Edge Chaining Framework for Black Ice Road Fingerprinting

Vittorio Cozzolino
Technical University of Munich
Munich, Germany
cozzolin@tum.de

Aaron Yi Ding
Delft University of Technology
Delft, Netherlands
aaron.ding@tudelft.nl

Jörg Ott
Technical University of Munich
Munich, Germany
ott@tum.de

ABSTRACT

Detecting and reacting efficiently to road condition hazards are challenging given practical restrictions such as limited data availability and lack of infrastructure support. In this paper, we present an edge-cloud chaining solution that bridges the cloud and road infrastructures to enhance road safety. We exploit the roadside infrastructure (e.g., smart lampposts) to form a processing chain at the edge nodes and transmit the essential context to approaching vehicles providing what we refer as road *fingerprinting*. We approach the problem from two angles: first we focus on semantically defining how an execution pipeline spanning edge and cloud is composed, then we design, implement and evaluate a working prototype based on our assumptions. In addition, we present experimental insights and outline open challenges for next steps.

1 INTRODUCTION

The presence of a smart roadside infrastructure in the cities of tomorrow (e.g. smart lampposts [3]) offers opportunities for building new applications working with and providing fine-grained, localised information. Consequently, more detailed and precise maps of metropolitan areas can be generated to support applications in, for instance, the health and safety domains. City-wide pollution fingerprinting can enable pedestrians and cyclists to select less polluted routes, while infrastructure-supported black ice detection can allow drivers to predict the presence of patches of black ice outside their field of view.

Numerous applications fully exploit crowdsourcing to generate *augmented* maps which, however, have limited dimensionality in terms of collected data. In fact, they rely on users' hand-held devices, which may not be equipped pollution or particulate sensors or infra-red (IR) thermal cameras, which are fundamental for the aforementioned functions.

As smart cars become widespread, the availability of sensors will expand the information domain of crowdsourcing applications; however, there are multiple limitations. First, data availability: data collected by car manufacturers is often not publicly available, thus, creating a disparity in the quality of service offered among providers. Second, bad weather conditions, limited range of car sensors and lack of enough reliable data to properly map the road condition can reduce the effectiveness of crowdsourcing solutions and eventually provide false predictions putting at serious risk the drivers. This problem is exacerbated in areas with poor network connectivity where vehicles are hardly reachable from the cloud and left in the dark about the presence of road hazards. Section §2 will provide a more detailed discussion about such issues.

The problem we wish to solve concerns *how to provide reliable information regarding road hazards to vehicles in challenging conditions with the support of a smart roadside infrastructure.* Our primary use case is black ice detection, which we tackle with an *edge-cloud pipelining* concept to create on-demand execution pipelines spanning edge and cloud nodes. Involved edge nodes (ENs) form execution chains and follow a specific protocol in order to collaboratively contribute to the task completion. Assessed road conditions are then broadcast to approaching vehicles. In this paper, we build upon our previous groundwork [7], develop a new edge chaining framework, and contextualize it for the above-mentioned use-case.

The remainder of this paper is structured as follows: framing the problem (§2), background (§3), edge-cloud pipeline (§4), system design (§5), implementation (§6), evaluation (§7), conclusion and future work (§8).

2 FRAMING THE PROBLEM

With the growth in deployed smart devices and the presence of physical infrastructure in proximity to end-users, there arises the challenge of constructing a platform that can provide **accurate**, **reliable** road conditions information at **scale**.

Detecting road conditions and potential hazards is a problem that has been explored and approached in the literature using different approaches. Both crowdsourcing solutions, where vehicles exchange collected information to identify bumps [5], and infrastructure based solutions as in [13], where IR cameras mounted on lampposts are used to detect ice formations on the road, have been explored. Through different approaches and tools, various studies examining the effectiveness of detecting road conditions have been conducted [11, 12].

Eriksson et al. [8] proposed pothole patrol (P2), a mobile sensing application used for detecting and reporting road surface conditions. In a similar system used in traffic sensing and communication, Mohan et al. [16] proposed the use of mobile devices connected to exterior sensors. Mednis et al. [15] improved and extended the P2 system using a customised embedded gadget and with the aid of a smartphone hardware platform for sensing road surface conditions [21]. Edge-computing-based approaches have been also explored, as in [5]. However, their focus is on security implications, while the focus of this study is to formalize the system requirements and build a working distributed edge computing platform.

