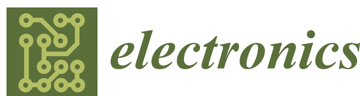

Article

A Distributed Edge-Based Scheduling Technique with Low-Latency and High-Bandwidth for Existing Driver Profiling Algorithms

Mehdi Pirahandeh , Shan Ullah and Deok-Hwan Kim *

Department of Electronic Engineering, Inha University, Incheon 22212, Korea; mehdi@inha.ac.kr (M.P.); shan.ullah@inha.edu (S.U.)
* Correspondence: deokhwan@inha.ac.kr; Tel.: +82-(0)32-860-7424

Abstract: The gradual increase in latency-sensitive, real-time applications for embedded systems encourages users to share sensor data simultaneously. Streamed sensor data have deficient performance. In this paper, we propose a new edge-based scheduling method with high-bandwidth for decreasing driver-profiling latency. The proposed multi-level memory scheduling method places data in a key-value storage, flushes sensor data when the edge memory is full, and reduces the number of I/O operations, network latency, and the number of REST API calls in the edge cloud. As a result, the proposed method provides significant read/write performance enhancement for real-time embedded systems. In fact, the proposed application improves the number of requests per second by 3.5, 5, and 4 times, respectively, compared with existing light-weight FCN-LSTM, FCN-LSTM, and DeepConvRNN Attention solutions. The proposed application also improves the bandwidth by 5.89, 5.58, and 4.16 times respectively, compared with existing light-weight FCN-LSTM, FCN-LSTM, and DeepConvRNN Attention solutions.

Keywords: driver behavior profiling; edge computing; memory scheduling; key-value storage

Citation: Pirahandeh, M.; Ullah, S.; Kim, D.-H. A Distributed Edge-Based Scheduling Technique with Low-Latency and High-Bandwidth for Existing Driver Profiling Algorithms. *Electronics* **2021**, *10*, 972. https://doi.org/10.3390/electronics10080972

Academic Editor: Tony Givargis

Received: 15 March 2021
Accepted: 16 April 2021
Published: 19 April 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Over the years, deep learning algorithms have revolutionized the autonomous car industry by achieving higher accuracy and performance for the comfort of people. However, there is an end-to-end latency issue owing to the need for a higher level of computational resources when autonomous cars simultaneously request driver profiling. Edge computing enables driver-profiling services to reduce the end-to-end latency by providing services to users closer their vicinity. If resources are exhausted in edge computing, service migration must be performed seamlessly to fulfill requirements of the user. However, the ultimate answer will be offloading such services using edge-based solutions. In this paper, we propose a new in-memory data scheduling technique to provide locality awareness for real time execution and fulfillment of users/client (embedded systems) requirements. The data scheduling technique is aimed at decreasing end-to-end latency. In addition, a novel architecture is proposed to deploy a deep learning framework for driver profiling inside the edge server with lower latency, despite a higher number of responses to requests.

The following are the key contributions of this study:

- We present a new architecture for driver-profiling with deep learning techniques in cars with embedded system.
- We achieve a greater number of responses from a driver profiling service.
- We successfully re-implement all the algorithms in an edge server environment.
- We conduct extensive experiments to confirm the advantages of our approach.

The remainder of the paper is structured as follows. We review background information and related work in Section 2. Section 3 presents driver profiling fundamentals.

Section 4 presents detailed experimental results from our approach. We provide conclusions in Section 5.

2. Related Work

Driver behavior profiling using driving features is an emerging trend for multiple markets, such as traffic safety, user-based insurance, and monitoring. In this regard, it is one of the fertile areas of research containing ample studies. In this section, we describe the existing research in two major categories: machine learning models and applied platforms.

2.1. Existing Driver-Profiling Deep Learning Models

Studies dealing with the modeling of individual driver activities use many state-of-the-art machine learning algorithms. These include statistical classification techniques such as the decision tree, random forest, K-nearest neighbors [1], hidden Markov model [2], Gaussian mixture model [3], K-means [4,5], support vector machine (SVM) [5], and many others. However, many of them have various shortcomings, such as data dependence and working under only specific conditions, which are overcome by the vigorous nature of deep learning algorithms [6] having a more significant advantage in feature learning. Scalar data from driving information can be considered a spatiotemporal task because it involves feature extraction per sample (spatial features) as well as includes temporal information, such as the relationships of the samples over time. However, in some studies, a convolutional neural network (CNN) was employed in individual time series to capture local dependencies along temporal dimensions of sensor signals for similar applications, such as action recognition [7]. However, state-of-the-art research offered promising results by using the combination of a CNN (for spatial feature extraction) and long short-term memory (for temporal feature extraction) for driver identification [6,8] and behavior analysis [9,10]. In this research, we compared the proposed architecture with state-of-the-art deep learning algorithms for driver identification, such as the DeepConvRNN [6], and FCN-LSTM [8]. The proposed architecture uses a lightweight FCN-LSTM [11] model with network pruning and offers sparse learning for new classes.

2.2. Applied Embedded Deep Learning Platforms

To deploy driver behavior-profiling models for real-time applications, there are various options, such as a smartphone integrated with the vehicle (e.g., Automotive Grade Linux (AGL) [8,12]), in-vehicle dedicated embedded computers, such as an Advanced Driver-Assistance System (ADAS) [13], cloud- or edge-based services in connected car ecosystems [14], etc. There are many dedicated embedded system solutions available in the market providing energy-efficiency and low -power profile, such as Intel Movidius (Neural Compute Stick -I, -II), Raspberry Pi 3+, and the NVIDIA Jetson series (e.g., Nano, TX1, TX2, and Xavier) replete with high-speed GPUs. Among these, the NVIDIA Jetson series is most favorable because it offers a wide range of developer kits (CudaToolkit, CuDNN) with various specifications. Jetson series provides energy efficiency (low power consumption), less weight, a compact form factor, high performance per watt, and low-power GPU cores [15]. As per comparative studies [16], the Jetson series offers higher peak performance than Raspberry Pi 3+ and Intel Movidius. In this regard, we opted for the Jetson series, implementing the proposed driver behavior identification on a Jetson nano as a client.

