A Software/Hardware Co-Design Framework for the ‘Internet of Eyes’

Cathal Garry, Derek Molloy
Entwine Centre, School of Electronic Engineering, Dublin City University, Ireland
cathal.garry3@mail.dcu.ie, derek.molloy@dcu.ie

***Abstract*— This paper examines the challenges involved in bringing real-time image analysis to the Internet of Things (IoT) and thereby develops a software/hardware co-design framework that takes account of the power and computational requirements of IoT edge devices. Current state-of-art solutions are typically suited to applications that require low power or low latency image analysis, but not both. This paper describes an architecture that can be used to perform low latency, low power processing on an edge device, thus making it suitable for wireless ‘Internet of Eyes’ (IoE) applications. Such applications require the processing of vast amounts of high-resolution video data in order to extract small amounts of salient information, which can then be transmitted to cloud platforms using low-bandwidth network communications protocols. This novel approach is tested by applying it to a real-time vision-based distributed motorway vehicle counting application, and evaluated for suitability to an energy harvesting deployment.**

***Index Terms*— Internet of Eyes, Internet of Things, Image Processing, Computer Vision, Linux, MQTT, Programmable logic, SDSoC**

I. Introduction

This paper examines the challenges of bringing ‘eyes’ to the Internet of Things (IoT) in real time. Image analysis applications typically involve significant computational processing, but IoT devices are often needed in remote power-isolated locations and thereby have limited computational capability. This means many IoT applications do not perform image processing in real time at the edge of the network, and instead transmit image data to cloud services for processing. The low-bandwidth wireless configuration of many IoT applications means that cloud processing is often infeasible. A number of image processing edge solutions are available, but they are suited to applications that either require low power consumption or low latency image processing, but not both [1] [2].

In order to accommodate IoT applications that require real-time image processing and low power consumption, this work investigated software-defined system-on-a-chip (SDSoC) platforms. An SDSoC is an integrated circuit (IC) that contains a processor, a number of peripherals, and some amount of programmable logic (similar amounts to a field programmable gate array [FPGA]). The programmable logic can be used to add new custom hardware components to the IC using software. This effectively allows software to redefine hardware, either before, or dynamically during run-time operations. This paper describes an architecture that uses an SDSoC solution to enable edge image processing for IoT applications.

II. Prior Work

Current research in this domain is largely focused around the use of cloud computing or discrete graphics processing units (GPUs). Cloud computing involves offloading large amounts of computation to a remote server. This approach can be quite useful in general-purpose IoT applications with low capacity edge devices. Assuming sufficient communications bandwidth is available, this approach can still suffer from large latencies while processing data, largely due to data access times and congestion on a network [1]. The discrete GPU solution has an opposite trade off, whereby it offers very low latency image processing but requires high power consumption [2]. These solutions are suitable for a large number of IoT applications, but they are unsuitable for IoE applications where both low power consumption and real-time image analysis is required.

Other solutions that offer low power consumption and real-time image processing are FPGAs [3] and neural network hardware platforms, such as the Movidius (Intel) Neural Compute Stick [4] [5]. Such solutions provide the capability to process data at the edge of the network, however, there is significant complexity involved in layering such hardware platforms with an IoT communications stack and network connectivity, which is essential for IoT devices and is commonplace on processor-based hardware. Another solution that can overcome these disadvantages is an Application Specific Integrated Circuit (ASIC). An ASIC solution offers a custom design that meets many of the requirements; however, the cost and development time for such a solution is immense. The Xilinx Zynq SDSoC is chosen for this paper as it offers similar advantages to an ASIC, but also provides a mechanism for ease of processor-integrated software development using the Xilinx tool suite [6]. As illustrated in Fig.1, the number of steps required for non-SDSoC software development is far greater than that of the SDSoC solution. This reduces the complexity in developing a hardware/software co-design. The SDSoC solution also includes network connectivity, which is necessary for IoT applications.

