2018 IEEE International Conference on Edge Computing

Data Distillation at the Network's Edge: Exposing Programmable Logic with InLocus

Lucas R. B. Brasilino, Alexander Shroyer, Naveen Marri, Saurabh Agrawal, Catherine Pilachowski, Ezra Kissel, Martin Swany

School of Informatics, Computing and Engineering - Indiana University
Bloomington, IN 47408
{lbrasili,ashroyer,navmarri,agrasaur,calupila,ezkissel,swany}@indiana.edu

***Abstract*—With proliferating sensor networks and Internet of Things-scale devices, networks are increasingly diverse and heterogeneous. To enable the most efficient use of network bandwidth with the lowest possible latency, we propose InLocus, a stream-oriented architecture situated at (or near) the network's edge which balances hardware-accelerated performance with the flexibility of asynchronous software-based control.**

In this paper we utilize a flexible platform (Xilinx Zynq SoC) to compare microbenchmarks of several InLocus implementations: naive JavaScript, Handwritten C, and High-Level Synthesis (HLS) in programmable hardware.

I. Introduction

Deployments within the context of Smart Cities and the Internet of Things (IoT) are expected to generate massive data inflows and, critically, a shift from data primarily generated in the cloud to data generated at or near the edge of the network. In fact, the widespread use of IoT devices is already here. From the areas of health sciences, security and environmental monitoring, to industrial and commercial applications, the prevalence and variety of wireless sensing devices and capabilities continues to grow. By 2025, it is projected that the global IoT market will reach upwards of 27 billion devices [1] with a corresponding explosion in sensor data volume that far exceeds the growth of traditional business data [2].

One of the key unanswered research questions in this space involves the effective processing and analysis of the aforementioned data deluge. Simply routing sensor traffic to centralized data centers presents a number of challenges which are often couched in terms of current and future network bandwidth, computation, and storage capacities. While data warehousing strategies are adapting, there is an acknowledgment that hybrid approaches, including *in-situ* computation at edge devices, will play a critical role in the future of the modern networking landscape [2]. Also from a practical standpoint, the task of identifying those data flows which have immediate value becomes particularly important in the context of latency-sensitive applications. To meet these challenges, we envision the edge consisting of network-driven facilities for data aggregation, deduplication, filtering, and fusion supported by a wide range of device classes.

Nascent standards such as those defined within Mobile Edge Computing (MEC) [3] are laying the groundwork for mobile network operators to deploy services and virtualized network functions closer to the source of data. At the same time, there is recognition that a class of devices supporting Field Programmable Gate Array (FPGA) technology shows promise in managing the heterogeneity of IoT implementations [4], and as we contend, has the potential to accelerate workflows by enabling network and computational offload at the edge.

We also note that within the data center the trend to incorporate FPGAs has been accelerating within recent years. The introduction of so-called SmartNICs based on System-on-Chip (SoC) and FPGA parts [5] are enabling application acceleration and offloading techniques, particularly in the high-performance storage arena. For non-fixed scenarios, where reliable infrastructure may not be available, FPGAs provide mobile, low power computational flexibility on a multitude of SoC-class devices [6] suitable for edge deployment. In this latter area, the ability to orchestrate and easily reconfigure the FPGA-enabled edge becomes an important concern.

It is within this context of function virtualization, service orchestration, and embedded SoC FPGA platforms that we introduce the InLocus architecture. InLocus has been designed as a distributed stream processing framework that enables *in-situ* computation on a wide range of edge compute devices. The InLocus approach is twofold:

- Develop a network functional unit (NFU) model for topologically-aware stream processing at the edge;
- Tightly integrate compute with network processing using *programmable logic* (PL), particularly FPGAs, on otherwise resource constrained embedded platforms.

With InLocus we recognize that a key performance bottleneck limiting the scalability of SoC compute elements is the network packet processing overhead as data rates increase. To that end, the primary contribution of this paper is the design of a computing architecture that moves stateful computation toward the edges of the network, along with a comparative evaluation of InLocus service implementations that demonstrates the viability of a PL approach to IoT sensor data processing on an exemplar ARM-based SoC with an integrated FPGA. The key aspects of the InLocus architecture include offloading processing from cloud to edge, responsiveness to heterogeneous workloads, low-latency computation over IoT data and reduction of overall bandwidth requirements.

