2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)

DeepPicar: A Low-cost Deep Neural Network-based Autonomous Car

Michael G. Bechtel†, Elise McEllhiney†, Minje Kim⋆, Heechul Yun†
† University of Kansas, USA. {mbechtel, elisemmc, heechul.yun}@ku.edu
⋆ Indiana University, USA. minje@indiana.edu

***Abstract*—We present DeepPicar, a low-cost deep neural network based autonomous car platform. DeepPicar is a small scale replication of a real self-driving car called DAVE-2 by NVIDIA. DAVE-2 uses a deep convolutional neural network (CNN), which takes images from a front-facing camera as input and produces car steering angles as output. DeepPicar uses the same network architecture—9 layers, 27 million connections and 250K parameters—and can drive itself in real-time using a web camera and a Raspberry Pi 3 quad-core platform. Using DeepPicar, we analyze the Pi 3's computing capabilities to support end-to-end deep learning based real-time control of autonomous vehicles. We also systematically compare other contemporary embedded computing platforms using the DeepPicar's CNN-based real-time control workload.**

We find that all tested platforms, including the Pi 3, are capable of supporting the CNN-based real-time control, from 20 Hz up to 100 Hz, depending on hardware platform. However, we find that shared resource contention remains an important issue that must be considered in applying CNN models on shared memory based embedded computing platforms; we observe up to 11.6X execution time increase in the CNN based control loop due to shared resource contention. To protect the CNN workload, we also evaluate state-of-the-art cache partitioning and memory bandwidth throttling techniques on the Pi 3. We find that cache partitioning is ineffective, while memory bandwidth throttling is an effective solution.

***Keywords*-Real-time, Autonomous car, Convolutional neural network, Case study**

I. Introduction

Autonomous cars have been a topic of increasing interest in recent years as many companies are actively developing related hardware and software technologies toward fully autonomous driving capability with no human intervention. Deep neural networks (DNNs) have been successfully applied in various perception and control tasks in recent years. They are important workloads for autonomous vehicles as well. For example, Tesla Model S was known to use a specialized chip (MobileEye EyeQ), which used a vision-based real-time obstacle detection system based on a DNN. More recently, researchers are investigating DNN based end-to-end real-time control for robotics applications [5], [21]. It is expected that more DNN based artificial intelligence (AI) workloads may be used in future autonomous vehicles.

Executing these AI workloads on an embedded computing platform poses several additional challenges. First, many AI workloads, especially those in vehicles, are computationally demanding and have strict real-time requirements. For example, computing latency in a vision-based object detection task may be directly linked to the safety of the vehicle. This requires a high computing capacity as well as the means to guaranteeing the timings. On the other hand, the computing hardware platform must also satisfy cost, size, weight, and power constraints, which require a highly efficient computing platform. These two conflicting requirements complicate the platform selection process as observed in [25].

To understand what kind of computing hardware is needed for AI workloads, we need a testbed and realistic workloads. While using a real car-based testbed would be most ideal, it is not only highly expensive, but also poses serious safety concerns that hinder development and exploration. Therefore, there is a need for safer and less costly testbeds.

In this paper, we present DeepPicar, a low-cost autonomous car testbed for research. From a hardware perspective, DeepPicar is comprised of a Raspberry Pi 3 Model B quad-core computer, a web camera and a small RC car, all of which are affordable components (less than $100 in total). The DeepPicar, however, employs a state-of-the-art AI technology, namely end-to-end deep learning based real-time control, which utilizes a deep convolutional neural network (CNN). The CNN receives an image frame from a single forward looking camera as input and generates a predicted steering angle value as output at each control period in *real-time*. The CNN has 9 layers, about 27 million connections and 250 thousand parameters (weights). DeepPicar's CNN architecture is identical to that of NVIDIA's real-sized self-driving car, called DAVE-2 [5], which drove on public roads without human driver's intervention while only using the CNN.

Using DeepPicar, we systematically analyze its real-time capabilities in the context of end-to-end deep-learning based real-time control, especially on real-time *inferencing* of the CNN. We also evaluate other, more powerful, embedded computing platforms to better understand achievable real-time performance of DeepPicar's CNN based control system and the performance impact of computing hardware architectures. From the systematic study, we want to answer the following questions: (1) How well does the CNN based real-time inferencing task perform on contemporary embedded multicore computing platforms? (2) How susceptible is the CNN inferencing task to contention in shared hardware resources (e.g., cache and DRAM) when multiple tasks/models are consolidated? (3) Are the existing state-of-the-art shared resource isolation techniques effective in protecting real-time performance of the CNN inferencing task?

2325-1301/18/$31.00 ©2018 IEEE
DOI 10.1109/RTCSA.2018.00011

Our main observations are as follows. First, we find that real-time processing of the CNN inferencing is feasible on contemporary embedded computing platforms, even one as inexpensive as the Raspberry Pi 3. Second, while consolidating additional CNN models and tasks on a single multicore platform is feasible, the impact of shared resource contention can be catastrophic—we observe up to 11.6X slowdown even when the CNN model was given a dedicated core. Third, our evaluation of existing cache partitioning and memory bandwidth throttling techniques [32], [33] shows that cache partitioning is not effective in protecting the CNN inferencing task while memory bandwidth throttling is quite effective.

This paper makes the following **contributions**:

- We present DeepPicar, a low-cost autonomous car testbed, which employs a state-of-the-art CNN based end-to-end real-time control [1].
- We provide extensive empirical performance evaluation results of the CNN inferencing workload on a number of contemporary embedded computing platforms.
- We apply the state-of-the-art shared resource isolation techniques and evaluate their effectiveness in protecting the inferencing workload in consolidated settings.

The remainder of the paper is organized as follows. Section II provides a background on the application of neural networks in autonomous driving. Section III gives an overview of the DeepPicar testbed. Section IV presents extensive real-time performance evaluation results we collected on the testbed. Section V offers a comparison between the Raspberry Pi 3 and other embedded computing platforms. We review related work in Section VI and conclude in Section VII.

II. Background

In this section, we provide background on the application of deep learning in robotics, particularly autonomous vehicles.

A. End-to-End Deep Learning for Autonomous Vehicles

To solve the problem of autonomous driving, a standard approach has been decomposing the problem into multiple sub-problems, such as lane marking detection, path planning, and low-level control, which together form a processing pipeline [5]. Recently, researchers have begun exploring another approach that dramatically simplifies the standard control pipeline by applying deep neural networks to directly produce control outputs from sensor inputs [21]. Figure 1 shows the differences between the two approaches.

