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

Michael G. Bechtel
Electrical Engineering and Computer Science
University of Kansas
Lawrence, USA
mbechtel@ku.edu

Elise McEllhiney
Electrical Engineering and Computer Science
University of Kansas
Lawrence, USA
e908m428@ku.edu

Minje Kim
Electrical Engineering and Computer Science
Indiana University Bloomington
Bloomington, USA
minje@indiana.edu

Heechul Yun
Electrical Engineering and Computer Science
University of Kansas
Lawrence, USA
heechul.yun@ku.edu

***Abstract*—We present DeepPicar, a low-cost deep neural network (DNN) based autonomous car platform. DeepPicar is a small scale replication of a real self-driving car called Dave-2 by NVIDIA, which drove on public roads using a deep convolutional neural network (CNN), that takes images from a front-facing camera as input and produces car steering angles as output. DeepPicar uses almost the exact same network architecture—9 layers, 27 million connections and 250K parameters—and can be trained to drive itself, in real-time, using a web camera and a modest 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 software as a workload. We find all tested platforms, including the Pi 3, are capable of supporting deep-learning based real-time control, from 20 Hz up to 100 Hz depending on hardware platform. However, shared resource contention remains an important issue that must be considered in applying deep-learning models on shared memory based embedded computing platforms.**

***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 deep neural network for vision-based real-time obstacle detection and avoidance. More recently, researchers are investigating DNN based end-to-end control of cars [4] and other robots. It is expected that more DNN based Artificial Intelligence 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 in vehicles are computationally demanding and have strict real-time requirements. For example, 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 [22].

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 strong need for safer and less costly testbeds. There are already several relatively inexpensive RC-car based testbeds, such as MIT's RaceCar [25] and UPenn's F1/10 racecar [1]. However, these RC-car testbeds still cost more than $3,000, requiring considerable investment.

Instead, we want to build a low cost testbed that still employs the state-of-the art AI technologies. Specifically, we focus on an end-to-end deep learning based real-time control system, which was developed for a real self-driving car, NVIDIA DAVE-2 [4], and uses the same methodology on a smaller and less costly setup. In developing the testbed, our goals are (1) to analyze real-time issues in DNN based end-to-end control; and (2) to evaluate real-time performance of contemporary embedded platforms for such a workload.

In this paper, we present DeepPicar, a low-cost autonomous car platform for research. From a hardware perspective, DeepPicar is comprised of a Raspberry Pi 3 Model B quad-core computer, a web camera and a RC car, all of which are affordable components (less than $100 in total). The DeepPicar,

however, employs state-of-the-art AI technologies, including a vision-based end-to-end control system that utilizes a deep convolutional neural network (CNN). The network 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 network has 9 layers, about 27 million connections and 250 thousand parameters (weights). The network architecture is almost identical to that of NVIDIA's DAVE-2 self-driving car [4], which uses a significantly more powerful computer (Drive PX computer [20]) than a Raspberry Pi 3. We chose to use a Pi 3 not only because it is affordable, but also because it is representative of today's mainstream low-end embedded multicore platforms found in smartphones and other embedded devices.

We apply a standard imitation learning methodology to train the car to follow tracks on the ground. We collect data for training and validation by manually controlling the RC car and recording the vision (from the webcam mounted on the RC-car) and the human control inputs. We then train the network offline using the collected data on a desktop computer, which is equipped with a NVIDIA GTX 1060 GPU. Finally, the trained network is copied back to the Raspberry Pi 3, which is then used to perform inference operations—locally on the Pi 3—in the car's main control loop in real-time. For real-time control, each inference operation must be completed within the desired control period (e.g., $33.\overline{33}$ ms period for 30 Hz control frequency).

Using the DeepPicar platform, we systematically analyze its real-time capabilities in the context of deep-learning based real-time control, especially on real-time deep neural network inferencing. We also evaluated other, more powerful, embedded computing platforms to better understand achievable real-time performance of DeepPicar's deep-learning based control system and the impact of computing hardware architectures.

From the systematic study, we have made a number of interesting observations from the perspective of real-time systems. First, we find that DNN inferencing is highly predictable—from the timing perspective—as the amount of computation needed to complete a single inference is fixed at the DNN architecture design time and does not change at runtime over different inputs (e.g., different image frames). This predictable timing behavior is obviously a desirable property for real-time systems.

Second, we find that real-time processing of DeepPicar's CNN is feasible on today's embedded computing platforms, even one as inexpensive as the Raspberry Pi 3. We find that the control loop completes in under $33.\overline{33}$ ms, or 30 Hz, using just two cores of the Raspberry Pi 3's quad-core CPU (w/o using its GPU), and can do so 100% of the time; or we can achieve 20 Hz control frequency using just one core. Other tested embedded platforms, the Intel UP and NVIDIA TX2, offer even better performance, and are capable of supporting deep-learning based real-time control over 100 Hz control frequency on the TX2 when its embedded GPU is used. We find that all the tested embedded computing platforms have enough computing capacity to consolidate multiple instances of DeepPicar's DNN workload.

Third, contention in shared hardware resources remains an important issue when consolidating multiple workloads. In particular, we find that DNN inferencing is highly sensitive to memory performance interference, as we observe up to an 11.6X control loop execution time increase when memory performance intensive applications are co-scheduled on idle CPU cores. On the other hand, we find that DNN inferencing is not sensitive to cache space. This finding suggests that in order to guarantee real-time performance of DNN-based control applications, solutions to guarantee their memory performance are more important.

The **contributions** of this paper are as follows:

- We present DeepPicar, a low-cost autonomous car testbed, which is based on a state-of-the-art end-to-end deep learning, as open-source [1], with the goal of lowering the economic and safety-related barriers to the study of autonomous cars.
- We systematically analyze real-time characteristics of DeepPicar's DNN inferencing workload on a number of representative contemporary embedded computing platforms. While larger embedded platforms have been evaluated, as in [22], to the best of our knowledge, no prior work investigated the impacts of shared resource contention on DNN inferencing workloads in the context of smaller embedded platforms like the Raspberry Pi 3.

