FARD: Accelerating Distributed Fog Computing Workloads through Embedded FPGAs

Samuele Barbieri
DEIB, Politecnico di Milano
Milano, Italy
samuele.barbieri@mail.polimi.it

Fabiola Casasopra
DEIB, Politecnico di Milano
Milano, Italy
fabiola.casasopra@mail.polimi.it

Rolando Brondolin
DEIB, Politecnico di Milano
Milano, Italy
rolando.brondolin@polimi.it

Marco D. Santambrogio
DEIB, Politecnico di Milano
Milano, Italy
marco.santambrogio@polimi.it

Abstract

In the last few years Internet of Things (IoT) applications are moving from the cloud-sensor paradigm to a more variegated structure where IoT nodes interact with an intermediate fog computing layer. To enable compute-intensive tasks to be executed near the source of the data, fog computing nodes should provide enough performance and be sufficiently energy efficient to run on the field. Within this context, embedded Field Programmable Gate Array (FPGA) can be used to improve the performance per Watt ratio of fog computing nodes. In this paper we present Fog Acceleration through Reconfigurable Devices (FARD), a distributed system that exploits FPGAs to accelerate compute-intensive tasks in fog computing applications. FARD is able to efficiently run distributed fog applications thanks to a well-defined application structure, a per-application isolated network overlay and thanks to the acceleration of tasks. Results show energy efficiency improvements while efficiently enabling cooperation across fog nodes.

CCS Concepts •**Computer systems organization** → *Embedded systems; Distributed architectures;* •**Hardware** → *Hardware accelerators;*

Keywords Fog computing; FPGA; Fog acceleration

1 Introduction

In the last years, the development of IoT sensors and communication infrastructures allowed to build many complex and powerful applications deployed on the field. This increased the challenges on how to extract knowledge, considering also that most of these applications rely on cloud infrastructures for all the compute-intensive tasks required to process the data. When dealing with IoT applications, sensors should send all the data they measure to the cloud platform, wait for the cloud application to analyze the data and receive back results if the IoT nodes have to actuate on the monitored system. This process introduces non-negligible latency that is unsustainable for safety-critical applications and extremely expensive in general.

Within this context, fog computing [2] tries to bring the computation closer to the data, adding intermediate nodes between cloud and IoT layers that are able to handle compute-intensive tasks [6]. This allows to reduce the latency from the IoT node perspective and allows to send only aggregated data to the cloud infrastructure to save bandwidth. Fog nodes are usually hierarchically structured in order to serve both as intelligent nodes for computations and as routers for the data moving from IoT sensors to the cloud.

Given the roles of the fog computing nodes, the amount of computing power needed depends on the complexity of the applications and on the complexity of the environment in which the fog layer will operate. Many fog nodes are based on small yet powerful ARM nodes [1, 18], others are instead based on the x86 architecture [3]. Issues arise when the cores are not powerful enough to handle the application requirements (as in the ARM based scenario) or when the cores are too power-hungry to operate with limited energy resources (as in the x86 based scenario). To overcome this limitation, specialized hardware can efficiently implement compute-intensive kernels at a fraction of the energy cost. Cerina et. al [4] showed the feasibility of FPGA-based platforms in the fog scenario. However, to the best of our knowledge, a distributed fog computing platform based on heterogeneous nodes (CPU and FPGA) is still missing.

Within this paper, we present FARD, a distributed system that exploits FPGAs to accelerate compute-intensive tasks in fog computing applications. FARD is a peer to peer network of fog nodes, where the developer can build the applications with different structures and hierarchies. To foster research on this topic, we released FARD as open-source on *Github*. Given the research challenges described so far, the contributions of this paper are the following:

- the design and implementation of a distributed system for heterogeneous fog computing nodes that provides

EWiLi '19, October 17, 2019, New York, USA.

a defined application structure, a network layer for each application in isolation and the flexibility to build both peer to peer as well as hierarchical applications.
- the design and implementation of a heterogeneous fog application based on FARD and the Python productivity for Zynq (PYNQ-Z1) [12] platform able to accelerate a traffic count algorithm that outperforms the software implementation both in terms of pure performance and in terms of performance per Watt.