The use of neural networks for end-to-end control of autonomous cars was first demonstrated in the late 1980s [26], using a small 3-layer fully connected neural network; and subsequently in a DARPA Autonomous Vehicle (DAVE) project in early 2000s [19], using a 6 layer convolutional neural network (CNN); and most recently in NVIDIA's DAVE-2 project [5], using a 9 layer CNN. In all of these projects, the neural network models take raw image pixels as input

[1]The source code, datasets, and build instructions of DeepPicar can be found at: https://github.com/mbechtel2/DeepPicar-v2

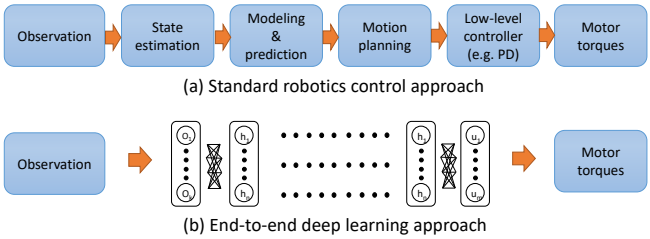


Fig. 1: Standard robotics control vs. DNN based end-to-end control. Adopted from [20].

and directly produce steering control commands, bypassing all intermediary steps and hand-written rules used in the conventional robotics control approach. NVIDIA's latest effort reports that their trained CNN autonomously controls their modified cars on public roads without human intervention [5].

Using deep neural networks involves two distinct phases [24]. The first phase is *training*, during which the weights of the network are incrementally updated by backpropagating errors it sees from the training examples. Once the network is trained—i.e., the weights of the network minimize errors in the training examples—the next phase is *inferencing*, during which unseen data is fed to the network as input to produce predicted output (e.g., predicted image classification). In general, the training phase is more computationally intensive and requires high throughput, which is generally not available on embedded platforms. The inferencing phase, on the other hand, is relatively less computationally intensive and latency becomes as important, if not moreso, as computational throughput, because many use cases have strict real-time requirements.

B. Embedded Computing Platforms for Real-Time Inferencing

Real-time embedded systems, such as an autonomous vehicle, present unique challenges for deep learning, as the computing platforms of such systems must satisfy two often conflicting goals: (1) The platform must provide enough computing capacity for real-time processing of computationally expensive AI workloads (deep neural networks); and (2) The platform must also satisfy various constraints such as cost, size, weight, and power consumption limits [25].

Accelerating AI workloads, especially inferencing operations, has received a lot of attention from academia and industry in recent years as applications of deep learning are broadening to include areas of real-time embedded systems such as autonomous vehicles. These efforts include the development of various heterogeneous architecture-based system-on-a-chip (SoCs) that may include multiple cores, GPU, DSP, FPGA, and neural network optimized ASIC hardware [16]. Consolidating multiple tasks on SoCs with a lot of shared hardware resources while guaranteeing real-time performance is also an active research area, which is orthogonal to improving raw performance. Consolidation is necessary for efficiency, but unmanaged interference can nullify the benefits of consolidation [18].

The *primary objectives of this study* are (1) to understand the necessary computing performance to realize deep neural network based robotic systems, (2) to understand the characteristics of the computing platform to support such workloads, and (3) to evaluate the significance of contention in shared hardware resources and existing mitigation techniques to address the contention problem.

To achieve these goals, we implement a low-cost autonomous car platform as a case-study and systematically conduct experiments, which we will describe in the subsequent sections.

III. DeepPicar

In this section, we provide an overview of our DeepPicar platform. In developing DeepPicar, one of our primary goals is to faithfully replicate NVIDIA's DAVE-2 system on a smaller scale using a low cost multicore platform, the Raspberry Pi 3. Because Raspberry Pi 3's computing performance is much lower than that of the DRIVE PX [22] platform used in DAVE-2, we are interested in if, and how, we can process computationally expensive neural network operations in real-time. Specifically, inferencing (forward pass processing) operations must be completed within each control period duration—e.g., a WCET of 33.3 ms for 30 Hz control frequency—locally on the Pi 3 platform, although training of the network (back-propagation for weight updates) can be done offline and remotely using a desktop computer.

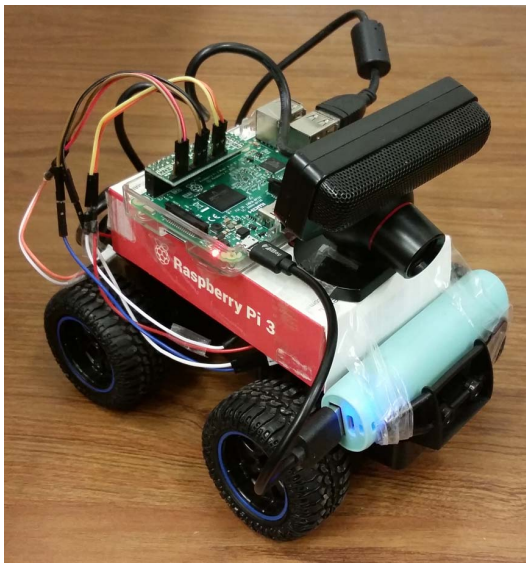


Fig. 2: DeepPicar platform.

Figure 2 shows the DeepPicar, which is comprised of a set of inexpensive components: a Raspberry Pi 3 Single Board Computer (SBC), a Pololu DRV8835 motor driver, a Playstation Eye webcam, a battery, and a 1:24 scale RC car. Table I shows the estimated cost of the system.

For the neural network architecture, we implement NVIDIA DAVE-2's convolutional neural network (CNN) using an open-source CNN model in [6]. Note, however, that the CNN model in [6] is considerably larger than NVIDIA's CNN model as it contains an additional fully-connected layer of approximately 1.3M additional parameters. We remove the additional layer to faithfully recreate NVIDIA's original CNN model. As in DAVE-2, the CNN takes a raw color image (200x66 RGB pixels) as input and produces a single steering angle value as output. Figure 3 shows the network architecture used in this paper, which is comprised of 9 layers, 250K parameters, and about 27 million connections as in NVIDIA DAVE-2's architecture.

Item	Cost ($)
Raspberry Pi 3 Model B	35
New Bright 1:24 scale RC car	10
Playstation Eye camera	7
Pololu DRV8835 motor hat	8
External battery pack & misc.	10
Total	70

TABLE I: DeepPicar's bill of materials (BOM)

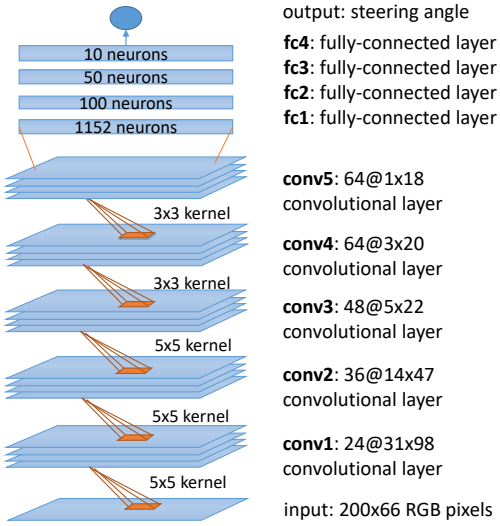
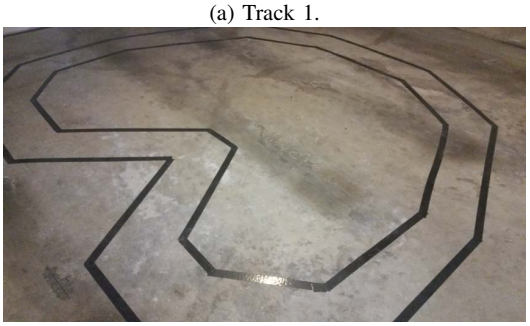


Fig. 3: DeepPicar's neural network architecture: 9 layers (5 convolutional, 4 fully-connected layers), 27 million connections, 250K parameters. The CNN architecture is identical to the one used in NVIDIA's real self-driving car [5].

