Pipelined Data-Parallel CPU/GPU Scheduling for Multi-DNN Real-Time Inference

Yecheng Xiang and Hyoseung Kim
University of California, Riverside
yxian013@ucr.edu, hyoseung@ucr.edu

***Abstract*—Deep neural networks (DNNs) have been showing significant success in various applications, such as autonomous driving, mobile devices, and Internet of Things. Although much research has been conducted to optimize the structure of DNNs, limited attention has been given to their timely execution, specifically on the scheduling of real-time inference requests to various DNN models. For instance, existing DNN frameworks, such as Caffe, TensorFlow and Torch, only provide a single-level priority, one-DNN-per-process execution model and sequential inference interfaces. They can be particularly problematic when used in edge computing and in-vehicle intelligence systems for multiple DNNs, as response time may become unpredictably long in the worst case while leaving system resources underutilized. This paper presents DART, a DNN scheduling framework that offers deterministic response time to real-time tasks and increased throughput to best-effort tasks. DART employs a pipeline-based scheduling architecture with data parallelism, where heterogeneous CPUs and GPUs are arranged into nodes with different parallelism levels. DART also includes pipeline stage design and node configuration schemes, admission control, execution time profiling, and runtime enforcement techniques. We evaluated DART on Intel x86 Xeon and Nvidia ARM platforms with GPUs. Experimental results indicate that DART significantly outperforms the existing approaches, by up to 98.5% shorter worst-case response time for real-time tasks while simultaneously achieving up to 17.9% higher throughput for best-effort tasks.**

I. Introduction

Deep neural networks (DNNs) have shown great potential for the use in many Internet of Things (IoT) and Cyber-Physical Systems (CPS), such as autonomous driving [9, 32], smart robotics [11], and precision agriculture [45]. Such systems often have multiple sensing tasks that capture raw sensor data and issue DNN inference requests (jobs) to various pre-trained DNN models in order to obtain a high-level representation of the environment. In the meanwhile, due to size, weight, power and cost constraints, running multiple DNN applications concurrently on a single hardware platform is gaining much interest. To ensure the usefulness and correctness of the obtained information, both timely and accurate response of these DNN jobs needs to be provided. A deterministic tail latency bound, i.e., the guaranteed worst-case response time, is particularly important for safety-critical applications as it is closely related to the performance, correctness, and even safety of individual components as well as the entire system.

The current state-of-art DNN frameworks, such as Caffe [1, 22], TensorFlow [4], and Torch [3], handle inference jobs in a sequential manner with a single process per DNN model. This approach is useful to achieve high throughput in handling batch jobs. However, they lack consideration for periodic or sporadic jobs with different timing requirements, which are crucial in many IoT and CPS applications. Moreover, while multi-core CPUs and hardware accelerators like GPUs have been used to improve the average-case response time of inference jobs, they are often underutilized when multiple DNN models are concurrently requested and their benefits in the worst case are not as significant as in the average case. Since recent embedded hardware platforms are increasingly equipped with heterogeneous CPU and GPU cores, it is important to enable scalable improvement in both the worst-case response time and the overall throughput of DNN tasks by utilizing all computing resources available in the system.

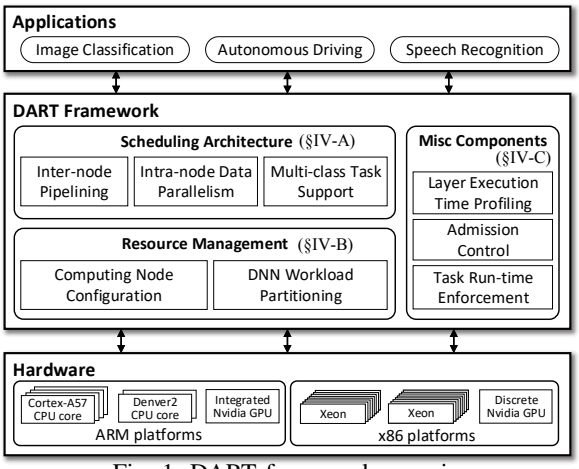


Fig. 1: DART framework overview

In this paper, we propose DART, a DNN scheduling framework with Analyzable Real-Time guarantee. The primary goals of DART are (a) to ensure the timing constraints of real-time DNN inference tasks while minimizing their response time, and (b) to maximize the throughput of best-effort DNN tasks by improving the utilization of heterogeneous CPU and GPU cores. To achieve those goals, we have designed DART as illustrated in Fig. 1. DART provides a unified runtime environment for the concurrent execution of tasks accessing diverse DNN models and addresses the limitations of the state-of-the-art. DART currently supports CPU and GPU cores on ARM and Intel platforms, e.g., Cortex-A57, Denver, and Xeon CPUs and Nvidia integrated and discrete GPUs, but the design of DART can also be applied to other processor architectures. DART consists of (i) a scheduling architecture integrating *inter-node pipelining* and *intra-node data parallelism* with multi-class task support, (ii) resource management with node configuration and DNN partitioning, and (iii) other miscellaneous components including layer execution time profiling, admission control, and task run-time enforcement. We will present the details of these components in Sec. IV.

2576-3172/19/$31.00 ©2019 IEEE
DOI 10.1109/RTSS46320.2019.00042

We have implemented DART on x86 and ARM platforms equipped with Nvidia GPUs. To realize our design, we have also developed OpenBLAS-rt, an extension of the standard OpenBLAS library with real-time priority and CPU affinity support. Experiments are conducted using this implementation and the run-time overhead is found to be acceptably small.

It is worth noting that our work does *not* claim to overcome timing jitters originated from underlying OS or drivers. Instead, the notion of guarantee in this paper is the bound on delays possibly caused by scheduling policies and resource arbitration rules when utilizing heterogeneous processors.

Contributions. The overarching contribution of this paper is the runtime framework design which brings algorithmic improvements into the real-time DNN scheduling problem. Our work extends existing knowledge in several ways: adding new system abstractions that do not exist in the current DNN frameworks, developing resource allocation schemes for the proposed abstractions, and presenting analytical extensions to bridge the gap between DNN execution and schedulability analysis. There may be other ways to utilize heterogeneous resources better than ours while ensuring analyzable real-time guarantees, but to the best of our knowledge, this paper is the first work to do so. Below are the detailed contributions:

- We introduce new abstractions to deal with the different resource requirements of individual layers of DNNs and to facilitate the co-utilization of CPU and GPU in inference job execution. A subset of DNN layers is grouped into task-level *stages* and allocated to node-level *workers* which form the foundations to achieve pipeline and data parallelism.
- We give a systematic formulation of the real-time DNN scheduling problem as a distributed acyclic scheduling problem. Our analysis captures DNN job execution over the proposed abstractions of stages, nodes, and workers, and takes into account system overheads including inter-node communication, GPU preemption, and data copy time.
- We develop resource management algorithms that (i) create the pipeline stages of each task in a way to balance the contention across a given set of processors, and (ii) allocate processor resources to meet timing constraints and to minimize task response time.
- From our experimental results, we found that DART significantly outperformed the existing representative methods, by up to 98.5% shorter worst-case latency for real-time tasks while simultaneously achieving up to 17.9% higher throughput for best-effort tasks. DART also dominated the existing methods in DNN taskset schedulability.

II. Background and Motivation

A. Deep Neural Networks

A DNN model can be viewed as a data-flow graph, where artificial neurons and their connections are represented in layers. Fig. 2 illustrates a general DNN model composed of one input layer taking raw data, one output layer extracting inference results, and multiple hidden layers in between responsible for capturing the features of data. From a high-level view, working with a DNN has a two-step process. First, training a DNN: it learns weight parameters from provided input training data and their associated output labels. Next, running inference from the trained DNN: it uses its trained parameters to classify, recognize, and process unknown inputs. While training typically takes enormous time even in cloud servers, inference from a trained model is deemed feasible and increasingly required in embedded systems. We thus focus on the timely execution of DNN inference jobs in this work.

The execution pattern of a DNN inference job can be viewed as *forward propagation*: input data is processed through an input layer, hidden layers, and an output layer in a layer-wise manner. Note that the execution of layers during a single inference job must be done in order. For instance, the first hidden layer can only be executed after the completion of the input layer, otherwise it would propagate erroneous results to its subsequent layers.

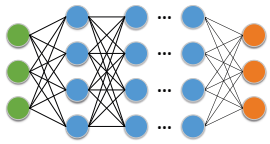


Fig. 2: An example DNN model with one input layer, several hidden layers, and one output layer

B. The Status Quo and Challenges

Modern deep learning frameworks, e.g., Torch, TensorFlow, and Caffe, have become popular in the research community and industry due to their convenience and portability. They handle inference jobs in a *sequential* manner using a *separate process per DNN model*. This means, if a system uses m distinct DNN models, at least m instances of the framework need to be created as separate processes. Also, they do not provide prioritization or real-time support for inference tasks. When multiple inference tasks with different timing constraints are given to such a framework, real-time tasks may experience nondeterministic delay and thus the schedulability of tasks is hard to be analyzed. This is a major limiting factor for the use of existing frameworks in safety-critical applications where a deadline miss may cause a huge loss in quality or stability.

