ECCO: Edge-Cloud Chaining and Orchestration Framework for Road Context Assessment

Vittorio Cozzolino, Jörg Ott
Technical University of Munich
{vittorio.cozzolino, ott}@in.tum.de

Aaron Yi Ding
TU Delft
aaron.ding@tudelft.nl

Richard Mortier
University of Cambridge
richard.mortier@cl.cam.ac.uk

***Abstract*—For road safety, detecting and reacting efficiently to road hazards is crucial and yet challenging due to practical restrictions such as limited data availability, which relies on network support. Moreover, from a system perspective we lack a computational model capable of providing to vehicles reliable and real-time assessment of the road context. As autonomous vehicles become widespread, the safety issues are further aggravated by the gap between cloud, roadside infrastructure and road users in terms of communication latency, software-hardware compatibility and data interoperability. To tackle this, we present ECCO: an orchestration framework that enables edge-cloud collaborative computing for road context assessment. ECCO can create on-demand task execution *pipelines* spanning multiple, potentially resource-constrained edge-nodes with the smart IoT infrastructure support. Our prototype lays the groundwork to support new services, which can use more efficiently the road infrastructure and deliver safety-critical applications for road users.**

***Index Terms*—Edge computing, Distributed computing, Unikernel**

I. Introduction

The development of ubiquitous road-side infrastructure through deployment of stationary and mobile roadside units (RSU)[1] [2] and street furniture such as lampposts [3] seeks out ways to ease congestion and improve road safety. For example, detailed metropolitan maps coupled with citywide pollution fingerprinting can improve citizen health, helping pedestrians and cyclists select less polluted routes. In spite of its great value, smart infrastructure development is still in its infancy with and it's tied to ad-hoc, vertically integrated solutions rather than open platforms offering shared data and compute resources. In fact, an open platform supporting multi-tenant access to a citywide compute edge-network infrastructure would facilitate development and deployment of a broad range of applications at a reduced cost.

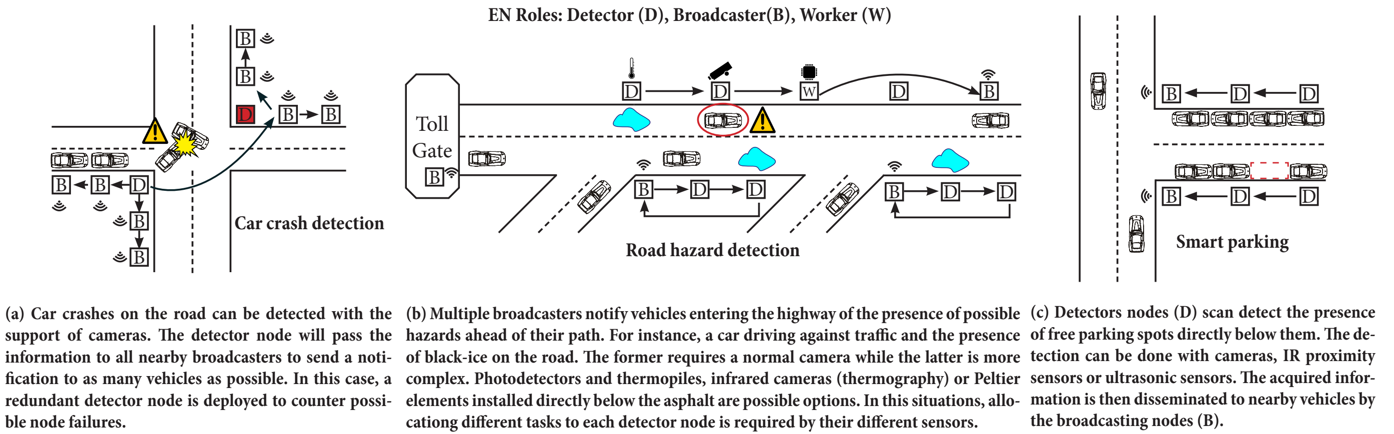
To support such applications, we propose a roadside infrastructure comparable to [4], which encompasses smart vehicles and devices, RSUs as intermediate computational units, and cloud servers. The challenge then becomes how to enable developers to write and efficiently deploy applications on such a heterogeneous infrastructure. As computing shifts to the edge and particularly the roadside infrastructure, one of the fundamental changes is that it will not be tied to a single vendor, regardless of how comprehensive their offerings may be [5]. Hence, solving the problem of balancing and controlling applications deployed by multiple providers is of crucial importance. Moreover, it is not about only edge or cloud — the key for innovation lies in their interplay. In our work, we build on top of these requirements a platform designed to deploy applications instantiated as *edge-cloud pipelines*. Therefore, we design and implement an orchestration framework enabling road context assessment by providing precise information about the road condition. Expanding on our previous work on computation offloading with unikernels [6], we propose a distributed, edge-cloud computational model to deploy multi-node execution pipelines on-demand. Comparing with existing frameworks such as KubeEdge [7] with generic computational model and cloud-only control plane, with ECCO we propose an edge-cloud chaining model dedicated to dynamic IoT scenarios (i.e., road context assessment) and with *responsibility repartition* between cloud and edge.

