CaRDIN: An Agile Environment for Distributed Computing on Reconfigurable Sensor Networks

Xuan Sang LE, Jean-Christophe Le Lann, and Loïc Lagadec
Lab-STICC, ENSTA Bretagne, France
{xuan_sang.le,lelannje,loic.lagadec}
@ensta-bretagne.fr

Luc Fabresse
and Noury Bouraqadi
Mines Douai-DIA, Univ. Lille France
{luc.fabresse,noury.bouraqadi}
@mines-douai.fr

Jannik Laval
DISP, University Lumière Lyon 2 France
jannik.laval@gmail.com

Abstract—Sensor networks (SNs) are playing an increasingly important role in smart systems and are now one of the key technologies for the Internet of Things (IoT). An IP-based (Internet Protocol) sensor network provides a natural way to connect the SNs to the internet, and allows the sensor nodes to participate to the IoT. In these smart systems, sensor nodes often do more than simply acquiring data and thus embed more complex features. All nodes are in need of more processing capability while being limited in power consumption. Reconfigurable architectures (i.e. FPGA) offer a good tradeoff between these two factors. In this paper, we consider the challenge of programming and deploying such sensor networks consisting of hybrid nodes built on top of processors coupled with FPGAs. We introduce CaRDIN, a dedicated software/hardware platform for that system. The aim is to provide an agile environment that facilitates the development and deployment of distributed applications over the network.

Index Terms—Smart Sensor Networks; Reconfiguration; FPGA; Internet of Things; Middleware; Distributed Computing

I. INTRODUCTION

Internet of things (IoT) is a concept increasingly supported by various stakeholders and market forces. It is foreseen to be a world-wide network of interconnected devices or objects (things) through wired and wireless connections [1]. The network is in charge of providing an unique addressing scheme and creates a pervasive environment where a person can interact with the digital and physical world. One of the key technologies for the Internet of Things is sensor networks (SNs) that are playing an increasingly important role in smart systems.

Sensor networks offer a virtual layer where the digital system can communicate with the physical environment. They are deployed in a wide range of environments, such as healthcare, industrial applications, agriculture, surveillance, smart-home, etc. Programming and operating these networks rely on middlewares.

There are various proposals of middleware architecture for SNs [2] such as: virtual database, Mobile agent, Virtual Machine or Message oriented middleware. Traditionally, these middlewares are not IP-based and thus are hardly adaptable to the internet. Moreover, they do not support hardware heterogeneity in SNs. Some efforts have been reported to address these issues. [3], [4] provide an intermediate integration process for the deployment of proxies at the edge between the SNs and IP-based WAN infrastructures. Recently, IP-based sensor networks have emerged as a new research trend [5], [6] that offers a more natural and direct way to bring SNs to the internet. Web services can be used on top of these systems and offer the compatibility between SNs and standard IoT infrastructures.

On the hardware side, in smart systems (surveillance, security centric applications, etc.), the functionality of sensor nodes often exceeds simple data collection and requires complex features. These systems may need to process a large amount of raw sensor data locally (e.g. signal processing, image processing, cryptography) in order to produce a compact and rich semantic information. These nodes, referred as Smart Sensor Nodes, therefore require processing capability while focusing on keeping the power consumption low. Reconfigurable architectures such as FPGAs are known to enable parallel and fast computations within a low energy budget, hence can play a crucial role in smart sensor nodes [7].

In the context of IoT, such smart sensor nodes can be built using a FPGA coupled with a processor (e.g. ARM). The processor greatly simplifies the implementation of an IP-based software stack with its associated web services. The FPGA process the sensor data. These nodes allow to naturally construct an IP-based sensor network that is IoT standards compliant. However, such hybrid systems, that combine hardware (FPGAs) and software, are very complex from development, deployment and maintenance perspectives. This raises the need for an agile environment for efficient development and deployment of such systems, and motivates our work.