Example: Self-driving Car. Let us consider a self-driving car using computer vision algorithms, e.g., [9, 32], where DNNs are used to detect physical objects and lane markings by analyzing images produced by multiple cameras mounted on the front, side, and rear of the car (Fig. 3). We refer to each task generating a DNN inference job as a *sensor client* and a DNN framework handing such a job as an *in-vehicle server*. Once a client collects an image from a camera, it sends an inference job to the server and waits for the completion of the job. Timely response is a necessity because each job has a deadline and its result is used as input for other components of the car. Given that the car moves forward most time, high priority can be assigned to a task for the front camera, medium for the side ones, and low for the rear ones. There also exist other DNN tasks running

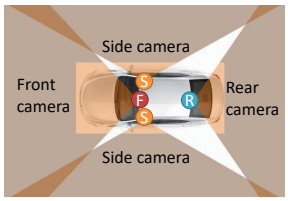


Fig. 3: A self-driving car with front, side, rear-view cameras and field of view

with no real-time requirements, i.e., best-effort tasks, such as for monitoring human driver's engagement [14].

Suppose that the low-priority task for the rear-view camera or best-effort tasks generate inference jobs in a burst manner, and the high-priority task for the front camera issues an inference job at the same time. Without real-time scheduling support in the server's framework, the response time of the high-priority job can become unpredictably long and even miss its deadline This can lead to a failure in the timely detection of objects from front-road images, thereby jeopardizing the driving context of the car. As more cameras are being used for wider coverage and other types of DNN tasks, e.g., motion planner [39], are introduced, such a timing problem will become more significant in future self-driving systems.

There are also other issues with the CPU and GPU scheduling of the existing frameworks, which lead to long response time and resource underutilization. We will discuss the details of such issues in Sec. IV-A with a comparison to our work.

III. System Model

This work considers a heterogeneous multi-core system where the main memory is shared among all CPUs. The system $\mathcal{R}$ is equipped with one or more types of CPU cores and GPU devices, i.e., $\mathcal{R} = \{c_1, c_2, ..., c_r\}$ where c_i is either a CPU core or GPU. In such a system, our framework constructs a set of *nodes* for DNN task execution. Each *node* p_k is a disjoint partition of $\mathcal{R}$. Hence, $\mathcal{R} = \bigcup p_k$, and if $i \neq j$, $p_i \cap p_j = \emptyset$. Each node is either a CPU or GPU node. If p_k is a CPU node, it has only one type of CPU cores, following the homogeneity assumption commonly made by real-time parallel scheduling work [7, 26, 30].[1] If p_k is a GPU node, it has one GPU as well as one CPU core to support GPU-related operations, which will be discussed in Sec. IV-A2. We let P denote the entire set of nodes, i.e., $P = \bigcup\{p_k\} = \{p_1, p_2, ..., p_k\}$. Given all the CPUs and GPUs available in the system $\mathcal{R}$, there can be multiple possible ways to group them into nodes. We define each of such ways as a node configuration P.

DNN inference tasks are characterized by the sporadic task model [35] which is widely used and accepted in the real-time systems community and industry. In this model, a task is a sequence of recurring jobs with the minimum inter-arrival time between any two consecutive jobs. We assume each task uses one DNN model and makes one inference request per job. If an application uses multiple DNN models, it can be represented as multiple tasks in our system model. A task τ_i is characterized by:

$$\tau_i := (C_i, T_i, D_i, L_i)$$

- C_i: the worst-case execution time (WCET) of a single job when it runs in isolation (i.e., no extrinsic temporal interference imposed by other tasks)
- T_i: the minimum inter-arrival time
- D_i: the relative deadline of each job
- L_i: the number of layers of the DNN model used by τ_i

This work considers two types of DNN tasks: real-time (RT) and best-effort (BE). If τ_i is an RT task, it has a constrained deadline, i.e., $D_i \leq T_i$, and the deadline should be strictly met all the time once the task is admitted to the system. If τ_i is a BE task, its deadline may be missed by some jobs. When there is no specific deadline required for a BE task τ_i, the deadline D_i can be set to ∞.

The execution of a job of τ_i can be decomposed into a sequence of L_i layers of the DNN model used by it. Due to data dependency, the latter layer can be executed only after the completion of the previous one. The execution time of each layer depends on the node where it executes. Hence, we use $\tau_{i,j}$ to denote the j-th layer of τ_i and $C_{i,j}(p_k)$ to represent the execution time of $\tau_{i,j}$ on a node p_k. If p_k is a CPU node consisting of more than one CPU core, BLAS multithreading is used and $C_{i,j}(p_k)$ is determined by the slowest thread. If p_k is a GPU node, $C_{i,j}(p_k)$ is further decomposed as follows:

$$C_{i,j}(p_k) := (G^{hd}_{i,j}(p_k), G^{e}_{i,j}(p_k), G^{m}_{i,j}(p_k), G^{dh}_{i,j}(p_k))$$

- $G^{hd}_{i,j}(p_k)$: the maximum memory copy time from the host to the device before GPU kernel execution
- $G^{e}_{i,j}(p_k)$: the worst-case GPU kernel execution time
- $G^{dh}_{i,j}(p_k)$: the maximum memory copy time from the device to the host after GPU kernel execution
- $G^{m}_{i,j}(p_k)$: the WCET of miscellaneous CPU operations in $\tau_{i,j}$, e.g., kernel launch preparation

This GPU execution model follows the latest real-time GPU work [25, 38], except that we distinguish memory copy time and miscellaneous CPU operation time. $C_{i,j}(p_k)$ is the sum of all these components. Thus, $C_{i,j}(p_k) = G^{hd}_{i,j}(p_k) + G^{e}_{i,j}(p_k) + G^{m}_{i,j}(p_k) + G^{dh}_{i,j}(p_k)$.

The worst-case execution time of a job of a task τ_i is therefore given by $C_i = \sum_{j=1}^{L_i} C_{i,j}(p_{k_{i,j}})$, where L_i is the number of layers of a DNN model used by τ_i and $p_{k_{i,j}}$ is the node assigned for the execution of the j-th layer.

IV. DART Framework

This section presents the detailed design of the DART framework. We first introduce the scheduling architecture with CPU and GPU node scheduling, and then explain the resource management algorithms for pipeline stage design, and node configuration and the other components including execution time profiling, admission control and runtime enforcement.

A. Scheduling Architecture

DART introduces several system abstractions to exploit pipeline and data parallelism over heterogeneous resources. Hence, we begin with defining the key abstractions as follows.

Def. 1. *The stage s of a task τ_i is a subset of consecutive layers of the corresponding DNN model used by the job of τ_i. Hence, the entire set of stages of τ_i includes all the layers of τ_i, i.e., $\cup s = \cup \tau_{i,j}$, and tasks may have a different number and set of stages even if they use the same DNN model.*

Def. 2. *The execution pipeline of a task τ_i is a sequence of stages executed over nodes following their precedence*

[1] One of the challenges is that if workload on one parallel segment is distributed into threads on heterogeneous cores, e.g., fast and slow cores, it is hard to predictably bound and ensure the benefit of fast cores unless assuming that the workload can be perfectly load-balanced.

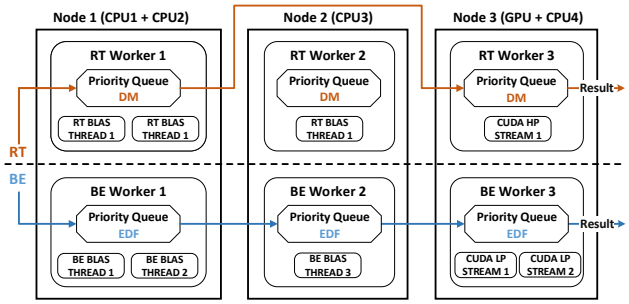


Fig. 4: DART scheduling architecture

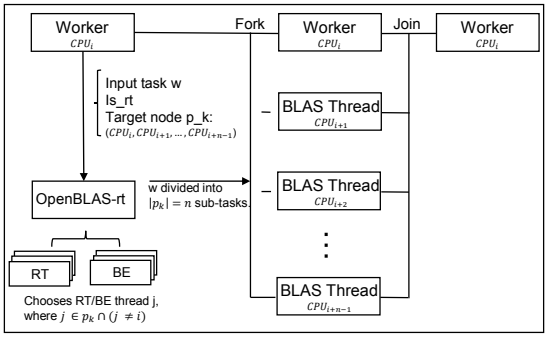


Fig. 5: OpenBLAS-rt workflow

constraints, e.g., the next stage of a task τ_i on a node p_k is eligible to run once the execution of its previous stage completes on the other node p_q.

Def. 3. *A worker is a scheduler unit that executes per-task stages arriving on the corresponding node. At the same time, the worker is a scheduling entity of the underlying OS, i.e., thread, meaning that the OS schedules workers on nodes.*

DART partitions the DNN inference execution of each task into stages. The stages are then assigned to different nodes and executed by the workers of the nodes in a pipeline manner. The main reasoning behind these abstractions is that each layer of a DNN task may have different sensitivity to the parallelization levels provided by heterogeneous computing resources. For instance, within a single DNN model, some layers may gain large performance improvement with more CPU cores and some others may not exhibit noticeable benefits. If a task using such a DNN model is simply allocated to only one type of resources, which is the case of the current DNN frameworks, the system may become underutilized and inefficient. The use of stages gives flexibility in allocating individual layers of a task to different resources for better utilization. The detailed steps on node configuration, stage partitioning, and resource allocation will be presented in the next subsection.

Each CPU and GPU node supports scheduling classes for intra-node task scheduling. DART currently has two classes, real-time (RT) and best-effort (BE), in accordance with our system model, but more classes can be added to achieve a finer classification of tasks. The RT class is strictly prioritized over the BE class. DART correspondingly creates two sets of workers, and assigns Linux real-time priority to RT workers and fair-share (normal) priority to BE workers. Then a pair of RT and BE workers is statically allocated to each node for the execution of RT and BE tasks by their respective workers. Each worker has a priority queue for pending tasks. In RT workers, we use the deadline-monotonic (DM) priority assignment for the queue as it gives deterministic guarantees even under overload conditions. In BE workers, we use the earliest-deadline-first (EDF) policy as it is known to achieve higher utilization than DM. While an RT worker can preempt a BE worker on the same node at any time, task execution within each worker is non-preemptive. The reason for the choice of non-preemptive scheduling within a worker is primarily due to the lack of fine-grained priority-based preemption support in the latest GPUs.[2] Moreover, it helps reduce memory consumption as it does not require each worker to store intermediate computation results of multiple preempted tasks.

