Received: 20 December 2019 | Revised: 4 March 2020 | Accepted: 25 March 2020
DOI: 10.1002/spe.2840

RESEARCH ARTICLE

WILEY

Timing isolation and improved scheduling of deep neural networks for real-time systems

Daniel Casini[1,2] | Alessandro Biondi[1,2] | Giorgio Buttazzo[1,2]

[1]Department of Excellence in Robotics & AI, Scuola Superiore Sant'Anna, Pisa, Italy
[2]TeCIP Institute, Scuola Superiore Sant'Anna, Pisa, Italy

Correspondence
Daniel Casini, Department of Excellence in Robotics & AI, Scuola Superiore Sant'Anna, Pisa, Italy.
Email: daniel.casini@santannapisa.it

Summary
In recent years, the performance of deep neural networks (DNNs) is significantly improved, making them suitable for many application fields, such as autonomous driving, advanced robotics, and industrial control. Despite a lot of research being devoted to improving the accuracy of DNNs, only limited efforts have been spent to enhance their timing predictability, required in several real-time applications. This paper proposes a software infrastructure based on the Linux operating system to integrate DNNs within a real-time multicore system. It has been realized by modifying both the internal scheduler of the popular TensorFlow framework and the SCHED_DEADLINE scheduling class of Linux. The proposed infrastructure allows providing timing isolation of DNN inference tasks, hence improving the determinism of the temporal interference generated by TensorFlow. The proposal is finally evaluated with a case study derived from a state-of-the-art benchmark inspired by an autonomous industrial system. Extensive experiments demonstrate the effectiveness of the proposed solution and show a significant reduction of both average and longest-observed response times of TensorFlow tasks.

KEYWORDS
deep learning, neural networks, predictability, real-time systems, temporal isolation, tensorflow

1 | INTRODUCTION

The huge performance improvement recently achieved by deep neural networks (DNNs) has motivated their use in a large number of fields, such as autonomous driving,[1] robotics,[2] particles detection,[3] and industrial control.[4] A measure of such an improvement is given by the results of a popular challenge for assessing the performance of DNNs:[5] the error rate for image classification tasks has been reduced from 28% in 2010 to 2.3% in 2017,[6] surpassing human capabilities in a similar context. A DNN is an information processing system inspired by the structure of a mammalian brain, hence consisting of a network of nodes (or neurons) exchanging data through directed channels modulated by synaptic weights. Weights are randomly initialized and then tuned by means of a *learning algorithm*. In a supervised learning paradigm, the neural network is trained by examples to match specific inputs with expected outputs and the weights are modified as a function of an error defined on the output neurons. DNNs are commonly developed and inferred by means of state-of-the-art frameworks, such as Tensorflow, Caffe, and Torch, which ease the development of new DNN architectures.

Unfortunately, none of the current frameworks are specifically optimized for being used in real-time applications, where computational activities (ie, tasks) are subject to timing constraints, and they are not supported by commercial real-time operating systems (eg, VxWorks, QNX). For instance, if adopted in automotive systems for image recognition and autonomous driving, DNNs should have a highly predictable behavior not only with respect to their functional aspects, but also in the time domain, responding within specific deadlines. With the aim of improving space, weight, energy, and costs, DNNs will likely be executed on top of the same hardware platform used for other real-time tasks (eg, image acquisition, actuation, etc.). As a consequence, DNN tasks may be subject to a variable inter-task interference, generated by the concurrent execution of other critical tasks, which can jeopardize their temporal correctness.

An effective solution to overcome this problem is to isolate the execution of time critical activities through a resource reservation mechanism,[7] which allocates and guarantees a fraction of the processor bandwidth to given task sets, thus limiting the interference from other isolated tasks. The adoption of a reservation algorithm is also supported by the fact that an accurate *worst-case execution time* (WCET) estimate is often not available for DNN tasks, especially when they are executed by means of inference frameworks.