Our paper targets this issue by proposing CaRDIN, a dedicated middleware/hardware platform for reconfigurable IP-based smart sensor networks. The proposed middleware combines the IP-based and Virtual Machine (VM) architectures. It aims to: (1) ease the integration of FPGAs into the sensor nodes by providing a generic SW/HW communication interface, (2) support distributed programming of the nodes using an interpreted language (thanks to the VM), and (3)
reconfigure the nodes (SW/FPGA) over-the-air. The proposed approach provides an agile environment that facilitates the development, deployment and interaction of user applications over the network.

The remainder of the paper is organized as follows. Section II describes related work. Section III provides an overview of CaRDIN. Section IV details the architecture of a sensor node. Section V and VI present how a node can be remotely reconfigured (software/hardware) and programmed (software) in a distributed setting. Section VII shows a practical application of the prototype platform as a proof of concept and uses it to benchmark the prototype. Last, section VIII concludes and provides future works.

II. STATE OF THE ART

Many researches have explored the benefit of hardware reconfiguration in FPGA for rapid prototyping of sensor networks. Some approaches only use FPGA, either with IP-based [8], [9] or without [10], [11] softcore (e.g. NIOS) to build the sensor nodes. These approaches are well suited for fast adaptation in reaction of hardware changes on sensor nodes. However they do not offer flexible reconfiguration as provided by software approaches nor do they support over-the-air reconfiguration. In [12], [13], the authors propose a non IP-based approach along with an entire workflow to generate, remotely configure and reconfigure the FPGA. These works use a micro-controller (μC) to reconfigure the FPGA. Both software (μC) and hardware (FPGA) can be reprogrammed/reconfigured remotely from host via a wireless link (e.g. ZigBee). None of these systems is compatible with an internet usage. To support IoT in these systems, the IP-stack (as well as the web services) must be implemented on μC or on FPGA which requires specific skills. The communication using ZigBee (non IP-based) can produce high latencies and is not suitable for IoT. Furthermore, the use of a μC limits the software capability of the smart node, (IP-stack, web service, etc.). The development is entirely baremetal (μC/FPGA) which is specific and limits the reusability of the system.

On the middleware side, to provide a flexible programming paradigm to the sensor node, some approaches rely on a Virtual Machine based solution [14]–[16]. They enable the distributed programming of the sensor node and hide the heterogeneity of the runtime environment and hardware resources. Nevertheless, with these approaches, it is difficult to integrate embedded device into a heterogeneous sensor network. Some IP-based approaches [5], [6], [17] handle the heterogeneity by providing an homogeneous communication layer for heterogeneous sensors network. However, they only provide an infrastructure without further software API for the operations and interactions of sensor nodes.

None of these work (middleware/hardware) fulfils the requirements of the proposed smart sensor nodes. Our approach complements these works by combining different solutions into one unique software/hardware platform: FPGA + processor (ARM) for the flexibility of hardware and IP-stack + VM middleware for a distributed software environment on the sensor node. This allows to build a heterogeneous smart sensors network with flexible nodes that can be remotely reconfigured and programmed.

III. CARDIN—AN OVERVIEW

CaRDIN consists of a predefined software/hardware architecture and a toolset that help to efficiently build and deploy smart sensor networks. The proposed Smart Sensor Networks System (SSNS) relies on an IP network in which all smart sensor nodes are connected (directly or via relay nodes) to a base station. Each node is set up from a processor coupled with an FPGA. Reconfigurable hardware (FPGA) adds more processing ability to the node and thus enables the node to locally execute more sophisticated tasks like signal processing, cryptography, etc. Furthermore, using FPGA improves the flexibility of the node hardware. That is, the hardware can be simply tailored to adapt to different kinds of sensors. The logic inside the FPGA that collects and/or processes sensor data can be easily reconfigured when performance improvements or bug fixes are available.