978-1-5386-7238-9/18/$31.00 ©2018 IEEE
DOI 10.1109/EDGE.2018.00011

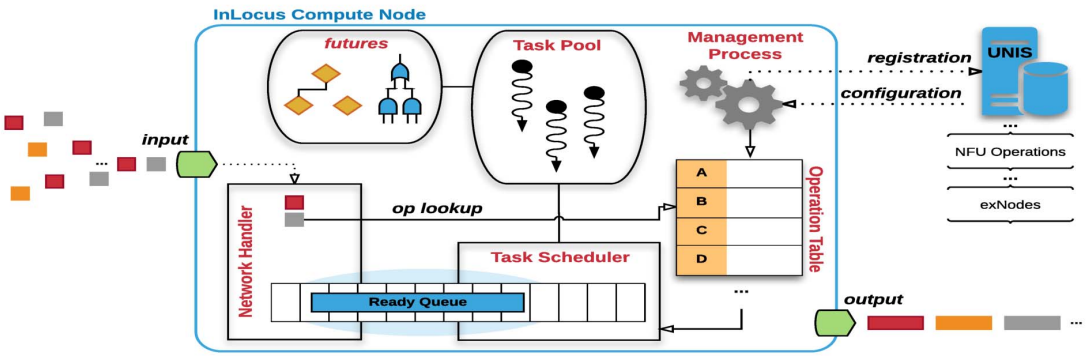

Fig. 1. Logical components of an InLocus compute node showing the tight coupling of the network handler and task scheduler. Platform-specific operations are instantiated as configured over the UNIS control connection while data dependencies may be realized using task-bound *futures*.

The remainder of the paper is structured as follows. We provide an overview of the InLocus architecture in Section II followed by a discussion of the software reference implementations in Section III. Section IV describes the PL design and implementation. We describe our testing environment and the results of our evaluation in Section V. Related work is surveyed in Section VI and we conclude the paper in Section VII.

II. The InLocus Architecture

The InLocus architecture centers around the creation of an abstraction of processor resources as a fundamental computing service. We borrow the term *Network Functional Unit* (NFU) [7] to describe this abstraction. To faithfully conform to the Internet design philosophy, the NFU, like IP, must offer a service that is not only simple and highly generic, but also provides only weak guarantees in regards to such features as the availability and the duration of the computation. In other words, the NFU provides best effort computation that allows for more fragmentation or packetization of application processes. All the stronger services that applications require must be built by aggregating units of the basic service provided by discrete and limited NFU operations, which we call *Ops*. The goal is to create a simple, generic, and limited computing service that can be integrated into the definition of the network and be used to create a programmable resource fabric that provides the kind of functionality, performance, and scalability that data-intensive applications in edge environments require. Further NFU details may be found in [7].

Our reference design for an InLocus compute node observing the NFU abstraction for edge computing is diagrammed in Figure 1. The key components within a node include a *task scheduler* with an associated *task pool*, an indexed NFU operation (or *Op*) table, and an integrated *network handler*. During stream processing, the network handler is responsible for characterizing arriving data elements and performing a lookup in the operation table. This lookup determines which task is placed in the scheduler's *ready queue* for execution of the appropriate NFU Op on the input data. Tasks are managed within in-memory pools that are sized based on resource availability and requested node configuration.

The asynchronous primitive in our system is a *future* [8], which represents a variable or memory region which is expected to eventually contain a value to be read. A concurrent system built on futures permits using *promises* or *tasks* for asynchronous control flow, where the *task* represents a computation depending on *future* data. In C++, for instance, futures are returned by *async* functions or *promise* objects, and calling code can execute the future's *wait* method to block execution until the result arrives [9]. This model of concurrency contrasts with the fork-join model by putting a scheduler in control of data dependency resolution and task ordering. Within InLocus computing nodes, tasks can be any of the executable functions performing *Ops* on streaming data. The future mechanism enables data-dependent scheduling, determining when tasks are added and removed from the task scheduler's ready queue based on data availability.

Crucial to the integration and orchestration of an InLocus network is the Unified Network Information Service (UNIS) [10]. UNIS maintains network topology, embedded service capabilities, and structural file metadata (referred to as *exNodes*), and provides a publish/subscribe interface for streaming updates. This InLocus management plane is responsible for the discovery, measurement, and management of NFUs *Ops*. As shown in Figure 1, each compute node runs a management process that communicates with an associated UNIS instance. Soft-state registration of the node allows our runtime to maintain resource and capability advertisements as well as reachability status for each compute resource in the network. Configuration of the node is also enabled via UNIS using pub/sub or regular polling of the appropriate service endpoint. Nodes may be either statically provisioned as single-use NFUs or have their *Op* table updated dynamically at runtime.

III. Reference Implementations

Our first task involved developing a generic InLocus runtime framework for compute nodes that could be ported to existing and future edge computing device targets. Our motivation was to create a "safe mode" version of the edge compute task scheduler that is not necessarily performance optimized

but instead works broadly across devices that run a *nix-like OS, from ARM SoC devices like the Raspberry Pi to workstation PCs that are suitable for prototyping and evaluation. This initial runtime environment was designed to be interoperable with existing IoT protocol stacks (e.g., Constrained Application Protocol (CoAP) [11] and Concise Binary Object Representation (CBOR) [12]) and serves as the reference implementation that adheres to the InLocus NFU architecture outlined in Figure 1.

A. Node.js reference implementation

The Node.js reference server (*ref-server*) implementation aims to provide portability and correct semantics, but not performance. The code is split into modules reflecting the key InLocus compute node components: *(a)* `server` - entry point; *(b)* `futures` - additional "structured" future/promise behaviors; *(c)* `scheduler` - runtime queue of operations to be run; and *(d)* `config` - default runnable operations, default port and "next hop" behavior.