Existing solutions focus on using either crowdsourcing or edge networks for transferring information. However, the quality of crowdsourced spatial data is often unreliable [6]. Consequently, the density of effective data points for estimating road conditions can be insufficient in low-traffic areas. Moreover, solutions based on on-board car sensors are also unsatisfactory in circumstances where road characteristics (e.g. buildings, trees, turns, crossroads) and adverse weather conditions effectively inhibit the ability to detect hazards at a distance. To overcome this challenge, we take advantage of road infrastructure by extending the sensory capacity

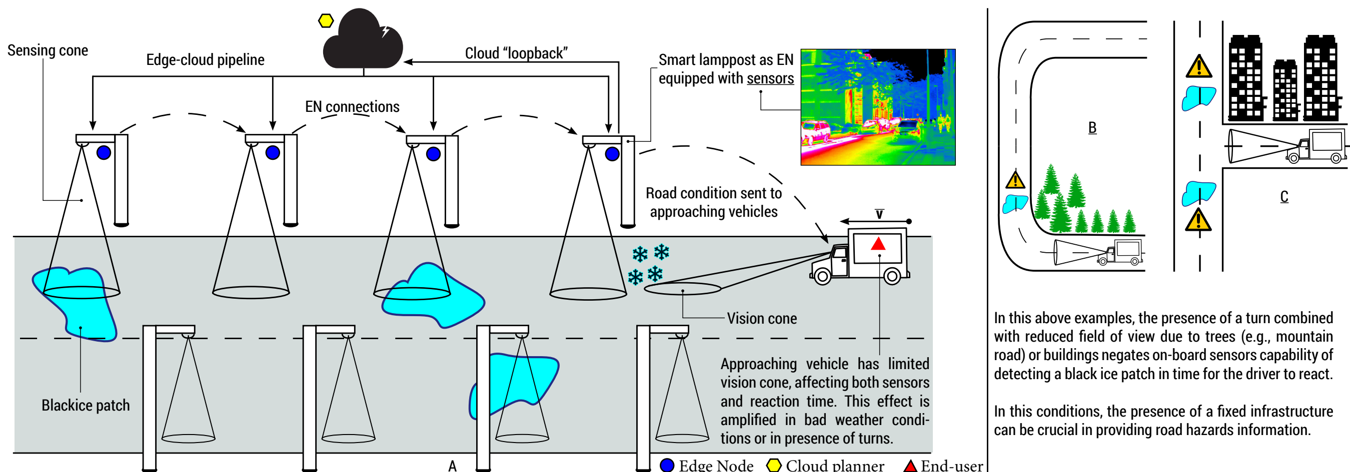

Figure 1: Black ice road fingerprinting (A); critical scenarios (B, C)

of cars beyond what can be captured by on-board sensors, and use edge computing as a core technology.

In this paper, first, we tackle the problem of semantically representing a distributed task that spans multiple ENs. Then, we introduce the definition of an edge-cloud pipeline and describe the process of splitting it into local sequences. Finally, we identify the required software components and emerging technologies that can fulfil the desired functions.

3 BACKGROUND

Here, we introduce several concepts and definitions that are used throughout the remainder of the paper.

Edge node (EN). The definition og this term is quite broad. We agree with the definition provided in [20], in which edge computing generally occurs in proximity to datasources. Hence, an EN is a device very close to the end user, such as a base station, mobile phone or private PC. Other classifications of ENs extend the definition to RAN microservers [14]. In this work, we focus on ENs in the range of micro-servers such as Intel NUC and Dell Optiplex which were used in our experiments.

Edge network. This is a network of ENs interconnected via a wired or wireless connexion. The ENs are in physical proximity to one another, such as lampposts on the side of a road. Detailed characteristics of an edge network are not described here, as they are discussed in other papers.

Task and edge function (EF). In this paper, we use the term *tas* to refer to an operation to be carried out by the network. At a high level, a task can be expressed as follows: *'find black ice patches at road intersection 1A and 3B'*. A task can be reduced to an ensemble of EFs: self-contained, *atomic* applications in which a small fraction of the task logic is embedded, but can be executed in a standalone fashion. A task contains at least one edge function. Distributed complex event processing [18, 19] is an example of a task composed of multiple sub-functions. To improve performance and scalability, these sub-functions are moved closer to the data sources.