[1]A Roadside Unit is a V2X direct link transceiver that is mounted along a road or pedestrian passageway [1].

II. Model of Computation

Deployment of the roadside infrastructure poses the non-trivial challenge of assessing the **road context** as the ensemble of *precise* and *trustworthy* road events information, at *scale*. This problem assumes even greater relevance in combination with fully autonomous vehicles, which rely on content delivery through mobile or edge communication to precisely understand the real-time driving environments [8]. With the support of edge computing, we can build an infrastructure able to deliver fresh information to nearby vehicles, enhancing their context awareness. Such approach can enable new services or enhance existing ones such as incident warning broadcasting, traffic signal violation warning, pre-crash sensing, cooperative forward collision warning, lane change warning, black-ice detection.

We can identify static and dynamic entities at work in the roadside scenario which need to communicate and exchange information. Based on these, we devise a model of computation pivoting on three elements: the inputs received by the roadside infrastructure, the functions (*edge functions*, *EF*) processing them, and the outputs enabling different services. To provide a thorough description, we select three use-cases: *(a)* car crash detection, *(b)* road hazards detection, and *(c)* smart parking as shown in Figure 1. The latter illustrates an example of a linear pipeline where: (*i*) the inputs are the



(a) Car crashes on the road can be detected with the support of cameras. The detector node will pass the information to all nearby broadcasters to send a notification to as many vehicles as possible. In this case, a redundant detector node is deployed to counter possible node failures.

(b) Multiple broadcasters notify vehicles entering the highway of the presence of possible hazards ahead of their path. For instance, a car driving against traffic and the presence of black-ice on the road. The former requires a normal camera while the latter is more complex. Photodetectors and thermopiles, infrared cameras (thermography) or Peltier elements installed directly below the asphalt are possible options. In this situations, allocationg different tasks to each detector node is required by their different sensors.

(c) Detectors nodes (D) scan detect the presence of free parking spots directly below them. The detection can be done with cameras, IR proximity sensors or ultrasonic sensors. The acquired information is then disseminated to nearby vehicles by the broadcasting nodes (B).

Fig. 1: Smart roadside infrastructure use-cases. (a) Car crash detection, (b) road hazards detection, (c) smart parking.

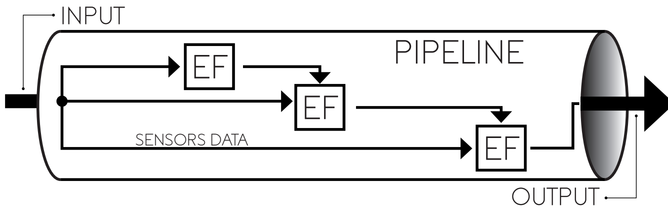


Fig. 2: Visual representation of an ECCO pipeline.

sensors readings detecting free parking spots, (*ii*) the roadside equipment is the compute infrastructure at the edge of network on which the EFs are deployed and executed, and (*iii*) the output is a list of available parking spots which is sent to nearby vehicles. Figure 2 illustrates how these elements are connected to form an ECCO **pipeline**.

We next describe the pipeline components (network, functions, nodes) in more detail, as well as their deployment and execution strategy.

A. Pipeline Components

In our scenario, the vehicles are the recipients of the pipelines output. They receive information from the infrastructure via long-range communication radios such as LoRaWAN [9] or LTE-V2X for Vehicle Fog Computing [10], [11]. As we focus on the computational model and system design, we do not delve deeper into the specifics of V2I transmission mechanisms, a topic explored in other research efforts [12].

Edge Nodes. Following the definition of Shi et al. [13] that edge computing occurs in proximity to datasources, we define an Edge Node (EN) to be a device close to the end-user, such as a mobile phone, PC, or wireless access point. In other contexts, the definition could be extended to include Radio Access Network (RAN) micro-servers [14]. In our case, we focus on the already mentioned RSUs, which are deployed on the road to monitor it and collect data. As they are stationary, we assume good connectivity to the cloud and to other ENs forming what we call an *Edge Network*.

Edge Functions. An Edge Function (EF) is a self-contained, atomic function which embeds a small piece of the application logic that can be executed standalone. When chained together, EFs form an execution pipeline[2]. Each instance of EF plays a specific role and is hosted on a EN. They need to be placed strategically based on the available datasources, the current load status and the geographic position.

Edge-Cloud Pipelines. An ECCO pipeline is a distributed task involving a set of ENs. ENs, listed in the pipeline, take part in execution chains and collaborate to run it. In the next section, more details are provided regarding how such pipelines are deployed.

B. Pipeline Deployment

We envision two levels of control in the pipeline deployment and management process: (*i*) the cloud, which defines the high-level, application driven pipeline deployment plan and (*ii*) the edge which locally makes scheduling decisions based on the parameters described in the rest of this section. The detail of a pipeline structure is defined by the cloud provider, which also monitors its execution.