Thanks to the coupled processor, a small footprint middleware is integrated on top of every node and plays three roles: (1) it allows to reconfigure FPGA and provides a flexible software API to remotely access data from FPGA; (2) it implements a software stack (i.e. web services) for the communication with the base station; and (3) it embeds a small language Virtual Machine (VM) that turns the node into a distributed programming environment. Then, sensor nodes can be remotely programmed (SW/HW at design time) and automatically reconfigured over-the-air when needed (SW/HW at run time).

---

**Fig. 1:** Workflow of CaRDIN. Developers only need to: (1) import the legacy IP to system for software/bitstream generation; (2) use the generated classes to develop their application.

---
The base station plays the role of application server that monitors, manages and collects data from the sensor nodes. Figure 1 shows the workflow of CaRDIN both at design time and runtime. At design time, the base station has a toolset dedicated to interface integration. The toolset takes a legacy IP (VHDL) as input and generates all the required interfaces to the communication between: (1) the FPGA circuit and the processor on the node, (2) the node and the base station (section V-A). This toolset outputs a bitstream and software API classes that will be deployed on the nodes at runtime. Developers can use the generated API classes along with our distributed object API to remotely access the FPGA on the nodes, from their application. From the developer perspective, the entire application is developed on the base station as a regular program, while being actually executed in a distributed manner (section VI). For that, at runtime, the base station automatically partitions the software and deploy software/bitstream on the nodes (section V).

Web services are a well-established mechanism for task invocation through internet. Since the system relies on an IP-based network, the communication between nodes and the base station are based on web services. This promotes an easy integration to other systems (e.g. IoT systems) that are built with standard (and general purpose) IT components. Two standards exist for web services: SOAP and REST. Some researches [6] have shown that, the SOAP-based solution is less flexible and has an higher overhead than the REST-based one. Therefore, the SSNS implementation in this paper uses REST as the base communication method. The REST architecture does not have the definition of common data formats. In CaRDIN, we choose JSON for data formatting since its syntax is simple and compact. JSON requires a very simple parser with less computational overhead (compared to XML), and hence is well suited for embedded applications.

IV. CARDIN SMART SENSOR NODES

Distributed development on SSNS requires an unified mechanism for remote reconfiguration and runtime access to the sensor nodes from the network. The hardware must be embed a generic communication interface that allows high level software to easily interact with the FPGA. Memory mapping is a natural way to offer this feature since it allows the software to treat the FPGA as a virtual memory region. On the other side, the middleware has to implement the network protocol and the distributed object API to allow remote reconfiguration and programming. An interpreted language VM (e.g. Smalltalk) is necessary to support distributed programming on the node.

A. Hardware architecture

Figure 2 shows our general architecture of a Smart Sensor Node. In the system, one or more physical sensors are connected to the FPGA and are controlled by a corresponding controller/processing unit (C/P unit). These units are application specific and are reconfigurable. From the processor perspective, the whole FPGA is viewed as a device file that can be mapped (byte-for-byte correlation) to a virtual memory region. Hence, each C/P unit occupies a segment in this memory. This permits the software to treat the FPGA as if it were part of the primary memory. Therefore, a register of C/P units can be read or written through a virtual memory address. To allow this on the FPGA, an addressing mechanism is needed to map each circuit register to the corresponding address.

Accessing circuit registers by address requires an address/data IO interface between the FPGA and the software framework. On the processor, this interface relies on the dedicated hardware driver. On the FPGA, we choose the Wishbone bus interface because it is simple and open-source. The principle, however, is generic and can be applied on any other bus interface (e.g. AXI4). Each C/P unit connects to a corresponding Wishbone slave that has a base virtual address assigned automatically. The slave maps each register of the C/P unit to an offset relative to its base address.

All of these slaves are connected to a shared bus controlled by a Wishbone master. This master is connected to a hardware specific wrapper close to the physical interface. This wrapper is used to adapt the physical interface to the Wishbone interface. It has to be redesigned when a new physical interface is adopted.

This IO interface additionally provides an interrupt mechanism that enables every slave to raise an IRQ signal to the processor. This allows the C/P unit to inform the processor whenever sensor data is available.