The application life-cycle begins when a compute node powers on and registers its capabilities with a central server (UNIS). The server *may* respond with specific operations for this node to perform, or the node may fall back to a default configuration. If there are multiple nodes available, there can be additional "next hop" targets in the network graph, otherwise the next hop defaults to returning to UNIS.

As data arrives at a node, processing is driven by the configured NFU Ops and control is forwarded to the next hop in asynchronous continuation-passing style. Individual nodes may maintain small amounts of local state in order to perform data reduction or aggregation, but are not used for global system state. Furthermore, nodes continue to process data until their operation is redefined via the management plane.

InLocus provides two built-in Ops atop the underlying *future* semantics: "default", which unconditionally forward to the next hop; and "counter", which performs data reduction by emitting messages at a rate of 1/count. Programmatically, these roughly correspond to streaming versions of *map* and *reduce*.

B. C/Linux implementation

The C implementation demonstrates a subset of the Node.js implementation's functionality, but with an emphasis on performance and message throughput, which allows this version to serve as a performance baseline in our evaluation below.

We developed a CoAP `blaster` which can generate many more CoAP messages per second than the Node.js implementation. As a brief description, it reads from the command line: the total number of messages, the desired message per second (mps) rate, and a CoAP URL containing the target node and service. Then, it generates messages containing CBOR encoded data of double-precision float-point format (64-bit), representing a large number of sensor outputs deployed in practice. Hence, `blaster` fits in the "synthetic workload" class of tools.

The C implementation also includes its own CoAP server with lightweight threading to handle NFU Ops. This server

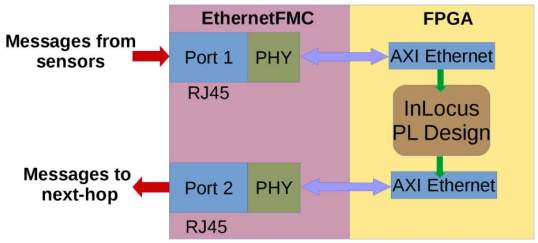

Fig. 2. Implemented system

functions as a benchmarking platform for measuring messaging performance and network overheads. While it may be possible to further optimize performance within the context of available Linux/SoC platforms, we believe that our implementations represent a faithful reflection of current IoT software best-practices and system tuning.

IV. Programmable Logic Implementation

The programmable logic (PL) implementation is designed to function in series with an existing network data-path, independent of a CPU. At the edge, InLocus nodes accept input from sensors and send output to a higher-level node in the data-path graph. This higher-level node can be either another function (node) or a destination application in the cloud.

We used the Xilinx ZedBoard development platform, which features programmable logic (Artix-7 based FPGA) and a processing system (dual-core ARM Cortex-A9) on the same chip. For networking, the ZedBoard was outfitted with an EthernetFMC expansion board, a separate peripheral which connects directly to the FPGA using the *FPGA Mezzanine Card* (FMC) interface. This card contains four 1Gbps Ethernet ports, each controlled by a Marvell Alaska 88E1610 [13] PHY transceiver. Figure 2 illustrates the overall system architecture.

The InLocus PL design is composed of two main parts. First, a *Function Controller* (FC), written in Verilog, manages the process of parsing network packets, extracting sensor data, and feeding data to the second part of the design: *Function Units* (FU). A Function Unit is written in Xilinx High Level Synthesis (HLS) [14], a tool which translates arithmetic calculations written in C/C++ source code into Verilog. This permits much more rapid prototyping, especially for application developers who are more likely to be familiar with C-family programming languages than with hardware description languages. In our experience, the generated module's[1] performance is comparable to handwritten Verilog, although the amount of FPGA resources used may be higher.

The design's input is a 32-bit AXI-Stream [15] bus from the Xilinx AXI Ethernet Subsystem [16], which is then converted to 256-bit bus. The reasons for choosing a 256-bit wide bus as FC's input were: *(a)* easier reuse on platforms where this width are native, such as NetFPGA [17] and *(b)* network protocol parsing is made simpler since data from different network

[1]Also referred to as *Intellectual Property (IP) cores*

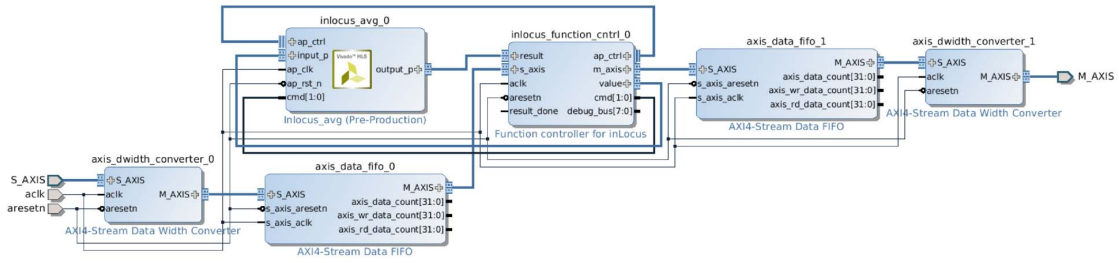

Fig. 3. High level block diagram of InLocus PL design

TABLE I
INLOCUS AVERAGE FUNCTION UNIT INSTRUCTIONS

Instruction	OPCODE	Description
`NOP`	`0x00`	Keeps FU inactive, but ready to run
`RESET`	`0x01`	Clear internal accumulator
`CALC`	`0x02`	Read new value, store average in accumulator
`RESULT`	`0x03`	Write accumulator value to output

layers can be accessed in a single clock cycle. Similarly, the design's output is converted back to a 32-bit bus and fed to another AXI Ethernet Subsystem module that places packets "*on-the-wire*" through Port 2. Both input and output AXI-Stream buses are running at 100Mhz, therefore supporting 3.2Gbps of streaming data.