In Figure 1, we illustrated the existing EDPA architecture including an interaction diagram, a client and server communication model, and two detailed client and server latency charts. Moreover, Algorithm 1 shows the pseudo-code of the traditional EDPA predicting the driver class using client and server functions. On Lines 1-5, the EDPA-client( ) function first initializes the configurations, reads data from sensors, and then stores data in memory ($data[K]$). Then, the client EDPA function calls a deep learning service named EDPA-server( ), which is initialized by the allocated sensor data, $data[K]$, in memory. As a result, the EDPA-server( ) function returns the driver class, the execution

time, and the accuracy of the driver profiling as a prediction on Line 4. Finally, on Line 5, the EDPA-client( ) function visualizes the prediction data in the embedded car interface. To be more specific, the EDPA-server( ) function is described on Lines 7–12, where it configures TensorFlow for predictions, and re-allocates sensor data on Lines 7–8. Consequently, the EDPA-server( ) function loads the trained model into TensorFlow on Line 10.

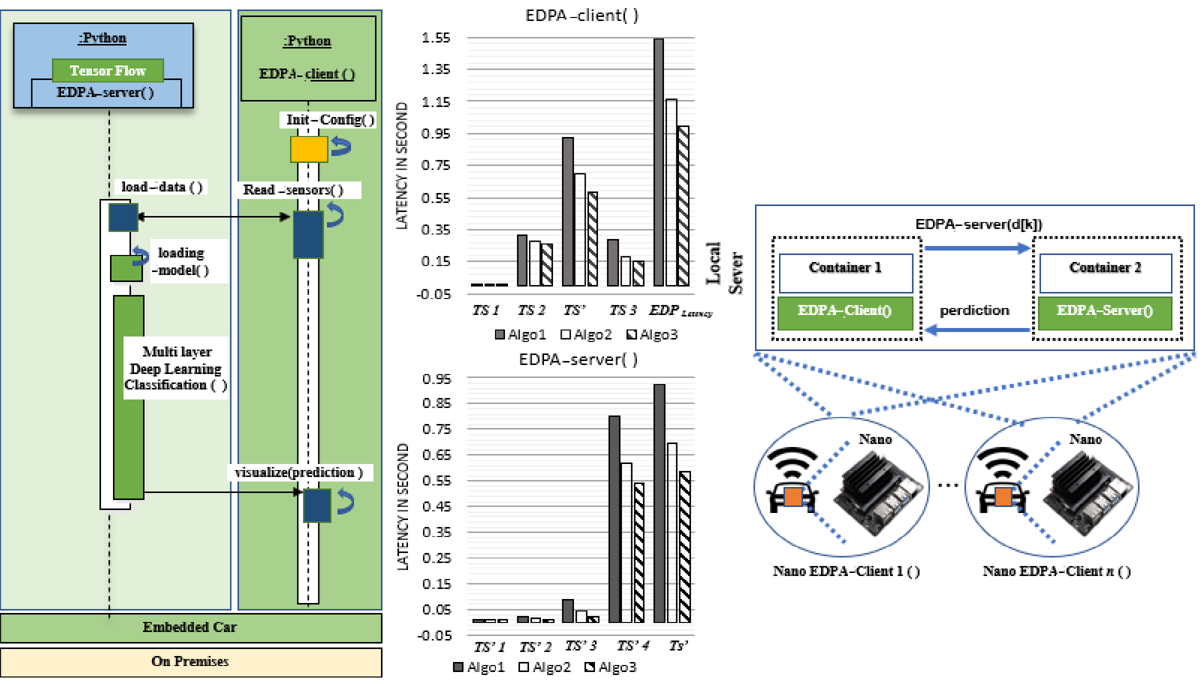

Figure 1. Existing EDPA architecture [6,8,11].

Algorithm 1: The traditional EDPA [6,8,11]

Function `EDPA-client()`:
- Initialize the configurations.
- $d[K] \leftarrow$ read data from sensors.
- $prediction \leftarrow$ EDPA-server($d[k]$).
- visualize(prediction).
- *return*.

Function `EDPA-server()`:
- Initialize the Tensorflow configurations.
- $data[K] \leftarrow$ Initialize the sensor data.
- load the trained model in *TensorFlow*.
- $prediction \leftarrow$ driver_profiling($data[k]$).
- *return prediction*.

Table 1 lists the parameters used in this paper, along with their symbols, representations, and ranges of usage. The end-to-end prediction latency in the proposed EDPA, $EDPA_{latency}$, is as follows:

$$\text{EDPA}_{latency} = TS2 + TS4 + TS5 + TS' + TS3, and TS' > TS2 > TS4 > TS3 > TS5 \quad (1)$$

The total prediction latency in the proposed EDPA using the EDPA-server() function, TS', is as follows:

$$TS' = TS5' + TS4' + TS6', and TS4' > TS5' > TS6' \quad (2)$$

Table 1. Parameter notations.

Symbol	Representation	Applied Function
$TS1$	Initialization latency	EDPA-client( )
$TS2$	Read latency from sensors	EDPA-client( )
$TS3$	Visualization latency	EDPA-client( )
$TS4$	Insert data to cache	EDPA-client( )
$TS5$	Delay	EDPA-client( )
$TS1'$	Initialization latency	EDPA-server( )
$TS2'$	Memory allocation latency	EDPA-server( )
$TS3'$	Trained model loading latency	EDPA-server( )
$TS4'$	Driver profiling latency	EDPA-server( )
$TS5'$	Latency when requesting a job	EDPA-server( )
$TS6'$	Latency when updating prediction	EDPA-server( )
TS'	Total classification latency	EDPA-client( )
$EDPA_{latency}$	End-to-end latency	Embedded system

3. Edge-Based Data Scheduling for FCN-LSTM Driver Profiling

In this study, we only focused on edge-based solutions in order to provide uninterrupted services when the number of requests increases at the edge server. In this scenario, the client will call the driver-profiling API service and will exit accordingly. In the traditional platforms, for every API call, all the deep learning libraries (TensorFlow, etc.) and models (driver profiling model file *h*5, frozen graphs, etc.), will be loaded beforehand in order to execute particular deep learning applications. In the proposed platform, driver-profiling calls are separately managed, and the loading and pre-processing time will be consumed for every single call. In addition, a traditional solution is also computationally expensive, such that it consumes resources that cost the users embedded resource-sensitive computations every time for loading the necessary dependencies and deep learning models for inferences.