Edge-cloud pipeline. This pipelines is a tasks issued by a cloud provider to edge nodes, and contains information regarding involved nodes, chaining order and data sources. Nodes involved in the pipeline form execution chains and collaborate to solve the task; they are selected based on multiple parameters and execute only a subsection of the entire pipeline. Pipelines can be ephemeral or recurrent, based on the task requirements. More details are provided in §4.

4 EDGE-CLOUD PIPELINE

In this section, we describe in detail our representation of the edge-cloud pipeline and its application to our prime use case: blackice road fingerprinting. Three main elements are involved: the cloud planner, the edge infrastructure and the vehicles. Due to page limit, the vehicles are not discussed in details as their role is passive in relation to our system — they receive the information from the infrastructure via, for instance, long-range communication radios such as LoRaWAN [2] or Vehicle Fog Computing [10].

4.1 Cloud planner

Cloud services define the pipeline structure and monitor its execution. We assume knowledge is available regarding th reachability of edge devices, their available data and their current load (in terms of active EFs and pipelines). Based on this assumption, it is possible to plan a pipeline execution tree based on a set of parameters among which *data locality* has a prime role. Once offloaded, the pipeline can be configured to run independently from the cloud based on specific policies.

Once the structure of the pipeline is defined, the cloud prepares a meta-pipeline containing various information about the pipeline itself, as illustrated in Figure 2. In our use case, the cloud provider may be a car manufacturer that wishes to offer augmented maps to its fleet [1] and, thus, exploits edge infrastructure to collect and analyse road condition data.

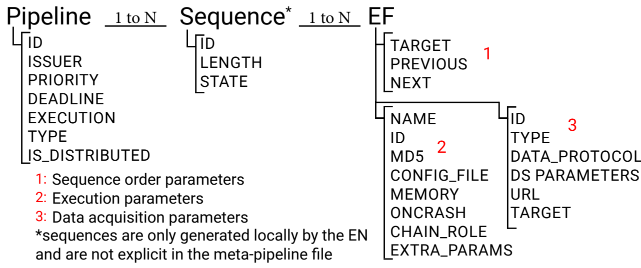

Figure 2: Meta-pipeline

4.2 Edge infrastructure

In relation to our use case, black ice detection requires each node to locally process thermal images acquired from IR cameras or similar sensors, identify patches of black ice and send the results to the following node. As Figure 1 shows, we assume that ENs are deployed inside smart lampposts that are equipped with an array of sensors able to detect road conditions.

The edge infrastructure is composed of manifold ENs each addressable by an unique identifier such as their location (e.g. GPS coordinates). When a meta-pipeline is offloaded, the involved ENs parse it and identify which sections has to execute and in which order relative to the other nodes. Each pipeline is split into sub-pipelines, which in turn are transformed into sections that can contain multiple stages, which are eventually, but not immediately, executable. Pipelines can be sorted by multiple parameters: priority, expected load and deadline which can alter the execution order.

Once the sub-pipeline execution order is assessed, each EN has to verify the reachability of the neighbour ENs in the pipeline and prepare to exchange and process data with them (here we assume that there are no issues in terms of network reachability).

4.3 Pipelines: flexible chaining at the edge

One concept that requires further explanation is the relationship between pipeline and sequence. As mentioned above, ENs scan the meta-pipeline and select the list of EFs to be executed locally. These EFs can have different roles, such as *head*, *transit* and *tail*. This classification is necessary because it determines the order in which the EFs should be chained and executed. The ordering is based on the identifiers associated with each EF. Identifiers (IDs) are assigned to pipelines, sequences, EFs and data acquisition rules, as demonstrated in Figure 2. *Head* and *tail* roles always represent the end of a sequence, while transits are EFs that serve as links between the head and tail. In other words, a sequence always begins with a head and ends with a tail; they are not necessarily deployed on the same EN, and there can be an arbitrary number of transit instances between the two. *Sequence order parameters* are used to correctly unfold the execution. Figure 4 illustrates different pipeline and sequence structures.