We assume ENs are reachable from the cloud and can report their available data and current load in terms of active EFs and pipelines. On this basis, the service can plan a pipeline based on a set of parameters to exploit data locality. Once offloaded, the pipeline can be configured to run independently from the cloud, based on specific policies. The need for a constant connection with the cloud stems from the specific scenario. Safety is a major concern in our use-cases as human lives are involved and the constant presence of the cloud as an overseer is deemed necessary to properly manage resources and system failures. For example, dissemination of wrong information or neglecting a car accident may put lives at risk.

As ENs have limited resources and are shared by multiple services, we use the *priority* and *execution* fields in the

[2]The composition of sources (inputs), edge nodes, and sinks (outputs) is similar to a directed acyclic graph (DAG). However, from a user perspective it can be abstracted to a linear flow so that we use the term pipeline to emphasize this relationship.

pipeline configuration to decide when to execute a pipeline. The *priority* field assumes different values based on the use-case and it is static, meaning that a specific use-case will always have the same priority. It is defined by the cloud provider orchestrating the service. For instance, car crash detection will always have higher priority than smart parking. This information allows the system to dynamically shut-down low priority services when additional resources are required by the high priority ones.

Another parameter is *execution*, which can assume only two values: on-demand and automatic. On-demand pipelines are only deployed when requested explicitly by the service provider. Black-ice detection is deployed on-demand as it only manifests in specific conditions (e.g., low temperature at night). Likewise, smart parking is not required in the early hours of the day or when there is very low traffic density detected. Conversely, car crash detection will be flagged as *automatic* as it is a safety critical application running with the highest priority.

Pipelines are flexible and adapt to the use-case and ENs at our disposal. The execution flow can be represented as a directed acyclic graph (DAG) or directed cyclic graphs with topological ordering [15]. We focus on the system aspects as theoretical challenges in service composition techniques have been explored in other studies [16]. For instance, in Figure 1a the pipelines branch to disseminate the alert regarding a car crash as quickly as possible to as many repeater nodes in close proximity. The same behaviour is expected in case of node malfunction, where branching might be necessary to bypass an unresponsive EN. When an EN is not reachable, a substitute is found to replace it or the pipeline is adjusted to skip the node and remove it from the execution tree. For Figure 1a, this means that we will not be able to reach some vehicles directly from our broadcasting ENs if the failure affects a broadcaster node. Conversely, if a detector node goes down, there will be another node ready to replace it and able to detect the car crash. Intersections require redundant ENs deployment as they are often involved in accidents: in 2007, approximately 2.4 million intersection-related crashes occurred, representing 40% of all reported crashes and 21.5% of traffic fatalities [17].

Another reason for branching is that each EN has different resource. One EN might only have cameras, another one only a proximity sensor and a broadcasting interface. This information is collected by the cloud and used to opportunely plan the pipelines structure. ENs without a broadcasting interface can only have a detector role which in turn is defined by its sensors' capabilities. By analogy, there can be ENs playing both the detector and broadcaster role. In the smart parking use-case, the data flow generated by the detectors is progressively enriched along the pipeline. In this case, a small delta of processing is carried out by each detector leaving the broadcasting node only with a task of actually sending the results as shown in Figure 3a. Finally, broadcasting nodes might not support all the radio access technologies required for vehicular communication which is a problem currently discussed by the research community [18].

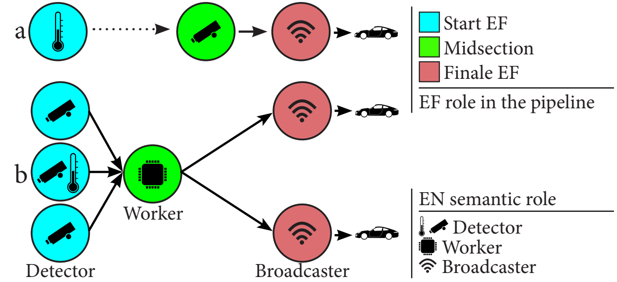


Fig. 3: Pipelines' execution graphs based on ENs capabilities and EFs roles.

In other cases, we might need an additional worker node to perform a computationally intensive task. For example, integration of multiple sensors feeds to detect road hazards as shown in Figure 1b. Another example is an intersection where all the data generated is sent to the worker node, processed, and sent back to manage more efficiently the traffic lights based on the current traffic conditions. The processing node might be a micro-server in close proximity which sends the refined information to selected broadcasting nodes (Figure 3b). The need for multiple broadcasting nodes is twofold: greater communication range and available network interfaces. In fact, radio access technologies required by vehicular communication are changing rapidly and it is expected that not all will be supported by a single RSU [19].

C. Pipeline Execution

The cloud provider generates the pipeline configuration which contains details about the execution plan. When the configuration is offloaded, the ENs involved parse it and each identifies sections it can execute in relation to other nodes. Each pipeline is thus split into sub-pipelines, and transformed into multiple stages which eventually become executable. Execution order of EFs within an EN can be based on various parameters, e.g., priority, expected load, and deadline.