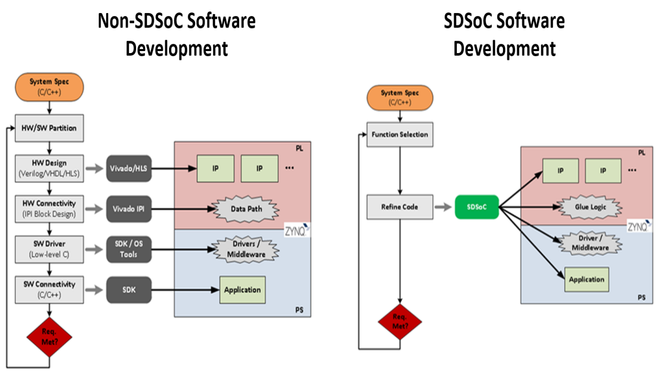


Fig.1. – Comparison of software development between non-SDSoC and SDSoC solutions [6]

SDSoCs are a relatively new technology and much of the existing research on this platform is based on their use for general-purpose image processing [7]. They have not yet been applied to full-stack IoT applications, hence the novelty of the IoE architecture described in Section III of this paper.

III. Technical Description

Fig.2 shows an overview of the architecture described in this paper. This architecture consists of three main components:

1) The Producer: Performs image processing and analysis, along with other local processing on the video data. It creates small messages that have high information content for transmission to the network.
2) The Handler: Is used as an intermediate agent between the producer and the consumer. The handler can also be used to provide some additional processing if needed (e.g., aggregate data from several producers) and can be used for long-term information storage.
3) The Consumer: An IoT device that acts on the information received from the handler (e.g., an actuator, display).

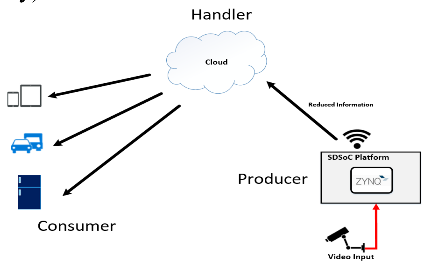


Fig.2. – Overview of the high-level architecture

This high-level architectural design was chosen because it is suited to IoE applications where large amounts of video data is reduced to only salient information. The architecture is also scalable, allowing the number of producers and consumers to be increased as needed. For this work the SDSoC chosen for evaluation as the producer was the Xilinx Zynq chipset. The handler illustrated in Fig.2 could be implemented using a number of different devices ranging from a PC to a high-end hosted server. Similarly, the consumer devices could be implemented using a range of IoT devices containing a number of sensors and actuators. The exchange of information rich short messages means that the communication between the three major components can be implemented using a number of different communications protocols, including those for ISM radio band LPWAN (including NB-IoT, Sigfox).

To evaluate this architecture as a solution for bringing eyes to the IoT in real time, an application use case was selected. The chosen application is a variable speed limit (VSL) controlled motorway, as it is a typical example of an IoE application that processes a large amount of raw video data (24-bit HD video at 60fps requires approx. 3Gbit/s) in order to produce a small amount of salient information. In this application the raw data is video from a HD camera and the salient information is an estimate of the traffic flow, which is described as a vehicle count per second. This approach is implemented using a distributed image analysis solution, as a controller can best make decisions based on multiple adjacent locations along a motorway route. The following devices were selected for the major components of the architecture implementation:

1) The Producer: The SDSoC chosen for this application is the Xilinx Zynq SoC.
2) The Handler: For this application the handler is implemented on an Ubuntu Linux virtual machine.
3) The Consumer: The consumer in this application is implemented on a Raspberry Pi SBC.

The communication protocol selected for this implementation was message queuing telemetry transport [8] which was transmitted over a Wi-Fi network. MQTT transmits data using a publisher/subscriber model. As outlined in Fig.3 this model is made up of three components:

1) Publisher: This publishes data to the broker.
2) Broker: Acts as an intermediate stage between the publisher and subscriber.
3) Subscriber: This subscribes to some data on the server.

This type of model differs from the regular client/server model where data can be transmitted in either direction. Instead, the flow of data with MQTT only goes from publisher to subscriber, via an intermediate broker. This model is well suited for applications where a decision is made using data from multiple locations. MQTT offers very small packet sizes when transmitting data (with a minimum packet size of two bytes). It also has a number of levels of quality of service (QoS) available when sending packets. Another useful feature of MQTT is the last will message. A last will message is a message sent by a broker on behalf of a publisher, when that publisher unexpectedly stops publishing. Such architecture support for catastrophic failure is important for this particular implementation.

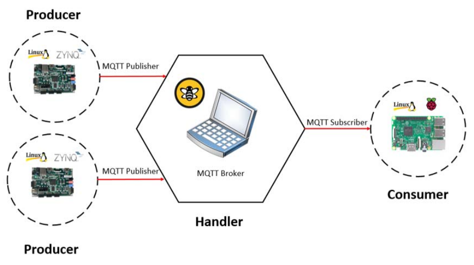


Fig.3. – Overview of VSL Controlled Motorway

The next subsections examine how each of the three major components are implemented.