Figure 3 depicts the core high-level block diagram generated by the Xilinx Vivado tool. For the sake of simplicity, other modules (AXI Ethernet Subsystem, clocks and reset processors) are omitted from this diagram. The principal modules are described in the following subsections.

1) Function Unit: The Function Unit performs arithmetic computations as specified by the Function Controller. It corresponds to the aforementioned NFU in the InLocus architecture.

The first FU initially developed has the primary goal of computing an arithmetic average over a streaming sensor input. The arithmetic average functionality represents an *Op*. In addition, to satisfy the programmable requirement of the InLocus architecture, a set of four instructions were created to dictate its behavior, as described in Table I. The FU has an accumulator register and an arithmetic computing unit. It was implemented in 25 lines of C code, but the generated Verilog design has 868 lines, instantiates 3 Digital Signal Processing (DSP) units, 4388 flip-flops and 5619 Look-Up Tables (LUT).

An implementation detail imposed by Xilinx HLS is that conforming modules must adhere to a *Block-level I/O protocol*, accomplished by the *ap_ctrl_hs* interface. This interface consists of 4 signals: *(a)* `ap_start` - starts module operation; *(b)* `ap_idle` - module is idle; *(c)* `ap_ready` - module ready to perform operation; *(d)* `ap_done` - computation is done and result is available. The Function Controller ensures that clock-cycle timing is internally consistent among these signals, and that instruction issuance is synchronized within Function Units.

2) Function Controller: The Function Controller is the central module, performing many tasks. It analyzes the network packets streaming in from the input AXI-Stream bus and tests that the following conditions hold:

1) *EtherType* field in the Ethernet header matches IPv4 (`0x0800`)
2) *Protocol* field in the IPv4 header matches UDP (`17`)
3) *Destination Port* field in the UDP header matches port `5683`, the CoAP protocol's *default* port

When all of the above conditions are true, we assume the packet is a CoAP message, and the FC extracts the 64-bit sensor value from the CoAP message body and generates the correct signaling for starting a computation in the FU. Otherwise, the packet bypasses further processing and is switched to the output AXI-Stream bus.

The interface between the FC and FU is done by the four buses illustrated in Figure 3, named after the FC's port names. `cmd` is the instruction bus, where the FC issues the instructions listed in Table I. `ap_ctrl` is the *ap_ctrl_hs* interface. Bus `value` is the FU's input interface where the FC places the sensor data extracted from CoAP message. Finally, bus `result` is the FU's output interface, where the computation result is placed when the `RESULT` instruction is issued.

V. EVALUATION

We concentrate on the feasibility of deploying an InLocus compute node with FPGA functionality in a real-world, operational network. To evaluate our approach, we first explore the processing capacity of CoAP message streams, which reveals the maximum amount of messages per second (rate) that can be handled, thus defining an upper bound for each implementation. Next we explore the latency introduced in the in-network computing data-path by the function (Op), to estimate the total computational time of a particular implementation, in order to compare viability of that version compared to conventional alternatives.

A. Testbed

We developed a testbed containing two categories of resources: *(I)* software resources, composed of *(I.a)* a CoAP messages sender application, *(I.b)* a CoAP endpoint server and *(I.c)* a network traffic capture tool; and *(II)* hardware resources, containing four major parts: *(II.a)* a server to send and receive

CoAP messages, *(II.b)* a board containing embedded ARM processors along with an FPGA, *(II.c)* a server for capturing network traffic, acting as an independent observer and *(II.d)* a managed Ethernet switch.

B. Software

In order to generate CoAP message streams, we utilized the `blaster` tool. In turn, an instance of the C ref-server was the final destination for streams, representing the immediate "next-hop" of compute service in the architecture. Both tools are described in section III-B. It is worthwhile to mention that this study focuses on the first level of NFUs at the network's edge, but the InLocus architecture may encompass additional levels of functions to be placed in the data-plane path towards the core cloud-computing application.

Finally, the `tcpdump` network packet capture tool was used to store all messages that ingress and egress the InLocus function processing node in PCAP files.

C. Hardware

To send and receive CoAP messages, we used a Dell PowerEdge R510 server with two Intel Xeon E5640 2.66Ghz CPUs (dual core, four threads each, totaling 16 processing units), 32GB RAM at 1333Mhz and an Intel Gigabit ET QuadPort NIC, where one port was dedicated to send and other to receive messages. The operating system was GNU/Linux Debian 9 x86_64 Kernel version 4.9.0 and GLIBC 2.24.