The computation power of edge computing services has been revolutionized along with a very high capacity for computations, databases, and flexible services. In our proposed architecture, deep learning services are handled using a distributed architecture based on in-memory caching for sensor data. In this regard, we used the proposed edge-based data scheduling architecture, as shown in Figure 2.

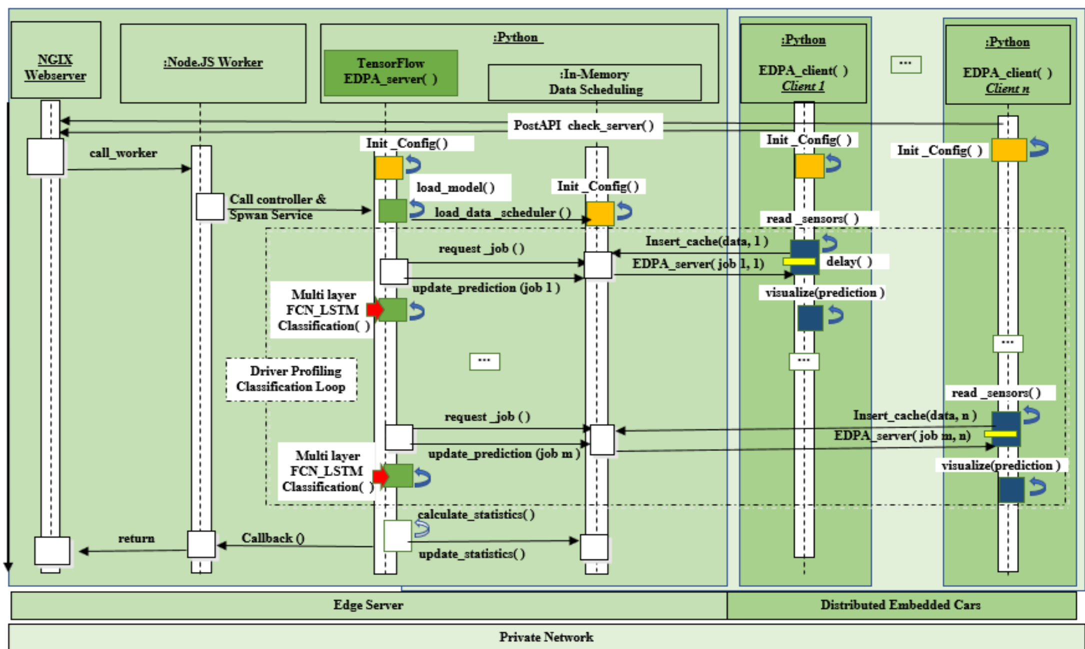

Figure 2. The proposed edge-based data scheduling architecture.

Algorithm 2 shows the pseudo-code of the proposed EDPA for predicting the driver class using distributed client/server functions. First, each $EDPA - client()$ function initializes the configurations and checks the server, and, if the server function was already activated, it is passed to the next line. In contrast, if the server function was not activated, the client will activate the server function using a REST API call. In the next step, iteratively on Lines 4–10, the client function reads data from the sensors and stores the data in memory as job_{data}. Then, the client function saves job_{data} to in-memory cache on Line 6. Later, after a fixed delay, the client function calls EDPA-server( ), which is initialized by the allocated sensor data, job_{id}, on Line 8. As a result, EDPA-server( ) returns the driver class, the execution time, and the accuracy of driver profiling as prediction on Line 8. Finally, EDPA-client( ) visualizes the prediction data in the embedded car interface on Line 9 for the given time series-based data sensors. The proposed distributed EDPA-server( ) function is described on Lines 12–23, where EDPA_server( ) initializes the configurations of TensorFlow for the prediction, and then loads the in-memory data scheduler on Line 14. Consequently, EDPA-server( ) loads the trained model only once for each Node.js worker in TensorFlow on Line 15. In the next step, iteratively on Lines 4–10, the server function reads data by requesting the prediction job from in-memory cache and stores the data in job_{data}. Then, the driver profiling function updates the prediction data for the given job_{id} by calling the $update_prediction$ function developed for in-memory cache on Line 19. Finally, the EDPA-server( ) call calculates the statistics of the prediction data for the given embedded car interface on Line 21 and updates the statistical data of the in-memory cache on Line 22.

Algorithm 2: The proposed EDPA

```
Function EDPA-client():
    Init_config( ).
    check_server( ).
    while true do
        job_data ← read_sensor().
        job_id ← Insert_cache(job_data, job_client_id).
        delay( ).// waiting for prediction.
        prediction ← EDPA-server(job_id, job_client_id).
        visualize(prediction).
    end
    return.
Function EDPA-server():
    Init_config( ).
    load_data_scheduler( ).
    load_model( ).
    while true do
        request_job( ).
        job_prediction ← driver_profiling(job_data).
        update_prediction(job_id).
    end
    calculate_statistics( ).
    update_statistics( ).
    return.
```

The end-to-end prediction latency in the proposed EDPA, $EDPA_{latency}$, is as follows:

$$EDPA_{latency} = TS1 + TS2 + TS^{'} + TS3, and TS^{'} > TS1 > TS3 > TS2 \quad (3)$$

The total prediction latency in the traditional EDPA using the EDPA-server() function, TS', is as follows:

$$TS' = TS1' + TS2' + TS3' + TS4', and TS2' > TS3' > TS4' > TS1' \quad (4)$$

In-Memory Data Scheduler

Key-value (KV) stores are suitable for latency-sensitive internet services, and they have been widely used in large-scale data-intensive internet applications [17]. High demand from users has increased the need for fast read/write performance in accessing databases [18–20]. Sensor-based key-value data consist of many small files, which create a latency bottleneck from low I/O performance in the system [17,21].