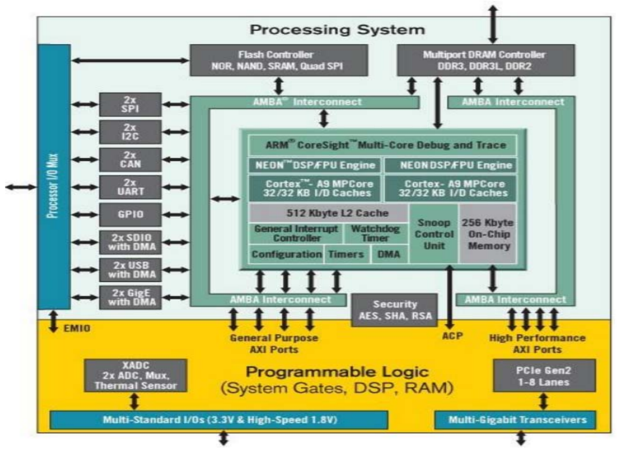
A. *The Producer*

The Xilinx Zynq SDSoC architecture outlined in Fig.4 consists of two major elements:

1) The processing system (PS) that contains the ARM CPU core, memory and peripherals.
2) The programmable logic (PL) that contains FPGA logic, which can be used by software to create new custom intellectual property (IP) and hence redefine the system hardware.

The separation of the PS and PL subsystems in the Zynq SDSoC is used to allow for flexible integration of different components or IPs in the PL. The PS and PL are integrated using AXI interfaces, which are defined in the ARM AMBA specification [10]. These interfaces can be added as needed, depending on the components and IP in the PL.

The PYNQ board was chosen for the implementation [11]. It is a PCB that integrates the Zynq SoC and includes a number of peripherals (such as HDMI and USB). This implementation involved the development of custom IP in the PL of the Zynq SoC. This IP block takes in an input image of a motorway and, using image analysis techniques, determines if a vehicle has crossed an arbitrary line in each lane, as illustrated in Fig.5. The result of this image processing is read by the PS which then transmits information about the number of cars passed as well as the amount of congestion for a given sample. The next few subsections will describe in more detail how this was achieved.



Processing System
Flash Controller NOR, NAND, SRAM, Quad SPI
Multiport DRAM Controller DDR3, DDR3L, DDR2
AMBA Interconnect
ARM CoreSight Multi-Core Debug and Trace
NEON DSP/FPU Engine
Cortex-A9 MPCore 32/32 KB I/D Caches
512 Kbyte L2 Cache
General Interrupt Controller
Watchdog Timer
Configuration
Timers
DMA
Snoop Control Unit
256 Kbyte On-Chip Memory
2x SPI
2x I2C
2x CAN
2x UART
GPIO
2x SDIO with DMA
2x USB with DMA
2x GigE with DMA
Processor I/O Mux
Security AES, SHA, RSA
EMIO
General Purpose AXI Ports
ACP
High Performance AXI Ports
XADC 2x ADC, Mux, Thermal Sensor
Programmable Logic (System Gates, DSP, RAM)
PCIe Gen2 1-8 Lanes
Multi-Standard I/Os (3.3V & High-Speed 1.8V)
Multi-Gigabit Transceivers

Fig.4. – Summary overview of Zynq architecture [9]

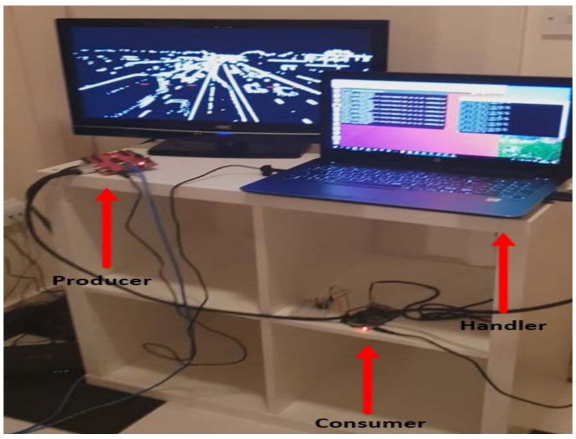


Fig.5. – Line crossing image processing Motorway application. See http://tiny.cc/InternetofEyes

A video outlining the full test demo can be found at the link here: http://tiny.cc/InternetofEyes

1) Processing System (PS)

The PS used in the Zynq chipset is made up of an ARM Cortex A9, a memory subsystem, and a number of peripherals. In this implementation, the PS was configured to run Ubuntu Linux [11]. This Linux image supports libraries and device drivers that allow the PS to interact with the PL using a number of AXI interfaces. These are vital to a generalized IoE architecture, as otherwise a custom shared memory solution would be required for each application.