A key property of the pipeline is **chaining**. ENs create temporary, dynamic execution chains based on the pipeline topology in order to form a collaborative network. As data flows from one link of the chain to the other, computation unfolds and the pipeline progresses towards the *tail* EN. In addition, pipelines are not necessarily linear, they can *branch* and *join* creating execution trees. For instance, branching may be required in situation where groups of ENs provide different information to be subsequently merged at the end of a pipeline.

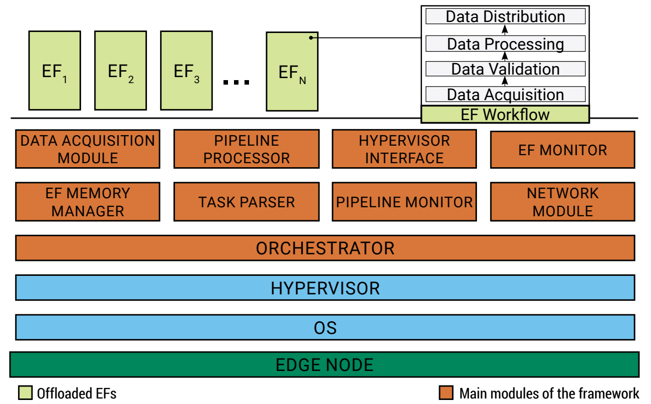

Figure 3: System modules overview

Each EF has data acquisition rules that can be split into input and output. The former determines where the data should be retrieved, while the latter specify whether the result of the computation should be sent to a remote node to continue along the pipeline or should be stored locally for future use. Additional details concerning this process are provided in section §5.

5 SYSTEM DESIGN

In this section, we provide an overview of the system (as presented in Figure 3) in relation to the discussed use case followed by a description of the core components necessary for pipelining. To conclude, we briefly describe the system workflow.

5.1 Overview

In the scenario in which roadside infrastructure is used exclusively as an adjunct of the cloud, a slow connexion or the lack thereof can invalidate the purpose of the entire system as follows: information about the road conditions is slowly retrieved or not retrieved at all, and vehicles are left without information about potential hazards, which can potentially cost lives. Hence, edge computing has the critical role of being a **reliable, resilient and autonomous infrastructure** that delivers services even when the cloud is unreachable.

As vehicle density on side-roads might not be sufficient to map reliably the road conditions, crowdsourcing in unable tackle this problem. Available spatial data is very sparse and limited, leading to incomplete or misleading information distributed to vehicles driving in low-traffic areas. In terms of on-board car sensors (when available), there are two key situations in which their effectiveness is hampered: harsh weather conditions where visibility is heavily reduced (e.g. blizzard, hailstorm, fog), or the scenarios illustrated in Figure 1 (B,C). In both cases, the vehicle and driver fields of view are reduced, limiting the sensor detection capabilities and reaction time, respectively.

Our framework is designed to solve the aforementioned challenges, as it provides a framework where multiple cloud services can share existing infrastructure, which in turn facilitates scheduling and handling multiple offloaded tasks. It offers computation

at the ENs, enabling both independence from the cloud in case connectivity loss and efficient processing and aggregation of information based on EFs chaining. In addition, local broadcasting of road hazards to approaching vehicles affects end-users, who can benefit from a standalone infrastructure without requiring a mobile connexion.

5.2 Core components

Our system pivots around the concept of edge offloading [17]: an emerging paradigm by which computation can be moved from the cloud towards edge nodes in order to provide multiple benefits. To differentiate from similar solutions, we design our system as a collaborative framework where multiple ENs can be chained to execute different tasks. In order to support and manage the offloaded EFs, we developed a set of modules residing and constantly running directly on each EN.

Orchestrator. It's the core and entry point of our system, and it functions as both a coordinator and interface with the outside world. A REST interface is employed to interact with it. When one or more EFs are offloaded as part of a pipeline, the orchestrator handles the calling of the required modules to philtre, order and execute the EFs. Both single execution and recurrent pipelines are supported. For resiliency purposes, checkpoints of the pipeline status plus EFs intermediate results are stored in a local database.