As shown in Figure 3 and Algorithm 3, the proposed in-memory data scheduler provides multi-level buffering (MLB) and a flushing flowing-down mechanism, which incurs significant write performance enhancement and makes the KV items move much faster by performing the pipelining process at the edge level. The proposed in-memory data scheduler is suitable for both put-intensive workloads and scan-intensive data analysis workloads and, thus, can be used as the back-end storage engine for edge storage systems. A three-level memtable architecture was designed for the proposed in-memory data scheduler. Memtables and key-value sensor data are constructed using two of linked list (1) and linked list (2) data structures.

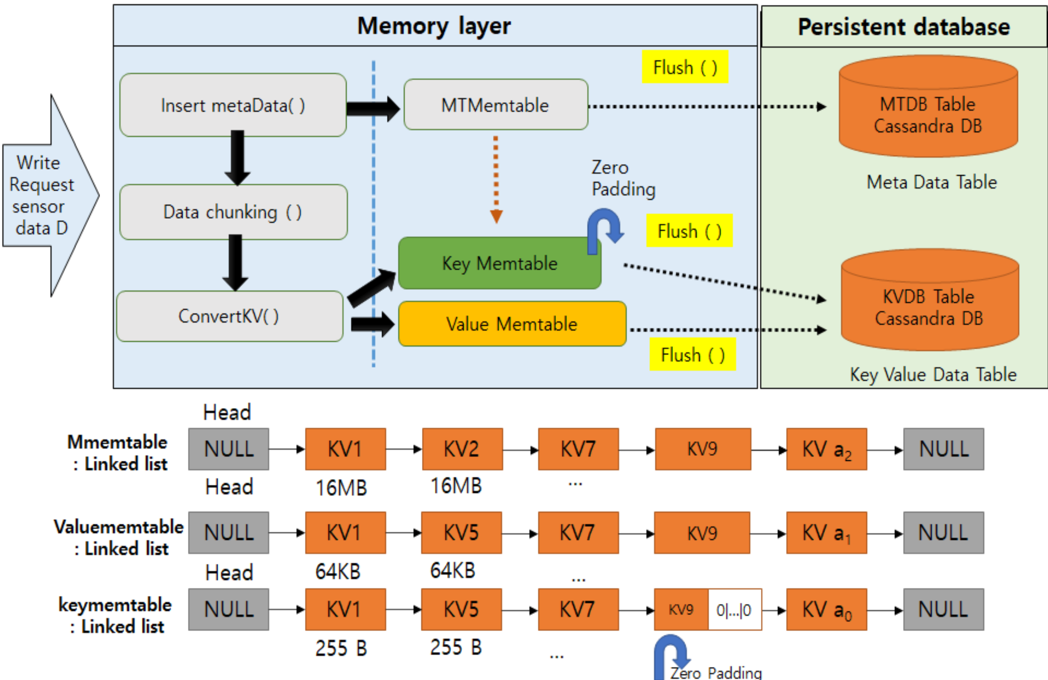

Figure 3. In-memory data scheduling implemented as insert_cache().

In Algorithm 3, $Flush()$ function applies on KV items which are sequenced in advance. KV items are copied at certain intervals (the flush size) to each memtable level. KV items are queued in sequence (flush size = MAX). Each KV item's related meta information is stored in MTMemtable, describing how sensor data are stored in the memtable. A recovery mechanism is needed when the system suffers a sudden power-off, losing all the in-buffer KV items. Therefore, data stored in memtables are flushed to the Cassandra database at each checkpoint. Based on the methods described in [22–24], in-memory schedulers such as Redis have a much lower read write latency compared to the other KV database such as Cassandra and Leveldb. Therefore, in the proposed in-memory scheduler, linked list(1) and (2) in-memory structures play the role of the cache system, which accelerates the read and write performs and sharply reduces the latency of the data scheduler.

Algorithm 3: Data scheduling in Insert_cache function

```
Function Insert_cache():
    Insert metadata and job_client_id into Linked list(1).
    data[K] ← Allocate data chunks for given job_data.
    KV[k][v] ← Convert data[K] into key-value data.
    Insert key-value data KV[k][v] → Linked list(2).
    return job_id
Function Flush():
    Interval ← Set the flushing interval as Interval.
    FS[2] ← Set the flushing size for linked list(1) and (2).
    while true do
        delay (Interval ).
        if Linked list(1) size > = FS[1] then
            Insert All Key-value meta information → CassandraDB.
        end
        if Linked list(2) size > = FS[2] then
            Insert All Key-value sensor data → CassandraDB.
        end
    end
    return.
```

4. Experimental Results

In this section, we evaluate the performance of the proposed low-latency distributed driver behavior-detection system. The system detects five types of driving behaviors through multi-class classification. We used an array unison shuffle technique to randomly shuffle samples of all five classes. Then, we divided the dataset into two non-overlapping sets, including 75% for a training set and 25% for a test set. We experimented with various CNN configurations, including different filter sizes, numbers of convolutional layers, and numbers of filters, to achieve a simple yet efficient network. Moreover, we experimented with a CNN configuration to achieve a model with low computational costs and high efficiency, which is appropriate for embedded applications.

The proposed edge-based driver profiling application (EDPA) was compared to existing EDPAs in terms of the deep learning algorithm (DA), the data scheduling technique (DST), the service architecture type (SAT), the request-handling level (RHL), the scalability level (SL), and end-to-end latency (EEL), as shown in Table 2.

Table 2. A summary of the related EDPA research.