The remaining sections of the paper are as follows. Section II provides a background of applications in autonomous driving and related works. Section III gives an overview of the DeepPicar platform, including the high-level system and the methods used for training and inference. Section IV presents our evaluation of the platform and how different factors can affect performance. Section V offers a comparison between the Raspberry Pi 3 and other embedded computing platforms to determine their suitably for autonomous driving research. 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 [4]. 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 [19]. 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 [23],

[1]The source code, and build instructions for the DeepPicar can be found at: https://github.com/heechul/picar

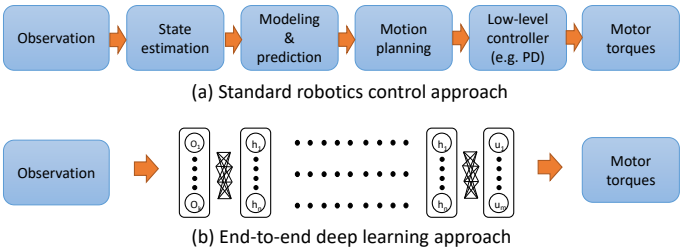


Fig. 1: Standard robotics control vs. DNN based end-to-end control. Adopted from [18].

using a small 3-layer fully connected neural network; and subsequently in a DARPA Autonomous Vehicle (DAVE) project in early 2000s [17], using a 6 layer convolutional neural network (CNN); and most recently in NVIDIA's DAVE-2 project [4], using a 9 layer CNN. In all of these projects, the neural network models take raw image pixels as input 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 [4].

Using deep neural networks involves two distinct phases [21]. 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 (e.g., search query latency).

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 [22]: (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.

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-chips (SoCs) that may include multiple cores, GPU, DSP, FPGA, and neural network optimized ASIC hardware [14]. 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 [16]. For these reasons, finding a good computing platform is a non-trivial task, one that requires a deep understanding of the workloads and the hardware platform being utilized.

The *primary objectives of this study* are to understand (1) the necessary computing performance for applying AI technology-based robotics systems, and (2) what kind of computing architecture and runtime supports are most appropriate for such a workload. 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 Overview

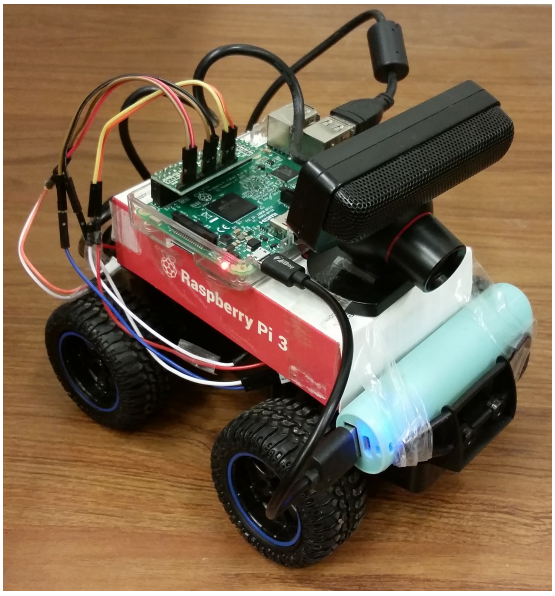


Fig. 2: DeepPicar platform.

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)

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 the Raspberry Pi 3's computing performance is much lower than that of the DRIVE PX 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 $33.\overline{33}$ ms for 30Hz 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. 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.

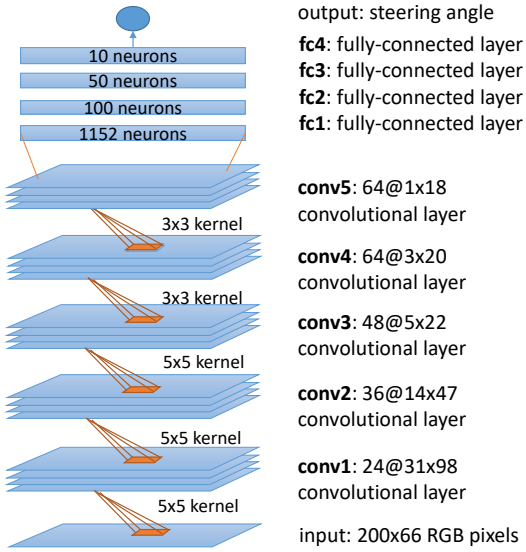
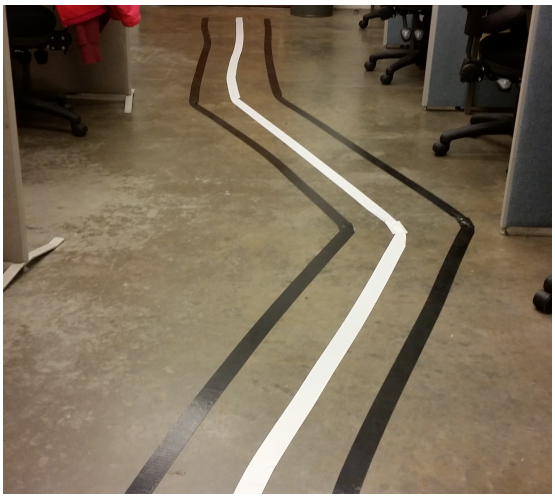


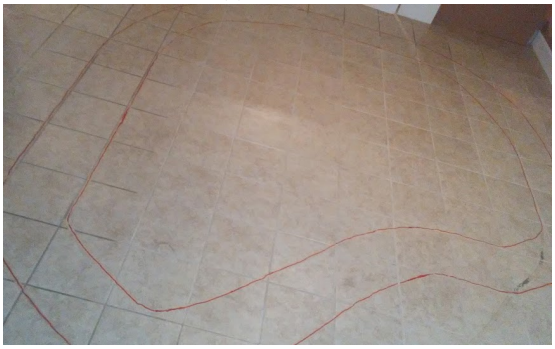
Fig. 3: DeepPicar's neural network architecture: 9 layers (5 convolutional, 4 fully-connected layers), 27 million connections, 250K parameters. The CNN architecture is identical to the one used in NVIDIA's real self-driving car [4].

For the neural network architecture, we implement NVIDIA DAVE-2's convolutional neural network (CNN) using a TensorFlow based CNN model in [5]. Note, however, that the CNN model in [5] was considerably larger than NVIDIA's CNN model as it contained an additional fully-connected layer of approximately 1.3M additional parameters. We eliminated the additional layer to faithfully recreate the 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. Note, however, that we did not apply the normalization mentioned in [4], as it does not include trainable parameters and its computational demand with respect to the overall CNN processing is minimal.