In turn, the Zedboard plus EthernetFMC kit was running the ARM hard-float 32-bit port of GNU/Linux Debian 9 (armhf), GLIBC 2.24, and a Linux Kernel version 4.4.0 provided by Xilinx within its Petalinux distribution, compiled from sources. This distribution plus kernel combination were chosen to support this study in two ways: First, Debian equipped Zedboard with compilers, libraries and runtimes allowing both C and Node.js ref-servers to be compiled and executed. Second, Linux Kernel distributed by Xilinx enabled *bitstream* files to be loaded into the FPGA *on-the-fly*, essentially reprogramming it from the command line without need of reboots, special programs (*e.g.Xilinx Hardware Server*), or JTAG cables/ports.

The *independent observer* server received copies of all network traffic between the sender/receiver server and the Zedboard kit running InLocus functions. It was a Dell PowerEdge 860 with a Intel Xeon 3050 2.13Ghz CPU (dual core) and a DualPort 10Gbps Mellanox ConnectX-3 NIC dedicated for packet capturing. The server was also running the same GNU/Linux Debian 9 x64_64 version as the sender/receiver server. This server was easily capable of capturing packets at rates exceeding 1Gbps and ran only a minimal Linux OS and `tcpdump` for packet capture to minimize system noise. Packet captures were written to a *tmpfs* RAM filesystem to minimize I/O latency.

For Ethernet connectivity and port-mirroring, a Pronto Switch model P-3290 running firmware Pica8 PicOS 2.0.4 was deployed. This switch has 48 1Gbps and 4 10Gbps ports. Two VLANs were enabled for splitting broadcast domains and traffic between the sender (to Zedboard kit) and receiver (from Zedboard kit) sides. In other words, each VLAN was assigned two 1Gbps ports (one connected to the originator or receiver server ports and other to Zedboard's EthernetFMC). Finally, the two ports connected to the Zedboard were mirrored by the switch to a 10Gbps port, where the Mellanox 10Gbps NIC on the independent observer captures both network traffic that is about to ingress or egress the Zedboard kit.

D. Stream processing capability

We first focused on the stream processing capability of the three initial implementation of InLocus functions on the network edge, performing a direct comparison between them. The main inquiry was how many messages per second (mps) a particular reference implementation can handle. In other words: what is the *drop-rate* of each one.

In this study each implementation performed a summarization over 10 CoAP messages containing random temperature values around 50°F generated by `blaster` software. Every individual message has 121 bytes in total, where the CoAP payload has 75 bytes. We performed 10 rounds of stream processing for each message per second rate, and each round was 30 seconds long – sufficient time to stabilize and stress every component on the implementation. Consequently, the number of messages in a single round is $N = R \times T$. For instance, 37500 messages were sent per round at 1250mps. Since 10 rounds were performed, a total of 375000 messages were analyzed to obtain a single data point.

Figure 4 depicts the results. The box-and-whisker plot shows the minimum drop-rate (lower whisker), the first quartile (lower box edge), the average (black horizontal trace), the third quartile (upper box edge) and the maximum drop-rate (upper whisker). An important aspect that can be directly visualized is the *interquartile range* (IQR), represented by the box height. It is a measure of dispersion, carrying information on how disperse is 50% of the population from the average.

At 250mps the Node.js ref-server had 0% drop-rate, along with C and the PL implementation. However, at 500mps the average drop-rate was 30% in average with a large IQR. At 1500mps (not depicted), Node.js ref-server had almost 100% of drop-rate, establishing its barrier.