The rest of this paper is organized as follows: Section 2 lists the related works in the field, Section 3 describes the FARD design, Section 4 presents the *counting cars fog application*, Section 5 evaluates the fog computing system while Section 6 draws the conclusion and future work.

2 Related work

The foundational work of *Bonomi et al.* [2] identified the main characteristics that a fog computing layer should have to efficiently operate between the cloud computing layer and the sensor one. The fog layer usually has a large number of geographically distributed nodes receiving data from an even larger amount of sensors deployed on the field. It has to provide low latency communication between fog nodes, between sensors and the fog layer and between the fog layer and the cloud layer. It should support mobility when dealing with mobile devices and it should support real-time interactions (service-oriented instead of batch-oriented). It should support wireless access, it should be heterogeneous, and, finally, it should operate with many different protocols.

Previous works in the state of the art do not provide node to node communication [1, 3, 15], just supporting communication between each node with a remote cloud application or service. This approach is sufficient when a single fog node provides gateway functionalities to sensors, but more complex applications require coordination that this model is not able to support. To overcome this limitation, *Wen et. al* [18] addressed node-to-node interplays to parallelize the computation and then collect a single general result. Within this context, FARD enhances node to node cooperation by providing a network layer for each application and defining a communication mechanism able to implement both hierarchical structures as well as peer to peer ones.

For what concerns the hardware architecture, fog computing nodes usually leverage one of these standard architectures: ARM and x86. *Bellavista et al.* [1] explored and leveraged energy efficient ARM cores that can last on batteries at the cost of lower performance. *Caligoo* [3], instead, leverages x86 processors to handle compute-intensive tasks at the cost of lower energy efficiency that poses challenges when such devices need to be deployed on batteries. Another option is represented by embedded FPGAs, as they are able to meet both low energy requirements as well as enough computing capabilities. *Cerina et al.* [4] demonstrated that FPGAs are suitable as fog computing nodes. FARD builds on top of the PYNQ-Z1 platform to provide acceleration of fog computing tasks.

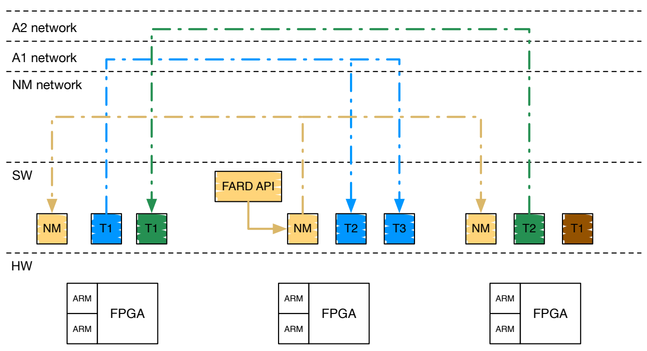

Figure 1. FARD distributed system with hardware platform, software stack, networking infrastructure and FARD APIs. Each node has a node manager and tasks depending on available resources. The FARD APIs connects to a node manager to control the applications and collect output.

For what concerns our case study, we implemented a counting cars algorithm on FPGA. This kind of algorithms are built as a pipeline of image filters able to isolate the background from the foreground elements, then isolate blobs and count them. Cameras used in traffic monitoring are usually fixed ones, that led us to implement a *basic motion detection* variant algorithm [7]. Generally speaking, blob counting algorithms grows in complexity depending on how many blobs the system should detect [5]. In our case, instead, the application is able to identify and count blobs with linear complexity w.r.t. the size of the input image. The only constraint is that the blob should not contain holes.