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



(a) Track 1.



(b) Track 2.

Fig. 4: Custom tracks used for training/testing

NVIDIA GTX 1060 GPU, where we train the network to accelerate training speed. For comparison, training the network on the Raspbeery Pi 3 takes approximately 4 hours, whereas it takes only about 4 minutes on the desktop computer using the GTX 1050 Ti GPU.

```
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

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 controller, which feeds an image frame from the web camera as input to the network. In each control period, the produced steering angle output is

then converted into the PWM values of the steering motor of the car. Figure 5 shows simplified pseudo code of the controller's main loop. Among the five steps, the 3rd step, network inferencing, is the most computationally intensive and dominates the execution time.

Note that although the steering angle output of the network ($angle$) is a continuous real value, the RC car platform we used unfortunately only supports three discrete angles—left (-30°), center (0°), and right (+30°)—as control inputs. Currently, we approximate the network generated real-valued angle to the closest one of the three angles, which may introduce inaccuracy in control. In the future, we plan to use a different (more expensive) RC car platform that can precisely control the car's steering angle.

Other issues we observed in training/testing the network, which can affect the prediction accuracy of the CNN, are camera and actuator (motor) control latencies. In DeepPicar's context, the camera latency measured from 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 performance of the Pi. Higher camera latency could negatively affect control performance, because the DNN would analyze stale scenes. Actuator (motor) control latency also can be a factor, because moving the steering motor to a desired position takes time. In our platform, the combined latency is measured to be around 50 ms. In other words, in the recorded video and control data, a control action (left, center, right) taken by the CNN appears to be applied 50 ms later. Considering that camera's framerate is 30Hz ($33.\overline{33}$ ms/frame), this is about two frames after the control action. If this value is too high, control performance may suffer. Our initial prototype suffered this problem as the observed camera and control 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 [27].

Despite these issues, the trained models were still able to achieve a reasonable degree of accuracy, successfully navigating several different tracks we used for training. In our testing, we found that the DeepPicar could remain on a fairly challenging track 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 the issues related to the CNN's accuracies have no impact on the *computational aspects of the system*, and that our main focus of this study is not in improving network accuracy but in closely replicating the DAVE-2's network architecture and studying its real-time characteristics.

[2]A collection of self-driving videos can be found at: https://photos.app.goo.gl/ce93sU7jPk4ywO8u2

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
DNN inferencing	18.49	19.30	18.99	0.20
Total Time	22.87	23.74	23.38	0.20

TABLE II: Control loop timing breakdown.

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.

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 a 32K private Instruction cache and a 32K private Data cache, and all cores share a 512KB L2 cache. The chip also includes Broadcom's Videocore IV GPU, although we did not use the GPU in our evaluation due to the lack of sofware support (TensorFlow is not compatible with the Raspberry Pi's GPU). For software, we use Ubuntu MATE 16.04, TensorFlow 1.1 and Python 2.7. We disabled DVFS (dynamic voltage frequency scaling) and configured the clock speed of each core statically at the maximum 1.2GHz.

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 DNN 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 DNN inference operations.

Figure 6 shows the measured control loop processing times of the DeepPicar over 1000 image frames (one per each control

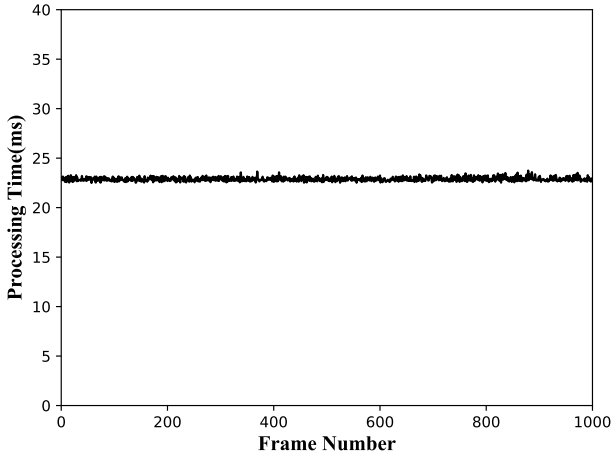


Fig. 6: DeepPicar's control loop processing times over 1000 input image frames.

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 DNN 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 22.87 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 (its GPU is not supported by Tensorflow). In comparison, NVIDIA's DAVE-2 system, which has the exact same neural network architecture, reportedly runs at 30 Hz [4]. 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 DNN inferencing operation is fixed at the DNN 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 DNN inferencing an attractive real-time workload.

C. Effect of the Core Count to Inference Timing

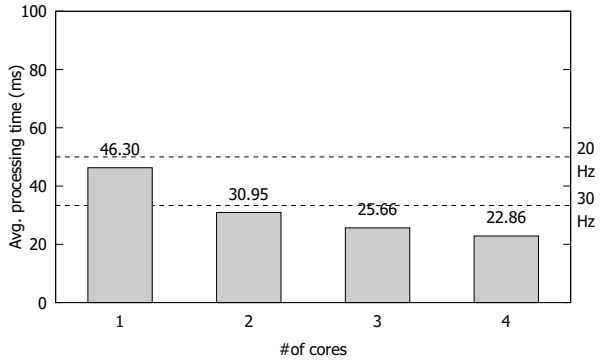


Fig. 7: Average control loop execution time vs. #of CPU cores.

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 same experiment described in the previous subsection with varying numbers of CPU cores—from one to four.

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 mere 2.80 ms, or 12%. In short, we find that the scalability of DeepPicar's deep neural network is not ideal on the platform.

As noted in [21], DNN 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 [21], which generally does not scale very well.