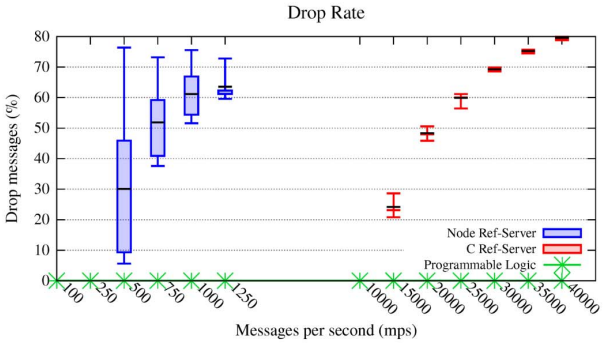

Fig. 4. Message drop rate

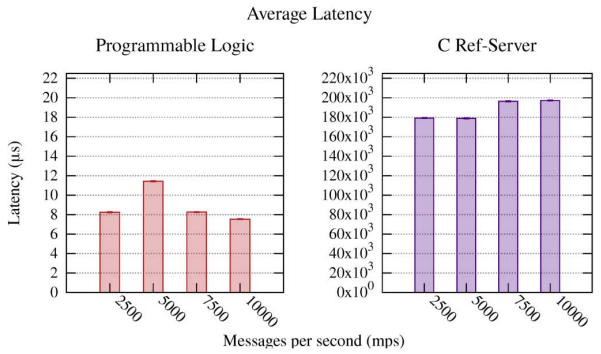

Fig. 5. Average PL vs C ref-server latency

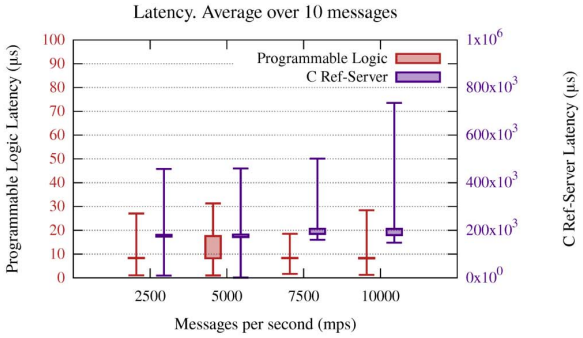

Fig. 6. C ref-server and PL latency

On the other hand, the C ref-server was capable of processing without any loss up to approximately 10000mps. To keep Figure 4 cleaner, we present C ref-server results in steps of 5000mps. The C implementation effectively started dropping messages at 14000mps and at 15000mps achieved 23% of drop-rate. At 40000mps it reached an average drop-rate of 80% with almost no dispersion (IQR→0).

The PL implementation had shown no message loss at any condition. Going further, the design was stressed up to 1000000mps, again with no message loss. At this rate the network bandwidth reaches 0.97Gbps – essentially full bandwidth.

The Node.js ref-server was the implementation least capable of handling CoAP streams, not only by reaching saturation at lower rates, but also presenting a large IQR: its drop-rate is less predictable. Although Node.js engine compiles JavaScript applications into a bytecode, its runtime manages memory allocations, thread schedulings, and I/O operations concurrently at the same time of execution. That higher level of abstraction tends to be CPU intensive.

Moreover, the C ref-server performed much better than the Node.js counterpart, by up to a factor of 40x when considering message latency. We note that the C ref-server is compiled to an ARM architecture binary, which is scheduled and run directly by the Linux kernel and can better take of advantage of the underlying SoC hardware.

Finally, the PL implementation presented the best results, capable of processing at the full network bandwidth of 1Gbps. This outcome can be explained by the facts that the PL implementation does not interact/interrupt Zedboard OS and/or CPU, it is *fully pipelined* and the internal AXI-Stream bus delivers 3.2Gbps of throughput.

E. Message latency

Another important aspect that must be evaluated in an *in-situ* edge computing function is the latency at which it computes a result. A way to model latency is by the disturbance it introduces in the network when placed along the data-path. Lower latency yields better performance.

Here we compare the C ref-server and PL implementations (the Node.js version is omitted due to being at least 40 times slower than the C version).

For this experiment, each implementation computes a moving average of synthetic sensor readings over a "window size" of 10 messages. This emulates the intended use case of handling N sensors (ideally located in close physical proximity) at once, in order to reduce the upstream network traffic by a factor of N. Since the last message in the set is the only one to trigger a summary message, we measure the latency between it and the resulting egress message.

`Blaster` generated 300000 messages (30s duration) regardless of message rate. The maximum mps was limited to 10000 to prevent the C ref-server from dropping them. Figure 5 shows the average latency results for 2500, 5000, 7500, and 10000 mps. The PL scale is in tens of μs while the C ref-server is at hundreds of thousands of μs, resulting in a ratio of 10^4. In the best case, the PL implementation achieved a latency of 7.53μs at 10000 mps, while the C ref-server achieved 178875.31μs (178.86ms) at 5000mps.

While the average results were promising for the PL implementation, we also study the latency variance to measure the determinism of each implementation. Figure 6 shows both PL and C ref-server results, however the former scale is depicted on the left and the latter on the right. Again, the ratio between both scales is 10^4.

The PL implementation results demonstrate highly deterministic behavior, with almost zero IQR. The only exception was at 5000 mps where there was an increase of average latency (Figure 5) and dispersion. This happened only at 5000mps and might be due to an external factor, such as an incidental delay on the Xilinx Ethernet MAC module, for instance. This behavior will be the subject of further investigation.

Although the C ref-server had a slightly wider IQR (tens of ms), Figure 6 it shows that they are also near the average, and results (average and IQR) do not abruptly change with increasing mps. A final analysis considers the average speedup afforded by the PL implementation at **26,000** times faster than the C ref-server.

F. Latency breakdown

The work done by each reference implementation can be decomposed into a sequence of smaller steps. For instance, the C ref-server issues syscalls to bind to an interface and UDP port, reads from a network file descriptor, parses CoAP messages, executes summarization functions, assembles egress messages, and (optionally) writes to another network file descriptor. Therefore a full understanding of internal bottlenecks is essential for future designs.

For internal latency analysis of the PL implementation, we utilize Xilinx Vivado development environment's RTL simulation. Vivado provides a timing-accurate model for the design, producing a reliable hardware behavior model.

We repeat the measurement from section V-E: the latency between the *(a)* the final CoAP message in the set, and *(b)* its summarized output. The summarized message construction is the only occasion where the FC issues two instructions sequentially: `CALC` and `RESULT`.

Vivado simulation reveals 0.070µs completion time for the `CALC` instruction, and 0.370µs for the `RESULT` instruction.