Fig. 4 illustrates the example of the DART scheduling architecture. In this example, there are one RT and one BE tasks. The RT task has two stages: the first stage is assigned to Node 1 and the second stage to Node 3; Node 2 is not used by the RT task. On the other hand, the BE task has three stages and each allocated to a different node. Although Nodes 1 and 3 are used by both tasks, the RT task is not delayed by the BE task as RT workers can preempt BE workers. Node 1 has two CPU cores and the RT and BE workers of Node 1 each have two BLAS threads for intra-node data parallelism. Node 2 has only one CPU core. Node 3 is a GPU node and each worker on Node 3 has a set of CUDA streams. We below explain more details of CPU and GPU node scheduling in DART.

1) CPU Node Scheduling: Multi-thread BLAS libraries, such as ATLAS and OpenBLAS, can achieve data parallelism in DNN inference tasks, but they provide only limited control over BLAS thread scheduling. For example, in the latest version of OpenBLAS (v0.3.5), the number of threads and their CPU allocation cannot be configured for individual BLAS operations, but only the total number of threads to be spawned can be set at the time of launching a target process. General multi-threading libraries such as OpenMP cannot be used since they do not support BLAS operations natively and designers have to implement them from scratch.

To address such limitations and achieve intra-node data parallelism in DART, we have developed OpenBLAS-rt, which is an extension of the existing OpenBLAS library. Fig. 5 depicts how workers work with OpenBLAS-rt. OpenBLAS-rt creates two sets of BLAS threads (RT and BE), following the scheduling class design of DART, and assigns Linux real-time priority to RT BLAS threads and normal priority to BE BLAS threads. Also, for incoming BLAS requests, it provides an interface to tell their task class and node information. With OpenBLAS-rt, the number of BLAS threads on a node p_k is determined by the number of CPU cores of p_k, and each thread is strictly mapped to one core to prevent potential overhead caused by thread migration.

As an example, Fig. 6 compares CPU scheduling under three scenarios: (a) launching multiple instances of an existing

[2]GPUs using the Nvidia Pascal architecture, e.g., TX2 and GTX 1080, support only two priority levels for kernel preemption.

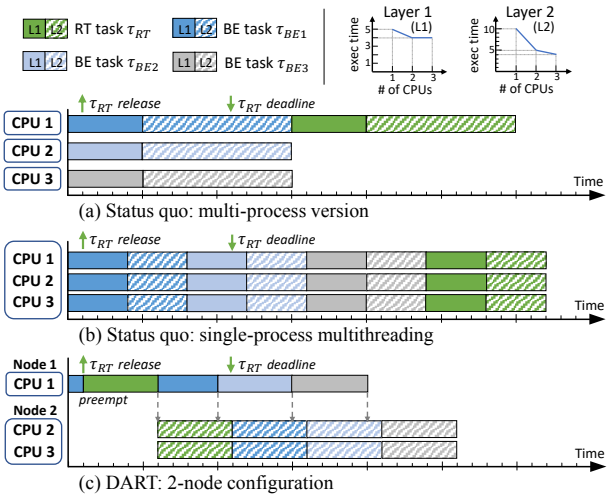


Fig. 6: CPU scheduling in the status quo and DART

DNN framework to utilize all CPUs in the system without a multi-threaded BLAS library, (b) a single-process with an existing multi-threaded BLAS library, and (c) DART with OpenBLAS-rt. In this figure, there are four tasks, one RT and three BE, and all of them use the same DNN model consisting of two layers. The system has three CPUs. Layer 1 does not yield much speed-up with more CPUs but layer 2 does. Note that these reflect typical speed-up patterns of DNN workloads which we will show in the evaluation section. All the three BE tasks arrive at time 0. The RT task arrives at 1 and has an absolute deadline at 11. In both (a) and (b), the RT task misses the deadline as the existing framework is oblivious of the priority of DNN tasks. The total completion time of all tasks in (b) is even greater than in (a), because the existing BLAS library always uses a pre-configured number of threads regardless of the presence of diminishing speed-ups. In contrast, DART meets the deadline by RT worker preemption and achieves shorter total completion time. Although DART uses a 2-node configuration in this example, it subsumes the other configurations used in (a) and (b) as those can be found by the resource management schemes of DART.

2) GPU Node Scheduling: In DART, each GPU node is configured to have one CPU core and one GPU device. The reason for this is because a GPU needs CPU operations, such as for launching a kernel and transmitting data, and any unnecessary delay on the CPU side can cause long idle time to the GPU. The workers of a GPU node makes use of the CUDA stream prioritization and thread-level preemption of Nvidia Pascal and later architectures. Hence, RT and BE workers use CUDA high-priority and low-priority streams, respectively, to execute the GPU kernels of tasks.

The RT worker executes one kernel at a time and determines the execution order of kernels using the priority of their tasks. This approach follows the majority of existing real-time GPU schemes [12, 25, 38] in order not to cause nondeterministic slowdown from concurrent kernel execution [37, 48]. Hence, each RT worker utilizes one high-priority CUDA stream. On the other hand, each BE worker uses multiple low-priority streams in the same CUDA context to maximize the parallelism provided by internal computing units of the GPU. This enables multiple BE kernels to concurrently execute as long as the GPU resources permit, thereby improving overall throughput and GPU utilization. The number of CUDA streams per BE worker is configurable and can be determined by kernel parameters and the compute capability of GPU, e.g., thread block size and maximum resident threads.

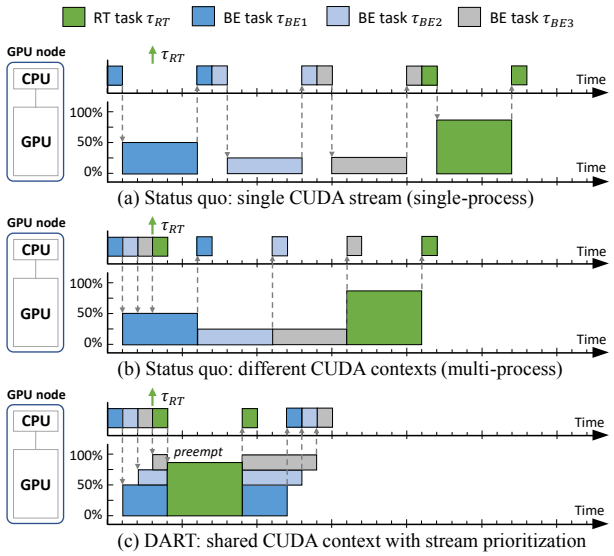
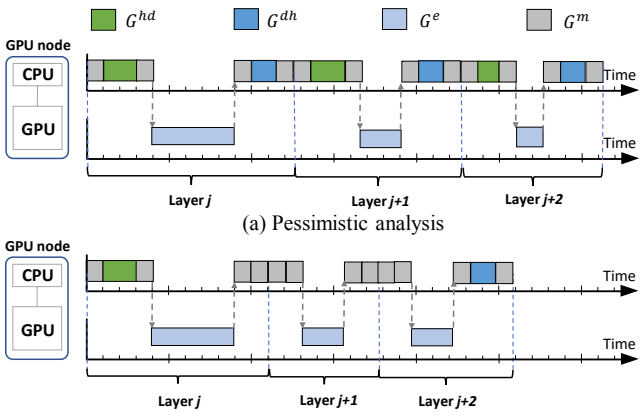


Fig. 7: GPU scheduling in the status quo and DART

Fig. 7 compares the GPU scheduling of DART with the two possible cases of the status quo. For simplicity, we skip the time for memory copies in this figure. Existing DNN frameworks, such as Caffe [1, 22], use only one default CUDA stream with no kernel prioritization. Hence, if one instance of the framework is launched, which is the case (a) in the figure, all operations on CPU and GPU are serialized. One may launch multiple instances of the existing DNN framework, which is the case (b). It is worth noting that this is the only way to serve two or more DNN models using the existing framework, including the latest real-time DNN inference work [52]. Then some operations on the CPU side can be overlapped with GPU kernel execution due to the CPU concurrency provided by multiple processes. However, while the BE kernels have low GPU utilization, e.g., only 25% of the GPU is used by τ_{BE2} and τ_{BE3}, this approach cannot support the concurrent execution of the BE kernels as they are launched from different CUDA contexts. Moreover, the execution of the RT kernel is delayed by the BE kernels. In case of DART, all the BE kernels run concurrently, achieving high GPU utilization, and the RT kernel preempts the BE kernels immediately, resulting in short response time.

One may question if the problems can be addressed in existing frameworks by simply assigning different CUDA streams to different models. However, it does not work because CUDA stream priority is effective only within a given context and does not work across different contexts. In addition, kernels from different CUDA contexts do not execute concurrently and each CUDA context time-shares the GPU, thereby leading to long waiting time and low utilization in GPU execution.



(a) Pessimistic analysis

(b) Improvement

Fig. 8: GPU stage execution time

3) Batched Execution: The batched execution of multiple DNN requests is a well-known optimization technique to increase throughput [28, 34, 52]. However, it is not suitable for RT tasks since they may suffer from unpredictable delay to fill the batch queue due to their sporadic arrival patterns. Therefore, we allow the batched execution only for BE tasks in DART. The batch size can be configured offline by the user. If the batch size is greater than one, batched execution is enabled and BE workers execute the entire batch of BE jobs as a single large request. This approach can bring higher throughput and better resource utilization to BE tasks, while minimizing interference on the execution of RT workers and the response time of RT tasks. We will discuss the effect of batching in RT task schedulability in the next subsection and present experimental results in Sec. VI.