On the other hand, the poor 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 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 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 DNN Models

In this experiment, we launch multiple instances of DeepPicar's DNN 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. These instances are not identical copies of the same model, but are instead copies of different models. This is done to ensure that no L2 cache memory is shared between the models as they are running.

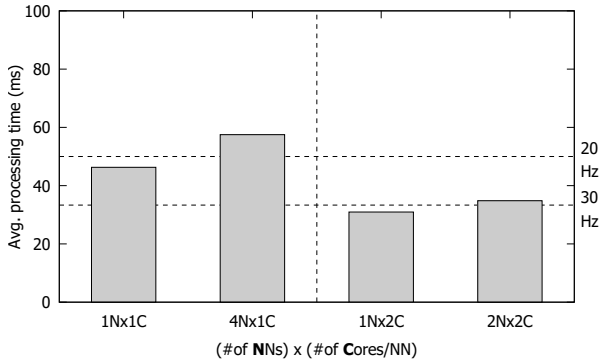


Fig. 8: Timing impact of co-scheduling multiple DNNs. 1Nx1C: one DNN model using one core; 4Nx1C: four DNN models each using one core; 1Nx2C: one DNN model using two cores; 2Nx2C: two DNN models each using two cores.

Figure 8 shows the results. In the figure, the X-axis shows the system configuration: #of DNN models x #of CPU cores/DNN. For example, '4Nx1C' means running four DNN models each of which is assigned to run on one CPU core, whereas '2Nx2C' means running two DNN 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 DNN models. Compared to executing a single DNN model on one CPU core (1Nx1C), when four DNN models are co-scheduled (4Nx1C), each model suffers an average inference time increase of approximately 11 ms, ∼24%. On the other hand, when two DNN 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 and are caused by contention in the shared hardware resources, such as the shared L2 cache and/or the DRAM controller.

E. Effect of Co-scheduling Memory Performance Hogs

In this experiment, we investigate the impact of contended DRAM requests to the DNN inference timing of DeepPicar. For the experiment, we use a synthetic memory intensive benchmark from the IsolBench suite [28]. We run a single DNN model on one core, and co-schedule an increasing number of the memory intensive synthetic benchmarks [3], on the remaining idle cores.

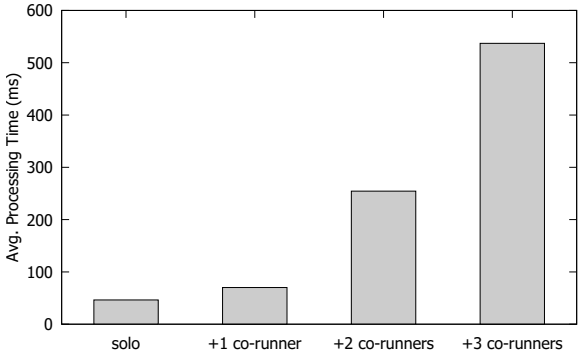


Fig. 9: Average processing time vs. the number of memory memory intensive co-runners introduced.

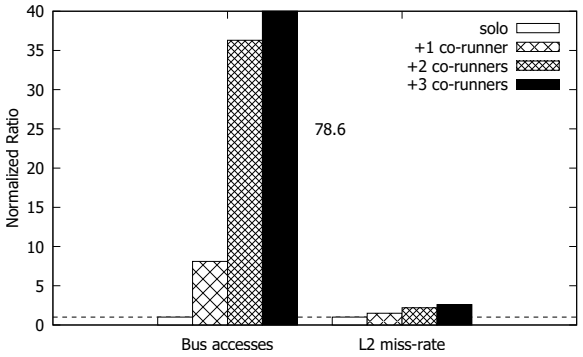


Fig. 10: Effect of memory performance hogs on the shared resources. The DNN model uses Core 0 and memory-hog co-runners use the rest of the cores.

Figure 9 shows the execution time of the DNN model and Figure 10 shows both the normalized total memory bus accesses (first group) and L2 miss rates (second group). In both figures, the DNN model is running on Core 0 as a function of the number of co-scheduled memory intensive synthetic benchmarks. First, as we increase the number of co-runners, the DNN model's execution times are increased exponentially—by up to 11.6X—even though the DNN model is running on a dedicated core (Core 0). On the other hand, the DNN model's L2 cache-miss rates do not increase as much. In other words, the DNN model's execution increase is not fully explained by increased contention in the L2 cache space. Instead, the exponential increase in the total number of bus accesses, due to the memory-hog co-runners, appears to be more closely correlated with the DNN model's execution time increase. This means the high memory bandwidth pressure from the co-scheduled memory intensive benchmarks increased the DNN model's memory access latency.

[3]We use the *Bandwidth* benchmark in the IsolBench suite, with the following command line parameters: `$ bandwdith -a write -m 16384 -t 1000`

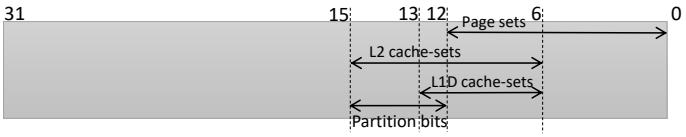


Fig. 11: A memory mapping of the cache sets for the Broadcom BCM2837 processor used in the Raspberry Pi 3 Model B.

This observation suggests that shared cache partitioning techniques [8], [16] may not be effective isolation solutions for DeepPicar's DNN processing workload, as cache partitioning does not provide memory performance guarantees.

F. Effect of Cache Partitioning

In order to validate the effectiveness of cache partitioning, we patch the kernel with PALLOC [29], which is a kernel-level color-aware page allocator, supporting cache partitioning. Figure 11 shows the physical address mapping of Raspberry Pi3's BCM2837 processor. We use the bit 12, 13, 14 for coloring, which results in 8 page colors.

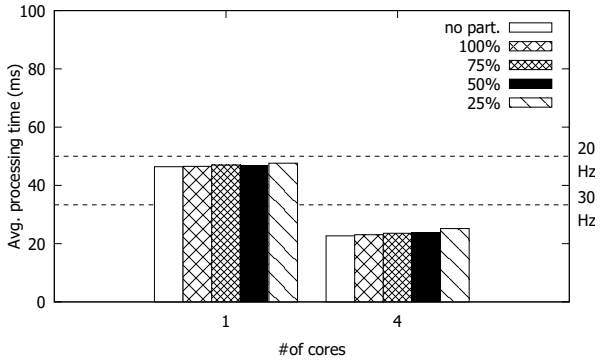


Fig. 12: Average control loop execution time vs. L2 cache partition size.

In the first experiment, we investigate the cache space sensitivity of the DeepPicar's DNN-based control loop. Using PALLOC, we create 4 different cgroups which are configured to use 2, 4, 6, and 8 colors (25%, 50%, 75%, and 100% of the L2 cache space, respectively). We then execute the control loop on each cache partition (cgroup) and measure the average processing time.

If performance improves as a result of partitioning the shared cache then we know that the DNN, to some extent, relies on shared cache performance in addition to shared DRAM performance. On the other hand, if there is no improvement in performance, then it can be observed that the shared memory is more critical to the performance of the DNN. The results can be seen in 12. As expected, the amount of L2 cache available to the model does not result in any significant changes in performance.

We also co-schedule multiple models to see how cache partitioning affects interference between them. Each model uses an equal number of colors, and are thus given equal amounts of L2 cache space, and there is no overlap in the colors used, so no L2 cache is shared between models. As can be seen in Figure 13, the performance of the models remains the same as when no cache partitioning was employed. Based of these results, we find that contention for the L2 cache is not the main source of interference between multiple DNN models.

