[ES6154 Report]: Javascript to FPGA compiler

Aditya Sundararajan
School of Computer Engineering
Nanyang Technological University
50 Nanyang Avenue, S639798
Email: ad0001an@e.ntu.edu.sg

Milin Hariharakrishnan
School of Computer Engineering
Nanyang Technological University
50 Nanyang Avenue, S639798
Email: milin001@e.ntu.edu.sg

Navneet Ramachandran
School of Computer Engineering
Nanyang Technological University
50 Nanyang Avenue, S639798
Email: navneet001@e.ntu.edu.sg

***Abstract*— A Javascript-based programming and code deployment environment can make Zynq-based FPGA accelerators available to an emerging class of embedded IoT developers allowing high level abstraction of complexities involved in code development. We first show how to use Javascript proxies to automatically generate C/C++ for high-level synthesis. After high-level synthesis the C/C++ code will then be converted to bitstreams. Next, we develop a bitstream distribution and delivery framework using an end-to-end Javascript-based environment. The objective is to achieve a server-client model capable of programming and loading bitstreams to any device in IoT network on-the-fly.**

I. Introduction

Today, with the advent of miniaturization of electronics and efficient communication technologies, we live in a world surrounded by devices running on micro-computers which can communicate with each other. Devices, which were never imagined to have a 'brain' of their own, now possess processing powers much powerful than some of the top-class computers of the previous century. This, inter-networking of devices is aptly termed as Internet of Things (IoT). IoT is truly a revolution, in the sense, that a user can access a plethora of devices at the convenience of a laptop or smartphone.

Embedded hardware development has traditionally been the exclusive recluse of a small band of programmers and hardware developers. Unlike desktop, server and even mobile development, the embedded platforms considered in this study are far less powerful and the programming platforms and environments far less developer friendly. Yet, as we enter the IoT (Internet of Things) era, we must prepare to widen the audience for developers targeting these IoT platforms.

By 2020, the number of smartphones tablets and PCs in use will reach about 7.3 billion units. In contrast, the IoT will have expanded at a much faster rate, resulting in a population of about 26 billion units at that time [1]. Thus, to capitalize on this technology it is essential to invest in its research and development and understand the underlying architecture of the system. The IoT is a convergence of multiple technologies, ranging from wireless communication to the Internet and from embedded systems to micro-electromechanical systems (MEMS). Not everyone is exposed to such sophisticated technologies, therein lies the problem of commercialization and development of IoT. Brown University computer scientist Michael Littman has claimed that successful execution of the Internet of Things requires consideration of the interface's usability as well as the technology itself. These interfaces need to be not only more user friendly but also better integrated: "If users need to learn different interfaces for their vacuums, their locks, their sprinklers, their lights, and their coffeemakers, its tough to say that their lives have been made any easier." [2] Thus, we are posed with the following questions: How do we bring forth this technology to every potential user, irrespective of their background and understanding of embedded systems and wireless communication? How do we provide users with powerful platforms like the MicroZed with which they can maximize the capabilities of their IoT systems?

In this project, we develop FPGA.js, a friendly high-level Javascript library targeting low-power FPGA-based SoCs. Javascript is primarily aimed at web-developers which outnumber hardware developers by a 100:1 ratio. Javascript runtimes are actively supported by Apple, Google, Mozilla, and many other companies and has seen dramatic 20-100x performance improvement over the past decade (note that this rate is faster than Moore's Law). While ECMAScript6 has proposed GPU accelerator and SIMD NEON support for Javscript running on embedded devices, there is no direct support for FPGA acceleration. This is no surprise, as FPGA programming is hard and out of the reach of most Javascript programmers. We aim to bridge this gap by developing FPGA.js, a library for (1) streamlining distribution of FPGA bitstreams to IoT devices exploiting node.js push model, and (2) using ECMAscript proxies to extract HLS-friendly C code from dataflow Javascript descriptions that can then be synthesized into hardware bitstreams.

Based on the streaming capture of dependencies in a Node.js framework, and our FPGA.js library for Javascript, we provide a convenient front-end for loading bitstream blobs directly on FPGAs. This provides preliminary support for compiling Javascript descriptions into C/C++ code for HLS compilation using proxy object. The proxy object is used to define fundamental operations which will enable translation of Javascript code to C/C++ code. In the context of IoT, the compiler enables on-the-fly code conversion and hot swappable bitstreams.

In this project, we make the following contributions: 1. Development of FPGA.js for describing dataflow computations using the ECMAscript Proxy handling support. 2. Design and engineering of an FPGA bitstream distribution flow.

The remainder of this paper is structured as follows: Section II presents our implementation and the methodology used.

The JavaScript proxies, the flow from C/C++ code to FPGA bitstreams, and communication between server-client has been explained in detail in Section III. In Section IV, we present a brief idea about the various approaches we took, before arriving at the current final implementation. Finally, the results of our project are presented in Section V.