To collect the training data, a human pilot manually drives the RC car on tracks we created (Figure 4) to record timestamped videos and contol commands. The stored data is then copied to a desktop computer, which is equipped with a NVIDIA Titan Xp GPU, where we train the network to accelerate training speed.

Once the network is trained on the desktop computer, the trained model is copied back to the Raspberry Pi 3. The network is then used by the car's main controlller, which feeds image frames from the web camera as input to the network in real-time. At each control period, the network produced steering angle output is converted into the PWM values of the car's steering motor. Figure 5 shows simplified pseudo code of

(a) Track 1.



(b) Track 2.

Fig. 4: Custom tracks used for training/testing

```
while True:
  # 1. read from the forward camera
  frame = camera.read()
  # 2. convert to 200x66 rgb pixels
  frame = preprocess(frame)
  # 3. perform inferencing operation
  angle = DNN_inferencing(frame)
  # 4. motor control
  steering_motor_control(angle)
  # 5. wait till next period begins
  wait_till_next_period()
```

Fig. 5: Control loop

the controller's main loop. Among the five steps, the 3rd step, network inferencing, is the most computationally intensive and is expected to dominate the execution time.

Note that although the steering angle output of the CNN ($angle$) is a continuous real value, the RC car we used only supports three discrete angles—left (-30°), center (0°), and right (+30°)—as control inputs. We approximate the network generated real-valued angle to the closest one of the three angles. Although this may introduce inaccuracy in control, the observed control performance of the system is respectable, likely due to the inherently stochastic nature of CNNs.

Other factors that can potentially affect the prediction accuracy of the CNN, are camera and actuator (motor) control latencies. The camera latency is defined as the time the camera sensor observes the scene to the time the computer actually reads the digitized image data. This time can be noticable depending on the camera used and the data processing time of the computing platform. Higher camera latency could negatively affect control performance, because the CNN would analyze stale scenes. The actuator (motor) control latency is defined as the time the control output is sent to the steering motor to the time the motor actually moves to a desired position, which also can take considerable time. In our platform, the combined latency is measured to be around 50 ms, which is reasonable. If this value is too high, control performance may suffer. Our initial prototype suffered from this problem as the observed latency was as high as 300 ms, which negatively affected control performance. For reference, the latency of human perception is known to be as fast as 13 ms [30].

Our trained CNN models showed good prediction accuracy, successfully navigating several different tracks we used for training. For instance, the DeepPicar could remain on Track 1 (Figure 4a) for over 10 minutes at a moderate speed (50% throttle), at which point we stopped the experiment, and more than one minute at a higher speed (75% throttle) [2]. Running at higher speed is inherently more challenging because the CNN controller has less time to recover from mistakes (bad predictions). Also, we find that the prediction accuracy is significantly affected by the quality of training data as well as various environmental factors such as lighting conditions. We plan to investigate more systematic ways to improve the CNN's prediction accuracies.

We would like to stress, however, that our main focus of this study is not in improving the network accuracy but in closely replicating the DAVE-2's network architecture and studying its real-time characteristics, which will be presented in the subsequent section.

IV. Evaluation

In this section, we experimentally analyze various real-time aspects of the DeepPicar. This includes (1) measurement based worst-case execution time (WCET) analysis of deep learning inferencing, (2) the effect of using multiple cores in accelerating inferencing, (3) the effect of co-scheduling multiple deep neural network models, and (4) the effect of co-scheduling memory bandwidth intensive co-runners, and (5) the effect of shared L2 cache partitioning and memory bandwidth throttling for guaranteed real-time performance.

A. Setup

The Raspberry Pi 3 Model B platform used in DeepPicar equips a Broadcom BCM2837 SoC, which has a quad-core ARM Cortex-A53 cluster, running at up to 1.2GHz. Each core has private 32K I/D caches and all cores share a 512KB L2 cache. The chip also includes Broadcom's Videocore IV GPU, although we do not use the GPU in our evaluation, due to the lack of sofware support [3]. For software, we use Ubuntu MATE 16.04 and TensorFlow 1.1. We disable DVFS (dynamic voltage frequency scaling) and configure the clock speed of each core statically at the maximum 1.2GHz. We use the SCHED_FIFO

[2]Self-driving videos: https://photos.app.goo.gl/q40QFieD5iI9yXU42

[3]TensorFlow currently only supports NVIDIA's GPUs.

real-time scheduler to schedule the CNN control task while using the CFS when executing memory intensive co-runners.

B. Inference Timing for Real-Time Control

For real-time control of a car (or any robot), the control loop frequency must be sufficiently high so that the car can quickly react to the changing environment and its internal states. In general, control performance improves when the frequency is higher, though computation time and the type of particular physical system are factors in determining a proper control loop frequency. While a standard control system may be comprised of multiple control loops with differing control frequencies—e.g., an inner control loop for lower-level PD control, an outer loop for motion planning, etc.—DeepPicar's control loop is a single layer, as shown earlier in Figure 5, since a single deep neural network replaces the traditional multi-layer control pipline. (Refer to Figure 1 on the differences between the standard robotics control vs. end-to-end deep learning approach). This means that the CNN inference operation must be completed within the inner-most control loop update frequency.

To understand achievable control-loop update frequencies, we experimentally measured the execution times of DeepPicar's CNN inference operations.

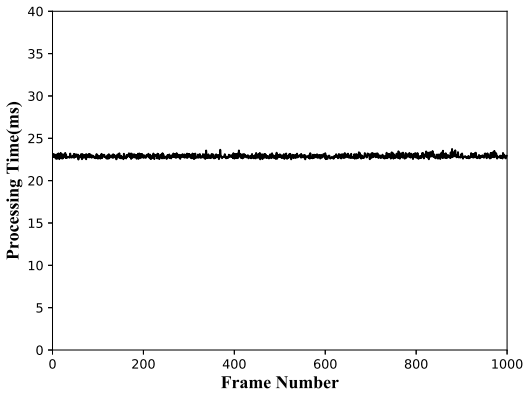


Fig. 6: DeepPicar's control loop processing times over 1000 input image frames.

Operation	Mean	Max	99pct.	Stdev.
Image capture	1.61	1.81	1.75	0.05
Image pre-processing	2.77	2.90	2.87	0.04
CNN inferencing	18.49	19.30	18.99	0.20
Total Time	22.86	23.74	23.38	0.20

TABLE II: Control loop timing breakdown.