EDPA Reference	DA	DST	SAT	RHL	SL	EEL
Algorithm 1 [6]	DeepConvRNN-Attention	-	Sequential processing	Low	Low	Low
Algorithm 2 [8]	FCN-LSTM	-	Sequential processing	Low	Low	Low
Algorithm 3 [11]	light-weight FCN-LSTM	-	Sequential processing	Low	Low	Medium
This work	FCN-LSTM, DeepConvRNN-Attention, light-weight FCN-LSTM	Yes	Distributed-Parallel processing	High	High	High

4.1. Data Sources

Driving features can be extracted using various sources, mainly using in-vehicle sensor data and smartphone sensor data. In [1,4,6,8,25,26], the authors exploited CAN-Bus data for identification of drivers using footprint. CAN-BUS (OBD-II protocol) communication data include parameters related to (1) the engine (coolant temperature, friction torque, etc.), (2) fuel (long-term fuel trim bank, fuel consumption, etc.), and (3) the transmission (wheel velocity, transmission oil temperature, etc.). Similarly, in other studies, [27–29], smartphone sensor data is used for driver behavior profiling and various other applications [30]. Smart-

phones can provide speed, acceleration, rotation rates, and other parameters for driver behavior profiling. Conversely, some researchers exploit hybrid approaches by combining vision (cameras) and other sensors (LiDAR, GPS, IMU, etc.) for driver profiling [9,10,31]. Besides, few studies included physiological sensor data to classify distracted drivers [32]. CAN-BUS is the most favorable and reliable candidate among the data sources mentioned above [4]. Among various CAN-BUS datasets, a security dataset [1] provides up to 51 features captured from CAN-BUS, and has been used by several researchers recently for driver identification [6,8,11]. The authors of [1] further shortlisted 15 features out of 51 features of CAN bus using InfoGainAttributeEval evaluation method, previously implemented by Weka [33], which is one of the ranker search methods. These 15 features include "Long Term Fuel Trim Bank1", "Intake air pressure","Accelerator Pedal value", "Fuel consumption", "Torque of friction", "Maximum indicated engine torque, Engine torque", "Calculated LOAD value, Activation of Air compressor", "Engine coolant temperature, Transmission oil temperature", "Wheel velocity front left-hand", "Wheel velocity front right-hand", "Wheel velocity rear left-hand", and "Torque converter speed". In this paper, we targeted the same set of aforementioned features, previously utilized by several researchers for driver identification [1,6,8,11]. In this regard, our input size of time series is 15 (features) $\times$ 60 (Window size Wx, as mentioned in Table 2). Subsequently, Algo1 receives multivariate time-series data while Algo2 and Algo3 further process both uni-variate and multivariate time series data by shuffling the dimensions, as depicted in Figure 4a,b.

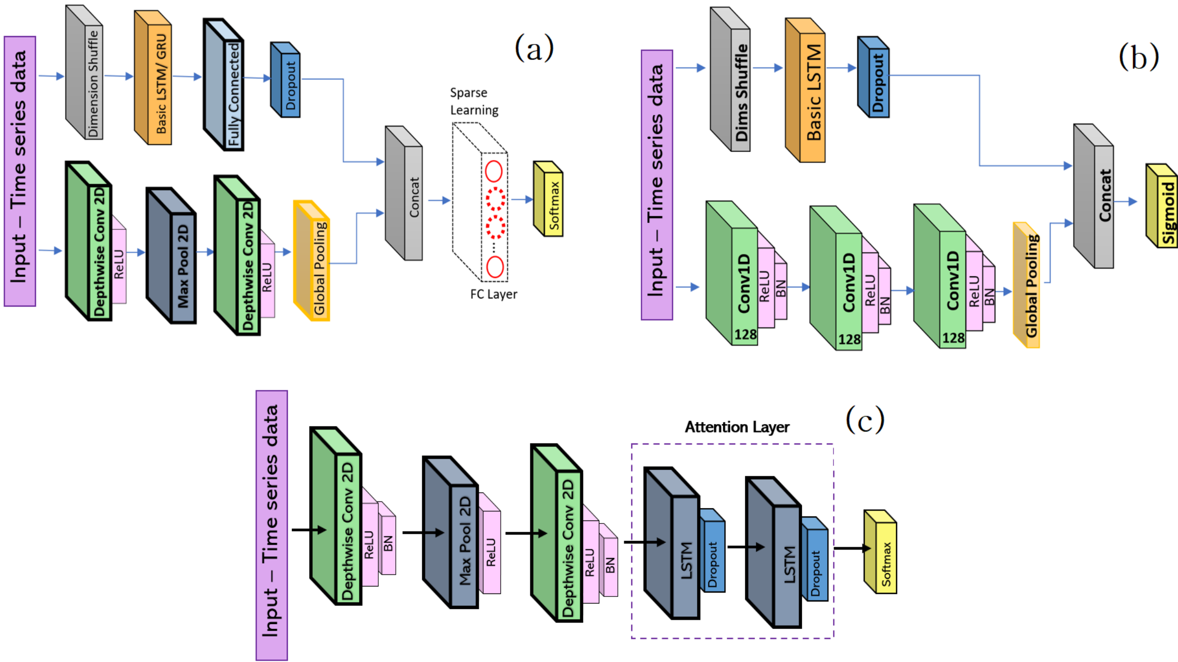

Figure 4. (**a**) Algo3, Depthconv-RNN, Reprinted from ref. [11]; (**b**) Algo2, FCN-LSTM [8], Reprinted from ref. [34] (initially proposed for time-series classification) and (**c**) Algo1, DeepConvRNN-Attention [6].

Table 3. Specifications of embedded devices and edge servers.

Domain	Platform	Hardware Specifications	Implementation Details
Edge Server	Desktop	CPU: Intel Core.i7, RAM: 4 GB, NVME: 128 GB	An edge-Server for four clients
Embedded system	Jetson Nano	CPU: ARM Cortex-A57, RAM: 4 GB, HDD: 128 GB	A local server for a client *

* Note that Jetson Nano is only used for a traditional EDPA as a server function.

Table 4. Driver identification algorithms with their specifications.