Edge function. The EFs are virtual instances moved from the cloud to edge devices as part of the pipeline. They are composed of four parts: data acquisition, validation, processing and distribution. During the first phase, an EF awaits the necessary data from the orchestrator (intra-node communication). Next, it proceeds to the validation phase, in which received data is checked for errors in case of faulty transmission, eventually requesting a re-transmission. The data processing phase contains the developer code and is the core of the EF. By customising this part of the EF, it is possible to execute any computation inside the EF, granted that eventual external dependencies and libraries have been handled. Finally, the distribution phase contains rules that specify whether outputted data should be dumped to the host or sent to the next EF in the local sequence.

Intra-node and inter-node communication. Based on the pipeline structure, data can be exchanged in two ways. Intra-node communication occurs when the transfer involves two co-located EFs or an EF and the orchestrator. In contrast, inter-node communication occurs when the transfer takes place between two EFs that are not co-located. For the former, a set of parameters is provided for identifying the source and destination addresses of shared memory pages blocks used to transfer the data. When two co-located EFs are involved, the transfer is performed automatically and requires no involvement from the host modules. In other cases, the host *memory manager* module has the task of allocating, deallocating and cleaning up the memory pages that are continuously used to communicate with each EF. EFs are unable to fetch data from a local or remote source; they exist in a completely sealed environment. Hence, the host framework must also handle the collection of the required data from the data source specified in the respective meta-pipeline section.

Network module. This module enables the communication between non-co-located ENs. It has two groups of queues containing data structures called *bundles*, which contain a set of parameters used to unequivocally identify a pair of producer and consumer ENs. A combination of IDs extracted from the meta-pipeline is used for this purpose. The bundles in the inbound queue are stored until consumed by one or more local EFs which remain in a waiting state until the specific data is available. The outbound queue contains the bundles that are ready to be forwarded to the next EN in the pipeline. The outbound queue is also necessary in case of failures in some stages of the pipeline; data bundles are in fact stored until the malfunctioning nodes are prepared to continue. For additional resiliency, the bundles are also stored inside a database to allow hot restarts of the system in case of local failure.

5.3 Workflow

The cloud service sends the generated pipeline to each selected EN. The orchestrator extrapolates relevant information from the meta-pipeline file with the *task parser* module. Next, the *pipeline processor* generates the local sequences and boot-up the required EFs while the *pipeline and the EF monitor* supervise the status of the respective components. Depending on the data provenance, either the *data acquisition module* or the *network module* prepare a data bundle subsequently passed to the *EF memory manager*. As the data starts flowing, the computation takes place with EFs being allocated and deallocated following the prescribed order. Once the local sequence is complete, collected data is either sent to the approaching vehicles, in case we reached the end of the pipeline, or sent to the next EN.

6 IMPLEMENTATION

Our platform was developed as a unikernel-based system running on top of the Xen hypervisor (stock version). We exploited virtualisation for multiple reasons to hide hardware discrepancies between off-the-shelf devices, achieve fine-grained control over running VMs, obtain stronger isolation and maintain compatibility with existing cloud computing platforms. We considered Xen [4] to be the most suitable hypervisor for our implementation as it directly supports MirageOS [9] — the unikernel of choice for our implementation. We opted for this specific technology, since one of our goals was to offload compact, ready to launch, small virtual instances, embedding only the required code; unikernels are the perfect fit for this scenario.

Three languages were used to implement the entire system: C for the EF memory module (kernel module), Python for the majority of the core modules and OCaml for the EF code (MirageOS). Existing MiniOS and MirageOS libraries were modified and extended for compatibility with our system.

7 EVALUATION

We profiled our system under different loads, devices and pipeline topologies, as shown in Figure 4. The basic case A represent black ice detection done by a pipeline involving only two lampposts, the results reflect the processing load of the edge computing nodes. Other cases grow in complexity and allow to profile our system under different configurations. Each pipeline begins and ends with a red and green block, respectively. Yellow blocks represent transit

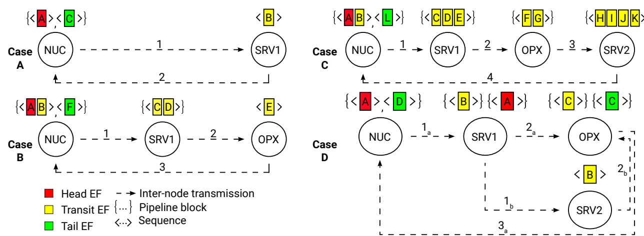

Figure 4: Pipeline topologies

EFs, which can be found in the middle of the chain. A pipeline block is delimited by curly braces and contains sequence blocks delimited by angular braces. A block is an EF processing images or other sensor data locally collected by the lamppost. Case D has two pipelines delimited by curly brace blocks. Each node executes only the blocks assigned to it, which are expected to be executed in a specific order (represented here with a character in each block).