B. Resource Management

The resource management scheme of the DART framework includes two algorithms. The one is to design the stages of one DNN task, either RT or BE, for a given node configuration, and to allocate the stages to the nodes. Using this, the other one is to find a node configuration that satisfies the timing constraints of all RT tasks and minimizes their response time. We first analyze the schedulability of RT tasks in our framework and then present the details of the two algorithms.

1) Schedulability Analysis: Before analyzing RT task schedulability, we define the stage execution time of a task and the overhead incurred by our framework for stage execution. The execution time of a stage of a task τ_i assigned to a node p_k, $\mathbb{C}_{i,k}$, is the cumulative execution time of the corresponding layers on p_k plus overhead. Thus, $\mathbb{C}_{i,k}$ is given by:

$$\mathbb{C}_{i,k} = \begin{cases} \left(\sum_{\tau_{i,j} \in p_k} C_{i,j}(p_k)\right) + \epsilon_{i,k}, & : \exists \tau_{i,j} \in p_k \\ 0 & : \text{otherwise} \end{cases} \quad (1)$$

where $\tau_{i,j} \in p_k$ denotes all the layers of τ_i execute on p_k, and $\epsilon_{i,k}$ is the total amount of overhead for stage execution on the node p_k (will be analyzed later).

While Eq. (1) can be used for both CPU and GPU nodes, we can reduce pessimism on GPU nodes. Fig. 8 shows the execution time of a GPU stage comprising three layers. Each layer has memory copies (G^{hd} and G^{dh}), kernel execution (G^e), and miscellaneous CPU operations (G^m). Eq. (1) captures the execution time of all these segments (shown as (a) in the figure). However, if multiple adjacent layers are executed on the same GPU node, the output of an intermediate layer can remain on the GPU memory and be directly used as input to the next layer. This eliminates the need for unnecessary memory copy and thus reduces the stage execution time. Based on this observation, we compute the stage execution time $\mathbb{C}_{i,k}$ on a GPU node p_k as follows:

$$\mathbb{C}_{i,k} = \sum_{\tau_{i,j} \in p_k} \left(G^e_{i,j}(p_k) + G^m_{i,j}(p_k)\right) + G^{hd}_{i,j1}(p_k) + G^{dh}_{i,j2}(p_k) + \epsilon_{i,k} \quad (2)$$

where $j1$ and $j2$ denote the first and the last layers of τ_i on the node p_k, respectively.

The overhead $\epsilon_{i,k}$ mainly comprises four factors: (i) the inter-node communication overhead $\epsilon^s_{i,k}$, which accounts for the time for signaling from the current node p_k of τ_i to its next node $p_{k'}$[3], (ii) the CPU preemption overhead ϵ^{cp}_k, which is the time to preempt the BE-class worker (and OpenBLAS threads if exist), (iii) the GPU preemption overhead ϵ^{gp}_k for preempting BE-class CUDA streams while they are executing kernels, and (iv) the non-preemptive CUDA memory copy blocking time caused by BE tasks, ϵ^{gm}_k, since CUDA stream preemption does not occur during memory copy. The first three factors need to be measured from the implementation. On the other hand, the last factor ϵ^{gm}_k can be obtained using task parameters:

$$\epsilon^{gm}_k = \max_{\tau_l \in p_k \wedge \tau_l \in \Gamma^{\text{BE}}} \max_{1 \le j \le L_l} \left(G^{dh}_{l,j}(p_k), G^{hd}_{l,j}(p_k)\right) \quad (3)$$

This equation takes the longest memory copy time among the layers of all BE tasks running on the GPU node p_k because once the memory copy that has been already executing finishes, the CUDA stream of an RT task can start. If batched execution is enabled, the memory copy time should be captured as the cumulative amount of copy time for b BE tasks where b is the batch size. Therefore, RT tasks may experience higher overhead if a larger BE batch size is used. Based on these, we can capture the total overhead $\epsilon_{i,k}$ as follows:

$$\epsilon_{i,k} = \epsilon^s_{i,k} + \epsilon^{cp}_k + \max(\epsilon^{gp}_k, \epsilon^{gm}_k) \quad (4)$$

It is worth noting that the equation takes the maximum of ϵ^{gp}_k and ϵ^{gm}_k as they do not occur simultaneously.

In DART, the stages of RT tasks on each node are scheduled non-preemptively (although they can preempt those of BE tasks on the same node) and the execution sequence of stages of a task across nodes can be modeled as a *directed acyclic path* in a graph of nodes. Hence, we use the schedulability analysis in [21] which is developed for non-preemptive distributed acyclic system scheduling. This analysis is based on the non-preemptive DAG delay composition theorem, and reduces the DAG into an equivalent uniprocessor system [21]. The analysis bounds the worst-case response time R_i of a task τ_i by the following iterative equation:

$$R^{(0)}_i = \mathbb{C}^*_e(i); \quad R^{(k)}_i = \mathbb{C}^*_e(i) + \sum_{\tau_h \in hp(\tau_i)} \lceil \frac{R^{(k-1)}_i}{T_h} \rceil \mathbb{C}^*_h \quad (5)$$

where $\mathbb{C}^*_h$ is the maximum stage execution time of a task τ_h

[3] $\epsilon^s_{i,k}$ is zero if p_k is the node that executes the last stage of a task τ_i.

among all of its stages (denoted as $\mathbb{C}_{h,max}$), $hp(i)$ is the set of higher-priority RT tasks than τ_i, and $\mathbb{C}^*_e(i)$ is given by:

$$\mathbb{C}^*_e(i) = \sum_{\tau_w \in hep(i)} (\mathbb{C}_{w,max} + \mathbb{C}_{w,max} \cdot SM_{w,i}) + \sum_{\substack{p_k \in path_i \wedge \\ k \leq N_P - 1}} \max_{\tau_u \in \Gamma^{\text{RT}}} \mathbb{C}_{u,k} + \sum_{p_k \in path_i} \max_{\tau_l \in lp(i)} \mathbb{C}_{l,max} \quad (6)$$

where $SM_{w,i}$ denotes the total number of splits-and-merges between the paths of τ_w and τ_i, $path_i$ is the set of nodes visited by τ_i, $hep(i)$ is the set of RT tasks with priority higher than or equal to τ_i, $lp(i)$ is the set of lower-priority RT tasks than τ_i, Γ^{RT} is the set of all RT tasks. Eq.(5) finishes when $R_i^{(k)} = R_i^{(k-1)}$ and the task τ_i is schedulable if $R_i^{(k)} \leq D_i$.

2) Designing Task Pipeline Stages: Consider a task τ_i using an DNN model m that has L_i layers. The system has a node configuration P consisting of N_P nodes, i.e., $P = \cup_k^{N_P}\{p_k\}$ where p_k is the k-th node. There may be stages of other tasks that have been already allocated to these nodes. The goal here is to construct the stages of τ_i for its L_i layers and to allocate each stage to a node in P so that the utilization of N_P nodes is balanced after the allocation. The number of stages of τ_i can be smaller than or equal to N_P, and the stage execution order has to be maintained due to the data dependency between layers. The reason for balancing utilization is to reduce contention on nodes. If one node is particularly contended for by many DNN tasks, it can be a bottleneck in the pipeline of nodes, thereby increasing the response time of RT tasks and reducing the throughput of BE tasks.

To solve this problem, we present an algorithm by extending the dynamic programming approach for the list partition problem [41]. Our algorithm aims at minimizing the maximum utilization of a node in the presence of other pre-allocated tasks. Let $M[n,k]$ denote the the utilization of the most loaded node when the first n layers of a task τ_i are allocated to the first k nodes, i.e., $p_1, p_2, ..., p_k$. The recurrence of the dynamic programming algorithm is given by:

$$M[n,k] = \min_{x=0}^{n} \max(M[x,k-1], w[k] + \sum_{y=x+1}^{n} U_{i,y}(p_k)) \quad (7)$$

where $w[k]$ represents the utilization of a node p_k with pre-allocated tasks (given as input), and $U_{i,y}(p_k)$ is the utilization of the layer y of τ_i when it executes on p_k, i.e., $U_{i,y}(p_k) = C_{i,y}(p_k)/T_i$. The initial conditions are:

$$M[0,k] = 0, \quad M[1,k] = \min_{q=1}^{k}(w[q] + U_{i,1}(p_q)) \quad (8)$$

With Eq. (7), the stage allocation of a task τ_i yielding balanced node utilization is found by $M[n = L_i, k = N_P]$.

3) Finding a Node Configuration for Tasks: Alg. 1 shows the procedure to find a node configuration for all RT and BE tasks in the system. It takes as input a taskset, Γ, and a set of candidate node configurations to be explored, $\mathbb{P}$. For each configuration $P \in \mathbb{P}$, the algorithm sorts tasks in descending order of of their average utilization, U_i^{avg}, which is computed as $U_i^{\text{avg}} = (1/k) \cdot \sum U_i(p_k)$. Then, it allocates the tasks to the nodes of P by using the dynamic programming approach given in (7) (line 9 of Alg. 1), and updates the w array based on the resulting allocation of each task. The algorithm checks if all RT tasks can meet their deadlines under this configuration, by using the schedulability test given by Eq. (5). If they pass the test, it computes the sum of the *weighted worst-case response time* of all RT tasks (line 15). The weight is a normalized weight computed as $\pi_i/|\Gamma^{\text{RT}}|$, where π_i is the relative order of τ_i's real-time priority and $|\Gamma^{\text{RT}}|$ is the number of RT tasks. The algorithm uses this weight to better represent the relative importance of each RT task's response time and to quantitatively compare different node configurations. Finally, the algorithm chooses the one with the minimum sum of weight response time and returns it. If it cannot find any configuration that satisfies the schedulability of RT tasks, an empty set is returned as failure.