Moreover, using the same simulation technique reveals the total latency between the last CoAP message to get in the core PL implementation and sending the result is 1.37µs. Since the time for instructions being processed is 0.44µs in total, it accounts for 32.11% of the core's latency. The other 67.88% is spent in operations such as converting AXI-Stream width, controlling the HLS core, etc. However, considering that the complete PL design has shown a latency around 8µs (Figure 5), the core PL design's (Figure 3) overall impact is marginal, being responsible for 17.13% of latency.

The C ref-server has different runtime execution semantics than the PL design. Because it is an OS process, it is subject to scheduling, context switching, blocking by kernel services (*syscalls*), etc.

For C ref-server profiling, we made use of the Linux *perf* tool. *perf* works by collecting program counter samples during the program execution and registering which function/component were running, then records the time spent in each function as a percentage of total execution time.

Again, using the same technique as above to maintain consistency, we performed a round of 10 executions over C ref-server processing 300000 messages at 10000mps. Table II summarizes percentage of time spent in each component along with their total time, taking in account an average latency of 197.30µs (Figure 5).

The results show that most of the execution time is spent in Kernel and GlibC functions. Table III highlights that the former most expensive procedures are doing DMA transfer, processing interrupts and polling RX queue at AXI Ethernet Subsystem driver, while the latter most ones are related to memory management.

The function that computes the average (`arith_avg`) spends 0.12µs to deliver results. Looking back at the PL version, considering that (1) associates `CALC` instructions are issued in 9 messages and `CALC+RESULT` are issued in 1 message, we calculate that the average time spent by the PL to deliver the result is 0.107µs. Hence, comparing only the functionality of computing *average over 10* values, the PL implementation is only 12.15% faster than the C ref-server running on the CPU. However, the PL's clock is 100Mhz, while CPU's is 667Mhz, showing that the former can achieve better performance even using approximately 1/6 of the latter clock frequency. Despite this fact, the large PL design gain (26000 times less latency) is indicative of the PL eliminating the traditional application and OS runtime model overheads (scheduling, context switches, interrupts, etc), and in particular, offloading expensive network stack overheads that would otherwise be processed by the CPU.

TABLE II
C REF-SERVER PROFILING

Component	Mean (%)	Std Dev (%)	Time (µs)
Kernel	51.62	0.22	101.84
GlibC	18.68	0.18	36.85
libCoAP	13.43	0.15	26.50
libCBOR	13.37	0.08	26.38
Arith_avg	0.06	0.01	0.12

TABLE III
KERNEL & GLIBC MOST EXPENSIVE FUNCTIONS

Kernel		GlibC	
Function	Time (µs)	Function	Time (µs)
`v7_dma_inv_range`	8.09	`malloc`	8.07
`do_softirq`	7.99	`free`	1.64
`xaxienet_rx_poll`	4.44	`memset`	1.56

$$PL\_avg\_comp\_time = \frac{9 \times 0.07 + 1 \times 0.44}{10} = 0.107\mu s \quad (1)$$

VI. RELATED WORK

We recognize two main approaches to stream processing at the network's edge as "micro" versus "macro". At the micro level, existing approaches move infrastructure (from sensor data measurements to operating system) into smaller, more widely distributed nodes near the edges of the network. The macro level approach utilizes existing server infrastructure (still inside a datacenter, but not necessarily at the "center" of the network) to offload aggregation and reduction functionality in order to streamline the central services.

The hallmark "macro" approach in our view is Twitter's Heron architecture as described by Kulkarni et. al [18]. Heron performs streaming computation in a virtualized/shared cluster context. In our view, this is much closer to the network's "center" than its "edge", and is more suitable for Twitter-like data (TCP/IP, moderate to high latency, moderate to large payloads). Reducing virtual machine overhead using Linux containers [19], or exposing functions-as-a-serveice at the datacenter's edge [20] are similar "macro" solutions.

At the "micro", or "fog" level [21], the implementation is physically close to sensors. Form factors are often smaller and more resource-constrained, such as single-board computers

(Raspberry Pi, BeagleBone Black) or within consumer wireless access point devices. Fog computing applications may also utilize geographic contextual data, and have the opportunity to complete some of their work within 1 network hop.

Gomez, et. al [4] use a very similar technology "stack" to InLocus consisting of FPGAs as accelerators and CoAP for inter-node communication at the application level. Their approach defines a platform at the single node level, with a built-in OS and configurable "soft peripherals", whereas our approach spans many nodes which are managed externally.

One approach conceptually between our "macro" and "micro" definitions is employed by Liu et. al in their system called Paradrop [22]. It aims to cast consumer-grade Wi-Fi access points (APs) as an alternative platform for applications formerly deployed only to the cloud. These applications can benefit from local contextual network information and lower latencies within the local network of of IoT sensors and actuators.

VII. Conclusion and Future Work

In the present work we introduced the InLocus architecture with a focus on network edge computing nodes, where fundamental computing services are provided by Network Functional Units (NFUs). We presented three reference implementations of the InLocus architecture in a functional ARM-based SoC with an integrated FPGA, and investigated their characteristics and processing capabilities.