ENs scan the received pipeline configuration and identify the group of EFs it should execute. The classification determines the order to execute and chain EFs, plus the respective roles. An EF has one of three roles: (*i*) *Start*, starting a sequence; followed by (*ii*) zero or more *Midsections*; culminating in (*iii*) a *Finale* which closes the sequence. Sequence ordering parameters are used to correctly unfold execution onto the ENs. The nomenclature adopted in Figure 1 (*detectors*, *broadcasters* and *workers*) applies to the EN while the one just introduced only to the EFs and it is used internally by the system to properly order the pipeline graph. What matters for the pipeline processor is the *relative execution order* of the EFs and not their actual task in relation to the EN capabilities. The relationship between these two concepts is shown in Figure 3 with two simple topologies.

ECCO creates temporary, dynamic execution chains based on the pipeline topology to form ad-hoc collaborative networks of ENs. As data flows from one EF to the next, computation unfolds and progresses toward the Finale. As already

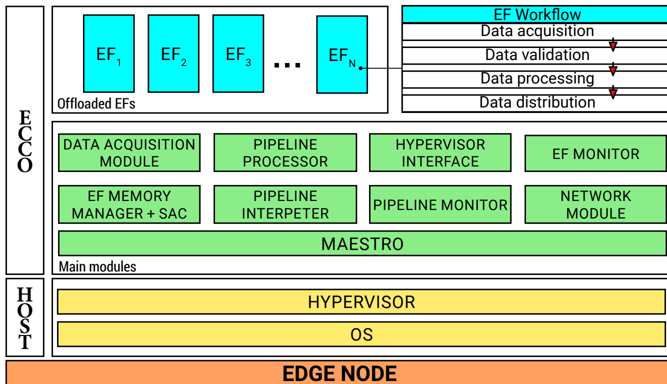


Fig. 4: Overview of ECCO modules.

discussed, pipelines need not be linear but can branch and join to create execution DAGs.

III. ECCO: Design

In this section, we provide an overview of the system depicted in Figure 4 and its components, relate them to our use case, and describe the system workflow. ECCO was designed to achieve two goals: (*i*) provide a landing platform to offload lightweight and fine-grained services orchestrated by the cloud and running on constrained devices at the edge; and (*ii*) support seamless cooperation and interconnection of ENs to support pipelines offloaded from the cloud.

If roadside infrastructure was only usable as an extension of the cloud, reliant on the cloud to work, then a slow or intermittent network connection could render the whole infrastructure useless. Information about road conditions would be retrieved slowly or not at all, and vehicles would be left without information about imminent hazards, potentially costing lives. Treating roadside infrastructure as an edge computing infrastructure, able to use but not reliant upon the cloud, offers a reliable, resilient, and independent infrastructure delivering services even when the cloud is unreachable from end-users.

Crowdsourcing cannot provide this because vehicle density on less heavily used roads will often be insufficient to reliably map road conditions. Available spatial data are sparse and inadequate, leading to incomplete or misleading information distributed to vehicles driving in low-traffic areas. Effectiveness of onboard car sensors is also reduced in common situations as adverse weather conditions which reduce visibility.

ECCO addresses these challenges by providing a platform where multiple cloud services can share existing edge infrastructure for scheduling and handling multiple offloaded pipelines. It offers computational power at the ENs, enabling both independence from the cloud in case of intermittent connectivity, and dynamic processing of information based on chaining EFs.

A. Components

Our system relies on edge offloading: a paradigm that moves computation from the cloud to edge nodes [20]. To differentiate from similar solutions, we design our system as a collaborative framework where multiple ENs are chained to execute different pipelines. To orchestrate the offloaded EFs at the edge, we developed a set of modules running on each EN. The components listed below are associated with the blocks in Figure 4.

Maestro. This is the core of and entry point to our system, functioning as both a coordinator and an interface with the outside world. When one or more EFs are offloaded as part of a pipeline, *maestro* handles the calling of the required modules to filter, order and execute the EFs. During pipeline execution, each EF is tracked and monitored to assess its state in conjunction with the pipeline's. Since multiple parties can access the same ENs, the execution of parallel pipeline is also supported as the allocated resources are completely independent. For resiliency purposes, checkpoints of the pipeline status together with EFs intermediate results are stored in a local database. This modules takes care of bootstrapping ECCO by notifying the presence of an EN to the cloud by advertising its capabilities in terms of hardware resources (e.g., RAM, CPU), sensors, cameras, and communication interfaces. These parameters allow a correct placement of the EFs to minimize distance from the datasource without overloading the EN. In fact, EFs are mapped to ENs based on the required data and type of processing.

EF workflow. Each EF is composed of four phases: data acquisition, validation, processing, and distribution as shown in Figure 4.