3 System design

FARD is a distributed system for fog computing nodes. It provides hardware acceleration for compute-intensive tasks as well as distributed run-time management of fog tasks to enhance their cooperation. The hardware side of FARD leverages the Xilinx Vivado Design Suite [17] as well as the libraries provided by the PYNQ-Z1 platform. The software side, instead, is built on top of *ZeroMQ* [10] and *Pyre* [13].

3.1 System overview

The distributed system behind FARD was designed with four principles in mind: *application structure*, *event driven computation*, *network cooperation* and *application isolation*. The *application structure* of workloads running inside FARD is well defined. The system is built upon the concepts of *peer*, *task*, *event* and *application*. An application is composed of a set of different *tasks*. Each instance of a *task* is called *peer*, and many *peers* can be a replica of the same *task*. Given that the fog nodes computation happen as a response to the generation of data from the IoT sensors, the natural way

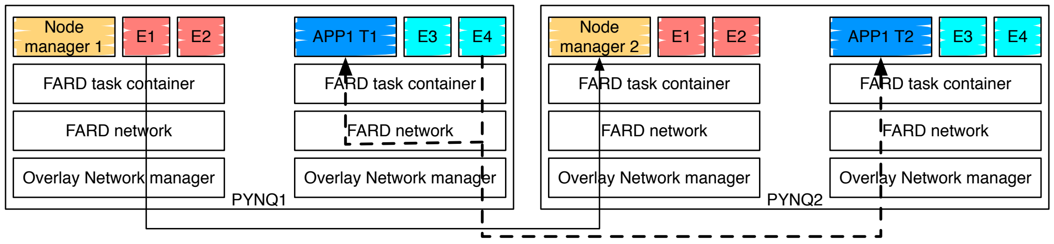

Figure 2. FARD events can be exchanged as broadcast, multicast or unicast. As an example, event 1 of node manager 1 is sent as unicast to node manager 2 while event 4 of APP1 is sent as broadcast to all the tasks of APP1, bypassing the network for the local sender task.

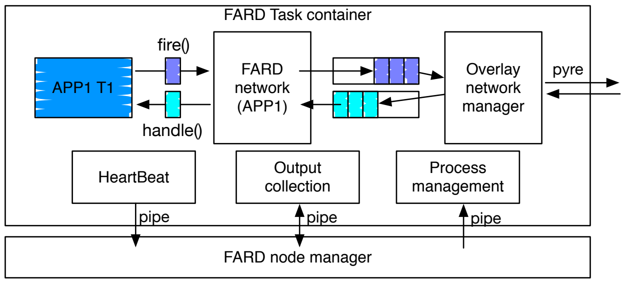

Figure 3. FARD task structure with network components, task container, heartbeat system, output collection and process management.

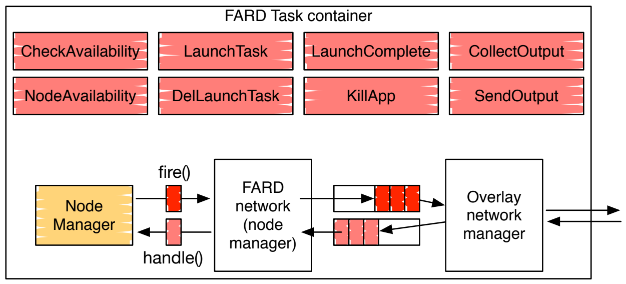

Figure 4. FARD node manager structure with events handled and network components.

to represent this behavior is through *event driven computation*. *Peers* of the same *application* communicate with each other using *events*. Each event can be sent in broadcast to all the *peers* in the *application*, in multicast to a set of *peers* belonging to the same *task* or in unicast to a specific *peer*. This allows a FARD *application* to enable *network cooperation* across nodes. Finally, FARD should be able to support multiple different applications running inside the distributed system. For this reason we developed a *network overlay* system with *ZeroMQ* and *Pyre* to enforce *application isolation*.