1.1 | Contributions

This paper aims at providing support for enhancing the timing predictability of DNNs, managed by the popular TensorFlow framework, and running under Linux on top of a multicore platform shared with other real-time tasks. The following contributions are provided. First, a deep code inspection has been performed and reported to understand how TensorFlow dispatches DNN operations. Also, the parallel computational graph generated by Tensorflow when executing a complex state-of-the-art DNN has been profiled for understanding the peculiar features of such a workload. This knowledge has been used to introduce a *budget overrun* mechanism in Linux, which enables overcoming to possible performance degradation introduced by the usage of the `SCHED_DEADLINE` scheduling class of Linux for providing *temporal isolation* for TensorFlow. Furthermore, the TensorFlow scheduler has been modified by introducing a *localized work-stealing* policy, which aims at improving data locality during DNN inference tasks. The proposed techniques have been evaluated by means of a case study inspired by an autonomous industrial system. Experiments show that adopting the proposed solution allows improving both the average-case and the longest-observed response times of DNNs up to 30%, with respect to standard approaches. To the best of our knowledge, this is the first work addressing the problem of scheduling DNNs in real-time systems under temporal isolation (achieved by resource reservation techniques). In addition, as the DNN workload generated by TensorFlow can be modeled by parallel task graphs with precedence constraints, this paper provides a practical and effective solution for handling such kind of tasks under resource reservation.

2 | TENSORFLOW INTERNALS AND BACKGROUND

TensorFlow is a machine learning framework developed by Google that is capable of both training and inferring DNNs, currently released for Linux, Windows, and MacOS. Each DNN is represented by a directed acyclic dataflow graph,[8] denoted in the following as *TensorFlow graph* (TFG), whose nodes represent the fundamental algorithmic and mathematical operations (eg, matrix multiplications and convolutions) needed to run the DNN. The core of TensorFlow is developed in C++. To infer a DNN, TensorFlow uses a specific C++ object, denoted as `Session`, which is in charge of executing the operations of the DNN by respecting the corresponding precedence constraints specified by the edges of the TFG. Each DNN operation is partitioned on the available devices [ie, groups of Central Processing Units (CPUs), Graphics Processing Unit (GPUs), etc.] and then internally dispatched by the selected device.

This work focuses on the case in which DNNs are inferred on a multicore CPU platform with m identical cores (ie, no interdevice partitioning is needed). Under this configuration, TensorFlow (by default) exploits the Eigen[9] mathematical library for implementing the DNN operations. To exploit multicore platforms, Eigen parallelizes the mathematical operations creating a large set of small sequential execution units, referred to as *elementary nodes*, which are subject to precedence constraints. Elementary nodes are non-preemptively scheduled by *work-first scheduling*[10] (WFS) using a thread pool. The specific implementation of WFS adopted in Eigen includes per-thread work queues. Elementary nodes are dispatched to such work queues as follows: (i) if an elementary node n has been created by another elementary node n', then n is inserted in the work queue of the thread that executed n'; otherwise, (ii) n is inserted in a randomly selected work queue. Then, each thread accesses its local queue in a first-in-first-out (FIFO) manner to select the next operation to execute. Eigen also employs a randomized *work-stealing* technique to ensure load balancing between the various threads of the pool: whenever a thread finds its queue empty, it steals work from the work queues of other randomly selected threads by accessing them in last-in-first-out order. The stolen work must correspond to nodes that are still waiting to be selected for execution (ie, not already started to execute). Migration is not supported because the elementary nodes are user-space entities (the work queues manage pointers to C++ functions).

Overall, note that TensorFlow generates two levels of concurrency: a first one to handle the operations of the TFG, and a second one to serve the execution of the elementary nodes created by Eigen (ie, dealing with the inner parallelism present in the mathematical operations). Each TensorFlow `Session` object instantiates two thread pools to handle such two levels, denoted as *inter-op* and *intra-op* thread pools, respectively. The threads of both thread pools are finally scheduled by the underlying operating system, which in our case is Linux. The TensorFlow runtime environment is shown in Figure 1, where only a single thread pool is considered for simplicity. By default, each pool comprises a number of threads equal to the number of available cores.