In the *data acquisition* phase, an EF awaits the necessary data from the *maestro* which identifies the correct datasource and retrieves the data on the EF behalf. In fact, *maestro* exposes to EFs different end-point to access sensors or local databases identified during the bootstrapping phase. Moreover, the specific steps of the data retrieval phase change depending on the type of end-point. For instance, in the case of hardware sensors, the code to pilot them is embedded directly into the EFs, while for external sources (e.g., databases) *maestro* would use libraries from the host to read the data and then pass it to the EF. Contextualizing, in the example use-case of black ice detection such data are images produced by an infrared camera or readings from a Peltier element. The *data validation* phase checks the received data for errors, eventually requesting a re-transmission. The *data processing* phase is the core of the EF as it contains the developer code. By customizing this part of the EF, it is possible to execute arbitrary code in the EF, granted that eventual external dependencies and libraries have been opportunely handled. In relation to our use-cases, it can contain algorithms to manipulate and process images from cameras or do sensors fusion. Finally, the *data distribution* phase determines whether the outputted result should be passed to the host module which takes care of sending it to nearby road users or sent to the next EF in the local sequence. For instance, a *midsection* EF (detector) in the pipeline can output a post-processed image to be sent along the pipeline for further analysis while a *finale* EF (broadcaster) will signal to nearby vehicles the presence of potential hazards.

Data communication primitives. Dependent on pipeline

structure, data can be exchanged in two ways: (*i*) *Intra-node* communication occurs when the transfer involves two co-located EFs or an EF and the *maestro*; and (*ii*) *Inter-node* communication occurs when the transfer takes place between two EFs on different ENs. To do so, we adopted a shared memory approach to transfer data between EFs using a custom module called *EF memory manager* (EF-MM).

Network module. It enables communication between distinct ENs. It has two groups of queues that contain data structures called *bundles*, a set of parameters to unequivocally identify a pair of producer and consumer ENs. A combination of IDs extracted from the pipeline configuration serves this purpose. The bundles in the inbound queue are stored until consumed by one or more local EFs, which remain on standby until 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 used as a fail-safe measure in case of a misstep in a pipeline stage whereas data bundles are stored until the malfunctioning nodes are ready to proceed. For added resiliency, the bundles are preserved inside a database to allow hot restart of the system in case of local failure.

Other components. Of the remaining components, the *data acquisition module* is a library wrapper, loaded on-demand based on the requirement specified in the pipeline configuration to interact with different datasources. The *hypervisor interface* exposes an API to control and monitor the VMs running on Xen. The *pipeline processor* contain the core algorithm to unfold the pipeline into ordered sequences. It filters the EFs, identify their roles, bundled them in order sequences (longest sequence). Finally, there is a *pipeline monitor* for each running pipeline as their execution is independent. It spawns multiple *EF monitors* to track each running EF. It uses the *hypervisor interface* to track the status of each running instance and report back in case of failure.

IV. Implementation and Evaluation

ECCO was developed as an orchestration platform for unikernels: specialised, single address space machine image constructed by using library operating systems [21]. Specifically, our unikernel of choice was MirageOS [22] which we ran on top of the Xen hypervisor [23]. We used virtualization to abstract over hardware discrepancies between ENs, and to obtain fine-grained control over running VMs, stronger isolation, and compatibility with existing cloud computing platforms. Our implementation uses C for the EF memory module (a kernel module), Python for the core modules, and OCaml for the EF code (mandated by use of MirageOS).

ECCO computational model is **platform-independent**, meaning that it can be potentially implemented using other sets of technologies as Docker containers on top of Kubernetes. We decided to develop our system based on unikernels due to the multiple advantages in terms of isolation, memory footprint and fine-grained function encapsulation. These are crucial properties in a multi-tenant, resource-constrained scenario.

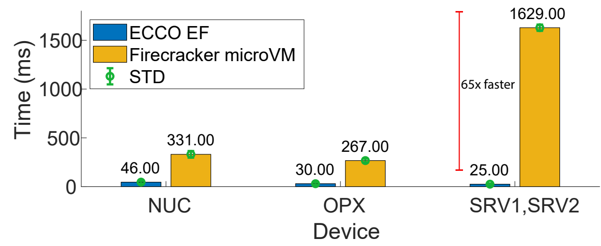


Fig. 5: Average ECCO EF vs. Firecracker microVM boot-up time and standard deviation (STD).

Xen was the most suitable hypervisor for our implementation as it directly supports MirageOS, our chosen unikernel framework due to it producing compact, bootable images that embed only the required OS functionality. Other unikernel technologies were available but MirageOS is one of the most mature and has already been applied to similar IoT use-cases [24]. However, our framework is in principle compatible with any unikernel framework that supports Xen adding flexibility in terms of usable programming languages (e.g., C++, Java, Haskell).

A. Evaluation

Our preliminary evaluation of ECCO focuses on: (*i*) the overhead introduced by the technology choices made in ECCO (Xen, MirageOS) and (*ii*) what is the impact of these overheads on a specific application, driven by our use-cases. The default EF used for in our experiments is a MirageOS unikernel supporting basic image processing operations fed with an image size of approximately 280 kB. This EF is used for all our subsequent benchmarks.