It should be unsurprising that it is possible to increase performance by porting software from JavaScript to C (or from C to programmable hardware). However, even performance-sensitive applications rarely take this last step of porting functions to hardware, due in part to additional cost, but especially the vastly different programming model.

A key aspect of InLocus is to facilitate incremental adoption. Because the overall model is realizable in JavaScript, C, and hardware, application developers can gradually implement more high-performance approaches as needed, without having to completely rearchitect the whole application each time.

We believe that by providing *(a)* a standardized interface for IoT applications (CoAP and CBOR), and *(b)* a programming model which facilitates a more familiar development workflow (procedural JavaScript, C, or HLS), the increased performance potential of C is made more accessible to JavaScript programmers, and likewise the performance of FPGAs is made more accessible to both JavaScript and C programmers.

In this direction, we envision as future work: the development of CoAP and CBOR processor modules in the FPGA, feasible for both PL and software NFUs; exploring Partial Reconfiguration to dynamically place new NFUs in the network data-path; augmenting InLocus management plane to take advantage of Partial Reconfiguration capabilities on PL-enabled edge computing nodes; and finally inspecting other FPGA-enabled SoCs, comparing resource utilization and power consumption for a given set of NFUs.

References

[1] Machina Research. (2016, Aug) PRESS RELEASE: GLOBAL INTERNET OF THINGS MARKET TO GROW TO 27 BILLION DEVICES. http://bit.ly/2b5Bp8d.

[2] Peter ffoulkes, HP Enterprise. (2017) InsideBIGDATA Guide to The Intelligent Use of Big Data on an Industrial Scale. https://insidebigdata.com/white-paper/guide-big-data-industrial-scale/.

[3] ETSI White Paper No. 20. (2017) Developing Software for Multi-Access Edge Computing. http://www.etsi.org/images/files/ETSIWhitePapers/etsi_wp20_MEC_SoftwareDevelopment_FINAL.pdf.

[4] T. Gomes, S. Pinto, T. Gomes, A. Tavares, and J. Cabral, "Towards an fpga-based edge device for the internet of things," in *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*, Sept 2015, pp. 1–4.

[5] Caulfield et al., "A Cloud-Scale Acceleration Architecture," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016, pp. 1–13.

[6] J. Greene, S. Kaptanoglu, W. Feng, V. Hecht, J. Landry, F. Li, A. Krouglyanskiy, M. Morosan, and V. Pevzner, "A 65nm flash-based fpga fabric optimized for low cost and power," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 87–96. [Online]. Available: http://doi.acm.org/10.1145/1950413.1950434

[7] M. Beck, T. Moore, and J. S. Plank, "An end-to-end approach to globally scalable programmable networking," in *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 4. ACM, 2003, pp. 328–339.

[8] C. Hewitt and H. G. Baker, Jr, "The incremental garbage collection of processes," *ACM SIGPLAN Notices*, 1977.

[9] (2017, Oct) std::future. http://en.cppreference.com/w/cpp/thread/future.

[10] A. El-Hassany, E. Kissel, D. Gunter, and M. Swany, "Design and implementation of a Unified Network Information Service," in *10th IEEE International Conference on Services Computing (SCC 2013)*, 2013.

[11] K. H. Z. Shelby and C. Bormann, "The Constrained Application Protocol (CoAP)," Internet Requests for Comments, RFC Editor, RFC 7252, June 2014. [Online]. Available: https://tools.ietf.org/html/rfc7252

[12] C. Bormann and P. E. Hoffman, "Concise Binary Object Representation (CBOR)," RFC 7049, Oct. 2013. [Online]. Available: https://rfc-editor.org/rfc/rfc7049.txt

[13] Marvel. (2017, Dec) Marvel Alaska 88E1510. http://www.marvell.com/transceivers/assets/Marvell_Alaska_88E1510_18-002_product_brief.pdf.

[14] Xilinx. (2017, Dec) Vivado High-Level Synthesis. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[15] ARM. (2017, Dec) AMBA AXI4-Stream Protocol Specification. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0051a/index.html.

[16] Xilinx. (2017, Dec) AXI 1G/2.5G Ethernet Subsystem. https://www.xilinx.com/products/intellectual-property/axi_ethernet.html.

[17] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "NetFPGA SUME: Toward 100 Gbps as research commodity," *IEEE Micro*, vol. 34, no. 5, pp. 32–41, 2014.

[18] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 239–250.

[19] A. Celesti, D. Mulfari, M. Fazio, M. Villari, and A. Puliafito, "Exploring container virtualization in iot clouds," in *Smart Computing (SMARTCOMP), 2016 IEEE International Conference on*. IEEE, 2016, pp. 1–6.

[20] K. Varda. (2017, Sep) Introducing Cloudflare Workers: Run JavaScript Service Workers at the Edge. https://blog.cloudflare.com/introducing-cloudflare-workers/.

[21] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM, 2012, pp. 13–16.

[22] P. Liu, D. Willis, and S. Banerjee, "Paradrop: Enabling lightweight multi-tenancy at the networks extreme edge," in *Edge Computing (SEC), IEEE/ACM Symposium on*. IEEE, 2016, pp. 1–13.