Note that, the entire interface can be generated automatically by the dedicated toolset. This process is described in the section V-A.
B. Software stack

The software stack is built on top of a lightweight Linux OS. It has two separated layers: the system layer and the API layer. The System Layer consists of a hardware driver and the IP Stack implementation. The driver on the first hand, handles the communication between FPGA and the processor. In addition, it provides a generic device special file that can be mapped to a virtual memory region. Similarly to the Wishbone wrapper circuit on FPGA, this driver is hardware specific and must be updated accordingly when a new physical interface is adopted. The IP Stack is a part of the OS and is used in the API layer to implement the communication protocol (HTTP, REST).

The second layer, named API Layer, relies on a lightweight (small memory footprint) HTTP server that supports plugins. These plugins can be loaded at runtime on demand. There are two core plugins: the REST engine and an embedded dedicated interpreted language Virtual Machine (e.g. Smalltalk VM).

The REST engine implements the web service mechanism that handles the communication with the base station. All commands from the base station (in JSON format) are decoded to software objects (e.g. Smalltalk objects) by the REST engine and passed to the VM. This VM has a dedicated API and primitives to access to the FPGA registers (via the virtual memory region) and to reconfigure the FPGA given the bitstream. Data objects from the VM can be serialized to JSON format and sent to the base station via the REST engine.

The VM along with the REST web services offers two benefits: first, since the node supports interpreted language, the software on the node can be evolved at runtime; second, the node can be reconfigured (software/hardware) remotely without the needing to restart the node. The system enables the distributed programming on the node. In other words, the software can be written on the base station but finally runs on the node.

V. OVER-THE-AIR RECONFIGURATION OF SSN

A. Toolset for software/bitstream generation

The toolset is developed using Pharo Smalltalk [18] and is based on the auto-generation code feature of the meta-model presented in [19]. This meta-model captures a subset of synthesizable VHDL structures [20]. From this meta-model, developers can express descriptions (i.e. models) of digital circuits as plain objects that can automatically be processed and exported to VHDL code if needed. Such an object-based representation of circuits eases the introduction of modifications into existing VHDL design such as adding an interface.

Figure 3 shows the workflow of the toolset. At the beginning, the hardware description (VHDL code) of the sensor controller/processing logic is imported to the system with the help of a dedicated VHDL Parser. The parser processes any valid VHDL code and automatically generates a circuit model of that design. The toolset has a predefined generic Wishbone specification. This specification takes the circuit model as input and generates the corresponding wishbone interface based on its structure. The circuit registers are also addressed automatically in this process. The final circuit model (including interface) can be automatically exported to VHDL and passed to the vendor tools to be further synthesized and produce bitstream.

The VHDL parser has been verified using some standard benchmarks: ANTRL [22], IWLS 2015 [23] and ITC’99 [24]. Our interface specification is similar to IP-XACT [21], a well-known standard for this kind of problem, but exhibits extended refactoring facilities.

Beside the interface generation process, the toolset also generates the wrapper classes, based on the user input design. These classes offer abstract software representations of the circuit on FPGA. They allow to access to FPGA registers through dedicated virtual addresses. These special classes are part of our distributed object API and enable user software to remotely access to circuit registers (FPGA) on the node. While the wrapper classes are completely written (generated) on the base station, some parts of their code can run locally while others (e.g. FPGA access) operate remotely on the smart sensor node (see section VI). This mechanism allows the entire software to be written on one place and to run distributively on many places.

The toolset assigns automatically a unique signature to the user logic and a version number to the corresponding wrapper class. These numbers enable the API to verify whether the software/bitstream on the sensor node is up to date or not. An update process will take place automatically and dynamically (at runtime) when necessary.

The interfacing specification is made at the meta-model level. As a consequence, the solution is reusable and generic, and only a change in the physical interface between the FPGA and the processor leads to a modification. In this case, the Wishbone specification must be modified to support the new physical interface. This change is made once, and then can be reused for different nodes using the same physical interface.