Device	CPU	RAM	Ocaml	Xen	OS
Dell PowerEdge R530 (SRV1, SRV2)	Intel Xeon E5-2640 2.60GHz — 32 Cores	128 GB	4.04.2	4.6.0	Ubuntu 14.04 Kernel 3.19.0
Intel NUC (NUC)	Intel i5-6260U 1.80GHz — 4 Cores	16 GB	4.04.2	4.6.6	Ubuntu 14.04 Kernel 4.4.0
Dell Optiplex 7050 (OPX)	Intel i5-7500T 2.70GHz — 4 Cores	8 GB	4.04.2	4.6.6	Ubuntu 14.04 Kernel 4.4.0

TABLE I: Devices specifications.

Different devices were used to understand the performance gap between the edge and cloud (all connected to the same LAN network). As revealed in Table I, SRV1 and SRV2 have identical configuration, hence their results are bundle together in all the plots due to negligible differences. We performed each experiment 100 times, except when clearly stated.

To evaluate baseline overheads we compare against Amazon Firecracker [25], a recently introduced lightweight serverless computing framework that delivers end-to-end orchestration for tiny VMs. To do so, we built a custom microVM based on an Alpine Linux v3.9 kernel, loaded it with OpenCV v3.4.6 and allocated it was 128 MB RAM and 1 vCPU. The size of its *rootfs* was roughly 4.5 GB.

Figure 5 shows boot times for unikernels compared to the Firecracker microVM. On all devices where we can compare to the Amazon Firecracker microVM, the unikernel

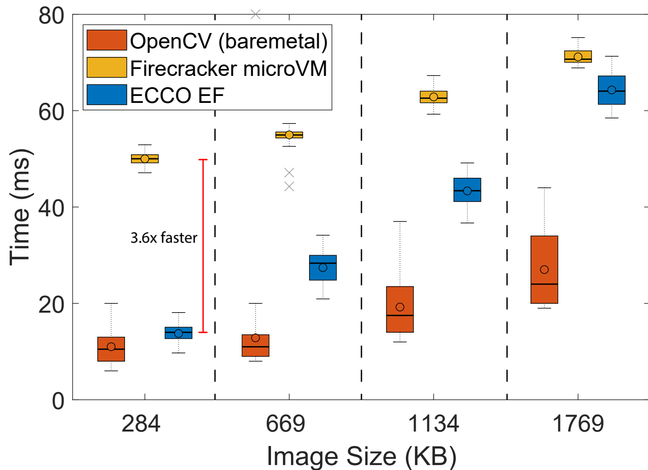


Fig. 6: ECCO Benchmarks (SRV1, color normalization).

boot time was substantially lower, below 50 ms. There is also a considerable difference in size between the images, which probably accounts for much of the difference in boot times. The ECCO EF unikernel is around 5 MB EF while the microVM is around 22 MB plus another 4500 MB of attached *rootfs*. Similarly, the RAM required for an EF is around 15 MB (x86) or 12 MB (ARM) while the microVM required at least 33 MB. We believe that this shows the ECCO approach is well suited to low-latency applications running on ENs with limited memory, as well as for situations where EFs may be updated and distributed frequently.

We compare EF performance against two baselines in Figure 6: (*i*) we developed multiple C++ applications with OpenCV v3.4.6 replicating the operations executed inside the EF; and (*ii*) we loaded the applications in the custom Firecracker microVM previously described. In this way we compared our system to both solutions. For this purpose, we developed a simple application for colour normalization. For space reasons, we present only the result for SRV1, but a similar behavior was shown for the other devices. While ECCO cannot outperform OpenCV running on bare-metal, it has a substantial advantage compared to Firecracker. Pairing this result with the substantial difference in boot time, ECCO outperforms the alternatives and is competitive with bare metal solutions for small image sizes. ECCO performance assumes even greater importance when executing distributed computation spanning multiple ENs, where both quick instantiation and execution time are crucial.

We identify a different execution time growth factors between ECCO and Firecracker, steeper for the former. This shows that our solution is suitable for processing a small amount of information, while the serverless Amazon approach shines with higher data loads.

V. Related Work

Our work draws on multiple strands of existing research which we split into two major branches: detection of road hazards and events, and distributed edge computing systems.

Detecting road conditions and possible hazards is a problem that has been solved in multiple ways: through crowdsourcing, where vehicles exchange collected data to spot bumps [4], or through new infrastructure, where infrared cameras on lampposts are used to identify ice formations on the road [26]. Various studies have examined the efficacy of different methods for detecting road conditions [27], [28].

Current solutions focus on using either crowdsourcing or edge networks for transferring road conditions information. However, the quality of crowdsourced spatial data is often unreliable [29], resulting in insufficient density of data to estimate road conditions in low-traffic areas. Solutions based on on-board car sensors can prove to be mediocre depending on the road characteristics and weather conditions. Edge computing can play a pivotal role in addressing these challenges by exploiting road infrastructure to augment vehicle sensory capacity beyond their on-board sensors. Using edge computing to support offload of computation to deliver particular applications is not new [13]. With ECCO we are concerned with providing a distributed framework to dynamically interconnect nodes based on the applications requirements.