2.1 | An example: the InceptionV3 DNN

To better understand the structure of a complex DNN workload, the TFG resulting from a state-of-the-art DNN, named InceptionV3,[11] has been profiled on a 8-core Intel i7-6700K machine running at 3.5GHz and equipped with 32GB of RAM memory under TensorFlow v1.5. InceptionV3 is a popular image classifier DNN.

The resulting TFG is composed of 702 operations. At the Eigen level, the workload is further decomposed, reaching more than 34 000 elementary nodes. Figure 2 shows the histogram of the execution times of the elementary nodes of InceptionV3, which are very lightweight, as only about 400 of them (less than the 1.2% of the total nodes) reported execution times larger than 100 microseconds. The operations involved in InceptionV3 are also particularly memory-intensive: the amount of data exchanged (following a producer-consumer paradigm) between two nodes of the TFG memory varies from 0 (a simple precedence constraint) to 8.2 MB, with an average of 334 kB computed over 1806 TFG edges. Overall, the amount of data exchanged by the nodes of the TFG of InceptionV3 amounts to 603 MB.

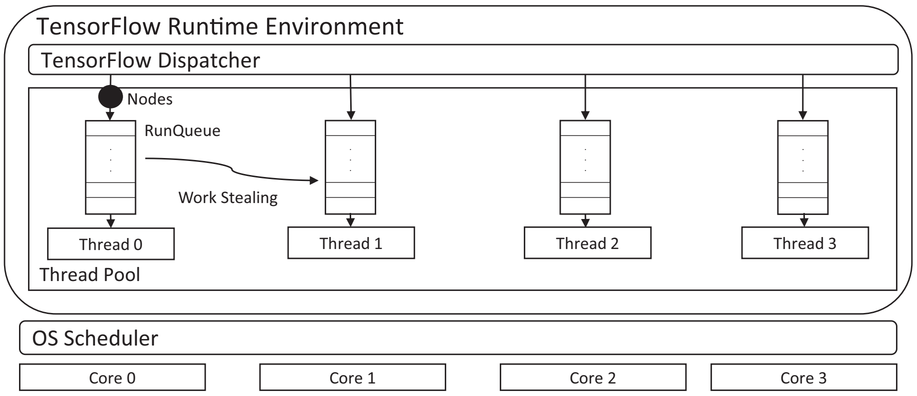


FIGURE 1 Illustration of the TensorFlow runtime environment with a single thread pool. Each thread disposes of a local queue. When the local queue is empty, threads can steal work from other queues. Threads are in turn scheduled upon the available cores by the operating system

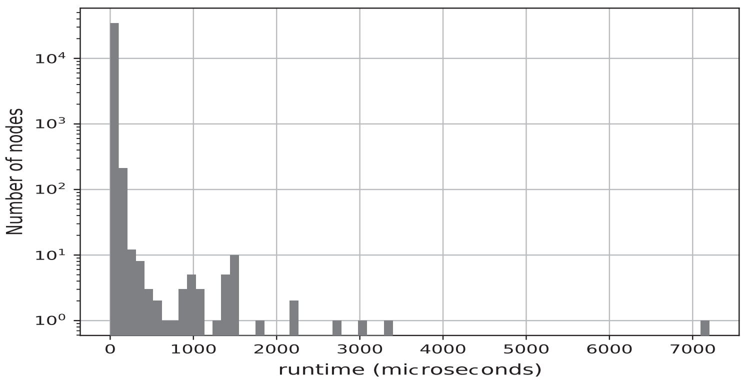


FIGURE 2 Histogram of the profiled node execution times of the InceptionV3 DNN. The *y*-axis has a logarithmic scale [Colour figure can be viewed at wileyonlinelibrary.com]

From the above results it is possible to derive two important observations.

Observation 1. Due to the large number of elementary operations dispatched by Eigen, state-of-the-art DNNs are likely to originate a very complex parallel workload and hence scheduling decisions may have a significant impact on their timing performance.

Observation 2. Due to the presence of memory-intensive workloads, scheduling decisions that privilege data locality may improve the timing performance of DNNs (eg, by reducing the delays related to cache misses).