Figure 1 shows the main components of the FARD distributed system. On top of the concepts described so far, we built the *Node Manager*, which is the component of FARD responsible for life-cycle management of the FARD *applications* and for the resource management of the PYNQ-Z1 nodes. Finally, we developed the *FARD APIs* to let users communicate with the node managers and the applications. The APIs connect with the local *node manager* to send commands. The APIs integrate with the PYNQ-Z1 platform and can be used within the Jupyter notebooks of the boards. Within this context, a developer can build an *application* both leveraging a peer to peer design as well as a fully hierarchical approach to follow the Open Fog consortium specifications [8].

In the next sections we will detail the design of FARD, starting from the *network design* in Section 3.2, moving to the *application structure* in Section 3.3 and finally describing the *node manager architecture* in Section 3.4.

3.2 Networking system

The networking system of FARD is based on *pyre*. This library provides client's naming and automatic discovery of nodes and peers through a local network. On top of *pyre* we built the *overlay network* system. FARD creates for each *application* a networking channel where each *peer* can attach to communicate with other *peers* in the local network. This allows to isolate messages on a per *application* basis, however, no other FARD component can access this network to send control commands to the *peers*. To overcome this limitation, the *Node Manager* creates a process pipe with each *peer* it handles to send the management commands.

Figure 2 shows the network components and some examples of communication between *peers* belonging to the same *application*. As detailed in Section 3.1, each *event* can be sent in broadcast to each *peer*, multicast to *peers* that are instance of a given *task*, or unicast from a *peer* to another. This allows the *peers* to share responsibilities in the computation and to build different application structures. Broadcast events can be used to notify *peers* of events that have to be managed at the whole *application* level, while multicast events can be used to launch computations specifically for a given *task*. Finally, unicast events can be used to build chains of computation with different *peers* geographically distributed. To properly work, each *peer* should instantiate the networking system components (e.g. the *FARD network* and the *Overlay network manager*).

Figure 3 and Figure 4 show instead how the networking system works for a FARD task and for the node manager respectively. When a *peer* sends an event, it calls the *fire()* method of that event providing a message as payload. The message is then sent through the FARD network and the overlay network manager, which is the component that handles the *pyre* socket. Then the sender can either wait synchronously or continue executing depending on the message type. On the receiver side the *handle()* method is called asynchronously when the message arrives.

3.3 Application structure

As we mentioned in Section 3.1, each *application* is composed of a set of *tasks*. A *task* is a Python class that extends the *FardTask* class, implementing the methods necessary to properly run it: *init()*, *run()* and *stop()*. The *init()* method is generally used to set up the *task* run, in particular it is used to configure the FPGA for the first time when the *task* is launched. The *run()* method is instead the main method of the task and runs continuously until the *task* is deleted from the system. When this happens, the *stop()* method is called, allowing the user code to clean the resources used by the *task* before deleting it from the node.

The *Node Manager* is the component that launches the *tasks* on its node. To do so, it encapsulates the user code inside the *FARD task container* (shown if Figure 3), which executes the *task* inside a thread, while it instantiates another thread to execute the health-checks required to monitor the *task*. This second thread is also responsible for the communication with the *Node Manager* and it periodically checks if the *task* should be restarted or removed from the system.

3.4 Node manager architecture

The FARD *node manager* (shown in Figure 4) is the component responsible for the life-cycle management of the applications running inside the distributed system. It provides facilities to launch and remove applications, it checks if an application can be scheduled on the system depending on its hardware requirements, it monitors each *peer* health and status, and it collects log data when requested. Thanks to the flexibility of the concepts that we leveraged to build FARD, the *node manager* is implemented in the same way as any other application that runs inside the distributed system, thus it represents a useful example of how a complex application can be designed within FARD. When the *node manager* is loaded onto each board, it loads a configuration file that states the amount of millicpus (1000 millicpus equals to 1 core) available, the number of FPGAs available and other useful runtime information.