Numerous authors have explored offloading computation and data, for different purposes and under different decision policies [30]–[34]. Cloudlets [35] were a particular pioneer in the field of computation offloading. Earlier work from Madhavapeddy et al. [36] proposed on-demand specialized VM instantiation within connection setup time. Airbox [37] presents a software platform based on onloading and backend-driven cyberforaging. It shares the general direction presented in our paper in terms of offloading the EF. Compared with Airbox, ECCO achieves fine-grained offloading by using unikernels instead of Docker technology. Databox [38] proposes a hybrid physical and cloud-hosted system for personal data management. Koller et al., [39] also proposed an unikernel-based serverless framework architecture while a more recent research effort proposes a WebAssembly solution [40].

VI. Conclusions

ECCO is a distributed edge computing framework developed to deliver road context assessment. We discuss the advantages of our approach in comparison to crowdsourcing, cloud computing, and onboard car sensors solutions. Given the fast adoption of autonomous vehicles, our work propose a computational model to bridge the gap between cloud, road infrastructure and road users to deliver rapidly instantiated, on-demand services. The logic, design and implementation of our system were described in relation to the analyzed scenario and encompass two crucial problems of edge computing: fine-grained orchestration and collaborative, multi-device task execution at the edge. At the core of our framework, the function chaining allows different nodes to cooperate in the execution of ECCO pipelines.

References

[1] "US government publishing office (2017) electronic code of federal regulations, 47 cfr part 90." http://www.ecfr.gov/cgi-