1 Users, however, must specify which ports of the circuit should be connected to the interface, and which ones should be wired to the IO pins [21]
If the class is not found or outdated, the base station generates it and streams it to the node. After this process, the skeleton object is correctly initialized on the node. This skeleton object takes care of FPGA access and data streaming back to the main object on the base station (see section VI). To enable this on the sensor node, a skeleton class needs to be generated and installed on the node before the skeleton object is initialized.

The reconfiguration process, illustrated in sequence diagram of figure 4, can be started at anytime when the wrapper object is created. First, the base station requests the sensor node to initialize an object of a wrapper class. The distributed objects API (DOA) on the node then finds the corresponding skeleton class and checks whether it is up to date or not (based on the class version number). If the class is missing or outdated, the base station generates it and streams it to the node. After this process, the skeleton object is correctly created.

Once the skeleton object is initialized, the base station asks for a circuit signature from the FPGA. This signature is needed to verify the presence of the last bitstream version on the FPGA. If the signature does not match, the base station streams it to the node for reconfiguration. At this point, the wrapper object is successfully initialized on the base station and ready to use.

The whole node reconfiguration process is handled by our distributed object API and is transparent to the end user. The wrapper classes or the bitstream can be changed at anytime and will be synchronized automatically to the node when needed.

**VI. DISTRIBUTED SOFTWARE PROGRAMMING ON SSN**

Distributed computing allows collaboration between objects across different address spaces of the network. These objects work together, share data and invoke (remote) methods. Regular DOAs [26], [27] require the server and client parts of the distributed program to be developed separately and deployed manually. CaRDIN offers a more dynamic distributed programming environment, with the ability to deploy software at runtime. The API allows remote and local methods to coexist in a same class. The system treats these methods as regular methods that can be used anywhere in the program. The API handles automatically the partition of both the local part and the remote part of the class at the deployment time (section V). These methods can be changed anytime. Beside, the remote part is be updated transparently if needed.

The proposed software API follows the principle of distributed objects architecture, as shown in figure 5. Remote and local methods are written in a single class (distributed class). They distinguish from another through method annotation. The distributed classes are subclasses of a special class named SSSynchronisableObject. This class handles the communication with the skeleton object (callee object) on the sensor node using the REST API. An instance of this class play the role of a caller object. Thanks to interpreted languages (e.g. Smalltalk), no server side class (skeleton) needs to be manually developed. Instead, it will be generated and deployed automatically on the node by the API (using the distributed class).

At deployment time, the software partition mechanism is based on the code annotation (e.g. pragma, method decorators, etc.) of the distributed class. All methods with an annotation will be tagged as remote methods and automatically partitioned to the generated skeleton class. The methods without annotation are local methods and thus are used locally by the caller object in the base station. At runtime, when a method is called on a caller object, the API will determine whether it is remote or not based on its annotation.

When a caller wants to perform remote call on the callee object (skeleton object), it initiates the communication with the remote skeleton. The caller arguments are then serialized to JSON format and passed to the skeleton object on the node via the REST API. On the sensor node, the REST engine receives the JSON data and reconstructs the argument objects, the corresponding method is then called on the skeleton object.

Fig. 4: If the software/bitstream is not deployed or outdated, the initialization of a wrapper (distributed) object will automatically trigger the reconfiguration of the node.

B. Over-the-air reconfiguration of sensor node

Reconfiguration of a smart sensor node consists of deploying the bitstream on the FPGA and installing the software wrapper classes on the VM. The deployment of the bitstream [25] is straightforward by streaming it from the base station to the node via HTTP, then downloading it to the FPGA using the VM. The software update process corresponds to the deployment of the remote code (skeleton classes) to support the distributed object mechanism of the base station. The key idea is that, when the base station initializes an object of a wrapper class, it creates also a "skeleton" of that object on the sensor node. This skeleton object takes care of FPGA access and data streaming back to the main object on the base station (see section VI). To enable this on the sensor node, a skeleton class needs to be generated and installed on the node before the skeleton object is initialized.

