 Prime Minister's Award for Australian University Teacher of the year.

**Dylan H. Hoskin** studied Electrical and Mechatronic Engineering at the University of South Australia, where he completed his Honours research project on Remote Reconfiguration of FPGA-based Wireless Sensor Nodes. Since graduating he now works as a Control Systems Engineer in the heavy-haul railway industry.

**Duc Minh Pham** is an experienced engineer with over 20 years' experience in FPGA and ASIC design and embedded digital systems development. His research interests are in efficient and high-performance architecture for Wireless Sensor Networks, DSP, Image and Video processing, and RADAR applications.

**Joarder Kamruzzaman** is a Professor of Information Technology at Federation University Australia. His research interests include wireless sensor networks, Internet of Things, and machine learning. He has published over 290+ peer-reviewed publications in journals, conferences and book chapters, received Best Paper award in four international conferences, and attracted over A$2.6m competitive research funding in Australia.