Figure 6 shows the measured control loop processing times of the DeepPicar over 1000 image frames (one per each control loop). We omit the first frame's processing time for cache warmup. Table II shows the time breakdown of each control loop. Note that all four CPU cores of the Raspberry Pi 3 were used by the TensorFlow library when performing the CNN inference operations.

First, as expected, we find that the inference operation dominates the control loop execution time, accounting for about 81% of the execution time.

Second, we find that the measured average execution time of a single control loop is 22.86 ms, or 43.7 Hz and the 99 percentile time is 23.38 ms. This means that the DeepPicar can operate at up to 40 Hz control frequency in real-time using only the on-board Raspberry Pi 3 computing platform, as no remote computing resources were necessary. We consider these results surprising given the complexity of the deep neural network, and the fact that the inference operation performed by TensorFlow only utilizes the CPU cores of the Raspberry Pi 3. In comparison, NVIDIA's DAVE-2 system, which has the exact same neural network architecture, reportedly runs at 30 Hz [5]. Although we believe it was not limited by their computing platform (we will experimentally compare performance differences among multiple embedded computing platforms, including NVIDIA's Jetson TX2, later in Section V), the fact that a low-cost Raspberry Pi 3 can achieve comparable real-time control performance is surprising.

Lastly, we find that the control loop execution timing is highly predictable and shows very little variance over different input image frames. This is because the amount of computation needed to perform a CNN inferencing operation is fixed at the CNN architecture design time and does not change at runtime over different inputs (i.e., different image frames). This predictable timing behavior is a highly desirable property for real-time systems, making CNN inferencing an attractive real-time workload.

C. Effect of the Core Count to Inference Timing

In this experiment, we investigate the scalability of performing inference operations of DeepPicar's neural network with respect to the number of cores. As noted earlier, the Raspberry Pi 3 platform has four Cortex-A53 cores and TensorFlow provides a programmable mechanism to adjust how many cores are to be used by the library. Leveraging this feature, we repeat the experiment in the previous subsection but with varying numbers of CPU cores—from one to four.

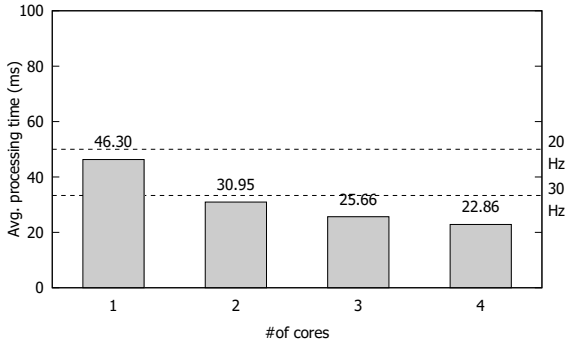


Fig. 7: Average control loop execution time vs. #of CPU cores.

Figure 7 shows the average execution time of the control loop as we vary the number of cores used by TensorFlow. As expected, as we assign more cores, the average execution time decreases—from 46.30 ms on a single core to 22.86 ms on four cores (over a 100% improvement). However, the improvement is far from an ideal linear scaling. In particular, from 3 cores to 4 cores, the improvement is a mere 2.80 ms, or 12%. In short, we find that the scalability of DeepPicar's deep neural network is not ideal.

As noted in [24], CNN inferencing is inherently more difficult to parallelize than training because the easiest parallelization option, batching (i.e., processing multiple images in parallel), is not available or is limited. Specifically, in DeepPicar, only one image frame, obtained from the camera, can be processed at a time. Thus, more fine-grained algorithmic parallelization is needed to improve inference performance [24], which generally does not scale well.

On the other hand, the limited scalability opens up the possibility of consolidating multiple different tasks or different neural network models rather than allocating all cores for a single neural network model. For example, it is conceivable to use four cameras and four different neural network models, each of which is trained separately for a different purpose and executed on a single dedicated core. Assuming we use the same network architecture for all models, then one might expect to achieve up to 20 Hz using one core (given that 1 core can deliver 46 ms average execution time). In the next experiment, we investigate the feasibility of such a scenario.

D. Effect of Co-scheduling Multiple CNN Models

In this experiment, we launch multiple instances of DeepPicar's CNN model at the same time and measure its impact on their inference timings. In other words, we are interested in how shared resource contention affects inference timing. For this, we create four different neural network models, that have the same network architecture, and run them simultaneously.

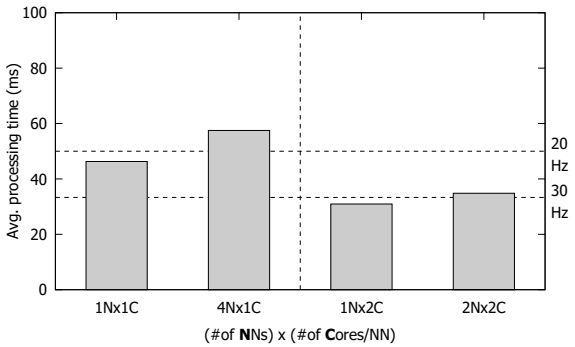


Fig. 8: Timing impact of co-scheduling multiple CNNs. 1Nx1C: one CNN model using one core; 4Nx1C: four CNN models each using one core; 1Nx2C: one CNN model using two cores; 2Nx2C: two CNN models each using two cores.

Figure 8 shows the results. In the figure, the X-axis shows the system configuration: #of CNN models x #of CPU cores/CNN. For example, '4Nx1C' means running four CNN models each of which is assigned to run on one CPU core, whereas '2Nx2C' means running two CNN models, each of which is assigned to run on two CPU cores. The Y-axis shows the average inference timing. The two bars on the left show the impact of co-scheduling four CNN models. Compared to executing a single CNN model on one CPU core (1Nx1C), when four CNN models are co-scheduled (4Nx1C), each model suffers an average inference time increase of approximately 11 ms, or 24%. On the other hand, when two CNN models, each using two CPU cores, are co-scheduled (2Nx2C), the average inference timing is increased by about 4 ms, or 13%, compared to the baseline of running one model using two CPU cores (1Nx2C).

These increases in inference times in the co-scheduled scenarios are expected because co-scheduled tasks on a multicore platform interfere with each other due to contention in the shared hardware resources, such as the shared cache and DRAM [9], [33].

E. Effect of Co-scheduling Synthetic Memory Intensive Tasks

In this experiment, we investigate the *worst-case* impact of shared resource contention on DeepPicar's CNN inference timing using a synthetic memory benchmark. Specifically, we use the *Bandwidth* benchmark from the IsolBench suite [31], which sequentially reads or writes a big array; we henceforth refer to BwRead as Bandwidth with read accesses and BwWrite as the one with write accesses. The experiment setup is as follows: We run a single CNN model on one core, and co-schedule an increasing number of the Bandwidth benchmark instances on the other cores. We repeat the experiement first with BwRead and next with BwWrite.