The PS in this implementation is used to read the result register from the custom image processing IP block in the PL. This register value indicates whether a vehicle is currently present or not present on a count line in each motorway lane. Each lane on the motorway is represented by a single bit in the returned result value. The PS then builds up a series of these results in order to determine if a vehicle has crossed the line and hence increments the vehicle count. A determination of the amount of congestion on the motorway is measured by the amount of time it takes for a vehicle to cross this line. At a fixed interval of time the PS publishes the latest results for the count and congestion level to the MQTT broker. The PS implementation is performed using Python, due to the availability of support libraries, but real-time performance is still possible. The MQTT publisher is developed using an open-source client implementation called paho-mqtt [12].

2) Programmable Logic (PL)

The PL is responsible for performing the image analysis necessary for measuring traffic on the motorway. This image processing IP was developed in Vivado HLS. Vivado HLS is a development tool for creating custom IP for Xilinx FPGAs. It allows the user to create and simulate IP using the C or C++ programming languages. Once complete, the tool uses synthesis to convert the C/C++ code to a hardware description language (HDL) like Verilog or VHDL, which can then be implemented in FPGA logic.

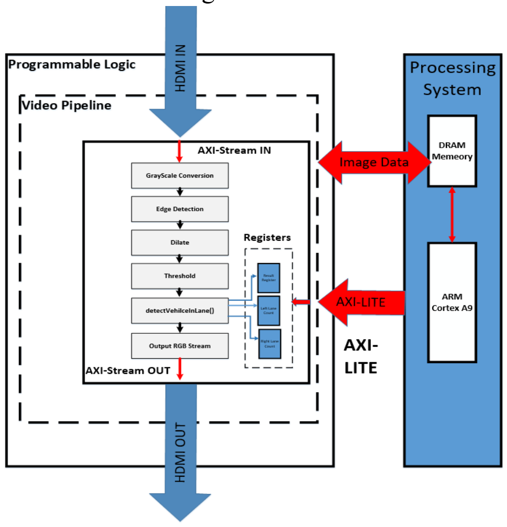


Fig.6. – Overview of vehicle counting IP

The IP developed using Vivado HLS was the vehicle counting IP. An overview of the structure of this IP is outlined in Fig.6 and described below:

1) Grayscale conversion: Reduces the RGB input image to a single gray scale image.
2) Edge detection: Uses Sobel filters to perform convolution using a 3×3 kernel to find the edges in the image. This is used to find the outline of the vehicles on the motorway.
3) Dilate: Convolution using a 3×3 kernel to dilate the black region in the image by increasing the white areas further.
4) Threshold: Sets pixels below a threshold to 0 and all other pixels to a fixed value. Used to remove non-distinct edges.
5) DetectVehicleLane: Custom function that uses six regions of interest to determine if a vehicle is currently crossing the count line on a motorway. The result register is then updated using a one-hot encoded vector.

The video data is inputted and outputted from this IP using an AXI-stream interface, and the registers are accessed using a standard AXI-Lite interface. Both of these interfaces are defined in the ARM AMBA specification [10]. The next section describes how this IP is integrated into the SDSoC so that it is available to the PS.

3) Overlays

Once a custom IP has been created using Vivado HLS, it can be exported to a user repository. Xilinx's Vivado tool is used for the integration of custom IP into the SDSoC subsystem. The hardware structure of an SDSoC is defined, and once complete, a bitstream can be generated for use on the SDSoC platform. A bitstream is a series of binary values that are used to control the functionality of the programmable logic. These values can either be loaded from memory when booting the platform (e.g., via U-Boot) or dynamically by the processor at run time (e.g., via Linux userspace). The hardware subsystem in this work uses a video pipeline in the PL to control the flow of data in to and out of the two HDMI ports on the platform. Internally, the custom image processing IP has been inserted in the video pipeline and the data from the HDMI ports are passed through it. The PYNQ platform has a custom method of loading these new hardware subsystems called *overlays*. Overlays are used by the software on PYNQ platform to load the generated bitstream along with a description file of the hardware subsystem. Once a new overlay has been loaded, software is aware of any new hardware available and can interact with it.