To generate candidate node configurations, $\mathbb{P}$, we present Alg. 2 which takes two input parameters: $\mathcal{R}$, a set of CPU cores and GPU devices available in the system, and $N_P^{\max}$, the maximum number of nodes allowed for each configuration. In

Algorithm 1 Find a Node Configuration for Tasks

Require: $\Gamma = \{\tau_1, \tau_2, \tau_3, ..., \tau_n\}$: taskset
Require: $\mathbb{P}$: a set of candidate node configurations
Ensure: P^{sol}: a node config. found (solution); $P^{\text{sol}} = \emptyset$, if failed.
1: **function** FIND_NODE_CONFIGURATION(Γ, $\mathbb{P}$)
2: $P^{\text{sol}} = \emptyset$ /* initialization */
3: $W^{\text{sol}} = \infty$ /* weighted response time of RT tasks for P^{sol}
4: **for all** $P \in \mathbb{P}$ **do**
5: $N_P = |P|$
6: Initialize $w[1...N_P]$
7: **for all** $\tau_i \in \Gamma$ in descending order of U_i^{avg} **do**
8: L_i = the number of layers of τ_i
9: Compute $M[L_i, N_P]$ for τ_i by Eq. (7)
10: Store the stage-to-node allocation of τ_i
11: Update $w[1...N_P]$ with τ_i
12: Γ^{RT} = a set of all RT tasks in Γ
13: **if** $\forall \tau_i \in \Gamma^{\text{RT}}$ passes the schedulability test of Eq. (5) **then**
14: $\forall \tau_i \in \Gamma^{\text{RT}}$, R_i = worse-case response time of τ_i
15: $W = \sum_{\tau_i \in \Gamma^{\text{RT}}} (\pi_i/|\Gamma^{\text{RT}}|) * (R_i/D_i)$ /* π_i: priority */
16: **if** $W < W^{\text{sol}}$ **then**
17: $P^{\text{sol}} = P$
18: $W^{\text{sol}} = W$
return P^{sol}
19: **end function**

Algorithm 2 Generate Candidate Node Configurations

Require: $\mathcal{R} = \{c_1...c_n\}$: a set of available CPUs and GPUs
Require: $N_P^{\max}$: the maximum number of nodes per config P
Ensure: $\mathbb{P}$: a set of node configurations
1: **function** GENERATE_CANDIDATE_CONFIGURATIONS(P)
2: $\mathbb{P} = \emptyset$
3: $\mathbb{V}$ = permutations of $\mathcal{R}$ with duplicate core types
4: **for all** $V \in \mathbb{V}$ **do**
5: **for** $k = 1$ to $N_P^{\max}$ **do** /* number of nodes per config */
6: **for all** case Θ of $\binom{|V|}{k-1}$ **do** /* split V into k nodes */
7: $\Theta = \{\theta_1, \theta_2, ..., \theta_{k-1}\}$
8: /* each bracket [] in P indicates a node */
9: $P = \{[c_1...c_{\theta_1-1}], [c_{\theta_1}...c_{\theta_2-1}], ..., [c_{\theta_{k-1}}...c_{|V|}]\}$
10: **if** all nodes $\in P$ satisfies the system model **then**
11: $\mathbb{P} = \mathbb{P} \cup P$
return $\mathbb{P}$
12: **end function**

$\mathcal{R}$, each GPU has been already paired with one CPU core and that CPU core is not present. $N_P^{\max}$ can be chosen considering system parameters, e.g., the number of CPU clusters and the number of GPUs. However, since the delay caused by lower-priority tasks in non-preemptive scheduling is proportional to the the number of nodes (not the number of lower-priority tasks) [21], it is better not to use an overly large $N_P^{\max}$. The algorithm first obtains distinct permutations of $\mathcal{R}$ considering the same type of CPU cores as duplicates. For example, for an Nvidia TX2 platform with 1 GPU pre-paired with 1 A57 core, 3 other A57 cores, and 2 Denver cores, the number of permutations is $(1+3+2)!/(3! \cdot 2!) = 60$. Then, for each permutation V, it splits V into k nodes, where $1 \leq k \leq N_P^{\max}$, by considering the combinations of $k-1$ partitions between nodes (line 6). If the resulting node configuration P satisfies our system model where a CPU node has only one type of cores and a GPU node has no other CPU core rather than its pre-paired one, P is considered legal and added to $\mathbb{P}$.

DART uses the above two algorithms in the initialization phase with a static taskset. In the runtime phase, DART can accept new tasks via admission control, but does not change the node configuration, as doing so may cause excessive overhead and affect other tasks being executed. To reduce such overhead and enable runtime node re-configuration, one may consider other heuristics or manual tuning, instead of Alg. 2.

C. Other Miscellaneous Components

1) Layer-wise Execution Time Profiling: To perform resource management and check task schedulability, the worst-case execution time (WCET) of individual layers of DNN models on each node needs to be estimated. DART takes a measurement-based approach for WCET estimation using a layer-wise profiling mechanism. Fig. 9 illustrates the procedure of the profiling mechanism. When DART starts, it first checks if a profile database exists for all DNN models it supports. If not, it executes all the supported models on all nodes of candidate node configurations (from Alg. 2), and measures their execution time by running n times that the user can choose. Then it estimates the WCET by taking the maximum among the observed execution time history. After this, DART enters the runtime phase and starts accepting inference jobs.

Note that the WCET estimated by the above procedure may be violated at any time when longer execution time is observed. To mitigate this problem, DART continuously updates the profile database at runtime. Upon the completion of each layer execution, DART checks if the execution time exceeds the WCET observed before. If so, DART updates the WCET for that layer and triggers *run-time task enforcement*. The profiling mechanism of DART also allows using more robust WCET estimation methods, e.g., extreme value theory [31], as it stores historical execution time data in the database. This is an interesting topic but beyond the scope of this paper.

2) Admission Control: DART includes an admission control mechanism to ensure that all accepted RT tasks can meet their deadlines. The schedulability of RT tasks is analyzed by using Eq. (5), and it is used by Alg. 1 to find a node configuration for a static taskset during the initialization phase.

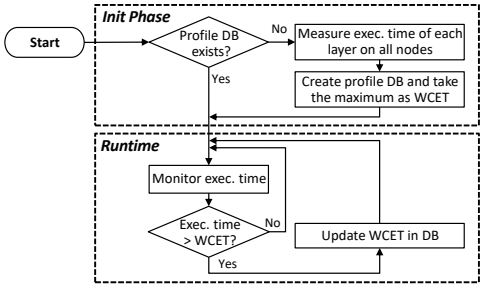


Fig. 9: Procedure of layer-wise execution time profiling

At runtime, when there is a new task, the admission control first determines the stage-to-node allocation of the new task using Eq. 7, and then checks if the new task is RT or BE. In case of RT, the task is accepted only when all RT tasks including the new one are schedulable. In case of BE, it is accepted as long as its longest GPU memory copy time is smaller than or equal to the existing $\epsilon_k^g m$ value, because the other parts of the BE task will not delay existing RT tasks due to the scheduling class design of DART. It is worth noting that the runtime admission control executes with normal priority and thus does not interfere with RT workers.

3) Run-time Task Enforcement: The execution time of each layer is monitored at runtime by the profiling mechanism. If it exceeds the WCET previously recorded in the database and is caused by an RT task, DART immediately demotes the task to BE and executes the subsequent layers of that task in the BE scheduling class. If the WCET exceedance is caused by a BE task, its deadline is changed to infinite so as to minimize the negative impact on other BE tasks. Then DART performs the schedulability test of RT tasks with the new WCET value. If all RT tasks remain schedulable, the task demoted to the BE class is recovered to the RT class. Otherwise, the user is alerted that the system can no longer meet the deadlines of all RT tasks and needs to be reconfigured.

V. Implementation

DART is implemented for Intel Xeon and Nvidia TX2 platforms running Ubuntu 16.04. We used Caffe v1.0 [1, 22] as our implementation basis and CUDA 9.0 for GPU programming[4]. The WCET database of the profiling mechanism, taskset information, and node configurations are managed in the JSON format for convenience.

The stock version of Caffe allocates memory for a `Net` instance, which stores trained network model parameters and intermediate computation results together in the same data structure. When multiple tasks use the same DNN model, each task needs to have its own `Net` instance, but it causes memory wastage because multiple copies of the same model parameters are created. To address this issue, our implementation introduces a new `dataNet` class that allows the separation of the computation results from the model parameters. Hence, for tasks using the same DNN model, only one memory instance

[4]We have chosen Caffe to implement DART as Caffe has been one of the most widely used frameworks driven by academia. It is implemented in C++. It also has less dependencies compared to Tensorflow or Pytorch.

of model parameters can be allocated and shared by the tasks, thereby saving memory.

We designed a custom application-level messaging protocol based on TCP/IP so that DART can accept new tasks at runtime and receive inference jobs of admitted RT/BE tasks. With this protocol, both local and remote tasks can be served by DART. The protocol includes: i) messages sent from tasks to DART for timing constraints, DNN models to use, and input data for inference, and ii) messages sent from DART to tasks for admission control results, inference output, and plain text for user alerts. For the synchronization of workers, we implemented an event notification mechanism using `condition_variable` from C++11 std library.