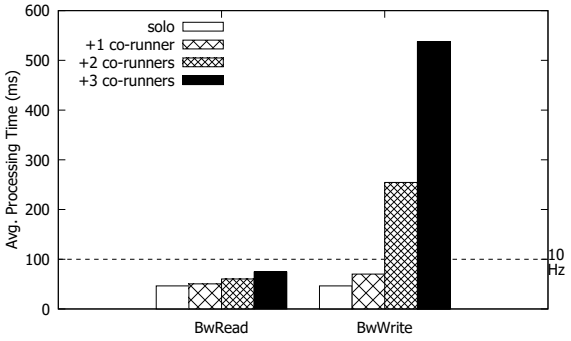


Fig. 9: Average processing time vs. the number of memory intensive co-runners introduced.

Figure 9 shows the results. Note that BwWrite co-runners cause significantly higher execution time increases—up to 11.6X—on CNN inferencing while BwRead co-runners cause relatively much smaller time increases. While execution time increases are expected, the degree to which it is seen in the worst-case is quite surprising—*if the CNN controller was driving an actual car, it would result in a car crash!*

Given the importance of predictable timing for real-time control, such as our CNN based control task, we wanted to: (1) understand the main source of the timing and (2) evaluate existing isolation methods to avoid this kind of timing interference. Specifically, shared cache space and DRAM bandwidth are the two most well-known sources of contention in multicore systems. Thus, in the following sections, we investigate whether and to what extent they influence the observed timing interference and the effectiveness of existing mitigation methods.

F. Effect of Cache Partitioning

Cache partitioning is a well-known technique to improve isolation in a multicore system by giving a dedicated cache space to each individaul task or core. In this experiment, we use PALLOC [32], a page-coloring based kernel-level memory allocator for Linux. Page coloring is an OS technique that controls the physical addresses of memory pages. By allocating pages over non-overlapping cache sets, the OS can effectively partition the cache. Using PALLOC, we investigate the effect of cache partitioning on protecting DeepPicar's CNN based controller.

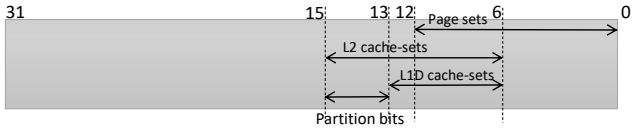


Fig. 10: Physical address mapping of L1/L2 caches of Broadcom BCM2837 processor in Raspberry Pi 3.

Figure 10 shows the physical address mapping of the Raspberry Pi 3's BCM2837 processor, which has 32K private L1 I&D (4way) caches and a shared 512KB L2 (16 way) cache. In order to avoid partitioning the private L1 caches, we use bits 13 and 14 for coloring, which results in 4 usable page colors.

In the first experiment, we investigate the cache space sensitivity of the DeepPicar's CNN-based control loop. Using PALLOC, we create 4 different cgroups which are configured to use 4, 3, 2, and 1 colors (100%, 75%, 50% and 25% of the L2 cache space, respectively). We then execute the CNN control loop (inference) on one core using a different cgroup cache partition, one at a time, and measure the average processing time.

Figure 11 shows the results. As can be seen, the CNN inference timing hardly changes at all regardless of the size of the allocated L2 cache space. In other words, we find that the CNN workload is largely insensitive to L2 cache space.

The next experiment further validates this finding. In this experiment, we repeat the experiment in Section IV-E—i.e., co-scheduling the CNN model and three Bandwidth (BwRead or BwWrite) instances—but this time we ensure that each task is given equal amounts of L2 cache space by assigning one color to each task's cache partition.

Figure 12 shows the results. Compared to Figure 9 where no cache partioning is applied, assigning a dedicated L2

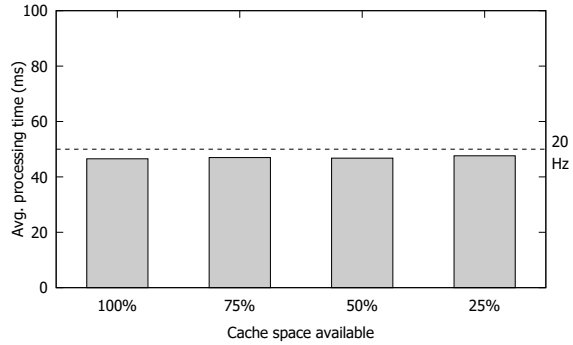


Fig. 11: Cache space sensitivity of the CNN controller.

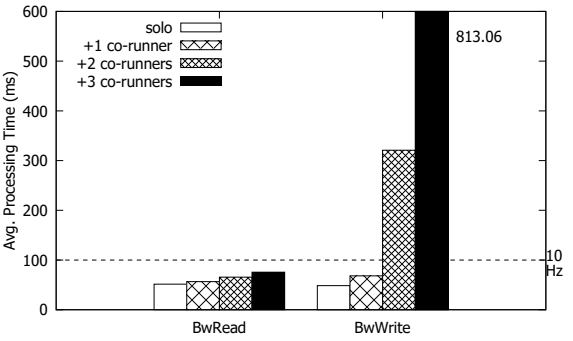


Fig. 12: Average processing time vs. the number of memory intensive co-runners; Each core (task) is given an equal-sized dedicated cache partition.

cache parititon to each core does not provide significant isolation benefits. For BwRead co-runners, cache partitioning slightly improves isolation, but for BwWrite co-runners, cache partitioning causes worse worst-case slowdown.

In summary, we find that the CNN inferencing workload is not sensitive to cache space and that cache partitioning is not effective in providing timing isolation for our CNN workload.

G. Effect of Memory Bandwidth Throttling

In this subsection, we examine the CNN workload's memory bandwidth sensitivity and the effect of memory bandwidth throttling in providing isolation. For the experiments, we use MemGuard [33], a Linux kernel module that can limit the amount of memory bandwidth each core receives. MemGuard operates periodically, at a 1 ms interval, and uses hardware performance counters to throttle cores if they exceed their given bandwidth budgets within each regulation period (i.e., 1 ms), by scheduling high-priority idle kernel threads until the next period begins.

In the first experiment, we measure the performance of the CNN model on a single core, first w/o using MemGuard and then w/ using MemGuard while varying the core's bandwidth throttling parameter from 500 MB/s down to 100 MB/s.

Figure 13 shows the results. When the core executing the CNN model is throttled at 400 MB/s or more, the performance

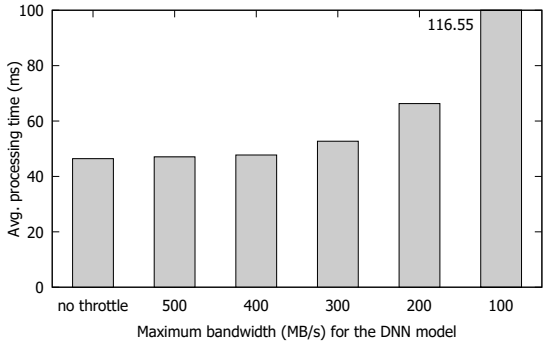


Fig. 13: Memory bandwidth sensitivity of the CNN control loop.

of the model is largely the same as the non-throttled case. However, as we decrease the assigned memory bandwidth below 300 MB/s, we start to observe noticeable decreases in the model's performance. In other words, the CNN model is sensitive to memory bandwidth and it requires 400 MB/s or more bandwidth to ensure ideal performance.