Input	Algorithm	Accuracy (%)	FLOPs *	Memory	Feature Engineering	Windowing
60 × 45	Algo1 in the proposed EDPA	97.72	1.524 M	7.53 MB	Yes	Wx = 60, dx = 6
60 × 45	Algo2 in the proposed EDPA	95.19	1.521 M	7.53 MB	Yes	Wx = 60, dx = 6
60 × 15	Algo3 in the proposed EDPA	95.1	0.46 M	3.09 MB	No	Wx = 60, dx = 10
60 × 45	Algo1 in the traditional EDPA [6]	97.83	1.624 M	7.88 MB	Yes	Wx = 60, dx = 6
60 × 45	Algo2 in the traditional EDPA [8]	95.29	1.623 M	7.88 MB	Yes	Wx = 60, dx = 6
60 × 15	Algo3 in the traditional EDPA [11]	94.9	0.56 M	3.28 MB	No	Wx = 60, dx = 10

* Floating point operations per second (mega flop/s).

4.2. Hardware Settings

The specifications of the Jetson Nano client used in embedded systems in cars and the edge server (such as the domain, platform, hardware specifications, and implementation details) are shown in Table 3. In addition, we illustrate the proposed distributed EDPA prototype in Figure 5. As shown in Figure 5, the proposed EDPA Edge Server can connect to four embedded systems in cars and, at the same time, uses four Node.js workers. This is because, edge server run the Node.js Application has multiple limitation such as the number of CPU cores, the number of thread, the size of memory, and the number of Node.js workers. Based on our edge server specification described in Table 3, we can support only four Node.js workers but, if, we use an edge server with higher number of the cores and memory size, we might be able to achieve a higher number of Node.js workers and autonomous cars. The performance of proposed EDPA for each autonomous cars does not depend on the number of workers because of we dedicate to each Node.js worker one autonomous car. Furthermore, if the number of worker increases or decreases we can dedicate less or more autonomous cars into edge server. However, the average overall performance may increase or decrease for a higher or lower number of dedicated autonomous cars.

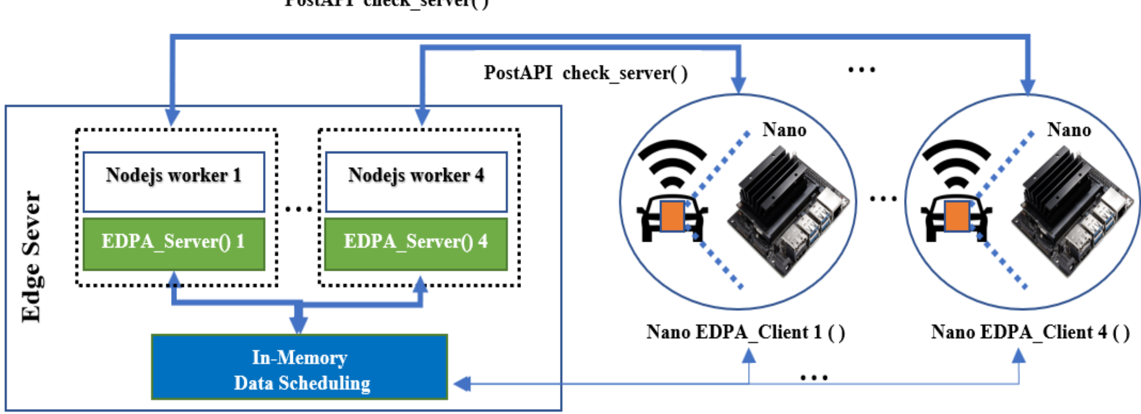

Figure 5. The prototype, including hardware and software settings of the proposed distributed EDPA.

4.3. Evaluation of Driver Profiling

We performed extensive experiments on the driver profiling algorithms, comparing specifications such as accuracy, FLOPs(floating-point operations per second), memory usage, feature engineering, and windowing size, while training models for Algo1, Algo2, and Algo3 in Table 4. We performed the experiment for the proposed EDPA using three mentioned driver profiling algorithms [6,8,11]. Algo1 and Algo2 algorithms require feature

engineering using moving standard deviation, mean, and median, whereas Algo3 algorithm directly normalizes raw features correspondingly. In Table 4, Wx denotes as time series size, and dx denotes as the shift between consecutive sliding window. In addition, the existing driver profiling algorithms require a 60 s window (time steps) for inferring the driver identity. Table 4 proves that the traditional and proposed EDPA architecture similar accuracy, FLOPSs and memory consumption for three driver profiling algorithms, while we tried to improve the latency and bandwidth in the proposed EDPA. Driver profiling latency and the number of requests per second are listed in Table 5. The traditional and the proposed $EDPA_{BW}$ for the existing driver profiling algorithms are listed in Table 6. The proposed EDPA uses applied to the three existing driver profiling algorithms [6,8,11] to train the models in the proposed EDPA. To increase the number of requests per second, we implemented the lightweight FCN_LSTM, which employs sparse learning. Each request defines as a driver profiling classification call. The number Node.js worker handing the driver profiling requests may increases if we use an edge server with higher number of the cores and memory size. The average $EDPA_{latency}$ is calculated using the following formula:

$$Average\ EDPA_{latency} = \frac{Total\ EDPA_{latency}}{The\ number\ of\ requests\ per\ second(Req/Sec)} \tag{5}$$

Moreover, the average bandwidth EDP_{BW} is calculated using the following formula for n input data frames and k clients:

$$Average\ EDPA_{BW} = \frac{\sum_{i=1}^{k}\sum_{j=1}^{n}(Size\ of\ input\ data\ frame)+(Size\ of\ the\ prediction\ data)}{Total\ EDPA_{latency}} \tag{6}$$

Table 5. End-to-end latency and the number of requests per second (Req/s).

Algorithm	Req/s	Total $EDPA_{latency}$	Average $EDPA_{latency}$
Traditional EDPA using Algo1 DeepConvRNN_Attention [6]	1	0.7910 s	0.7910 s
Traditional EDPA using Algo2 FCN-LSTM [8]	1	0.7510 s	0.7510 s
Traditional EDPA using Algo3 light-weight FCN-LSTM [11]	2	1.096 s	0.498 s
Proposed EDPA using Algo3	7	0.651 s	0.142 s
Proposed EDPA using Algo2	5	0.851 s	0.182 s
Proposed EDPA using Algo1	4	0.951 s	0.242 s

Table 6. The $EDPA_{BW}$ for the driver profiling algorithms.