Different devices were used to understand the performance gap between the edge and cloud. In our tests, the edge devices were comparable to micro-servers rather than base stations. The nodes used in our tests were an Intel NUC (NUC), Dell Optiplex (OPX) and two high-end Dell PowerEdge 730 servers (SRV1, SRV2), all connected to the same LAN network.

Table 1: Boot-up time for the EF unikernel on each device

NUC	OPX	SRV1	SRV2
46±15 ms	*30±13.2 ms*	*25±7.4 ms*	*29±11.5 ms*

We focussed on measuring the following parameters: intra-node data transfer overhead, pure computation time and pipeline completion time, as shown in Figure 5, 6 and 7 respectively. To this end, we baked a MirageOS unikernel supporting basic image processing operations such as colour normalization and continuously passed to it the same image with a size of approximately 250KB. Additionally, in our pipeline the nodes exchanged a complete post-processed image instead of a single value, as we expect in the case of black ice fingerprinting where a true/false is sufficient to at least communicate the presence of hazards. The average inter-node network transmission time is also specified for completeness: **20±0.039 ms**.

In Figure 6, a minimal difference in performance can be seen between cloud (server) and edge devices. Thus, we can assert that there is an effective profit margin in offloading computation as the potential loss in computational speed is countered by a reduced upload time. In the case of black ice detection, it would be necessary to upload images from each lamppost to the cloud, process them and then send results to cars on a specific road. This would eventually increase the round-trip time and become unfeasible under in poor network connectivity situations.

Figure 5 illustrates the overhead created by transferring data between the host machine and guest unikernels before the processing phase. Both read and write operations are very fast, on the order of hundreds of microseconds. Read operations, in which memory pages populated with processed data are mapped back to the host, are much slower. This results from different code used for the two

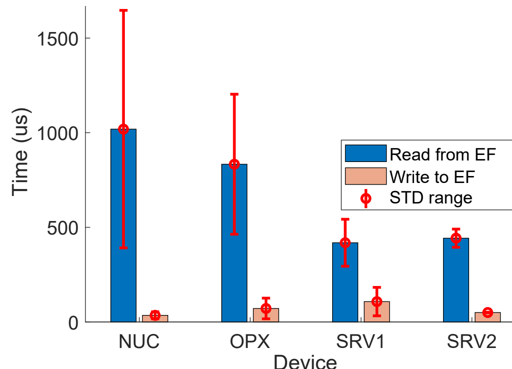

Figure 5: Intra-node transfer time (3000 data points evenly split between read/write per node)

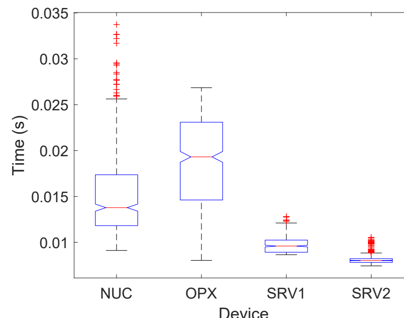

Figure 6: EF processing time (500 data points per node)

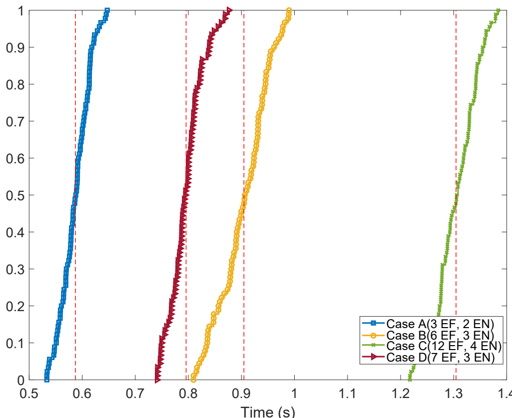

Figure 7: Pipeline execution time ECDF (100 iterations per topology)

operations; writing is more efficient as we can allocate blocks of free memory pages directly, whereas the read operation requires proceeding page by page. However, even in the average worst case, reading data is in the range of **1ms**. Thus, the memory module design has very low overhead on the overall system performance.