In the next experiment, we repeat the experiment in Section IV-E—i.e., co-scheduling memory intensive synthetic tasks—but this time we throttle the cores of the co-runners using MemGuard and vary their memory bandwidth budgets to see their impact on the CNN model's performance.

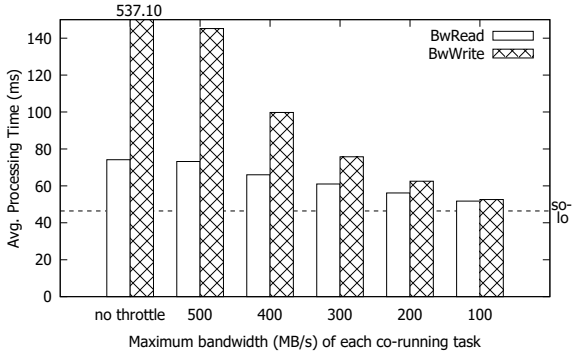


Fig. 14: Effect of throttling three memory intensive co-runners.

Figures 14 shows the results. As can clearly be seen in the figure, limiting the co-runners's memory bandwidth is effective in protecting the CNN model's performance for BwRead and BwWrite co-runners. The benefits are especially more pronounced in case of BwWrite co-runners as, when we throttle them more, the CNN's performance quickly improves.

In summary, we find that the CNN inferencing workload is sensitive to memory bandwidth and that memory bandwidth throttling is effective in improving the performance isolation of the CNN workload.

V. Embedded Computing Platform Comparison

In this section, we compare three computing platforms—the Raspberry Pi 3, the Intel UP [14] and the NVIDIA Jetson TX2 [23]—from the point of view of supporting end-to-end deep learning based autonomous vehicles. Table III shows the architectural features of the three platforms [4].

Our basic approach is to use the same DeepPicar software, and repeat the experiments in Section IV on each hardware platform and compare the results. For the Jetson TX2, we have two different system configurations, which differ in whether TensorFlow is configured to use its GPU or only the CPU cores. Thus, a total of four system configurations are compared.

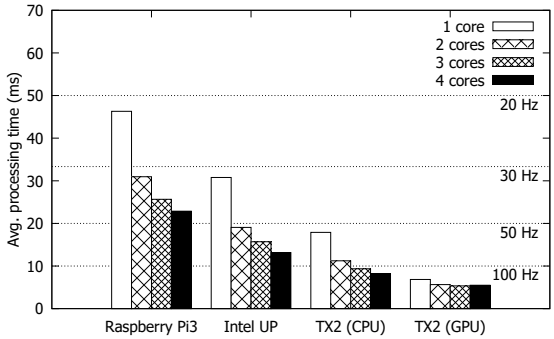


Fig. 15: Average control loop execution time.

Figure 15 shows the average control loop completion timing of the four system configurations we tested as a function of the number of CPU cores used. (cf. Figure 7) Both the Intel UP and Jetson TX2 exhibit better performance than Raspberry Pi 3. When all four CPU cores are used, the Intel UP is 1.33X faster than Pi 3, while the TX2 (CPU) and TX2 (GPU) are 2.79X and 4.16X times faster that the Pi 3, respectively. Thus, they all satisfy 33.3 ms WCET by a clear margin, and, in the case of the TX2, 50 Hz or even 100 Hz real-time control is feasible with the help of its GPU. Another observation is that the CNN task's performance on TX2 (GPU) does not change much as we increase the number of cores. This is because most of the neural network computation is done by the GPU.

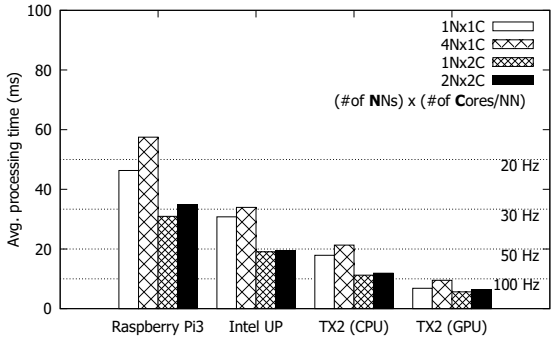


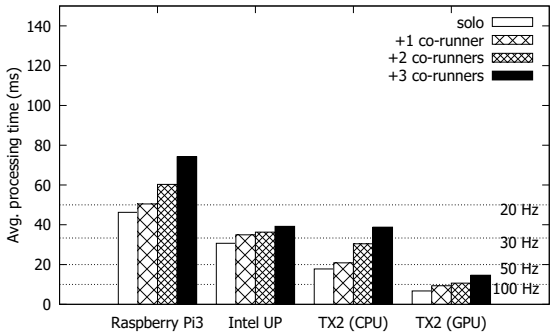
Fig. 16: Timing impact of co-scheduling multiple CNNs on different embedded multicore platforms.

[4]The GPU of Intel UP and the two Denver cores in the Tegra TX2 are not used in evaluation due to TensorFlow issues.

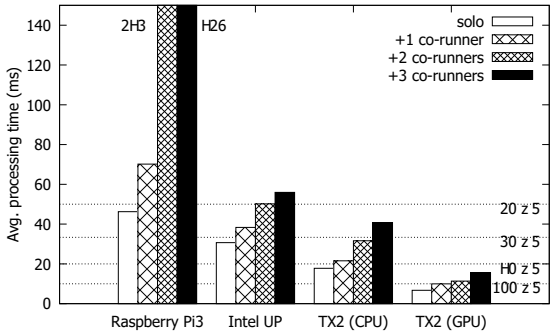
Item	Raspberry Pi 3 (B)	Intel UP	NVIDIA Jetson TX2
CPU	BCM2837 4x Cortex-A53@1.2GHz/512KB L2	X5-Z8350 (Cherry Trail) 4x Atom@1.92GHz/2MB L2	Tegra X2 4x Cortex-A57@2.0GHz/2MB L2 2x Denver@2.0GHz/2MB L2 (not used)
GPU	VideoCore IV (not used)	Intel HD 400 Graphics (not used)	Pascal 256 CUDA cores
Memory	1GB LPDDR2 (Peak BW: 8.5GB/s)	2GB DDR3L (Peak BW: 12.8GB/s)	8GB LPDDR4 (Peak BW: 59.7GB/s)
Cost	$35	$100	$600

TABLE III: Compared embedded computing platforms

Figure 16 shows the results of the multi-model co-scheduling experiment (cf. Figure 8). Once again, they can comfortably satisfy 30 Hz real-time performance for all of the co-scheduled CNN control loops, and in the case of the TX2 (GPU), even 100 Hz real-time control is feasible in all co-scheduling setups. Given that the GPU must be shared among the co-scheduled CNN models, the results suggest that the TX2's GPU has sufficient capacity to accomodate multiple instances of the CNN models we tested.