VI. Evaluation

A. Experiment Setup

We evaluate DART against two baselines: BaseCPU and BaseGPU. These represent the state-of-the-art DNN inference frameworks supporting multiple DNNs, such as TensorRT Inference Server[5] and Tensorflow Serving[6], which are front-end services using Caffe or Tensorflow as back-end. BaseCPU and BaseGPU each have a single run queue for each model to distribute workload across different processors. To make a fair comparison with DART, we also add priority queues to the two baselines, so that when multiple tasks are waiting for the start of execution, real-time tasks can be prioritized over best-effort ones. Then we create one process for each DNN model used since the state-of-the-art frameworks support one DNN model per process by default.[7] BaseCPU performs all of its operations on CPUs by the multi-thread OpenBLAS library, and BaseGPU uses the GPU. The two baselines are implemented based on stock Caffe V1.0. We evaluate DART and the two baselines on Intel Xeon and Nvidia TX2 platforms. The Xeon platform has an 8-core 2.1GHz Intel Xeon E2620 v4 CPU and an Nvidia GTX 1080 discrete GPU. The TX2 platform is equipped with an SoC that has a quad-core ARM Cortex-A57 CPU cluster, a dual-core Denver CPU cluster, and an integrated GPU. Four DNN models are considered in the evaluation: Alexnet [28], LeNet [29], VGGnet [2], for object classification, and Pilotnet [9], for self-driving. All the CPUs and GPUs on both platforms are configured to run at their maximum clock frequency. Unrelated system services, e.g. WiFi and lightDM, are disabled to avoid potential interference.

Runtime Overhead. Recall that, under DART, when the execution flow moves from one stage to another one, only a signal is sent to the next node. There is no performance loss due to cache refills since each node executes different part of layers and does not reuse the cache blocks of the previous node. Switching from CPU to GPU nodes requires memory copy, which is captured in our system model and in Eq. (2).

The major runtime overhead introduced by DART is therefore the time for synchronization and communication among nodes. This overhead may vary depending on the architecture type of CPU cores. Hence, we measured the overhead on the TX2 platform which has two different core types. The results are shown in Table I. We hypothesized that the communication overhead between different types of cores would be larger than that between the same type of cores, but it turned out the time between two Denver cores were the highest. However, these costs are acceptably small or marginal, compared with the typical layer execution time of DNN tasks and the improvement achieved by DART. Table II shows the GPU kernel preemption overhead caused by CUDA priority stream. We observe that such preemption overhead is relatively small, yet we already take into account by modeling it as the ϵ^{gp} term in Eq. (4).

TABLE I: Inter-node communication overhead on TX2

Time (us)	A57-A57	A57-Den	Den-A57	Den-Den
Average	20.83	36.33	42.58	51.45
Maximum	48	69	82	102

TABLE II: CUDA stream kernel preemption overhead

Time (us)	GTX 1080	TX2
Average	17.87	28.76
Maximum	52	121

B. DNN Execution Time Profiling

For all the DNN models used for evaluation, we have estimated the worst-case execution time of each layer under different node configurations by using the profiling mechanism of DART. Due to the space limit, we report only one of the results, LeNet on TX2, in Fig. 10. Three ARM A57 cores are shown here as one ARM core is paired with the GPU. The execution time of layers 1, 7, 8, and 9 are too small, compared to the others, and not discernible in this figure. We observe that the speed-up from an increased number of CPU cores varies significantly by layers. For example, layers 3 and 5 do not gain any noticeable benefit by increasing the number of A57 cores from one to three; on the other hand, layers 2 and 4 have speedups with more CPUs. In general, the speedup gradually diminishes on both Xeon and TX2 platforms as the number of cores increases. We also observe the performance characteristics of different types of processors. On both platforms, the GPU takes much less execution time than CPUs on most layers, but there are cases where the GPU is only < 2 times faster than CPUs, e.g., ARM*3 for layer 2 in the figure. Denver CPU cores overall perform better than ARM A57 CPUs. The results show the potential resource inefficiency that can be caused when resources are assigned blindly, and thus motivate the support for heterogeneous resources provided by DART.

C. Schedulability Experiments

Based on the collected layer WCET profiles, we have conducted schedulability experiments using randomly-generated bi-modal tasksets. The parameters we consider to generate tasksets are the number of tasks, periods (= deadlines), DNN models, and the ratio of tasks using heavy models to all tasks in a taskset. Based on the execution time measurement of the DNN model, we consider tasks using VGGnet and Alexnet as heavy-model tasks, and those using Pilotnet and LeNet as light-model tasks. Since our focus is on schedulability,

[5]https://github.com/NVIDIA/tensorrt-inference-server

[6]https://github.com/tensorflow/serving

[7]This is the case for the latest version of TensorRT Inference Server. While it allows creating more processes per model for better throughput, doing so may negatively affect the latency of other models.

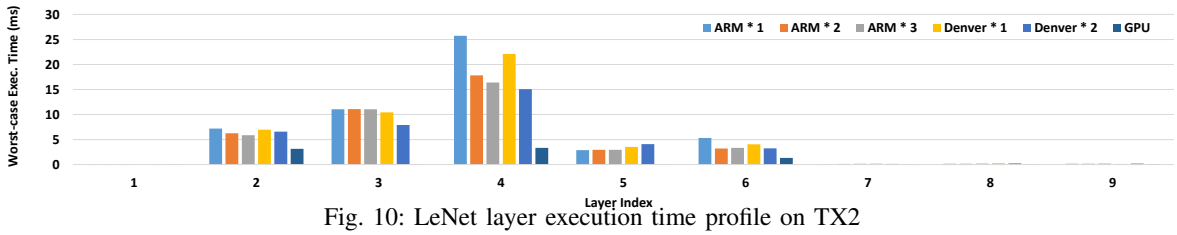
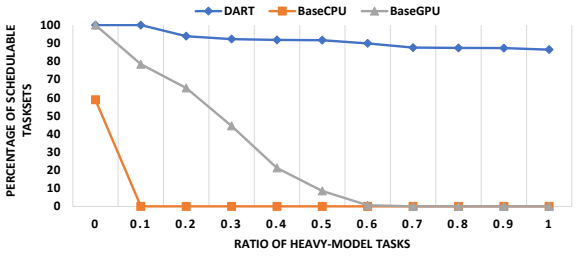
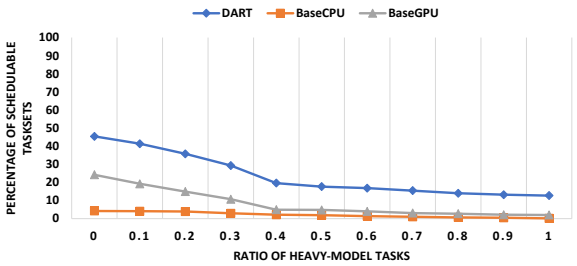


Fig. 10: LeNet layer execution time profile on TX2

only RT tasks are considered and no BE tasks are used here. The two baselines are both single-node systems and thus can be modeled as uni-processor systems with faster processors. Hence, we apply the standard iterative response time test to check schedulability of the taskset. For DART, we apply the schedulability analysis introduced in the previous section.

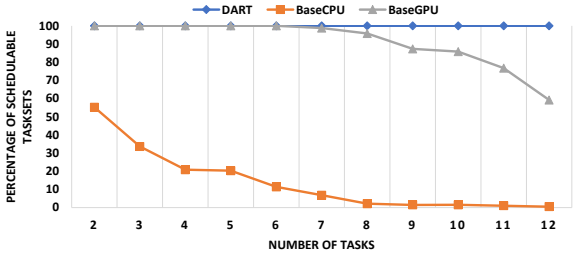


(a) Xeon platform

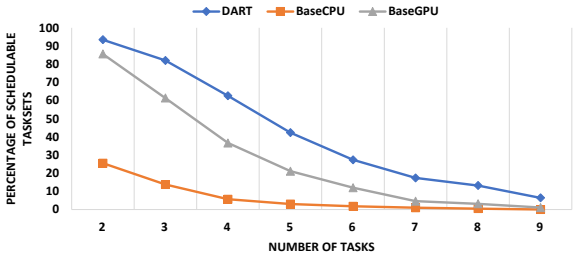


(b) TX2 platform

Fig. 11: Schedulability w.r.t. the ratio of heavy-model tasks



(a) Xeon platform



(b) TX2 platform

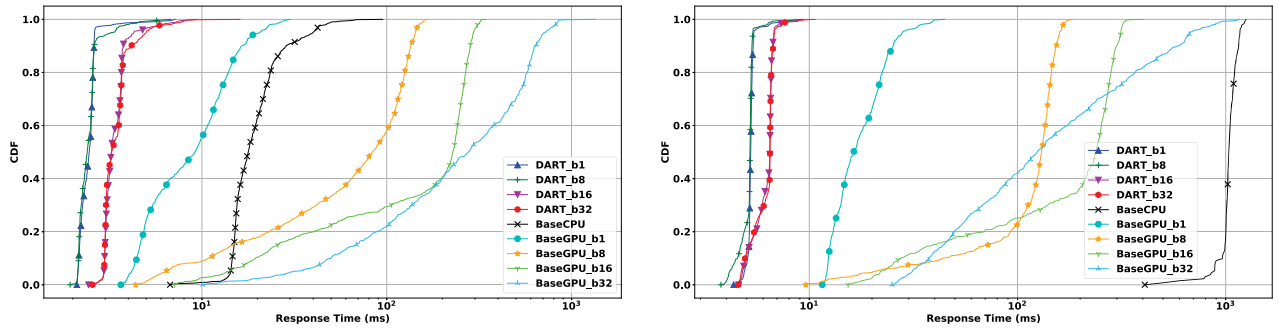
Fig. 12: Schedulability w.r.t. the number of tasks

We first show the schedulability of tasksets with different ratios of heavy-model tasks. Fig. 11 shows the results on Xeon and TX2 platforms. The task deadlines are randomly chosen between 300 and 500 ms. Each taskset has 10 RT tasks. We observe that, with higher heavy-model task ratios, the percentage of schedulable tasksets decreases for all the three approaches since the tasks with longer execution time are harder to meet deadlines. Xeon has higher schedulability than TX2 because the same taskset parameters are used on both platforms and Xeon is more computationally powerful than TX2. However, DART dominates the other two in all experimental conditions. Specifically, on Xeon with the ratio of 0.6, DART schedules about 90% of tasksets while the other two schedule none of them. We then change the number of tasks per taskset and compare the taskset schedulability. Fig. 12 depict the results on Xeon and TX2. As can be seen, tasksets become harder to schedule as more tasks exist in the system. DART, however, still always outperforms the two baselines. The results indicate that DART effectively utilizes given CPU and GPU resources and its benefit is significant in scheduling real-time DNN tasks.