The reconfiguration process, illustrated in sequence diagram of figure 4, can be started at anytime when the wrapper object is created. First, the base station requests the sensor node to initialize an object of a wrapper class. The distributed objects API (DOA) on the node then finds the corresponding skeleton class and checks whether it is up to date or not (based on the class version number). If the class is missing or outdated, the base station generates it and streams it to the node. After this process, the skeleton object is correctly created.

Once the skeleton object is initialized, the base station asks for a circuit signature from the FPGA. This signature is needed to verify the presence of the last bitstream version on the FPGA. If the signature does not match, the base station streams it to the node for reconfiguration. At this point, the wrapper object is successfully initialized on the base station and ready to use.

The whole node reconfiguration process is handled by our distributed object API and is transparent to the end user. The wrapper classes or the bitstream can be changed at anytime and will be synchronized automatically to the node when needed.

VI. DISTRIBUTED SOFTWARE PROGRAMMING ON SSN

Distributed computing allows collaboration between objects across different address spaces of the network. These objects work together, share data and invoke (remote) methods. Regular DOAs [26], [27] require the server and client parts of the distributed program to be developed separately and deployed manually. CaRDIN offers a more dynamic distributed programming environment, with the ability to deploy software at runtime. The API allows remote and local methods to coexist in a same class. The system treats these methods as regular methods that can be used anywhere in the program. The API handles automatically the partition of both the local part and the remote part of the class at the deployment time (section V). These methods can be changed anytime. Beside, the remote part is be updated transparently if needed.

The proposed software API follows the principle of distributed objects architecture, as shown in figure 5. Remote and local methods are written in a single class (distributed class). They distinguish from another through method annotation. The distributed classes are subclasses of a special class named SSSynchronisableObject. This class handles the communication with the skeleton object (callee object) on the sensor node using the REST API. An instance of this class play the role of a caller object. Thanks to interpreted languages (e.g. Smalltalk), no server side class (skeleton) needs to be manually developed. Instead, it will be generated and deployed automatically on the node by the API (using the distributed class).

At deployment time, the software partition mechanism is based on the code annotation (e.g. pragma, method decorators, etc.) of the distributed class. All methods with an annotation will be tagged as remote methods and automatically partitioned to the generated skeleton class. The methods without annotation are local methods and thus are used locally by the caller object in the base station. At runtime, when a method is called on a caller object, the API will determine whether it is remote or not based on its annotation.

When a caller wants to perform remote call on the callee object (skeleton object), it initiates the communication with the remote skeleton. The caller arguments are then serialized to JSON format and passed to the skeleton object on the node via the REST API. On the sensor node, the REST engine receives the JSON data and reconstructs the argument objects, the corresponding method is then called on the skeleton object.

2A local method can use any remote method inside its body.
The result of the call is finally serialized to JSON and send back to the caller object.

Note that the generated wrapper classes (from the toolset) are subclasses of SSynchronisableObject. They provide all necessary methods to remotely access the corresponding FPGA circuit registers on the sensor node. These classes can be easily extended by manually adding more methods or by being subclassed.

VII. CASE STUDY

A. Camera Sensor Node Performing Image Processing

This section describes an experiment that demonstrates and validates the proposed prototype platform. The middleware is implemented to support the Smalltalk language but the principle applies to any dynamic language (Python, Ruby, etc.). The node’s hardware is based on an Armadeus APF51 Single Board Computer which adopts a Freescale i.MX515 (Cortex-A8 @ 800MHz, 512MB DDR RAM) and a Xilinx Spartan 6 (LX9). The physical interface between the FPGA and the processor is WEIM (Wireless External Interface Module) with 16-bit dedicated data/address bus.

The experiment shows an implementation of a sensor node for image processing using our platform. An OV7670 camera is connected directly to the GPIOs of the FPGA (APF51). The FPGA acquires image from camera and filters it using a HSV filter based on a color pattern. This filtered image is then used to estimate the position of object as the barycenter of the largest connected component. Figure 6 shows the simplified block diagram of the object detection circuit (at 100MHz). This circuit can be configured (via dedicated flags) to work with either VGA or QVGA or QQVGA image. The base station and the node participate to the same LAN network using ethernet.