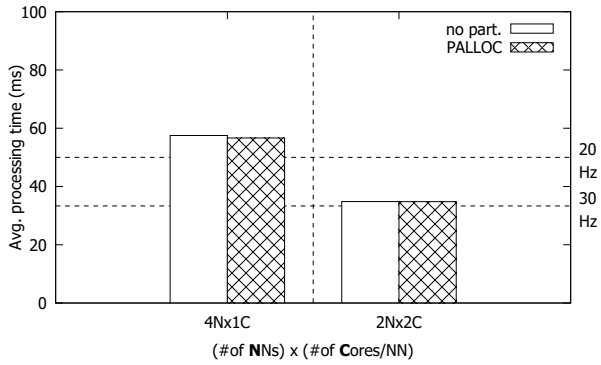


Fig. 13: Timing impact of co-scheduling multiple DNNs when cache partitioning is enabled.

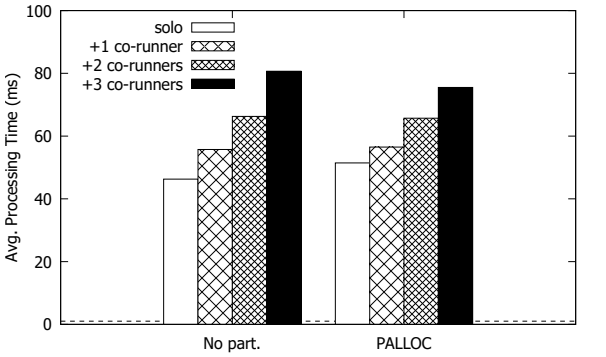


Fig. 14: Timing impact of co-scheduling memory intensive co-runners when cache partitioning is enabled.

Finally, we run a single model alongside memory intensive co-runners again, but ensure that each task is given equal amounts of L2 cache space. For this, we assign two colors to each task, regardless of the number of tasks being scheduled. As a result, the entire cache is not utilized in experiments with less than four tasks. The results can be seen in Figure 14. When the model runs by itself, performance is slightly worse due to less cache space being available (this was also seen in Figure 12). With one or two co-runners, the DNN performance doesnt't change relative to when the cache wasn't partitioned. When three co-runners are present, DNN performance becomes slightly better, with a improvement of ˜6.5%. However, as the performance improvement is minimal, we do not believe that cache partitioning is effective for protecting the model from interference from other memory intensive processes.

By partitioning the shared L2 cache, we find that no notica-

ble improvements are gained and that performance remains consistent, or slightly decreases, depending on the circumstances. In all experiements, the DNN shows no sensitivity to the shared L2 cache. As a result, we conclude that cache partitioning is not an effective isolation mechanism, and that the performance of the shared DRAM controller is of greater importance to the real-time efficiacy of the DeepPicar.

G. Effect of Memory Bandwidth Throttling

To determine the how sensitive the model is to the shared DRAM memory, we use MemGuard, a memory bandwidth reservation system that can limit the amount of bandwidth each core receives in a period. For all tests run using MemGuard, we use a period of one second.

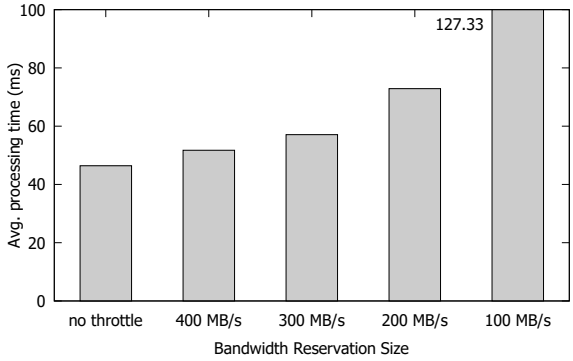


Fig. 15: Average control loop execution time vs. memory bandwidth reservation size.

We first run a single model on one core, Core 0, while varying the core's bandwidth reservation size, from 400 MB/s down to 100 MB/s. The results can be seen in Figure 15. When less memory bandwidth is available to the core the model's performance noticably decreases, and it performs the best when no memory bandwidth throttling is implemented. Compared to a reservation size of 100 MB/s, disabling memory bandwidth throttling results in ˜3x performance improvement. If the model was memory insensitive, then the amount of bandwidth avaiable to it would have little effect on its performance. However, since that is not the case, we conclude that, to a notable extent, the neural network model is dependent on shared DRAM memory performance.

We also test the effects of memory bandwidth throttling in the case of multiple models running concurrently on the Pi 3 by rerunning the 4Nx1C experiment. We use the same memory bandwidth reservation sizes from the previous experiment. The results can be seen in Figure 16. Once again, the performance of the models are affected by the amount of memory bandwidth that is available to the Pi 3's cores during each period, meaning that they are all memory dependent.

Finally, we test the performance of the model in the presence of memory intensive read co-runners in order to determine the effects of limiting the memory bandwidth of the co-runners. For each of the experiments, we give the model's core, Core 0, a constant 1000 MB/s memory bandwidth

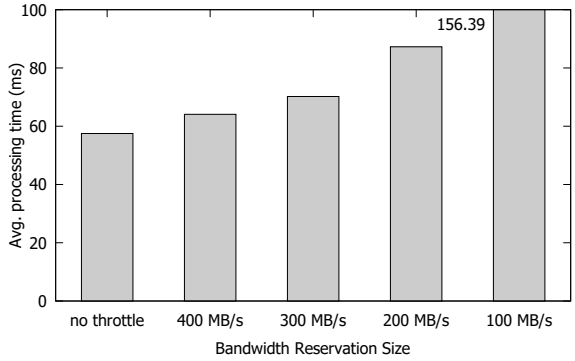


Fig. 16: Timing impact of co-scheduling multiple DNNs when memory bandwidth throttling is enabled.

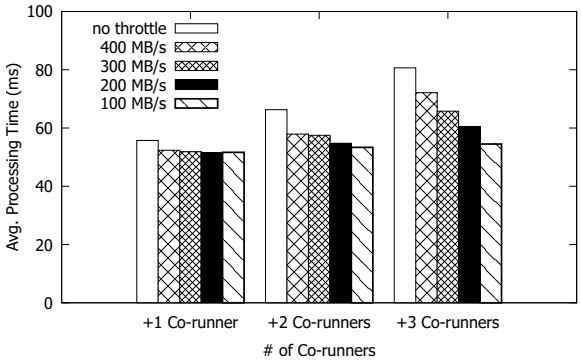


Fig. 17: Timing impact of co-scheduling memory intensive read co-runners when memory bandwidth throttling is enabled.

reservation, and then vary the amount of bandwidth the co-runners' cores receive following the same scheme used in the previous experiments. The results can be see in Figure 17. When less memory bandwidth is given to the read co-runners, the performance of the model does improve. In the case of one co-runner, the performance of the model remains constant even as the reservation sizes are decreased, however, it still improves compared to when throttling wasn't enabled. With two and three co-runners, performance continually improves as less bandwidth is given to the benchmarks. This is especially true with three co-runners as the improvement is linear, which indicates that there is a direct correlation between the co-runner bandwidth size and model performance. In all cases, model performance improved, to some extent, through the use of memory bandwidth throttling on the co-runners. Since this would not be the case if the model was memory insensitive, we find that the model is memory dependent in the presence of co-runners.

Based on all of the memory bandwidth throttling experiments performed, it is evident that the model is dependent on the shared DRAM memory, to a noticable extent. When a single or multiple models were run, their performance(s) decreased as less memory bandwidth was made available to them. Furthermore, when the bandwidths of memory intensive read co-runners were limited, the performance of the model