D. Response Time and Throughput

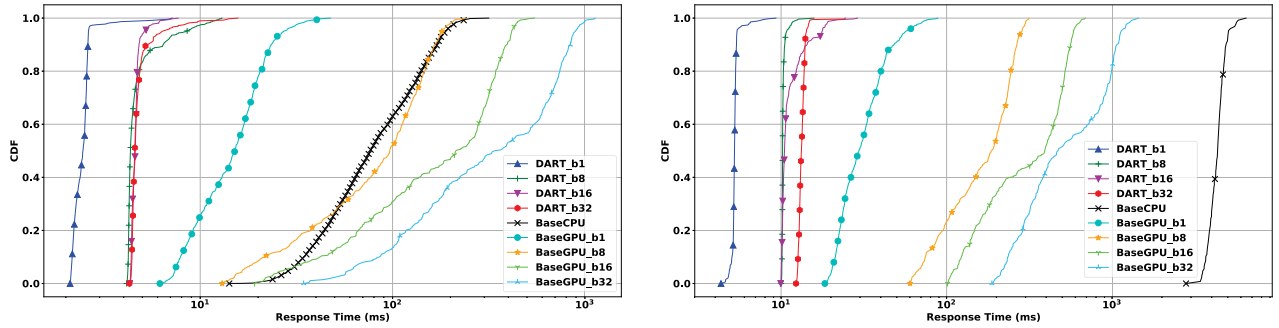
To understand the performance characteristics of DNN frameworks in response time and throughput, we have conducted a case study with a mixture of real-time (RT) and best-effort (BE) tasks on Xeon and TX2 platforms. The taskset we used is given in Table III. There are four RT tasks, two using pilotnet and the other two using alexnet. The deadlines of the RT tasks are set equal to their periods, and with tie-breaking, `pilot_rt_1` has the highest priority. There are also three BE tasks, each of which uses a different DNN model. The jobs of the BE tasks arrive in a back-to-back manner, i.e., a new job arrives as soon as the previous one finishes, in order to overload the system. All the BE and RT tasks are located in different machines and make requests to the system running a DNN framework over an Ethernet-based local area network (LAN). Note that the amount of input data and output results are considerably small compared to the bandwidth of the network ($< 1\%$). We also consider batched execution with the representative batch sizes of 8, 16 and 32 to evaluate their effects in response time and throughput. The batch size `n` is denoted as a suffix `_bn` in the label, e.g., `BaseGPU_b1` and `BaseGPU_b16` mean BaseGPU without batching and with the batch size of 16, respectively. For BaseCPU, we present only the results without batched execution as we observed inappreciable improvement in throughput but significant deterioration in response time when batching was enabled.

In this environment, we measure the end-to-end response time of RT tasks and the throughput of BE tasks under the



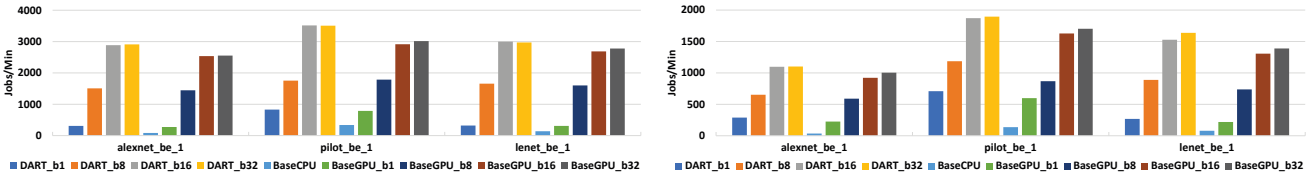
(a) CDF of pilot_rt_2 (b) CDF of alexnet_rt_2

Fig. 13: Response time CDF of real-time tasks on Xeon



(a) CDF of pilot_rt_2 (b) CDF of alexnet_rt_2

Fig. 14: Response time CDF of real-time tasks on TX2



(a) Xeon (b) TX2

Fig. 15: Throughput of BE tasks

three different DNN frameworks. Fig. 13 and Fig. 14 show the response time CDF of `pilot_rt_2` and `alexnet_rt_2` on Xeon and TX2, respectively (x-axis in log scale). These tasks have the second highest and lowest real-time priority in the taskset, which are good to demonstrate the impact of interference from higher-priority RT tasks and other BE tasks. Overall, DART gives much smaller tail latency than the others at the respective batch size. The tail latency increases with the batch size under all approaches, but such slowdown is much less in DART than BaseGPU. In case of `alexnet_rt_2` on TX2 with the batch size of 32, DART achieves 98.5% reduction in the maximum observed response time over BaseGPU. This is not only due to the RT and BE worker design of DART, but also due to its GPU node scheduling which uses separate streams for RT and BE kernels with CUDA stream prioritization. Conversely, under BaseGPU, an RT job may need to wait until the batch is filled up, and RT kernels may be delayed by BE kernels that have arrived earlier and been already executing. Even without batching, DART achieves 89.9% reduction in the maximum observed response time.

We have also measured the throughput of the BE tasks while the above four RT tasks are concurrently running in the same environment. Fig. 15 shows the throughput of each BE task on Xeon and TX2. Batched execution significantly improves the throughput under both DART and BaseGPU. However, the improvement diminishes after the batch size reaches 16. For all the BE tasks, DART gives the highest throughput, with as much as 17.9% higher throughput than BaseGPU for `lenet_be_1` on TX2 with the batch size of 32. This improvement of DART comes from two factors: the use of CPU cores together with the GPU for DNN inference tasks, and the use of multiple CUDA streams for concurrent kernel execution. In summary, the results give strong evidence that the efficient utilization of heterogeneous CPU and GPU resources is crucial in both low latency and high throughput. DART achieves these two simultaneously and dominates the other baselines that are representative of the state-of-the-art.

TABLE III: Taskset information on Xeon and TX2

Task name	DNN model	Deadline	Class	RT Prio
`pilot_rt_1`	Pilotnet	150 ms	RT	90
`pilot_rt_2`	Pilotnet	150 ms	RT	89
`alexnet_rt_1`	Alexnet	200 ms	RT	88
`alexnet_rt_2`	Alexnet	200 ms	RT	87
`pilot_be_1`	Pilotnet	–	BE	-
`alexnet_be_1`	Alexnet	–	BE	-
`lenet_be_1`	LeNet	–	BE	-

VII. Related Work

DNN Inference Frameworks. As DNNs have been widely adopted in real-time applications such as [9, 11, 43, 45], much effort has been invested in improving the performance of DNN inference. DeepSense [50] utilizes a GPU to speed up DNN computation. Glimpse [10] and MCDNN [16] are real-time mobile DNN frameworks collaborating with the cloud. Neurosurgeon [23] is a distributed inference framework with a light-weight scheduler, which partitions DNN computation between local and cloud machines in a layer granularity to improve latency. DART can be extended for such edge-computing scenarios by modeling a remote machine or cloud server as yet another node, but network delay needs to be taken into account. DeepEye [34] is developed to achieve DNN inference on a memory-limited embedded device by using model-compression and layer-interleaving. DeepMon [18] implements layer caching, decomposition and multiplication techniques to achieve a low-latency mobile-GPU DNN framework. S^3DNN [52] is developed for real-time workloads and improves average-case response time by applying supervised streaming and scheduling on GPUs. The work in [49] recently presents a case study on pipelining, parallelism, and image composition techniques for autonomous driving DNN applications. Note that both [52] and [49] assume only one type of DNN model, YOLO, and thus can be subject to the same problem as the status quo, when multiple DNN models are concurrently used. While all of the aforementioned frameworks have targeted on lowering inference latency, they do not focus on ensuring the timing constraints of DNN tasks in the worst-case scenario. In other words, they are not amenable to real-time schedulability analysis and do not offer deterministic worst-case response time while improving throughput. These limitations are addressed in this paper.

DNN Optimization Techniques. Much work has focused on compressing DNN models or layers [13, 15, 27, 46, 51]. They are motivated by trading-off output accuracy for performance gain. Some of these techniques achieve great speed-up, e.g., a 94.5% improvement in execution time without significant loss of accuracy [51]. Such optimization techniques can also be applied to DART to achieve shorter execution time of each DNN model and a more balanced pipeline, which helps improve latency and throughput.

Pipeline and GPU Scheduling. Jayachandran et al. proposed the delay composition theorem to analyze real-time pipelines under preemptive and non-preemptive scheduling [20] and then later extended it to DAG tasks [21]. DART utilizes the DAG-version of the theorem to check the schedulability of real-time tasks. However, other approaches, such as local deadline assignment [17], can also be potentially used in our framework and we wish to investigate it in the future.