The image processing VHDL code is imported to the toolset for interface generation and virtual address mapping. The bitstream and the corresponding wrapper class are then generated automatically and ready to be deployed on the node. Listing 1 shows the completed wrapper class. The first 5 methods (lines 1-7) are generated automatically by the toolset. The last 3 methods (lines 19-37) are developed manually for more complex processing features. At this point, no further development is needed for the node. In the developer point of view, the remote access to FPGA can be coded in a single simple class (17 active development lines). The system abstracts all the network communication, software partition, and hardware accessing that are performed transparently behind the code.

The API supports two kinds of pragma. The <remote> pragma on a method informs the caller that the method should be called remotely. When executing this kind of methods, the caller gathers all necessary arguments then passes them to the corresponding skeleton object. This latter will handle the execution and send back the result to the caller. The second kind of pragma, the <remote:....>, operates similarly to the first one, except that it allows to specify a callback for the returned data. This is especially helpful when continuously streaming data from the node to the base station. The #positionDo:n:s (line 26) method in the listing 1 uses this kind of pragma. It allows to stream a number of samples of object position (specified by anInt) to the base station at a frequency of 10 Hz (100 milliseconds). Each sample will be handled by the callback aBlock, which is a Smalltalk block closure. Since the method is executed remotely, the callback (line 31), on the opposite, is executed locally for every returned sample of data.

```
1 CameraUnitWrapper >> skelSuper
2 "DeviceMapper"
3 CameraUnitWrapper >> gateway
4 "ffvm/portal"
5 CameraUnitWrapper >> signature
6 <remote>
7 "self int16At:18"
8 CameraUnitWrapper >> x
9 <remote>
10 "self int32At:24"
11 CameraUnitWrapper >> y
12 <remote>
13 "self int32At:28"
14 CameraUnitWrapper >> position
15 |arr|
16 <remote>
17 arr <-(Array new:2).
18 arr at:1 put: self x; at:2 put: self y.
19 "arr"
```
The method #skelSuper (line 1) is used to define the superclass (e.g. DeviceMapper) of the corresponding skeleton class. All the remote methods on the wrapper class will be migrated to the skeleton class when deploying. For example, the generated skeleton class of CameraUnitWrapper is a subclass of DeviceMapper, and has 5 methods: #signature,#x,#y,#position, and #positionDo:ns:. These methods operate on the sensor node and have direct access to the FPGA register. The #gateway method enables the caller object to identify the REST resource (url) of the sensor node.

The method #stream in line 36 is a normal Smalltalk method that uses a remote method in its body. The code requires the #positionDo:ns: method to stream 500 samples of object position from the node and then simply prints each one of these on the base station.

B. Benchmarkings

At the beginning, when the node is powered on, only the software stack is running: the http server, the REST engine and the Smalltalk VM. Table I shows the static memory footprint of the software stack. This memory portion can be increased when the objects memory allocated for the Smalltalk image is full. In this case, the VM will claim for more dynamic memory allocation.

Table III shows the average percentage of the resources occupied by the software stack at different operation tests: (1) idle stage (with or without plugins). (2) The wrapper class is deployed on the Smalltalk image. (3) The bitstream is configured on FPGA. (4) The base station continuously fetches 500 samples of object position from the node with the rate of 10Hz. (5) The node streams 500 samples of object data to the base station at a frequency of 10Hz.

At the idle stage (with or without plugins), the software stack has no impact on the CPU consumption. The HTTP server takes around 0.3% of memory, up to 0.5% when the system is fully loaded. The memory used is nearly constant when the node is in operation mode.