3.4.1 Task distribution and admission control

The most important activity the *node manager* carries out during its execution is the admission control of tasks and its distribution across the fog nodes depending on its resource requirements. When a user requires to launch a new *application* through the APIs, the request is forwarded to the local *node manager* running on the same node of the APIs. The *node manager* then receives the application code and the bitstream, pack them in a zip file and send the archive to a set of *node managers* depending on the number of *tasks* and *peers* defined in the application configuration file. When a *node manager* receives an archive, it unpacks it and starts to execute the *peers* assigned to the given node.

In order to decide to which *node managers* we should send the archive and how many *peers* each *node manager* should run, we perform a *node availability check*. This procedure verifies whether the system is able to accept the application or not and it is based on five different events: *CheckAvailability*, *NodeAvailability*, *LaunchTask*, *DelLaunchTask*, and *LaunchComplete*. The local *node manager* sends a request for availability to all the nodes in the system through the *CheckAvailability* event. Each node then replies to the local *node manager* with its free resources using the *NodeAvailability* event. When the local *node manager* receives all the replies, it decides onto which nodes the application should run and sends to them a *LaunchTask* event with the code and the bitstreams, while it sends a *DelLaunchTask* to the remaining nodes to free the temporary allocated resources. At this point, the local *node manager* waits for the *LaunchComplete* events from the other *node managers* to return the application ID to the caller. If the process fails at some point, the local *node manager* asks to the other nodes to delete the *peers* already launched and returns a negative response to the FARD APIs.

3.4.2 Task removal and cleaning

Given that the FARD *applications* are designed as services, they usually do not end unless the user requires to remove them. To do so, the user has to call the delete command from the FARD APIs, sending as input the application ID. When this happens, the local *node manager* sends a *KillApp* event to all the nodes in the system. When a *node manager* receives this kind of event, it checks using the application ID if a *peer* of that application is running in its node. If the *peer* is found, the *node manager* sends a kill command to gracefully close it. If the *peer* does not respond, the *node manager* waits for a timeout and then kills the *peer* directly. Then, the files of the task are deleted and the state of the system cleaned for the given application.

3.4.3 Applications output management

Another important aspect of FARD is the collection of logs and outputs from the applications. When this functionality is enabled, the system collects the output from each peer and forwards it to the FARD APIs that requested it. To implement this functionality, we need to setup some channels to move data from the *peer* to the *node manager* and from the *node manager* to the APIs. From the *peer* to the *node*

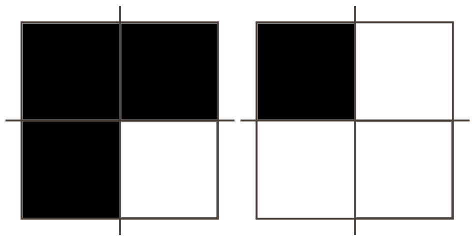

Figure 5. Graphic representation of the kernel that computes internal and external corners for black and white images.

manager we leverage the pipe already in place, while for the *node managers* communication we use the *SendOutput* event. When this kind of events arrive to the local *node manager* that requested the data, the content is unpacked and pushed to a queue consumed by the FARD APIs.

4 Case study: counting cars

We developed a case study to analyze the performance of FARD and to evaluate whether the use of FPGAs fit in a fog computing environment. In particular, we developed an image processing algorithm to retrieve the number of cars in a frame of a video coming from fixed cameras across a road.

The algorithm performs a series of image filters, then identifies and counts blobs in an image. At first, we perform background subtraction on the gray-scale image with a threshold of 12 to exclude background details from the foreground scene where we can find the cars. Then we apply 2 times the erosion filter. Then we apply 30 time the dilation filter and finally a last pass of the erosion filter. These steps allow to obtain images with blobs that do not contain holes. The final output images are composed of a black background and a white foreground representing the blobs to count.

The blob counting problem can be reduced to a corner counting problem, where a blob is an area delineated only by angles. These shapes have the characteristic of guaranteeing that the number of external corners minus the number of internal corners is constant, four in our case. Given the difference between internal and external corners, we can compute how many blobs the image contains by dividing the number by four. To identify the corners the algorithm scans the image with a kernel of 2×2 (shown in Figure 5). If the sum of the four pixels is one, we have an external corner, while we have an internal one if the sum is equal to three.