increased. As such, we conclude that the model is dependent on the performance of the shared DRAM memory.

H. Summary of the Findings

So far, we have evaluated DeepPicar's real-time characteristics from the perspective of end-to-end deep learning based real-time control, and made several observations.

First, we find that DeepPicar's computing platform, the Raspberry Pi 3 Model B, offers computing capacity to perform real-time control of the RC car at 30 Hz frequency (or $33.\overline{33}$ ms per control loop). Given the complexity of the DNN used, we were pleasantly surprised by this finding. The time breakdown shows that the DNN inferencing operation, performed by the Tensorflow library, dominates the execution time, which is expected.

Second, we find that scalability of Tensorflow's DNN implementation is limited. We find that using all four cores can double performance of the DNN, but compared to using three cores, performance improvement is relatively minimal.

Third, we find that consolidating multiple DNN models—on different CPU cores—is feasible as we find: (1) DNN performance using a single core is not much worse than using multiple cores; (2) multiple DNN models running simultaneously do not cause severe interference with each other.

Fourth, we find that consolidating memory (DRAM) performance intensive applications could jeopadize DNN performance, because DNN performance appears to be very sensitive to memory performance; we observe up to 11.6X slowdown in DNN performance by co-scheduling synthetic memory bandwidth intensive applications on idle cores.

Fifth, we find that partitioning the shared L2 cache provides practically no benefits to the real-time performance of the DNN. To the contrary, performance is constant regardless of the amount of cache space used. Furthermore, we find that performance remains consistent when multiple models are run simultaneously, and when memory intensive read co-runners are introduced. As such, we conclude that the shared L2 cache is not essential to the DNN's real-time performance.

Lastly, we find that memory bandwidth throttling results in significant changes to model performance. When less bandwidth is given to models, whether they're running alone or concurrently, we find that their performance decreases noticably. When memory intensive co-runners are present, we find that limiting the available bandwidth of the co-runners actually results in improved model performance. As such, we conclude that the shared DRAM memory is essential to the DNN's real-time performance.

V. Embedded Computing Platform Comparison

In this section, we compare three computing platforms—the Raspberry Pi 3, the Intel UP [4] and the NVIDIA Jetson TX2 [5]—from the point of view of supporting end-to-end deep learning based autonomous vehicles. Table III shows architectural features of the three platforms [6].

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.

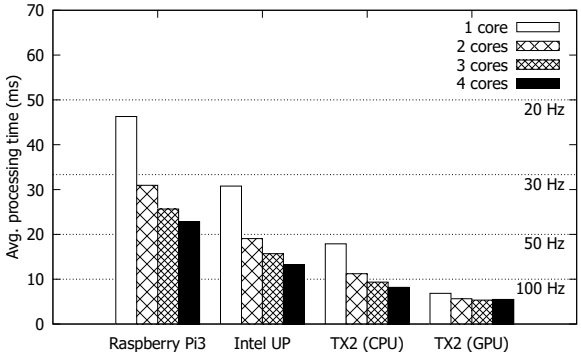


Fig. 18: Average control loop execution time.

Figure 18 shows the average control loop completion timing of the four system configurations we tested as a function of the number of CPU cores used. First, both the Intel UP and Jetson TX2 exhibit superior performance when compared with the Raspberry Pi 3. When all four CPU cores are used, the Intel UP is 1.33X faster than the Pi 3, while the TX2 (CPU) and TX2 (GPU) are 2.79X and 4.16X times faster, respectively, than the Pi 3. As a result, they are all able to satisfy the $33.\overline{33}$ 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 TX2 (GPU) performance does not change much, as most of the neural network computation is done by the GPU.

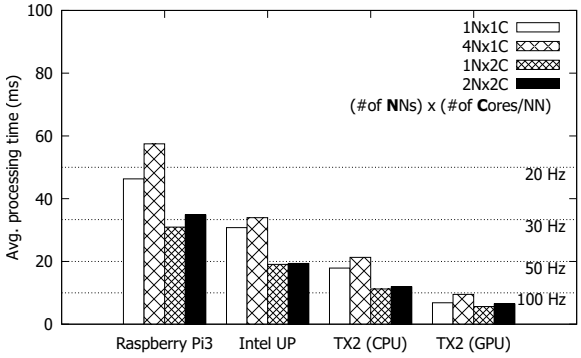


Fig. 19: Average control loop execution time when multiple DNN models are co-scheduled.

[4] http://www.up-board.org/up/

[5] http://www.nvidia.com/object/embedded-systems-dev-kits-modules.html

[6] 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@2GHz/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	2GB DDR3L	8GB LPDDR4
Peak Memory Bandwidth	8.5 GB/s	12.8 GB/s	59.7 GB/s
Measured Bandwidth	2127.94 MB/s	3951.94 MB/s	4447.90 MB/s
Cost ($)	35	100	600

TABLE III: Compared embedded computing platforms

The Intel UP board and Jetson TX2 also perform much better when multiple DNN models are co-scheduled. Figure 19 shows the results of the multi-model co-scheduling experiment. Once again, they can comfortably satisfy 30 Hz real-time performance for all of the co-scheduled DNN control loops, and in the case of the TX2 (GPU), 100 Hz real-time control is still feasible. Given that the GPU must be shared among the co-scheduled DNN models, the results suggest that the TX2's GPU has sufficient capacity to accomodate multiple instances of the DNN models we tested.

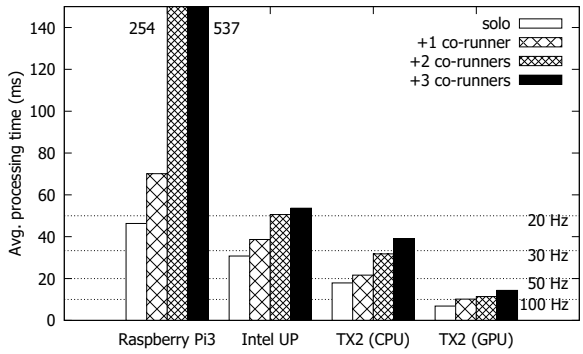


Fig. 20: Average control loop execution time in the presence of an increasing number of memory intensive applications on idle CPU cores.

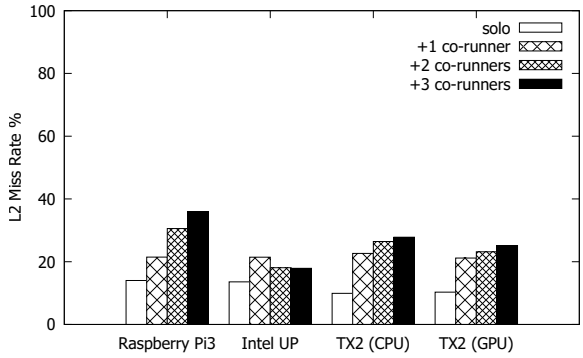


Fig. 21: L2 miss rates in the presence of an increasing number of memory intensive applications on idle CPU cores.