Real-time GPU scheduling has been extensively investigated in the literature. Many earlier schemes [12, 25, 38] launch one kernel at a time to guarantee bounded response time. Recent studies [37, 48] report that the concurrent execution of multiple kernels can improve throughput and GPU utilization, but at the cost of predictability. To achieve better performance isolation, techniques to split a GPU into multiple partitions have been proposed [19, 40]. Nvidia Multi-Process Service (MPS) [36] is a middleware that facilitates concurrent kernel execution when kernels are launched from different processes or CUDA contexts. However, at the time of writing, it is available only on x86 architectures and does not provide a direct control over the CUDA stream priority of real-time kernels. The GPU node scheduling of DART is designed to take the best of both predictability- and throughput-oriented approaches.

Memory-induced Interference. Temporal interference caused by shared memory resources, such as caches, DRAM, and memory buses, has been considered a serious problem in multi-core real-time systems [6, 24, 33, 42, 44, 47]. Recent work [5] reports that such memory-induced interference can cause as much as 3× slowdown to GPU kernel execution in CPU-GPU integrated SoCs like Nvidia TX2, when memory-bandwidth-intensive synthetic tasks are co-scheduled on CPUs. In this paper, we did not focus on this problem since our primary goal was to establish a real-time DNN framework with improvements on scheduling algorithms and abstractions. We also did not observe any significant slowdown of DNN tasks in our experiments. However, due to the increasing complexity of DNN models, memory interference may soon become a limiting factor in real-time DNN systems. We believe a more sophisticated system addressing such issues can be built upon DART. For instance, software techniques to protect against DoS attacks on the last-level cache shared among CPUs [8] and to partition the compute and memory resources of GPUs including streaming multiprocessors, caches, and DRAM [19] can be integrated into the CPU and GPU nodes of DART to achieve a higher degree of performance isolation.

VIII. Conclusions

In this paper, we presented DART, a real-time DNN framework that offers deterministic response time to real-time DNN inference tasks and supports concurrent execution of various types of DNN models. We have implemented the scheduling architecture of DART and its key components, including pipeline stage design, node configuration, execution time profiling, admission control, and runtime enforcement, on Intel Xeon and Nvidia TX2 platforms. Experiment results show that DART yields significant improvement in the maximum response time and throughput over the status quo, and gains substantial benefit from heterogeneous CPU and GPU resources.

The inter-node pipelining and intra-node parallelism design of DART offers several interesting directions for future work. First, remote machines can be modeled as one of the nodes in the pipeline and can be utilized to address the computational limit of local hardware. Second, other types of hardware accelerators, such as FPGAs, can be co-used with CPUs and GPUs for larger DNNs. Third, shared memory resources, such as caches and memory buses, and their performance interference are worth investigating to improve performance isolation in CPU and GPU parallelization. We plan to explore these topics in the future.

REFERENCES

[1] Caffe. http://caffe.berkeleyvision.org. Accessed: 2019-03-30.

[2] Ilsvrc-2014 model (vgg team) with 16 weight layers. https://gist.github.com/ksimonyan/211839e770f7b538e2d8#file-readme-md. Accessed: 2019-03-30.

[3] Torch. http://torch.ch. Accessed: 2019-03-30.

[4] M. Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[5] W. Ali and H. Yun. Protecting real-time GPU applications on integrated CPU-GPU SoC platforms. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2017.

[6] B. Andersson, H. Kim, D. D. Niz, M. Klein, R. R. Rajkumar, and J. Lehoczky. Schedulability analysis of tasks with corunner-dependent execution times. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(3):71, 2018.

[7] P. Axer et al. Response-time analysis of parallel fork-join workloads with real-time constraints. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[8] M. Bechtel and H. Yun. Denial-of-service attacks on shared cache in multicore: Analysis and prevention. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

[9] M. Bojarski et al. End to end learning for self-driving cars. *CoRR*, abs/1604.07316, 2016.

[10] T. Y.-H. Chen et al. Glimpse: Continuous, real-time object recognition on mobile devices. In *ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, 2015.

[11] J. S. Dyrstad and J. R. Mathiassen. Grasping virtual fish: A step towards robotic deep learning from demonstration in virtual reality. In *IEEE Conference on Robotics and Biomimetics (ROBIO)*, 2017.

[12] G. Elliott et al. GPUSync: A framework for real-time GPU management. In *IEEE Real-Time Systems Symposium (RTSS)*, 2013.

[13] Y. Guo, A. Yao, and Y. Chen. Dynamic network surgery for efficient dnns. *CoRR*, abs/1608.04493, 2016.

[14] M. Hajinoroozi et al. Prediction of driver's drowsy and alert states from EEG signals with deep learning. In *IEEE Workshop on Computational Advances in Multi-Sensor Adaptive Processing*, 2015.

[15] S. Han et al. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *CoRR*, abs/1510.00149, 2015.

[16] S. Han et al. MCDNN: An approximation-based execution framework for deep stream processing under resource constraints. In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2016.

[17] S. Hong et al. Meeting end-to-end deadlines through distributed local deadline assignments. In *IEEE Real-Time Systems Symposium (RTSS)*, 2011.

[18] L. N. Huynh et al. DeepMon: Mobile GPU-based deep learning framework for continuous vision applications. In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2017.

[19] S. Jain, I. Baek, S. Wang, and R. Rajkumar. Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

[20] P. Jayachandran and T. Abdelzaher. Delay composition in preemptive and non-preemptive real-time pipelines. *Real-Time Systems*, 40(3):290–320, Dec 2008.

[21] P. Jayachandran and T. Abdelzaher. Transforming distributed acyclic systems into equivalent uniprocessors under preemptive and non-preemptive scheduling. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2008.

[22] Y. Jia et al. Caffe: Convolutional architecture for fast feature embedding. In *ACM International Conference on Multimedia (MM)*, 2014.

[23] Y. Kang et al. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *SIGARCH Comput. Archit. News*, 45(1):615–629, Apr. 2017.

[24] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. R. Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2014.

[25] H. Kim et al. A server-based approach for predictable GPU access control. In *RTCSA*, 2017.

[26] J. Kim, H. Kim, K. Lakshmanan, and R. Rajkumar. Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car. In *IEEE/ACM International Conference on Cyber-Physical Systems (ICCPS)*, 2013.

[27] Y. Kim et al. Compression of deep convolutional neural networks for fast and low power mobile applications. *CoRR*, abs/1511.06530, 2015.

[28] A. Krizhevsky et al. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 2012.

[29] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[30] J. Li, J. J. Chen, K. Agrawal, C. Lu, C. Gill, and A. Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 85–96. IEEE, 2014.

[31] G. Lima, D. Dias, and E. Barros. Extreme value theory for estimating task execution time bounds: A careful look. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2016.

[32] S.-C. Lin et al. The architectural implications of autonomous driving: Constraints and acceleration. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[33] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2013.

[34] A. Mathur et al. DeepEye: Resource efficient local execution of multiple deep vision models using wearable commodity hardware. In *Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2017.

[35] A. K. Mok. Fundamental design problems of distributed systems for the hard real-time environment. *PhD Thesis, Massachusetts Institute of Technology*, 1983.

[36] Nvidia. Multi-process service. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf. Accessed: 2019-08-20.

[37] N. Otterness et al. 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.

[38] P. Patel et al. Analytical enhancements and practical insights for MPCP with self-suspensions. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2018.

[39] M. Pfeiffer et al. From perception to decision: A data-driven approach to end-to-end motion planning for autonomous ground robots. In *IEEE Conference on Robotics and Automation (ICRA)*, 2017.

[40] S. K. Saha, Y. Xiang, and H. Kim. STGM : Spatio-Temporal GPU Management for Real-Time Tasks. In *RTCSA*, 2019.

[41] S. S. Skiena. *The algorithm design manual*. Springer, 1998.

[42] N. Suzuki, H. Kim, D. de Niz, B. Andersson, L. Wrage, M. Klein, and R. R. Rajkumar. Coordinated bank and cache coloring for temporal protection of memory accesses. In *IEEE International Conference on Embedded Software and Systems (ICESS)*, 2013.

[43] F. Tang et al. On removing routing protocol from future wireless

networks: A real-time deep learning approach for intelligent traffic control. *IEEE Wireless Communications*, 25(1):154–160, February 2018.

[44] P. K. Valsan, H. Yun, and F. Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

[45] D. Vasisht et al. Farmbeats: An IoT platform for data-driven agriculture. In *NSDI*, 2017.

[46] Y. Wang et al. CNNPack: Packing convolutional neural networks in the frequency domain. In *Advances in neural information processing systems*, 2016.

[47] M. Xu, L. Thi, X. Phan, H.-Y. Choi, and I. Lee. vCAT: Dynamic cache management using CAT virtualization. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017.

[48] M. Yang et al. Avoiding pitfalls when using Nvidia GPUs for real-time tasks in autonomous systems. In *Euromicro Conference on Real-Time Systems*, 2018.

[49] M. Yang et al. Re-thinking CNN frameworks for time-sensitive autonomous-driving applications: Addressing an industrial challenge. In *RTAS*, 2019.

[50] S. Yao et al. Deepsense: A unified deep learning framework for time-series mobile sensing data processing. *CoRR*, abs/1611.01942, 2016.

[51] S. Yao et al. Compressing deep neural network structures for sensing systems with a compressor-critic framework. *CoRR*, abs/1706.01215, 2017.

[52] H. Zhou, S. Bateni, and C. Liu. S3DNN: Supervised streaming and scheduling for GPU-accelerated real-time DNN workloads. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2018.