We implemented this algorithm as a hardware accelerator for the PYNQ-Z1 platform, where the accelerator computes the filters on the image and counts the blobs, while the software layer collects the image from a camera or a file, sends it to the accelerator and collects the output of the algorithm. We then used FARD to build the fog application. The nodes are organized in a chain, where each node sends the number of outgoing cars to the next one. The last node collects the data, creates a per node balance of incoming and outgoing cars and sends it to the FARD APIs each second.

5 Experimental evaluation

Within this section, we will evaluate the proposed distributed system as well as the case study presented in Section 4. Section 5.1 will detail the experimental setup we realized to evaluate FARD and the hardware implementation. Section 5.2 will instead present the experimental results for the counting cars application, while Section 5.3 will present the results of the distributed system.

5.1 Experimental setup

To properly test the proposed system in a lab testing environment, we deployed three PYNQ-Z1 connected through a Gigabit Ethernet switch. On top of this deployment, we run both the tests on the case study application as well as the distributed system evaluation, where the case study was evaluated against a CPU implementation. Details of the hardware platforms are the following:

- **CPU:** Intel Core i7-4700MQ @ 3.40GHz, *RAM* 12GB DDR3, *OS* Arch Linux with 5.0.13 kernel;
- **PYNQ:** ARM Cortex-A9 with FPGA, *RAM* 512MB DDR3, *OS* Ubuntu 16.04 with 4.9.0 kernel. *FPGA* 280 BRAM 18K, 220 DSP, 106400 FF, 53200 LUT.

For what concerns the case study, we used as input a video coming from a highway traffic camera obtained from the MUOVEE[11] website. We then cropped and scaled it according to the hardware accelerator parameters (maximum size of the frames is 180×180 pixel). We then extracted the background frame used to exclude environment details. The hardware accelerator was designed with the Xilinx Vivado design suite according to the algorithm described in Section 4 and targeting the PYNQ-Z1 board. The accelerator was then integrated into the full embedded system. For what concerns the software implementation, we leveraged the Intel Core i7-4700MQ because it is quite common to have platforms with this kind of computing power in fog scenarios [9, 16]. The algorithm was developed in C++ as a single thread application and compiled with GCC with all optimizations active (*-O3*). Multithreading is not required in this case as the performance of the application already allows to process a frame in real-time and wait for the subsequent frame with a single worker.

For what concerns the distributed system evaluation, we implemented three simple applications that send messages in unicast, multicast and broadcast. Within unicast we used two nodes, while we leveraged three nodes in the multicast and broadcast cases. We performed experiments with different message rates and for each experiment we measured the Round Trip Time (RTT) of each message going from the sender to the receiver(s) and then back to the sender.

Table 1. Resource utilization of the kernel on the FPGA in terms of BRAMs, DSPs, FFs and LUTs.

	Total Used	Total Available	Utilization (%)
BRAM_18K	178	280	63
DSP48E	6	220	2
FF	4379	106400	4
LUT	5643	53200	10

Table 2. Comparison between software (i7-4700MQ) and hardware (ARM + PYNQ-Z1) implementations for execution time per frame, power consumption and FPS/Watt, with relative speedups.

System	Time per Frame [ms]		Average Power [W]		FPS/Watt
	Mean	Variance	Mean	Variance	
CPU	6.095	1.004	16.751	0.321	9.80
PYNQ	1.704	0.038	1.762	-	333.06
speedup	3.75x				33.98x