Finally, we compare the effect of co-scheduling memory bandwidth intensive synthetic benchmarks on the DNN control loop timing. Figure 20 shows the results. As discussed in Section IV-E, we observed dramatic execution time increases, up to 11.6X, in Raspberry Pi 3 as we increased the number of co-scheduled tasks. We also observe increased control loop execution timing in the Intel Up and Jetson TX2, but the degree of the increase is not as dramatic as the Pi 3. Compared to their respective solo timings (i.e., the model runs on a single core in isolation), Intel UP suffers up to 2.3X execution time increase; TX2 (CPU) and TX2 (GPU) suffer up to 2.2X and 2.1X increases, respectively. This is somewhat surprising because the Raspberry Pi 3's cores are in-order architecture based while the cores in the Intel Up and NVIDIA TX2 are out-of-order architecture based, and that the memory intensive tasks on out-of-order cores can generate more memory traffic. We believe that this is because the memory subsystems in the Intel UP and TX2 platforms provide higher performance than the memory subsystem of the Pi 3 as suggested from the measured memory bandwidth results in Table III ('Measured Bandwidth').

Another interesting observation is that the TX2 (GPU) also suffers considerable execution time increase (2.1X) despite the fact that the co-scheduled synthetic tasks do not utilize the GPU. In other words, the DNN model has dedicated access to the GPU. This is, however, a known characteristic of integrated CPU-GPU architecture based platforms where both CPU and GPU share the same memory subsystem [2]. As a result, the TX2 (GPU) fails to meet the 10ms deadline for 100 Hz control that would have been feasible if there was no contention between the CPU cores and the GPU.

Figure 21 shows the L2 miss rates of DNN control loop. The key takeaway of this result is that the L2 miss rate is a poor indicator to the execution time of the control loop. As shown already, the Raspberry Pi 3's increased L2 misses do not fully explain increases in processing times. In case of the Intel UP, as the number of interfering Bandwidth benchmark instances increases, the L2 miss rates drop slightly. In the case of the TX2 (CPU), from solo to one co-runner, L2 miss rates almost double, but then the execution time increase is a little less than 10%. However, when there are co-runners, even though the L2 miss rates are largely unchanged, the DNN processing time increases significantly. The TX2 (GPU) also shows similar behavior. All of these results essentially point to the fact that DNN inferencing workload is largely memory performance sensitive but not L2 cache space sensitive.

In summary, we find that todays embedded computing

platforms, even as inexpensive as a Raspberry Pi 3, are powerful enough to support vision and end-to-end deep learning based real-time control applications. Furthermore, availability of CPU cores and GPU on these platforms allow consolidating mutiple deep neural network based AI 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

In this paper, we demonstrated that a low-cost embedded platform such as the Raspberry Pi 3 can successfully run a CNN-based autonomous driving control system in real-time. However, it should be noted that the CNN we used here 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 [10] have between 3M to 54M parameters. 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 approrach is to reduce the required computational complexity itself. When a DNN is deployed in those implementations with limited resources, such as memory and power, the floating-point operations involved in the large matrix multiplications are a burdensome task. Many recent advances in network compression have shown promising results in reducing such computational cost during the feedforward process. The fundamental assumption in those techniques is that the DNNs are redundant in their structure and representation. For example, network pruning can thin out the network and provides a more condensed topology [9]. Another common compression method is to reduce the quantization level of the network parameters, so that arithmetic defined with the 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 [3], [7], [11], [12], [15], [24], [26]. 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 [24], it was shown that the binarized convolution can substitute the expensive convolutional feedforward operations in a regular Convolutional Neural Network (CNN), 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 GPU [11]. Moreover, FPGA implementations showed that the XNOR operation is 200 times cheaper than floating-point multiplications with single precision [3], [7]. XNOR-POP is another hardware implementation that reduced the energy consumption of a CNN by 98.7% [13].

These research efforts are expected to make complex DNNs more accessible for 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 instead of Raspberry Pi3.

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 almost identical to that of NVIDIA's real self-driving car.

Despite the complexity of the neural network, DeepPicar uses a low-cost Raspberry Pi 3 quad-core computer as its sole computing resource. We systematically analyzed the real-time characteristics of the Pi 3 platform in the context of deep-learning based real-time control applications, with a special emphasis on real-time deep neural network inferencing. We also evaluated other, more powerful, embedded computing platforms to better understand achievable real-time performance of DeepPicar's deep-learning based control system and the impact of computing hardware architectures. We find all tested embedded platforms, including the Pi 3, are capable of supporting deep-learning based real-time control, from 20 Hz up to 100 Hz, depending on the platform and its system configuration. However, shared resource contention remains an important issue that must be considered in applying deep-learning models on shared memory based embedded computing platforms.