II. Background

With the promising trend of IoT and the need to connect every device to each other and to the internet, JavaScript is gaining popularity in the Embedded domain as a gateway to provide web interfaces. Projects like Tessel, Espruino and Kinoma Create have showed promise in using JavaScript as the programming language for their platforms. Though they have been successful in reaching out to a larger crowd of hobbyists and enthusiasts, using a scripting language has its restrictions. Having said that, there is a huge potential of JavaScript on embedded devices to provide a simplified approach to embedded programming. This idea has just spawned and by actively contributing, we can help shape how JavaScript can be used in embedded systems in the future.

JavaScript, a dynamic computer programming language which is commonly used in web development is the language we chose for the users to realize their designs on the MicroZed. JavaScript has become one of the most popular programming languages of the Web. As already mentioned, the number of JavaScript developers exceed the number of embedded developers by a very high ratio. Thus by choosing JavaScript, we are making this technology available to all the web developers around the world. Since JavaScripts syntax is based on C, it is a natural choice of language to be translated into HLS friendly C. JavaScript is based on the ECMA scripts, which is a prototype based, functional, imperative scripting language. We use the proxy object which is a part of the ECMA6 (Harmony) proposal. The Proxy object is used to define custom behaviour for fundamental operations. The Proxy object uses traps method to provide property access. Below is the syntax of a simple Proxy variable p:

```
var p = new Proxy(target, handler);
```

Here the target object can be any sort of objects, including a native array, a function or even another proxy or function to wrap with Proxy. The handler is an object, whose properties are functions, which define the behaviour of the proxy when an operation is performed on it. We use the handler function to translate the user defined JavaScript code (our target) into custom C code for HLS compilation. The sequence of actions within the handler function is the heart of our interpreter. For the delivery model, we employ a node.js network application. Node.js is an open source, cross-platform runtime environment for server-side and networking applications. Node.js applications are written in JavaScript, and can be run within the Node.js runtime on several platforms. We use the node.js stream model to exchange data between the client and server. A stream is an abstract interface implemented by various objects in Node. For example, a request to an HTTP server is a stream, as is stdout. Streams are readable, writable, or both. In our case, we use streams to share files (bitstream) between the server and the clients. Node's modules have a simple and elegant API, reducing the complexity of writing server applications. The Socket.io framework allows creation of a simple web server that handles the functionality of file transfer.

Similar projects like the Connectal by Cambridgehackers have been trying to provide communication between the FPGA and the user logic. Currently the connectal framework provides a hardware-software interface for applications split between the user mode and custom hardware in a FPGA. The framework only provides an interface between the software and hardware modules and does not support compilation of the entire software module onto the FPGA, which is the focus of our project.

III. Idea

With FPGA.js, we provide solution to users who want to utilize platforms like MicroZed in their IoT framework. The framework has two functionalities: 1. Translating user input JavaScript code into HLS compatible C/C++ code and generating a bitstream using the Xilinx ISE. 2. Flashing a bitstream onto any device connected over the IoT network, using a web server interface. Fig. 1 represents the FPGA.js framework. To provide these functionalities we have divided our project into four sections: 1. GUI, Web Server and JavaScript Proxies to compile the user code to C/C++, 2. FPGA Bitstream generation and 3. Client-Server delivery model using node.js.

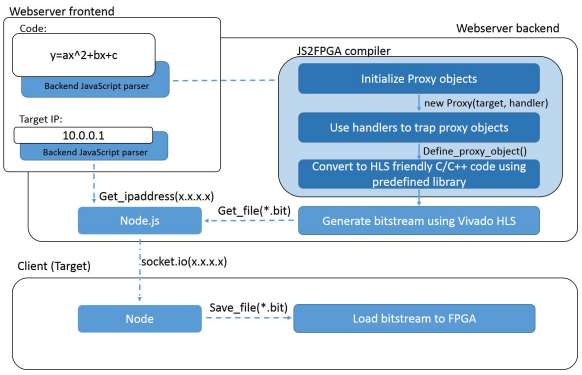

Fig. 1: FPGA.js framework

A. GUI, Web Server and JavaScript Proxies

The web interface allows the users to enter JavaScript(JS) code on a text box within a HTML page, as well as an option for uploading existing JS code. Also, the web page will show a list of previously saved JS codes, so that the user can modify the existing code or complete the unfinished code. When the user clicks on the 'Run' button, the web server backend will run JavaScript proxies that will perform code translation to C. The web server and Database are created using LAMP. LAMP, short for Linux, Apache, MySQL and PHP, is an open-source Web development platform, also called a Web stack,

that uses Linux as the operating system, Apache as the Web server, MySQL as the RDBMS and PHP as the object-oriented scripting language [3]. Once the user asks to run the code, the JavaScript Proxies will convert the user's code into C. This C code, as well as the user entered JS code, will both be saved in the database which resides in a x86 machine. The web interface also has a provision for selecting the device to be programmed, thus giving flexibility to the user, and providing a true IoT environment. The web interface shows a list of previously generated bitstreams. This enables the user to re-program the device using existing bitstream without the need to code it again.