(a) BwRead



(b) BwWrite

Fig. 17: Average processing time vs. the number of memory intensive co-runners.

Figure 17 shows the results of the synthetic memory intensive task co-scheduling experiments (cf. Figure 9). For read co-runners (BwRead), the performance of all platforms gradually decreased as additional BwRead instances were introduced: up to 1.6X for the Pi 3, up to 1.3X for the Intel UP, and up to 1.6X and 2.2X for the TX2 (CPU) and TX2(GPU), respectively. For write co-runners (BwWrite), however, we observe generally more noticieable execution time increases. As we discussed earlier in Section IV, the Pi 3 suffers up to 11.6X execution time increase, while the Intel UP and Jetson TX2 suffer less dramatic but still significant execution time increases.

Another interesting observation is that the TX2 (GPU) also suffers considerable execution time increase (2.3X) despite the fact that the co-scheduled synthetic tasks do not utilize the GPU (i.e., the CNN model has dedicated access to the GPU.) This is, however, a known characteristic of integrated CPU-GPU architecture based platforms in which both the CPU and GPU share the same memory subsystem [3] and therefore can suffer bandwidth contention as we observe in this case.

In summary, we find that todays embedded computing platforms, even as inexpensive as a Raspberry Pi 3, are powerful enough to support CNN based real-time control applications. Furthermore, availability of CPU cores and a GPU on these platforms allows consolidating multiple CNN workloads. However, shared resource contention among these diverse computing resources remains an important issue that must be understood and controlled, especially for safety-critical applications.

VI. Related Work

There are several relatively inexpensive RC-car based autonomous car testbeds. MIT's RaceCar [28] and UPenn's F1/10 [2] are both based on a Traxxas 1/10 scale RC car and a NVIDIA Jetson multicore computing platform, which is equipped with many sophisticated sensor packages, such as a lidar. However, they both cost more than $3,000, requiring a considerable investment. DonkeyCar [1] is similar to our DeepPicar as it also uses a Raspberry Pi 3 and a similar CNN for end-to-end control, although it costs more (about $200). The main contribution of our paper is in the detailed analysis of computational aspects of executing a CNN-based real-time control workload on diverse embedded computing platforms.

In this paper, we have analyzed real-time performance of a real-world CNN, which was used in NVIDIA's DAVE-2 self-driving car [5], on a low-cost Raspberry Pi 3 quad-core platform and other embedded multicore platforms. It should be noted, however, that DAVE-2's CNN is relatively small compared to recent state-of-the-art CNNs, which are increasingly larger and deeper. For example, the CNN based object detector models evaluated in Google's recent study [11] have between 3M to 54M parameters, which are much larger than DAVE-2's CNN. Using such large CNN models will be challenging on resource constrainted embedded computing

platforms, especially for real-time applications such as self-driving cars.

While continuing performance improvements in embedded computing platforms will certainly make processing these complex CNNs faster, another actively investigated approach is to reduce the required computational complexity itself. Many recent advances in network compression have shown promising results in reducing such computational costs during the feedforward process. The fundamental assumption in those techniques is that the CNNs are redundant in their structure and representation. For example, network pruning can thin out the network and provides a more condensed topology [10].

Another common compression method is to reduce the quantization level of the network parameters, so that arithmetic defined with floating-point operations are replaced with low-bit fixed-point counterparts. To this end, single bit quantization of the network parameters or ternary quantization have been recently proposed [4], [8], [12], [13], [17], [27], [29]. In those networks, the inner product between the originally real-valued parameter vectors is defined with XNOR followed by bit counting, so that the network can greatly minimize the computational cost in the hardware implementations. This drastic quantization can produce some additional performance loss, but those new binarized or ternarized systems provide a simple quantization noise injection mechanism during training so that the additional error is minimized to an acceptable level.

The XNOR operation and bit counting have been known to be efficient in hardware implementations. In [27], it was shown that the binarized convolution could substitute the expensive convolutional feedforward operations in a regular CNN, by using only about 1.5% of the memory space, while providing 20 to 60 times faster feedforward. Binary weights were also able to provide 7 times faster feedforward than a floating-point network for the hand written digit recognition task as well as 23 times faster matrix multiplication tasks on a GPU [12]. Moreover, FPGA implementations showed that the XNOR operation is 200 times cheaper than floating-point multiplications with single precision [4], [8]. XNOR-POP is another hardware implementation that reduced the energy consumption of a CNN by 98.7% [15].

These research efforts are expected to make complex CNNs accessible for a wider range of real-time embedded systems. We plan to investigate the feasibility of these approaches in the context of DeepPicar so that we can use even more resource constrained micro-controller class computing platforms in place of the current Raspberry Pi 3.

VII. Conclusion

We presented DeepPicar, a low cost autonomous car platform that is inexpensive to build, but is based on state-of-the-art AI technology: End-to-end deep learning based real-time control. Specifically, DeepPicar uses a deep convolutional neural network to predict steering angles of the car directly from camera input data in real-time. Importantly, DeepPicar's neural network architecture is identical to that of NVIDIA's real self-driving car DAVE-2.

Despite the complexity of the neural network, DeepPicar uses a low-cost embedded quad-core computer, the Raspberry Pi 3, as its sole computing resource. We systematically analyzed the platform's real-time capability in supporting the CNN-based real-time control task. We also evaluated other, more powerful, embedded computing platforms to better understand achievable real-time performance of DeepPicar's CNN based control system and the impact of computing hardware architectures. We find all tested embedded platforms, including the Pi 3, are capable of supporting the CNN based real-time control, from 20 Hz up to 100 Hz, depending on the platform. Futhermore, all platforms were capable of consolidating multiple CNN models and/or tasks.

However, we also find that shared resource contention remains an important issue that must be considered to ensure desired real-time performance on these shared memory based embedded computing platforms. Toward this end, we evaluated the impact of shared resource contention to the CNN workload in diverse consolidated workload setups, and evaluated the effectivness of state-of-the-art shared resource isolation mechanisms in protecting performance of the CNN based real-time control workload.

As future work, we plan to investigate ways to reduce computational and memory overhead of CNN inferencing and to evaluate the effectiveness of FPGA and other specialized accelerators.

Acknowledgements

This research is supported in part by NSF CNS 1815959 and the National Security Agency (NSA) Science of Security Initiative. The Titan Xp and Jetson TX2 used for this research were donated by the NVIDIA Corporation.