Algorithm	Total Prediction and Input Frames Data Size	Average $EDPA_{BW}$
Traditional EDPA using Algo1 DeepConvRNN_Attention [6]	10 MB	10.09 MB/s
Traditional EDPA using Algo2 FCN-LSTM [8]	10 MB	10.51 MB/s
Traditional EDPA using Algo3 light-weight FCN-LSTM [11]	5 MB	9.12 MB/s
Proposed EDPA using Algo3	5 MB	53.76 MB/s
Proposed EDPA using Algo2	10 MB	58.75 MB/s
Proposed EDPA using Algo1	10 MB	42.06 MB/s

The advantages of the proposed EDPA based on the experimental results are highlighted as follows:

- In the traditional EDPA, EDPA-Client( ) and EDPA-Server( ) are located in different containers inside each Embedded Nano board, which clearly shows they have a 1:1 relationship. Therefore, in the traditional EDPA, Algorithm 1 does not have a loop

structure in client/server, but Algorithm 2 does. In the proposed EDPA, Algorithm 2, EDPA-Server( ) uses Node.js applications which employ multiple workers in parallel which clearly show they are having 1:4 relationship. Besides, we can scale the number of Node.js applications using the load balancer. In the proposed EDPA-server function, initialization latency $TS1'$, memory allocation latency $TS2'$, and trained model loading latency $TS3'$ are excluded from end-to-end latency $EDPA_{latency}$, which results in improving and reducing the average $EDPA_{latency}$ of the EDPA system.
- The function driver_profiling employs the light-weight FCN-LSTM, which execute five requests per second.
- The proposed EDPA-server function connects four embedded cars via the in-memory scheduler, and at the same time uses four Node.js workers.
- The proposed EDPA-server function operates four Node.js workers in parallel.
- Each proposed EDPA-client activates a Nodejs worker using the check_server REST API.
- Each in-memory scheduler thread allocates key, value, and meta information related to each data sensor file using a linked-list structure in parallel.

5. Conclusions

In this study, we proposed a new method for put-intensive, edge-based data scheduling to decrease driver profiling end-to-end latency. The proposed in-memory scheduling stores sensor data in a key-value storage cache. The proposed memory scheduling reduces the number of I/O operations in the edge server by merging sensor data in memory using a linked-list structure. We achieved a greater number of responses for the driver profiling service. We successfully re-implemented all the algorithms in the edge server and conducted multiple experiments to verify the advantages of the proposed EDPA. The proposed application improves end-to-end latency and bandwidth significantly compared with traditional EDPA using the lightweight FCN-LSTM, DeepConvRNN, and FCN-LSTM deep learning algorithms.

Author Contributions: Conceptualization, M.P.; methodology, M.P.; software, M.P., S.U.; validation, M.P., S.U.; formal analysis, M.P., S.U.; investigation, M.P., D.-H.K.; resources, M.P., S.U.; data curation, M.P., S.U.; writing—original draft preparation, M.P.; writing—review and editing, M.P., S.U.; visualization, M.P.; supervision, M.P.; project administration, M.P., D.-H.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported in part by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2020R1I1A1A 01053724) and in part by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2018R1D1A1B07042602) and in part by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No.2019-0-00064, Intelligent Mobile Edge Cloud Solution for Connected Car).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Kwak, B.I.; Woo, J.; Kim, H.K. Know your master: Driver profiling-based anti-theft method. In Proceedings of the 2016 14th Annual Conference on Privacy, Security and Trust (PST), Auckland, New Zealand, 12–14 December 2016; pp. 211–218.
2. Zhang, X.; Zhao, X.; Rong, J. A study of individual characteristics of driving behavior based on hidden markov model. *Sens. Transducers* **2014**, *167*, 194–202.
3. Miyajima, C.; Nishiwaki, Y.; Ozawa, K.; Wakita, T.; Itou, K.; Takeda, K.; Itakura, F. Driver modeling based on driving behavior and its evaluation in driver identification. *Proc. IEEE* **2007**, *95*, 427–437. [CrossRef]
4. Fugiglando, U.; Massaro, E.; Santi, P.; Milardo, S.; Abida, K.; Stahlmann, R.; Netter, F.; Ratti, C. Driving behavior analysis through CAN bus data in an uncontrolled environment. *IEEE Trans. Intell. Transp. Syst.* **2018**, *20*, 737–748. [CrossRef]
5. Van Ly, M.; Martin, S.; Trivedi, M.M. Driver classification and driving style recognition using inertial sensors. In Proceedings of the 2013 IEEE Intelligent Vehicles Symposium (IV), Gold Coast City, Australia, 23–26 June 2013; pp. 1040–1045.
6. Zhang, J.; Wu, Z.; Li, F.; Xie, C.; Ren, T.; Chen, J.; Liu, L. A deep learning framework for driving behavior identification on in-vehicle CAN-BUS sensor data. *Sensors* **2019**, *19*, 1356. [CrossRef] [PubMed]