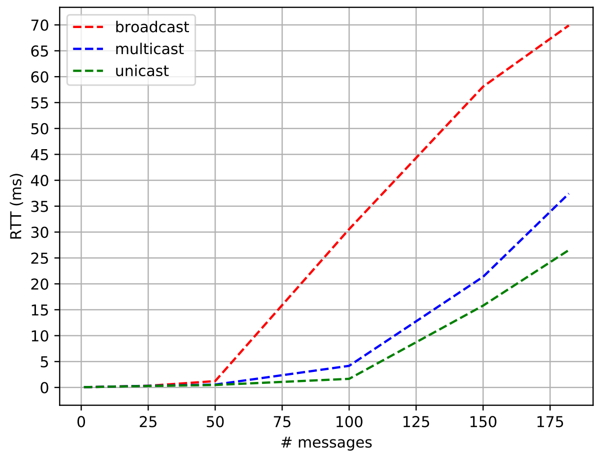

Figure 6. RTT of messages sent with different strategies (broadcast, multicast and unicast). Lower is better.

5.2 Counting cars performance

The utilization of the hardware resources of the PYNQ-Z1 for the counting cars accelerator is shown in Table 1. The application uses 63% of the total available BRAMs because both the background image and the current frame are stored in the on-chip memory. This is the heaviest resource utilization of the accelerator and can be mitigated by leveraging the DDR memory. This will enable to increase the number of cores implemented in the FPGA, to increase the image size, and to improve the performance of the algorithm. In fact, if we look at the other resources (e.g. DSPs, FFs and LUTs), we can see that they are used for at most $\simeq 10\%$.

The performance results of the counting cars application are shown in Table 2. For each platform, we list the execution time to compute each frame, the average power consumption and the performance per watt metric (expressed as Frames per Second (FPS)/Watt). We ran each experiment 10 times averaging the results and for each frame we compute the gray-scale, the filtering process and then the blob count algorithm. Power consumption measurements were taken from post-implementation power estimation for the *PYNQ*, while we used the Linux tool *perf* to collect the i7 power consumption leveraging Intel Running Average Power Limit (RAPL) [14] measurements for the CPU and the memory subsystem.

Considering the execution time per frame, Table 2 shows that both implementations are able to provide real-time processing of video frames, as most of the surveillance cameras generate 30 FPS. However, we can see that the hardware implementation is able to outperform the software one by 3.75x even if the PYNQ-Z1 includes only a small FPGA. Moreover, if we consider the energy efficiency and the FPS/Watt ratio, the hardware implementation outperforms the software one by 33.98x. These results are a first indication that FPGAs can be exploited in fog computing scenarios to provide better performance and energy efficiency. Finally, given the power consumption measurements of both platforms, we can say that the hardware implementation of the counting cars algorithm can run on batteries, while the software implementation will have issues if we consider also disks and other hardware subsystems.

5.3 Distributed system evaluation

To correctly evaluate the proposed distributed system design, we can leverage three metrics: fog network latency, network bandwidth savings and number of distinct services.

For what concerns fog network latency, Figure 6 shows the messages RTT for each communication mechanism for increasing amount of messages sent. Unicast shows the highest performance, as this mechanism implies direct messaging between two *peers* that already know each other's name. The RTT slightly increases for the multicast case, as the network layer of FARD has to look-up the *peers* belonging to the *task* that will receive the messages. Finally, broadcast shows the highest RTT, as it relies on the broadcast messaging system of *pyre*. The broadcast strategy is able to send 182 messages per second managing the replies. If the sender does not need to receive data back (as in counting cars), FARD can broadcast 780 messages per second.

For what concerns network bandwidth savings w.r.t. the communication with the cloud layer, we can leverage the counting cars case study to analyze the impact of the fog. The case study takes as input a video stream of 180×180 pixel. Each frame has an uncompressed size of $\simeq 94KB$. Thus for each second and for each camera, an IoT node would send to the cloud at most $\simeq 2.75MB$. If the video stream is compressed (as the videos of the MUOVEE[11] website), this number shrinks down to $\simeq 35KB$ per node per second. The fog implementation instead sends just the number of incoming and outgoing cars per node, reducing the bandwidth usage to 8 Bytes per second per node.