Figure 7 displays the ECDF of the edge-cloud pipeline execution time for the cases presented in Figure 4. Cases A to C represent single-pipeline, non-branching scenarios with an increasing

number of ENs and EFs. Case D represents the performance for a branched multi-pipeline scenario. The pipeline execution time includes the computational and memory overhead time in addition to the network inter-node transfer time and the unikernel boot-up time, as depicted in Table 1.

Our first insight into the results is the pipeline execution time grows linearly with the number of EFs. Even though they are not directly comparable, the first three topologies differ only in number of EFs and nodes, while the pipeline workflow remains roughly unchanged. We note that hardware differences do not play a major role, as performances is comparable. In each case, we double the number of EFs and add only one EN. By adding more nodes we amortise the overall completion time, thus, reducing the slope of the linear growth. This is critical in long pipelines, where finding the proper ratio of EF to EN has a large effect. Using our framework, to properly scale-up in terms of edge functions and nodes, different topologies can be tested to find the one with superior performance. In our scenario, EFs are expected to be quasi-uniformly distributed across the roadside equipment involved in the computation.

Case D is the sole case involving multiple pipelines. Rather than evaluation performance, the test serves as an example of the potential of our framework to run branching pipelines. The examples are situations where the set of information collectible by different EN is not uniform (e.g. smart lampposts equipped with different sensors). Hence, it is necessary to merge data collected from different branches of the pipeline to terminate the computation. In comparison to case B, whose computational load is the closest in terms number of EFs, we notice that there is a cost to running multiple parallel pipelines; with one additional EN, the results are only approximately 10% better than for the single pipeline case.

8 CONCLUSION AND FUTURE WORK

In this paper, we proposed a distributed edge computing framework to improve current solutions to road hazard detection. Our attention was focused on black ice detection under adverse road conditions. The logic, design and implementation of our system were described in relation to with the analysed use case scenario. We discussed the advantages of our approach in comparison to solutions based on crowdsourcing, cloud computing and on-board car sensors. Our current evaluation has not yet demonstrated the full potential of our system, as the thermal image processing necessary for detecting patches of black ice was not embedded in the tested EFs. Hence, in future work we intend to address this issue by integrating a state-of-the-art machine-learning black ice detection model into our system.

To strengthen and expand the capability of our framework, we plan on offloading non-virtualised EF. Unikernels do not actually require a virtual hardware abstraction; they can achieve similar levels of isolation when running as processes by taking advantage of existing kernel system call whitelisting mechanisms as demonstrated in [22]. This has the potential to make our system compatible with a larger range of devices and enable a much simpler integration of existing tools into our platform. An alternative option is to modify our system to be compatible with a Tier-2 hypervisor such as KVM. In fact, this would allow us to compare our system with other similar frameworks for serverless computing, such as Amazon Firecracker.

REFERENCES

[1] 2017. BMW Here HD Maps. https://www.forbes.com/sites/samabuelsamid/2017/02/21/bmw-here-and-mobileye-team-up-to-crowd-source-hd-maps-for-self-driving. [Online; accessed 08-January-2019].

[2] 2018. LoRaWAN. https://lora-alliance.org/sites/default/files/2018-04/what-is-lorawan.pdf. [Online; accessed 08-January-2019].

[3] 2018. Munich's Smart Lamp Posts Shine. https://www.smarter-together.eu/news/munichs-smart-lamp-posts-shine. [Online; accessed 07-January-2019].

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. *SIGOPS Oper. Syst. Rev.* 37, 5 (Oct. 2003), 164–177. https://doi.org/10.1145/1165389.945462

[5] Sultan Basudan, Xiaodong Lin, and Karthik Sankaranarayanan. 2017. A privacy-preserving vehicular crowdsensing-based road surface condition monitoring system using fog computing. *IEEE Internet of Things Journal* 4, 3 (2017), 772–782.