As future work, we plan to apply shared DRAM resource management techniques [29], [30] on the DeepPicar platform and evaluate their impact on the real-time performance of the system. We also plan to improve the prediction accuracy by feeding more data and upgrading the RC car hardware platform to enable more precise steering angle control.

References

[1] F1/10 autonomous racing competition. http://f1tenth.org.

[2] W. Ali and H. Yun. Work-in-progress: Protecting real-time GPU applications on integrated CPU-GPU SoC platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017.

[3] 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.

[4] M. Bojarski et al. End-to-End Learning for Self-Driving Cars. *arXiv:1604*, 2016.

[5] L. Fridman. End-to-End Learning from Tesla Autopilot Driving Data. https://github.com/lexfridman/deeptesla.

[6] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *13th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2010.
[7] G. Govindu, L. Zhuo, S. Choi, and V. Prasanna. Analysis of high-performance floating-point arithmetic on FPGAs. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, 2004.
[8] 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.
[9] 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.
[10] 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 *IEEE CVPR*, 2017.
[11] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized neural networks. In *Advances in neural information processing systems*, 2016.
[12] K. Hwang and W. Sung. Fixed-point feedforward deep neural network design using weights +1, 0, and -1. In *Signal Processing Systems (SiPS), 2014 IEEE Workshop on*, 2014.
[13] 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 IEEE/ACM International Symposium on*, 2017.
[14] N. P. Jouppi et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.
[15] M. Kim and P. Smaragdis. Bitwise neural networks. *arXiv preprint arXiv:1601.06071*, 2016.
[16] 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, IEEE*, 2016.
[17] 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.
[18] S. Levine. Deep Reinforcement Learning. http://rll.berkeley.edu/deeprlcourse/f17docs/lecture_1_introduction.pdf, 2017.
[19] S. Levine, C. Finn, T. Darrell, and P. Abbeel. End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*, 2016.
[20] NVIDIA. The AI Car Computer for Autonomous Driving. http://www.nvidia.com/object/drive-px.html.
[21] NVIDIA. GPU-Based Deep Learning Inference : A Performance and Power Analysis. Technical Report November, 2015.
[22] 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 *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017.
[23] D. a. Pomerleau. Alvinn: An autonomous land vehicle in a neural network. In *Advances in Neural Information Processing Systems 1*, 1989.
[24] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. XNOR-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, 2016.
[25] 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.
[26] 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*, 2014.
[27] Thomas Burger. How Fast Is Realtime? Human Perception and Technology — PubNub, 2015.
[28] 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.
[29] 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.
[30] 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 4a 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 a Xavier initializer [6], which is known to provide better initial values than the random weight assignment method. 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. When a model was trained with the aformentioned data, the training loss was 0.0188 and the validation loss was 0.0132. The change of the loss value over the course of model training can be seen in Figure 22.

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 huge discrepency between the training loss and the validation loss as the former was 0.009, while the latter was 0.0869—a 10X difference—indicating that the model suffers from an overfitting problem.

We continue to investigate ways to achieve better prediction accuracy in training the network, as well as improving the performance of the RC car platform, especially related to precise steering angle control.

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 and high real-time performance.

In all our experiments on the Raspberry Pi 3, the CPU operated at a preferred clock speed of 1.2 GHz. However, without care, it is possible for the CPU to operate at a lower frequency.

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 85C). DNN

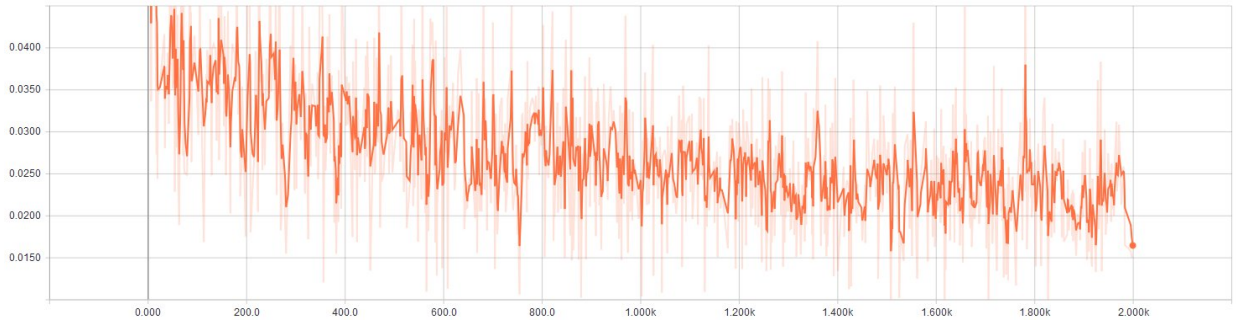


Fig. 22: Change in loss value throughout training.

model operations are computationally intensive, thus it is possible for the temperature of the CPU to rise quickly. This can be especially problematic in situations where multiple DNN models are running 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. From our experiences, the Pi 3 frequency throttling also kicks in when the power source can not provide the required minimum of 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, and instead, fluctuated between operating at 600 MHz and 1.2 GHz. 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.