Finally, if we consider the number of distinct services in the fog, FARD supports at most one *peer* connected to the FPGA per node, while many *peers* that use just the ARM cores can be instantiated on a single node depending on how many millicpus each *peer* requires.

6 Conclusion and future work

Within this paper we presented FARD, a distributed system for fog computing nodes that provides hardware acceleration for compute-intensive tasks. The distributed system behind FARD was designed with four principles in mind: *application structure*, *event driven computation*, *network cooperation* and *application isolation*. To show the feasibility of FPGAs in fog computing environments, we implemented and evaluated a geographically distributed fog application to count cars flowing in a camera video. The experimental results show good performance of the FARD distributed system in terms of latency, while the hardware implementation guarantees real-time execution and outperforms an x86 implementation in terms of pure performance and FPS/Watt.

Future work of FARD will revolve around the design, implementation, and management of stream operators and batch jobs throughout the distributed system. Finally, we will integrate fault-tolerance techniques to improve the availability of nodes and *peers* during the lifetime of the system.

References

[1] P. Bellavista and A. Zanni. Feasibility of fog computing deployment based on docker containerization over raspberrypi. In *Proceedings of the 18th International Conference on Distributed Computing and Networking*, page 16. ACM, 2017.

[2] 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*, pages 13–16. ACM, 2012.

[3] Caligoo. http://www.caligoo.com. [Online; accessed 30-Jun-2018].

[4] L. Cerina, S. Notargiacomo, M. G. Paccanit, and M. D. Santambrogio. A fog-computing architecture for preventive healthcare and assisted living in smart ambients. In *Research and Technologies for Society and Industry (RTSI), 2017 IEEE 3rd International Forum on*, pages 1–6. IEEE, 2017.

[5] T.-H. Chen, Y.-F. Lin, and T.-Y. Chen. Intelligent vehicle counting method based on blob analysis in traffic surveillance. In *Innovative Computing, Information and Control, 2007. ICICIC'07. Second International Conference on*, pages 238–238. IEEE, 2007.

[6] K. Dolui and S. K. Datta. Comparison of edge computing implementations: Fog computing, cloudlet and mobile edge computing. In *Global Internet of Things Summit (GIoTS), 2017*, pages 1–6. IEEE, 2017.

[7] E. GREYC. Comparative study of background subtraction algorithms.

[8] O. C. A. W. Group et al. Openfog reference architecture for fog computing. *OPFRA001*, 20817:162, 2017.

[9] K. Habak, M. Ammar, K. A. Harras, and E. Zegura. Femto clouds: Leveraging mobile devices to provide cloud service at the edge. In *2015 IEEE 8th international conference on cloud computing*, pages 9–16. IEEE, 2015.

[10] P. Hintjens. *ZeroMQ: messaging for many applications.* " O'Reilly Media, Inc.", 2013.

[11] Muovee. https://www.mouvee.com. [Online; accessed 30-Jun-2018].

[12] Pynq platform. http://www.pynq.io/. [Online; accessed 30-Jun-2018].

[13] Pyre runtime. https://github.com/zeromq/pyre. [Online; accessed 30-Jun-2018].

[14] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, and D. Rajwan. Power-management architecture of the Intel microarchitecture code-named Sandy Bridge. *IEEE Micro*, 32(2):20–27, Mar. 2012.

[15] S. Soma and A. Patil. Novel architecture for iot based video streaming over cloud.

[16] I. Stojmenovic. Fog computing: A cloud to the ground support for smart things and machine-to-machine networks. In *2014 Australasian Telecommunication Networks and Applications Conference (ATNAC)*, pages 117–122. IEEE, 2014.

[17] Vivado design tools. https://www.xilinx.com/products/design-tools/vivado.html. [Online; accessed 29-Jun-2018].

[18] Z. Wen, R. Yang, P. Garraghan, T. Lin, J. Xu, and M. Rovatsos. Fog orchestration for internet of things services. *IEEE Internet Computing*, 21(2):16–24, 2017.