B. *The Handler*

The demonstrator handler is implemented on a Linux VM. The handler acts as an intermediate agent, storing any messages from the publisher until they can be sent to the required number of subscribers. The MQTT broker is implemented using the open-source broker application Mosquitto [13].

C. *The Consumer*

The demonstrator consumer is implemented on a Raspberry Pi, a low-cost, low-power, basic SBC running Raspbian. The consumer subscribes to the broker topics using Paho MQTT [12]. The consumer processes the topic messages and can vary the motorway speed accordingly. For example purposes, an actuator circuit is connected to the consumer that flashes an LED when either an accident has occurred or one of the publishers has sent a last will message, indicating error state.

IV. Results Obtained

To test this work, three different image processing IPs were used in order to determine the variation in power and response times across varying levels of algorithmic complexity. These custom IP blocks include the following:

1) Grayscale Filter: This converts a red, blue and green image to a grayscale image.
2) Edge Detection Filter: This outputs an image with all the edges detected in an image.
3) Vehicle Count: This IP block is used for monitoring the traffic in a motorway image.

The level of complexity of these IPs increases from one IP to the other with the grayscale being the lowest and the vehicle count being the highest. Testing was performed on three different devices in this work; the PYNQ board, the Raspberry Pi and an Intel-based HP laptop. The next section describes the main areas of testing.

A. *Power Consumption*

Power consumption was measured using a USB power monitoring tool. Fig.7 (along with the description key in Table I) illustrates the power consumption of the PYNQ platform across a number of different PL and PS configurations.

TABLE I: Description of Test Conditions

Test Condition	**Description**
None	There is little or no programmable logic being used, only the ARM Cortex A9.
HDMI Pipeline Only	Only the video pipeline is being used in the PL.
HDMI Pipeline with image processing IP	The video pipeline has been modified to include a custom IP in the PL.
Idle	The ARM Cortex A9 is idle.
MQTT Transmit	The ARM Cortex A9 is being used to transmit MQTT messages.
Grayscale Conversion	The ARM Cortex A9 is being used to convert RGB images to grayscale images.

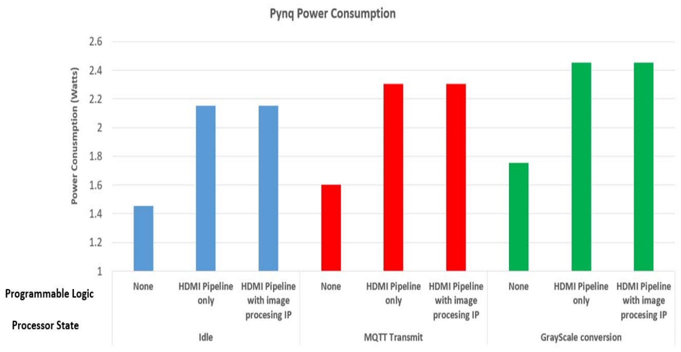


Fig.7. – Platform power consumption (see Table I) under varying conditions

The second series of power consumption tests were performed across the entire platform, whilst it is performing a number of different image processing tasks. All results in Fig.8 are measured while processing full HD data (1080p) at 60 fps, in order to obtain the worst case scenarios.

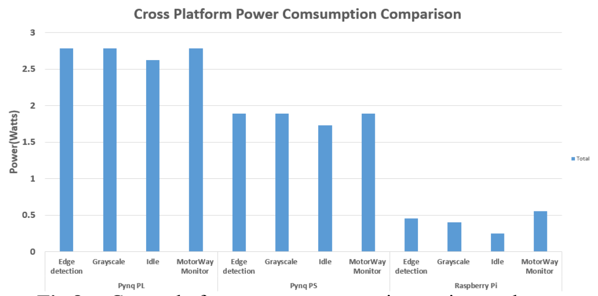


Fig.8. – Cross-platform power consumption against each test

For reference purposes, a power consumption estimation was also carried out on the HP laptop. This estimation found the laptop had a magnitude of roughly 16× greater power consumption when compared to any of the platforms shown in Fig.8.

B. *Image Processing Response Times*

Image processing response times were measured on the PYNQ PL, the PYNQ PS, the Raspberry Pi and the HP laptop. The testing of the PYNQ PS, the Raspberry Pi and the laptop was carried out by measuring the execution time in C++. The response time of the PYNQ PL was measured by labeling the frames of the input video and using a high speed camera to capture the difference between the input image and output image from the PL. The video used to measure these response times had a frame rate of 25 frames per second (fps). Fig.9. illustrates the response time for processing an individual frame in the test video.