The hardware reconfiguration takes most of CPU resource in the 5 tests. This process consists in streaming the bitstream from the base station to the node, then deploying the bit stream on the FPGA. Since it invokes some kernel modules to communicate with the FPGA, the CPU has a work overhead (26.2%). After the reconfiguration, the FPGA starts its processing and is ready for communication. The table II shows the resource used by the object detection circuit on the FPGA.

The result of the last two tests is particular interesting. These tests carry the same operation: fetching 500 samples of object position from the node at the rate of 10 Hz. The frequently fetching test requires to open repeatedly a connection between the node and the base station (500 connections). These connections add more overhead to the CPU consumption (21%). The streaming test involves only one connection, then continuously streams data via that connection. Therefore, the process consumes averagely only 5% of the CPU resource.

Figure 7 shows the network load on the node in different active operations. Since the software/hardware reconfiguration requires files transfer, the bandwidth used for this operation goes up to 235 KiB/s for reception (RX) and 9.58 KiB/s for transmission (TX) in around 2 seconds. Others operations require the bandwidth of less than 20 KiB/s for RX and less than 5 KiB/s for TX. Obviously, the frequently fetching mode uses more network resource than the streaming mode.

This experiment uses an existing VHDL code for the image processing on the FPGA. But the scenario can be expanded by starting with a pure software image processing implementation (e.g. in C). Modern high level synthesis tools are able to synthesize this code to RTL. Our toolset can then handle the generated VHDL.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we introduced CaRDIN, a platform for efficiently developing and deploying IP-based SSNS. Our proposal uses an FPGA, that offers high performance while reducing power consumption. We show that an hybrid hardware system (FPGA + processor) along with a web service oriented software platform enables remote reconfiguration/re-programming of sensor nodes. The use of web services fa-
Sensor networks consist of many smart nodes and have the distributed applications on a Smart Sensor Network. Such an ultimate goal is to use the prototype platform to deploy truly experiment in this article is bound to a proof of concept. The can be completely reconfigured/reprogrammed at runtime. The supporting the partial reconfiguration on FPGA. This allows to the base station using the distributed objects API.

To demonstrate the proposed platform, we have built a smart sensor node on an APF51 board as a proof of concept. This node connects to a camera and performs an object detection algorithm on the FPGA. It communicates the object position to the base station through the distributed software API hides all the details of node operations and interactions and let developers focus on the functionalities of their application.

To demonstrate the proposed platform, we have built a smart sensor node on an APF51 board as a proof of concept. This node connects to a camera and performs an object detection algorithm on the FPGA. It communicates the object position to the base station using the distributed objects API.

Regarding future work, we plan to improve CaRDIN by supporting the partial reconfiguration on FPGA. This allows to have a totally dynamic node where the software/bitstream can be completely reconfigured/reprogrammed at runtime. The experiment in this article is bound to a proof of concept. The ultimate goal is to use the prototype platform to deploy truly distributed applications on a Smart Sensor Network. Such sensor networks consist of many smart nodes and have the capacity to simply incorporate new nodes into the system.

REFERENCES


Fig. 7: Network load of the node on different operations: (7a) the software/bitstream reconfiguration process ([t, t + 2]); (7b) the frequently fetching test ([t, t + 70]) and lastly (7c) the streaming test ([t, t + 70]). t is the time when an operation begins.
[21] “Ieee standard for ip-xact, standard structure for packaging, integrating,
and reusing ip within tool flows,” IEEE Std 1685-2014 (Revision of IEEE
[22] T. Parr and K. Fisher, “Ll(*): the foundation of the ANTLR parser
generator,” in Proceednings of the 32nd ACM SIGPLAN Conference
on Programming Language Design and Implementation, PLDI 2011,
San Jose, CA, USA, June 4-8, 2011, 2011, pp. 425–436. [Online].
Available: http://doi.acm.org/10.1145/1993498.1993548
[24] F. Corno, M. Reorda, and G. Squillero, “Rt-level itc’99 benchmarks and
http://docs.oracle.com/javase/1.5.0/docs/guide/rmi, 2012.
[27] Cincom-Systems, VisualWorks ® Distributed Smalltalk Application