References

[1] DonkeyCar. http://www.donkeycar.com/.
[2] F1/10 autonomous racing competition. http://f1tenth.org.
[3] W. Ali and H. Yun. Protecting real-time GPU applications on integrated CPU-GPU SoC platforms. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2018.
[4] M. J. Beauchamp, S. Hauck, K. D. Underwood, and K. S. Hemmert. Embedded floating-point units in FPGAs. In *International symposium on Field programmable gate arrays (FPGA)*, 2006.
[5] M. Bojarski et al. End-to-End Learning for Self-Driving Cars. *arXiv:1604*, 2016.
[6] L. Fridman. End-to-End Learning from Tesla Autopilot Driving Data. https://github.com/lexfridman/deeptesla.
[7] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Artificial Intelligence and Statistics (AISTATS)*, 2010.
[8] G. Govindu, L. Zhuo, S. Choi, and V. Prasanna. Analysis of high-performance floating-point arithmetic on FPGAs. In *Parallel and Distributed Processing Symposium (IPDPS)*, 2004.
[9] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni. A Survey on Cache Management Mechanisms for Real-Time Embedded Systems. *ACM Computing Surveys*, 2015.
[10] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.
[11] J. Huang, V. Rathod, C. Sun, M. Zhu, A. Korattikara, A. Fathi, I. Fischer, Z. Wojna, Y. Song, S. Guadarrama, et al. Speed/accuracy trade-offs for modern convolutional object detectors. In *Computer Vision and Pattern Recognition (CVPR)*, 2017.

[12] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized neural networks. In *Advances in Neural Information Processing Systems (NIPS)*, 2016.
[13] K. Hwang and W. Sung. Fixed-point feedforward deep neural network design using weights +1, 0, and -1. In *Workshop on Signal Processing Systems (SiPS)*, 2014.
[14] Intel. UP Single Board Computer. http://www.up-board.org/up/.
[15] L. Jiang, M. Kim, W. Wen, and D. Wang. Xnor-pop: A processing-in-memory architecture for binary convolutional neural networks in wide-io2 drams. In *Low Power Electronics and Design (ISLPED)*, 2017.
[16] N. P. Jouppi et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.
[17] M. Kim and P. Smaragdis. Bitwise neural networks. *arXiv preprint arXiv:1601.06071*, 2016.
[18] N. Kim, B. C. Ward, M. Chisholm, C.-y. Fu, J. H. Anderson, and F. D. Smith. Attacking the One-Out-Of-m Multicore Problem by Combining Hardware Management with Mixed-Criticality Provisioning. In *Real-Time Technology and Applications Symposium (RTAS)*, 2016.
[19] Y. Lecun, E. Cosatto, J. Ben, U. Muller, and B. Flepp. DAVE: Autonomous off-road vehicle control using end-to-end learning. Technical Report DARPA-IPTO Final Report, 2004.
[20] S. Levine. Deep Reinforcement Learning. http://rll.berkeley.edu/deeprlcourse/f17docs/lecture_1_introduction.pdf, 2017.
[21] S. Levine, C. Finn, T. Darrell, and P. Abbeel. End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*, 2016.
[22] NVIDIA. DRIVE PX. http://www.nvidia.com/object/drive-px.html.
[23] NVIDIA. Jetson TX2 Developer Kit. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modules/.
[24] NVIDIA. GPU-Based Deep Learning Inference : A Performance and Power Analysis. Technical Report November, 2015.
[25] N. Otterness, M. Yang, S. Rust, E. Park, J. H. Anderson, F. D. Smith, A. Berg, and S. Wang. An Evaluation of the NVIDIA TX1 for Supporting Real-Time Computer-Vision Workloads. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017.
[26] D. a. Pomerleau. ALVINN: An autonomous land vehicle in a neural network. In *Advances in Neural Information Processing Systems (NIPS)*, 1989.
[27] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. XNOR-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision (ECCV)*, 2016.
[28] R. Shin, S. Karaman, A. Ander, M. T. Boulet, J. Connor, K. L. Gregson, W. Guerra, O. R. Guldner, M. Mubarik, B. Plancher, et al. Project based, collaborative, algorithmic robotics for high school students: Programming self driving race cars at mit. Technical report, MIT Lincoln Laboratory Lexington United States, 2017.
[29] D. Soudry, I. Hubara, and R. Meir. Expectation backpropagation: Parameter-free training of multilayer neural networks with continuous or discrete weights. In *Advances in Neural Information Processing Systems (NIPS)*, 2014.
[30] Thomas Burger. How Fast Is Realtime? Human Perception and Technology — PubNub, 2015.
[31] P. K. Valsan, H. Yun, and F. Farshchi. Taming Non-Blocking Caches to Improve Isolation in Multicore Real-Time Systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.
[32] H. Yun, R. Mancuso, Z. Wu, and R. Pellizzoni. PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.
[33] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.

APPENDIX

A. DNN Training and Testing

We have trained and tested the deep neural network with several different track conditions, different combinations of input data, and different hyper parameters. In the following paragraphs, we describe details on two of the training methods that performed reasonably well.

In the first method, we trained the neural network model across a set of 30 completed runs on the track seen in Figure 4b by a human pilot. Half of the runs saw the car driving one way along the track, while the remaining half were of the car driving in the opposite direction on the track. In total, we collected 2,556 frames for training and 2,609 frames for validation. The weights of the network are initialized using the Xavier initializer [7], which is known to perform better than a random weight assignment. In each training step, we use a batch size of 100 frames, which are randomly selected among all the collected training images, to optimize the network. We repeat this across 2,000 training steps.

In the second method, we use the same data and parameters as above except that now images are labeled as 'curved' and 'straight' and we pick an equal number of images from each category at each training step to update the model. In other words, we try to remove bias in selecting images. We find that the car performed better in practice by applying this approach as the car displayed a greater ability to stay in the center of the track (on the white tape). However, we find that there is a discrepency between the training loss and the validation loss, indicating that the model may suffer from an overfitting problem, despite its better real-world performance.

B. System-level Factors Affecting Real-Time Performance

In using the Raspberry Pi 3 platform, there are a few system-level factors, namely power supply and temperature, that need to be considered to achieve consistent performance.

In all of our experiments on the Raspberry Pi 3, the CPU is configured at the maximum clock speed of 1.2 GHz. However, without care, the CPU can operate at a lower frequency involuntarily. An important factor is CPU thermal throttling, which can affect CPU clock speed if the CPU temperature is too high (Pi 3's firmware is configured to throttle at 85 deg. C). DNN inferencing is computationally intensive, thus the temperature of the CPU could rise quickly. This can be especially problematic in situations where multiple DNN models run simultaneously on the Pi 3. If the temperature reaches the threshold, the Pi 3's thermal throttling kicks in and decreases the clock speed down to 600MHz— half of the maximum 1.2GHz—so that the CPU's temperature stays at a safe level. We found that without proper cooling solutions (heatsink or fan), prolonged use of the system would result in CPU frequency decrease that may affect evaluation.

Another factor to consider is power supply. The Pi 3 frequency throttling also kicks in when the power source can not provide 2A current. In experiments conducted with a power supply that only provided 1 Amp, the Pi was unable to sustain a 1.2 GHz clock speed. As a result, it is necessary, or at least highly recommended, that the power supply used for the Raspberry Pi 3 be capable of outputting 2 Amps, otherwise optimal performance isn't guaranteed.

Our initial experiment results suffered from these issues, after which we always carefully monitored the current operating frequencies of the CPU cores during the experiments to ensure the correctness and repeatability of the results.