All of the solutions tested in Fig.9 have a varying response time across the different tests. The only exception is the programmable logic in the PYNQ platform, which has a constant response time of 40ms.

C. *Overlay Switching Time*

The architecture design allows for the image processing PL bitstream to be switched to a different bitstream at run time. For example, a different image processing algorithm may be required at night time than during the day. This is a powerful feature of a heterogeneous platform solution, as the Linux OS on the PS can fully reconfigure the PL at run time. However, such switching may result in system downtime and it is important to determine the limitations. Overlay switching time was measured over a number of test iterations and results are illustrated in Fig.10.

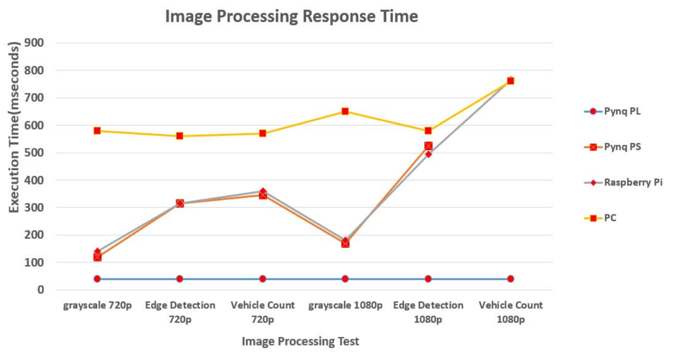


Fig.9. – Comparison of Image Processing Response Time

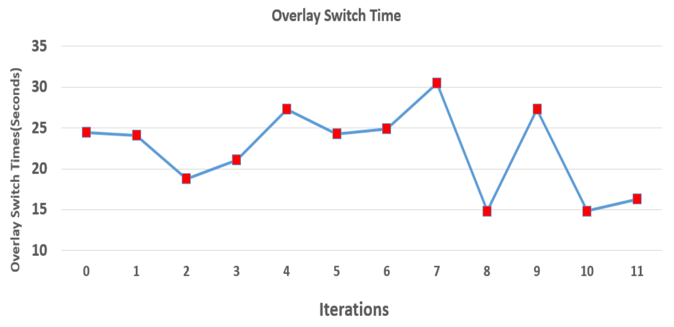


Fig.10. – PS overlay switching time over a number of iterations

D. Register Access Times

For this work the vehicle counting custom IP uses registers that are accessed over an AXI-Lite interface. In order to ensure the registers can be accessed quickly enough for the application, testing was performed to determine worst case access times by measuring execution time in a Python script, as shown in Fig.11.

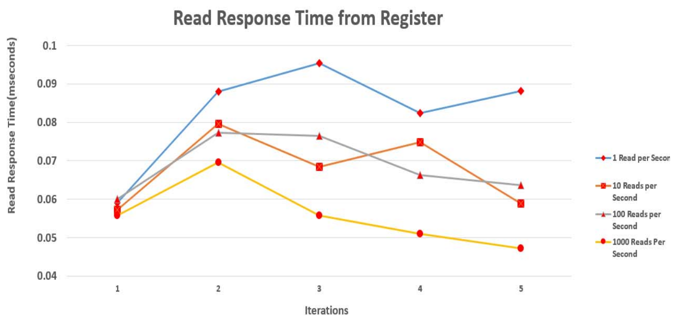


Fig.11. – Register access times over a number of iterations

E. MQTT Message Response Times

Another key aspect of the work is how fast the information can be passed from the SDSoC to the consumer device. In order to measure this latency, a test was developed where a time stamp was sent from the PYNQ platform to the Raspberry Pi and the difference between this and the current time were measured. In order to ensure accurate results both platforms had to have synchronized clocks. This was achieved using a network time protocol (NTP) server.

V. ANALYSIS

A. Power Consumption and Image Processing Latency

The key aim of this work is to develop a software/hardware co-design IoT architecture that is capable of achieving real-time image analysis within a low power envelope. Examining the power consumption in Section IV, the worst case scenario for the PYNQ platform was 2.7 watts while running a 1080p image at 60 fps. Looking at an energy harvesting application (e.g., solar based in this use case) with a typical reserve battery capacity of 600W/h, the implementation would have sufficient reserve capacity for 8-9 days of operation.