bin/textidx?tpl=/ecfrbrowse/Title47/47cfr90-main-02.tpl, 2017, [Online; accessed 07-October-2019].
[2] "V2X OBU deployed in new york's cv pilot," https://www.traffictechnologytoday.com/news/autonomous-vehicles/v2x-obu-deployed-in-new-yorks-cv-pilot.html, 2019, [Online; accessed 07-October-2019].
[3] "Smarter togheter - Munich's smart lamp posts shine," https://www.smarter-together.eu/news/munichs-smart-lamp-posts-shine, 2018, [Online; accessed 07-January-2019].
[4] S. Basudan, X. Lin, and K. Sankaranarayanan, "A privacy-preserving vehicular crowdsensing-based road surface condition monitoring system using fog computing," *IEEE Internet of Things Journal*, vol. 4, no. 3, pp. 772–782, 2017.
[5] "Satya Nadella looks to the future with edge computing," https://techcrunch.com/2019/10/08/satya-nadella-looks-to-the-future-with-edge-computing/, 2019, [Online; accessed 09-October-2019].
[6] V. Cozzolino, A. Y. Ding, and J. Ott, "Fades: Fine-grained edge offloading with unikernels," in *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems*. ACM, 2017, pp. 36–41.
[7] "Kubeedge," https://kubernetes.io/blog/2019/03/19/kubeedge-k8s-based-edge-intro/, 2019, [Online; accessed 21-October-2019].
[8] Q. Yuan, H. Zhou, J. Li, Z. Liu, F. Yang, and X. S. Shen, "Toward efficient content delivery for automated driving services: An edge computing solution," *IEEE Network*, vol. 32, no. 1, pp. 80–86, 2018.
[9] A. J. Wixted, P. Kinnaird, H. Larijani, A. Tait, A. Ahmadinia, and N. Strachan, "Evaluation of LoRa and LoRaWAN for wireless sensor networks," in *2016 IEEE SENSORS*. IEEE, 2016, pp. 1–3.
[10] X. Hou, Y. Li, M. Chen, D. Wu, D. Jin, and S. Chen, "Vehicular fog computing: A viewpoint of vehicles as the infrastructures," *IEEE Transactions on Vehicular Technology*, vol. 65, no. 6, pp. 3860–3873, 2016.
[11] Y. J. Li, "An overview of the DSRC/WAVE technology," in *International Conference on Heterogeneous Networking for Quality, Reliability, Security and Robustness*. Springer, 2010, pp. 544–558.
[12] K. Abboud, H. A. Omar, and W. Zhuang, "Interworking of DSRC and cellular network technologies for V2X communications: A survey," *IEEE transactions on vehicular technology*, vol. 65, no. 12, pp. 9457–9470, 2016.
[13] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.
[14] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 4, pp. 2322–2358, 2017.
[15] Wikipedia contributors, "Topological sorting — Wikipedia, the free encyclopedia," 2020, [Online; accessed 31-January-2020]. [Online]. Available: "https://en.wikipedia.org/wiki/Topological_sorting"
[16] U. Sadiq, M. Kumar, A. Passarella, and M. Conti, "Service composition in opportunistic networks: A load and mobility aware solution," *IEEE Transactions on Computers*, vol. 64, no. 8, pp. 2308–2322, 2014.
[17] "The national intersection safety problem," https://bit.ly/2pFzaUe, 2009, [Online; accessed 21-October-2019].
[18] L. GomesBaltar, M. Mucck, and D. Sabella, "Heterogeneous vehicular communications-multi-standard solutions to enable interoperability," in *2018 IEEE Conference on Standards for Communications and Networking (CSCN)*. IEEE, 2018, pp. 1–6.
[19] G. Naik, B. Choudhury, and J.-M. Park, "Ieee 802.11 bd & 5g nr V2X: Evolution of radio access technologies for V2X communications," *IEEE Access*, 2019.
[20] R. Morabito, V. Cozzolino, A. Y. Ding, N. Beijar, and J. Ott, "Consolidate iot edge computing with lightweight virtualization," *IEEE Network*, vol. 32, no. 1, pp. 102–111, Jan 2018.
[21] A. Madhavapeddy and D. J. Scott, "Unikernels: Rise of the virtual library operating system," *Queue*, vol. 11, no. 11, pp. 30:30–30:44, Dec. 2013. [Online]. Available: http://doi.acm.org/10.1145/2557963.2566628
[22] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," *Acm Sigplan Notices*, vol. 48, no. 4, pp. 461–472, 2013.
[23] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 164–177, Oct. 2003. [Online]. Available: http://doi.acm.org/10.1145/1165389.945462
[24] J. Zhao, T. Tiplea, R. Mortier, J. Crowcroft, and L. Wang, "Data analytics service composition and deployment on iot devices," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '18. New York, NY, USA: ACM, 2018, pp. 502–504. [Online]. Available: http://doi.acm.org/10.1145/3210240.3223570
[25] Amazon, "Firecracker – Lightweight Virtualization for Serverless Computing," 2020, [Online; accessed 31-January-2020]. [Online]. Available: "https://aws.amazon.com/blogs/aws/firecracker-lightweight-virtualization-for-serverless-computing/"
[26] M. Kutila, M. Jokela, J. Burgoa, A. Barsi, T. Lovas, and S. Zangherati, "Optical roadstate monitoring for infrastructure-side co-operative traffic safety systems," in *Intelligent Vehicles Symposium, 2008 IEEE*. IEEE, 2008, pp. 620–625.
[27] M. Jokela, M. Kutila, and L. Le, "Road condition monitoring system based on a stereo camera," in *Intelligent Computer Communication and Processing, 2009. ICCP 2009. IEEE 5th International Conference on*. IEEE, 2009, pp. 423–428.
[28] Y. Iwasaki, M. Misumi, and T. Nakamiya, "Robust vehicle detection under various environmental conditions using an infrared thermal camera and its application to road traffic flow monitoring," *Sensors*, vol. 13, no. 6, pp. 7756–7773, 2013.
[29] A. Comber, L. See, S. Fritz, M. Van der Velde, C. Perger, and G. Foody, "Using control data to determine the reliability of volunteered geographic information about land cover," *International Journal of Applied Earth Observation and Geoinformation*, vol. 23, pp. 37–48, 2013.
[30] A. Y. Ding, B. Han, Y. Xiao, P. Hui, A. Srinivasan, M. Kojo, and S. Tarkoma, "Enabling energy-aware collaborative mobile data offloading for smartphones," in *2013 IEEE International Conference on Sensing, Communications and Networking (SECON)*. IEEE, 2013, pp. 487–495.
[31] E. Hyytiä, T. Spyropoulos, and J. Ott, "Offload (only) the right jobs: Robust offloading using the markov decision processes," in *Proceedings of IEEE WoWMoM '15*, 2015.
[32] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *2012 Proceedings IEEE Infocom*. IEEE, 2012, pp. 945–953.
[33] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *Proceedings of the 8th international conference on Mobile systems, applications, and services*. ACM, 2010, pp. 49–62.
[34] D. F. Willis, A. Dasgupta, and S. Banerjee, "Paradrop: A multi-tenant platform for dynamically installed third party services on home gateways," in *Proceedings of the 2014 ACM SIGCOMM Workshop on Distributed Cloud Computing*, ser. DCC '14. New York, NY, USA: ACM, 2014, pp. 43–44. [Online]. Available: http://doi.acm.org/10.1145/2627566.2627583
[35] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 8, no. 4, 2009.
[36] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam *et al.*, "Jitsu: Just-in-time summoning of unikernels," in *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, 2015, pp. 559–573.
[37] K. Bhardwaj, M.-W. Shih, P. Agarwal, A. Gavrilovska, T. Kim, and K. Schwan, "Fast, scalable and secure onloading of edge functions using airbox," in *2016 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2016, pp. 14–27.
[38] R. Mortier, J. Zhao, J. Crowcroft, L. Wang, Q. Li, H. Haddadi, Y. Amar, A. Crabtree, J. Colley, T. Lodge *et al.*, "Personal data management with the databox: What's inside the box?" in *Proceedings of the 2016 ACM Workshop on Cloud-Assisted Networking*. ACM, 2016, pp. 49–54.
[39] R. Koller and D. Williams, "Will serverless end the dominance of linux in the cloud?" in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, ser. HotOS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 169–173. [Online]. Available: https://doi.org/10.1145/3102980.3103008
[40] A. Hall and U. Ramachandran, "An execution model for serverless functions at the edge," in *Proceedings of the International Conference on Internet of Things Design and Implementation*, ser. IoTDI '19.

New York, NY, USA: Association for Computing Machinery, 2019, p. 225–236. [Online]. Available: https://doi.org/10.1145/3302505.3310084