[6] Alexis Comber, Linda See, Steffen Fritz, Marijn Van der Velde, Christoph Perger, and Giles Foody. 2013. Using control data to determine the reliability of volunteered geographic information about land cover. *International Journal of Applied Earth Observation and Geoinformation* 23 (2013), 37–48.

[7] Vittorio Cozzolino, Aaron Yi Ding, and Jörg Ott. 2017. Fades: Fine-grained edge offloading with unikernels. In *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems*. ACM, 36–41.

[8] Jakob Eriksson, Lewis Girod, Bret Hull, Ryan Newton, Samuel Madden, and Hari Balakrishnan. 2008. The pothole patrol: using a mobile sensor network for road surface monitoring. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*. ACM, 29–39.

[9] Madhavapeddy et al. 2013. Unikernels: Library operating systems for the cloud. In *ACM SIGPLAN Notices*, Vol. 48. ACM.

[10] Xueshi Hou, Yong Li, Min Chen, Di Wu, Depeng Jin, and Sheng Chen. 2016. Vehicular fog computing: A viewpoint of vehicles as the infrastructures. *IEEE Transactions on Vehicular Technology* 65, 6 (2016), 3860–3873.

[11] Yoichiro Iwasaki, Masato Misumi, and Toshiyuki Nakamiya. 2013. Robust vehicle detection under various environmental conditions using an infrared thermal camera and its application to road traffic flow monitoring. *Sensors* 13, 6 (2013), 7756–7773.

[12] Maria Jokela, Matti Kutila, and Long Le. 2009. Road condition monitoring system based on a stereo camera. In *Intelligent Computer Communication and Processing, 2009. ICCP 2009. IEEE 5th International Conference on*. IEEE, 423–428.

[13] M Kutila, M Jokela, J Burgoa, A Barsi, T Lovas, and S Zangherati. 2008. Optical roadstate monitoring for infrastructure-side co-operative traffic safety systems. In *Intelligent Vehicles Symposium, 2008 IEEE*. IEEE, 620–625.

[14] Yuyi Mao, Changsheng You, Jun Zhang, Kaibin Huang, and Khaled B Letaief. 2017. A survey on mobile edge computing: The communication perspective. *IEEE Communications Surveys & Tutorials* 19, 4 (2017), 2322–2358.

[15] Artis Mednis, Atis Elsts, and Leo Selavo. 2012. Embedded solution for road condition monitoring using vehicular sensor networks. In *the 6th IEEE International Conference on Application of Information and Communication Technologies (AICT'12)*. 1–5.

[16] Prashanth Mohan, Venkata N Padmanabhan, and Ramachandran Ramjee. 2008. Nericell: rich monitoring of road and traffic conditions using mobile smartphones. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*. ACM, 323–336.

[17] R. Morabito, V. Cozzolino, A. Y. Ding, N. Beijar, and J. Ott. 2018. Consolidate IoT Edge Computing with Lightweight Virtualization. *IEEE Network* 32, 1 (Jan 2018), 102–111. https://doi.org/10.1109/MNET.2018.1700175

[18] Omran Saleh and Kai-Uwe Sattler. 2013. Distributed complex event processing in sensor networks. In *Mobile Data Management (MDM), 2013 IEEE 14th International Conference on*, Vol. 2. IEEE, 23–26.

[19] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. 2009. Distributed Complex Event Processing with Query Rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems (DEBS '09)*. ACM, New York, NY, USA, Article 4, 12 pages. https://doi.org/10.1145/1619258.1619264

[20] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge computing: Vision and challenges. *IEEE Internet of Things Journal* 3, 5 (2016), 637–646.

[21] Girts Strazdins, Artis Mednis, Georgijs Kanonirs, Reinholds Zviedris, and Leo Selavo. 2011. Towards vehicular sensor networks with android smartphones for road surface monitoring. In *2nd International Workshop on Networks of Cooperating Objects (CONET'11), Electronic Proceedings of CPS Week*, Vol. 11. 2015.

[22] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. 2018. Unikernels As Processes. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '18)*. ACM, New York, NY, USA, 199–211. https://doi.org/10.1145/3267809.3267845