7. Ha, S.; Choi, S. Convolutional neural networks for human activity recognition using multiple accelerometer and gyroscope sensors. In Proceedings of the 2016 International Joint Conference on Neural Networks (IJCNN), Vancouver, BC, Canada, 24–29 July 2016; pp. 381–388.
8. El Mekki, A.; Bouhoute, A.; Berrada, I. Improving Driver Identification for the Next-Generation of In-Vehicle Software Systems. *IEEE Trans. Veh. Technol.* **2019**, *68*, 7406–7415. [CrossRef]
9. Li, M.G.; Jiang, B.; Che, Z.; Shi, X.; Liu, M.; Meng, Y.; Ye, J.; Liu, Y. DBUS: Human Driving Behavior Understanding System. In Proceedings of the IEEE International Conference on Computer Vision Workshops, Seoul, Korea, 27–28 October 2019.
10. Ramanishka, V.; Chen, Y.T.; Misu, T.; Saenko, K. Toward driving scene understanding: A dataset for learning driver behavior and causal reasoning. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, USA, 18–23 June 2018; pp. 7699–7707.
11. Ullah, S.; Kim, D.-H. Lightweight Driver Behavior Identification Model with Sparse Learning on In-Vehicle CAN-BUS Sensor Data. *Sensors* **2020**, *20*, 5030. [CrossRef] [PubMed]
12. Automotive Grade Linux. 2020. Available online: https://www.automotivelinux.org/ (accessed on 18 April 2020).
13. Brookhuis, K.A.; De Waard, D.; Janssen, W.H. Behavioural impacts of advanced driver assistance systems—An overview. *Eur. J. Transp. Infrastruct. Res.* **2001**, *1*. [CrossRef]
14. Curry, E.; Sheth, A. Next-generation smart environments: From system of systems to data ecosystems. *IEEE Intell. Syst.* **2018**, *33*, 69–76. [CrossRef]
15. Mittal, S. A Survey on optimized implementation of deep learning models on the NVIDIA Jetson platform. *J. Syst. Archit.* **2019**, *97*, 428–442. [CrossRef]
16. Kim, C.E.; Oghaz, M.M.; Fajtl, J.; Argyriou, V.; Remagnino, P. A comparison of embedded deep learning methods for person detection. *arXiv* **2018**, arXiv:1812.03451.
17. Tulkinbekov, K.; Pirahandeh, M.; Kim, D.-H. CLeveldb: Coalesced leveldb for small data. In Proceedings of the 2019 Eleventh International Conference on Ubiquitous and Future Networks (ICUFN), Zagreb, Croatia, 2–5 July 2019; pp. 567–569.
18. Song, T.G.; Pirahandeh, M.; Ahn, C.J.; Kim, D.H. GPU-accelerated high-performance encoding and decoding of hierarchical RAID in virtual machines. *J. Supercomput.* **2018**, *74*, 5865–5888. [CrossRef]
19. Pirahandeh, M.; Kim, D.-H. Energy-aware RAID scheduling methods in distributed storage applications. *Clust. Comput.* **2019**, *22*, 445–454. [CrossRef]
20. Pirahandeh, M.; Kim, D.-H. EGE: A New Energy-Aware GPU Based Erasure Coding Scheduler for Cloud Storage Systems. In Proceedings of the 2018 Tenth International Conference on Ubiquitous and Future Networks (ICUFN), Prague, Czech Republic, 3–6 July 2018; pp. 619–621.
21. Pirahandeh, M.; Kim, D.-H. High performance GPU-based parity computing scheduler in storage applications. *Concurr. Comput. Pract. Exp.* **2017**, *29*, e3889. [CrossRef]
22. Diogo, M.; Cabral, B.; Bernardino, J. Consistency models of NoSQL databases. *Future Internet* **2019**, *11*, 43. [CrossRef]
23. Gupta, A.; Tyagi, S.; Panwar, N.; Sachdeva, S.; Saxena, U. NoSQL databases: Critical analysis and comparison. In Proceedings of the 2017 International Conference on Computing and Communication Technologies for Smart Nation (IC3TSN), Gurgaon, India, 12–14 October 2017; pp. 293–299.
24. Kabakus, A.T.; Kara, R. A performance evaluation of in-memory databases. *J. King Saud Univ. Comput. Inf. Sci.* **2017**, *29*, 520–525. [CrossRef]
25. Kang, Y.G.; Park, K.H.; Kim, H.K. Automobile theft detection by clustering owner driver data. *arXiv* **2019**, arXiv:1909.08929.
26. Park, K.H.; Kim, H.K. This Car is Mine!: Automobile Theft Countermeasure Leveraging Driver Identification with Generative Adversarial Networks. *arXiv* **2019**, arXiv:1911.09870.
27. Castignani, G.; Derrmann, T.; Frank, R.; Engel, T. Driver behavior profiling using smartphones: A low-cost platform for driver monitoring. *IEEE Intell. Transp. Syst. Mag.* **2015**, *7*, 91–102. [CrossRef]
28. Kashevnik, A.; Lashkov, I.; Gurtov, A. Methodology and Mobile Application for Driver Behavior Analysis and Accident Prevention. *IEEE Trans. Intell. Transp. Syst.* **2019**, *21*, 2427–2436. [CrossRef]
29. Warren, J.; Lipkowitz, J.; Sokolov, V. Clusters of driving behavior from observational smartphone data. *IEEE Intell. Transp. Syst. Mag.* **2019**, *11*, 171–180. [CrossRef]
30. Ferreira, J.; Carvalho, E.; Ferreira, B.V.; de Souza, C.; Suhara, Y.; Pentland, A.; Pessin, G. Driver behavior profiling: An investigation with different smartphone sensors and machine learning. *PLoS ONE* **2017**, *12*, e0174959. [CrossRef] [PubMed]
31. Fridman, L.; Brown, D.E.; Glazer, M.; Angell, W.; Dodd, S.; Jenik, B.; Terwilliger, J.; Kindelsberger, J.; Ding, L.; Seaman, S.; et al. Mit autonomous vehicle technology study: Large-scale deep learning based analysis of driver behavior and interaction with automation. *arXiv* **2017**, arXiv:1711.06976.
32. Taamneh, S.; Tsiamyrtzis, P.; Dcosta, M.; Buddharaju, P.; Khatri, A.; Manser, M.; Ferris, T.; Wunderlich, R.; Pavlidis, I. A multimodal dataset for various forms of distracted driving. *Sci. Data.* **2017**, *4*, 170110. [CrossRef] [PubMed]
33. Hall, M.; Frank, E.; Holmes, G.; Pfahringer, B.; Reutemann, P.; Witten, I.H. The WEKA data mining software: An update. *ACM SIGKDD Explor. Newsl.* **2009**, *11*, 10–18. [CrossRef]
34. Karim, F.; Mujamdar, S.; Darabi, H.; Chen, S. LSTM fully convolutional networks for time series classification. *IEEE Access* **2017**, *6*, 1662–1669. [CrossRef]