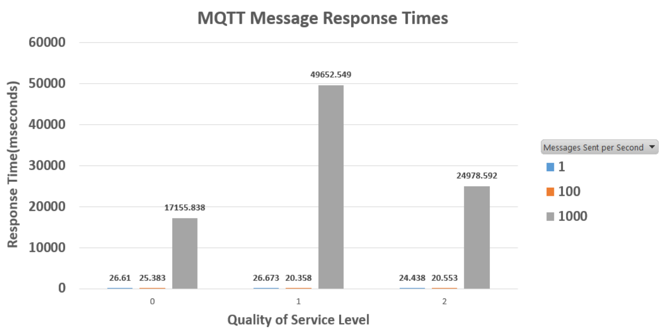


Fig.12. – MQTT Message response times on a local network

Reviewing the results of the image processing response times, it can be concluded that the PYNQ PL outperforms the other solutions evaluated. The time taken for image analysis scales according to the task complexity and image data size when run on the CPU. However, the results from the PL on the PYNQ platform shows that this time remains constant at 40ms per frame. At 60 frames per second, the PL response time increases to 50ms, while the response time of the other platforms remain unchanged from the results captured in Fig.9. It was also noted from the results shown in Fig.7 that the main factor in power consumption on the PYNQ platform results from the percentage of FPGA logic used in the design. Fig.7 shows that removing the HDMI pipeline completely reduces the power by roughly a half. While it is not possible to remove the HDMI pipeline completely in IoE applications, it may be possible to remove the output portion of the HDMI pipeline in order to save further power.

B. MQTT Message Latency

A requirement of IoT devices is the need to communicate with other devices in the network. This was tested by measuring the time it took for an MQTT message to pass from the publisher to the subscriber. Looking at the chart shown in Fig.12, it can be seen that the latency of sending a message is largely dependent on how quickly the publisher is publishing data. From Fig.12, it can clearly be seen that the transmission latency is roughly 20-30ms until the rate of publishing is increased to 1000 messages published per second. Once it has been increased to this rate the latency nearly increases by 1000. This is most likely caused by the broker's inability to cope with such a level of congestion. The latency also worsens depending on the different level of MQTT QoS being used. For the application chosen in this work, this latency is suitable, as the amount of data and the rate of publishing is quite low (e.g., one message every five seconds). However, other applications that require the transmission of larger data volumes may need to be cognizant of these results.

C. Register Access Times and Overlay Switching

Two other tests performed were the latency when accessing registers in the custom IP, and the latency switching from one overlay to another. It is important in this use case to measure the register access time latency, as the application chosen required that a result be read from the PL for each video frame passed into the HDMI pipeline. This means the application in PS Linux userspace must be able to read the result at the same speed or faster than the frame rate of the input video to the PL. At the worst case scenario of 60 fps, the userspace application must perform a read every 16ms. The result shown in Fig.11 indicates this could clearly be achieved as the worst read access time was ~95us. This illustrates that there is plenty of headroom for a more complex data exchange between the PS and PL within this IoE framework architecture.

The second test set illustrated in Fig.9, measures how long it takes to switch from one overlay to another. This could be important for applications that wish to change the type of image processing being used. Different image processing IP can be used for day mode and night mode using these overlays, while maximizing the available PL fabric to support more complex algorithms. Examining the results shown in Fig.10, the worst case latency for switching between overlays is ~30s.

D. *Other Analysis*

The implementation performed in this work identified that some types of image processing are better suited to an SDSoC solution than others. The image processing IP developed for this solution processes data using a stream, where one pixel is processed per clock cycle. This allows for very fast processing in hardware, but if an image processing technique requires access to a number of pixels around the current pixel, then memory is required to store these other pixels. The greater the distance from the current pixel the more memory is required. Since the programmable logic in this solution has a limited amount of memory it is only suitable for applications that process small localized sections of the image at one time. This means that the approach is suitable for techniques involving SIFT, HOG, SVM, but less so for CNNs, NNCs, etc.

VI. Conclusion

The work described in this paper presents the following core contributions:

- Development of a scalable IoT architecture using software / hardware co-design for real-time IoE applications.
- Implementation and evaluation of this architecture through the development of a full stack IoE application for the challenging task of low power VSL motorway control.

This architecture allows for the development of applications that support multiple producers and consumers, while offering low power and real-time image processing capabilities. Future work will consider sending segmented image data to the cloud for additional processing for complex cloud-based machine learning tasks. The SDSoC can segment the video stream to provide salient image segments or feature maps (e.g., identify the outline of a vehicle and transmit the segmented image only). Doing so would greatly reduce the workload required by the cloud device, reduce latency, and the overall network power profile. Not all types of image analysis is suitable for a PL solution, due to the sequential nature of implementing image processing in the PL. This architecture can be augmented to further split the image processing tasks between the PS and the PL, expanding the possible use cases.