B. Bitstream generation

Vivado High-Level Synthesis is then used to convert the generated C/C++ code into VHDL/Verilog code. High-level synthesis (HLS), sometimes referred to as C/C++ synthesis, electronic system-level (ESL) synthesis, algorithmic synthesis, or behavioral synthesis, is an automated design process that interprets an algorithmic description of a desired behavior and creates digital hardware that implements that behavior [4]. VHDL/Verilog file generated by Vivado HLS, is then passed to Xilinx ISE to generate FPGA bitstreams. The bitstreams are the configuration data to be loaded into an FPGA. Xilinx ISE (Integrated Synthesis Environment) is a software tool produced by Xilinx for synthesis and analysis of HDL designs, enabling the developer to synthesize ("compile") their designs, perform timing analysis, examine RTL diagrams, simulate a design's reaction to different stimuli, and configure the target device with the programmer[5]. Scripts are run on the x86 machine which perform the entire operation from C to VHDL to bitstream generation when the user selects the 'Generate bitstream' option on the web interface.

C. Client-Server delivery model

The delivery model used is a server(x86 machine) - client(device in IoT) model using the node.js runtime network application. The node.js framework establishes a socket connection between the server and the clients to enable bitstream sharing. The server maintaines a static list of the clients that are availalbe in the network. The user through the web interface can decide which bitstream has to be programmed onto a specific client avaiable on the network.

IV. Methodology

The challenges presented by the dynamics of modern computing problems and the complexity of their evolution demands the deployment of innovative and creative solutions. The scale, precision, flexibility and interactivity required for reproducing these solution demands a strict methodology to be followed. In the subsequent sections we describe the methodology to enable anyone to build and reproduce our compiler.

TABLE I: Hardware requirements

Type	Architecture	Memory	Operating System
Server	Intel core i5 x86-64 @ 2.2Ghz	4GB DDR3 RAM	Ubuntu 14.4 LTS
Client	Zynq-7000 AP SoC	1GB DDR3 RAM	Xilinux 1.1

A. Infrastructure

The infrastructure required can be split into two sub sections namely server and client as shown in Table **??**

The server is an Intel corei5 x86-64 based machine with 4GB DDR3 RAM, running Ubuntu 14.04 LTS and connected to a network. The network can be LAN for local setup or internet for true IoT deployment. The client can be a low-cost FPGA development board like MicroZed board running Ubuntu 12.04 based Xilinux OS. The board which is based on Xilinx Zynq-7000 All Programmable SoC, specifically the XC7Z010-CLG400-1 variant of SoC, consists of generic ARM cores and user-configurable FPGA space for customizable performance.

B. Backend compiler software

The JS2FPGA compiler and supporting software run in the backend of server to create an intuitive development environment for programmers and users in general. To make it easy to reproduce the server-client environment, Table **??** specifies the software version and specifications running in the server and client.

TABLE II: Software Requirement

Type	Server	Client
Browser support	Yes, using firefox v36.0.1	NA
Delivery model	node.js v	node v
Delivery support packages	stream.io, stream-server	stream-io, stream-client
Code translator	JS2FPGA compiler	NA
HLS	Vivado HLS	NA

The development environment used was Ubuntu 14.04 LTS. The following softwares are needed for development and subsequent deployment of the compiler in IoT: node node.js stream-io stream-server stream-client Vidado HLS Firefox x86 v36.0.6

The support packages and their specific versions are available at our git repository [6].

C. Delivery model

Once the setup is created, the client needs to get connected to server. To connect to server, open command promp in client and type :

"node client.js"

Once the script is run, open a command prompt at server side and type:

"node.js server.js"

Now both server and client are connected. The user has to now open firefox web browser and go to the IP address of the web server. The user can then type the code. The backend framework will parse the user input and convert it to C/C++ code using proxy objects.

V. REFERENCES

[1] Gartner Says the Internet of Things Installed Base Will Grow to 26 Billion Units By 2020 - http://www.gartner.com/newsroom/id/2636073

[2] Littman, Michael and Samuel Kortchmar. "The Path To A Programmable World". Footnote 14 June 2014

[3] http://www.webopedia.com/TERM/L/LAMP.html

[4] Springer Book High Level Synthesis From Algorithm to Digital Circuit ISBN 978-1-4020-8587-1

[5] https://en.wikipedia.org/wiki/XilinxISE

[6] http://codeventure.sce.ntu.edu.sg/stash/projects/CE4054P/repos/js2fpga/browse