The second contribution was demonstrated in the development of the variable speed limit (VSL) controlled motorway. While the application developed does not offer the most cutting edge algorithms when compared to other work currently taking place in traffic monitoring research [14], the implementation delivered a full-stack IoE solution with an average power consumption profile of 2.6W, and an image analysis response latency of 40ms. This solution could offer an alternative to other IoE applications; take the example of the current research taking place with the use of UAV (unmanned aerial vehicle) for traffic monitoring [15]. In this paper the cloud is used to perform the image processing, but using the IoE stack developed in this work the image processing could be performed by the UAV itself.

One challenge of this work is the complexity of the platform. There is little information available on how to develop full stack IoT applications on heterogeneous platforms, and there are particular challenges that are difficult to overcome – in particular, integrating the FPGA-based PL with the Linux-based PS using standardized memory sharing interfaces. With these challenges overcome, the architecture outlined in this paper offers a solution for general Internet of Eyes applications, which require a low power profile, with flexible and real-time image analysis capabilities.

VII. Acknowledgements

This research was supported by Xilinx Inc. who provided the PYNQ boards used in this work. In particular, we would like to thank Cathal McCabe and Peter Ogden from Xilinx Inc. who provided support during this research. We would also like to thank the Intel Corp. for their support throughout the DCU MEng program and in the development of this work.

VIII. References

[1] Kamburugamuve, S., Christiansen, L., “A Framework for Real-Time Processing of Sensor Data in the Cloud”, [Online]. Available: www.semanticscholar.org/paper/A-Framework-for-Real-Time-Processing-of-Sensor-Data-Kamburugamuve-Christiansen/38fe08755c6eead7c3c2f3938a8e92bf112502bf
[2] Shang, L., Lin, C.Y., Atif, M., Williams, A., “Evaluation of High Density GPUs as Sustainable Smart City Infrastructure”, Utility and Cloud Computing (UCC), 2015 IEEE/ACM 8th International Conference on utility and cloud computing.
[3] Bieszczad, G., “SoC-FPGA Embedded System for Real-time Thermal Image Processing”, Mixed Design of Integrated Circuits and Systems, 2016 MIXDES - 23rd International Conference
[4] Stromatias, E., Galluppi, F., “Power analysis of large-scale, real-time neural networks on SpiNNaker”, Neural Networks (IJCNN), The 2013 Intl. Joint Conf.
[5] CNX Software, “Intel-Movidius neural Network Compute stick brings deep learning artificial intelligence offline”, [Online]. Available: www.cnx-software.com/2017/07/21/intels-movidius-neural-compute-stick-brings-deep-learning-artificial-intelligence-offline/
[6] Xilinx, “SDSoC Overview”, [Online]. Available: www.xilinx.com/support.html
[7] Srijongkon, K., Duangsoithong, R., “SDSoC Based Development of Vehicle Counting System Using Adaptive Background Method”, Micro and Nanoelectronics (RSM), 2017 IEEE Reg. Symp.
[8] MQTT Org; “MQTT Specification”, [Online]. Available: http://mqtt.org/documentation
[9] Xilinx; “Zybo Reference Manual”, [Online]. Available: www.xilinx.com/support/documentation/university/XUP%20Boards/XUPZYBO/documentation/ZYBO_RM_B_V6.pdf
[10] ARM AMBA Specification [Online]. Available: www.gstitt.ece.ufl.edu/courses/fall15/eel4720_5721/labs/refs/AXI4_specification.pdf
[11] Python Productivity for Zynq; “PYNQ: Getting Started”, [Online]. Available: files.digilent.com/Products/PYNQ/pynq_z1_v2.1.img.zip
[12] Python Paho MQTT [Online]. Available: pypi.org/project/paho-mqtt/
[13] Eclipse Mosquitto [Online]. Available: mosquitto.org/
[14] Moayed, Z., Griffin, A., Klette, R., “Traffic Intersection Monitoring Using Fusion of GMM-based Deep Learning Classification and Geometric Warping”, 2017 Intl. Conf. on Image and Vision Computing New Zealand
[15] Niu, H., Gonzalez-Prelcic, N., Heath Jr, R.W., “A UAV-based Traffic Monitoring System”, 2018 IEE 87th Vehicular Tech. Conf.