2.2 | Linux and resource reservation

Since in the considered setting TensorFlow executes on Linux, it is important to briefly summarize the Linux scheduling capabilities. By default, Linux uses the Completely Fair Scheduler (CFS), which has the aim of maximizing both the overall CPU utilization and the performance of interactive tasks. Linux also includes other scheduling policies, such as fixed-priority, named `SCHED_FIFO` and `SCHED_RR`, and earliest deadline first (EDF), which is implemented by the `SCHED_DEADLINE` scheduling class.[12] `SCHED_DEADLINE` also implements a *resource reservation* mechanism by means of the Hard Constant Bandwidth Server (HCBS)[13] algorithm, which allows reserving, in a periodic fashion, a time budget of Q units every P units, to serve the execution of a given thread. In this way, threads are subject to *timing isolation*, in the sense that the response time of a thread is not affected by the timing behavior of other threads, but only depends on its own execution time and the amount of reserved budget. Such a scheduling mechanism is particularly useful to better control the timing performance of real-time tasks and shield the system from possible misbehaviors. Parameters Q and P are denoted as budget and period, respectively, and can be configured for each thread by means of a system call. Finally, it is worth mentioning that the Linux real-time performance can also be improved by applying the `PREEMPT_RT` patch,[14,15] a variant of Linux aimed at improving kernel latency by allowing the preemption of kernel functions.

3 | RELATED WORK

To our records, no other works focused on supporting the temporal isolation for DNN inference frameworks nor on extending their CPU scheduler to improve data locality.

The works related to this paper discussed in this section are classified in the following categories: (i) scheduling of deep neural networks and (ii) techniques for handling budget exhaustion of dependent reservations.

3.1 | Scheduling of DNN

Albaqsami et al[16] targeted the problem of optimizing the mapping of TensorFlow operations to computing devices for achieving a speed-up in the training phase. The authors used a genetic algorithm to determine whether to execute each operation on a CPU or a GPU, using different fitness functions. The operation-to-device partitioning does not require modifications to the TensorFlow internals, and it can be performed with the default Python API for TensorFlow. Zhou et al[17] proposed a pipeline scheduling solution aimed at optimizing the execution of DNN workload on GPUs, while Yang et al[18] identified a combination of techniques to support multiple cameras with an improved throughput in the context of automated-driving systems. In the context of mobile devices, Lane et al[19] proposed two runtime algorithms to decompose a DNN model across available processors with the purpose of improving performance and energy-efficiency. Very recently, a similar purpose has been pursued by Kang and Chung,[20] and Bateni et al.[21] Hong et al[22] presented an extended synchronous dataflow model aimed at explicitly expressing the parallelism of loop structures, allowing to model the computational graph of a DNN during the training phase. Casini et al[23] proposed approaches for bounding the worst-case response time of parallel tasks implemented with thread pools, using a task model inspired by Tensorflow. Bateni and Liu[24] explored methods for reducing the execution costs of DNN layers at the expense of a reduced accuracy. Finally, it is worth mentioning TensorRT,[25] a framework provided by NVIDIA to improve the performance-inference of DNNs on GPUs by implementing various optimizations, for example, quantizing floating-point numbers and merging successive layers. TensorRT is a proprietary product and its internal behavior is not publicly disclosed. Recently, much research has been devoted to improve performance and predictability when executing DNNs on GPU accelerators. For example, Forsberg et al[26] presented a mechanism to control the memory traffic on GPU-based SoCs (a prototype implementation has been developed on the Tegra TX1 platform by Nvidia). Similar mechanisms have been proposed by Capodieci et al[27] and Ali and Yun.[28] Capodieci et al[29] implemented the constant bandwidth server (also used in this paper to provide timing isolation of Tensorflow threads) for scheduling CUDA kernels on Nvidia GPUs. Unfortunately, their work is not publicly available.

3.2 | Techniques for handling budget exhaustion of dependent reservations

The problem of handling budget exhaustions for dependent reservations have been studied in the literature in the context of shared resources protected by mutexes. To the best of our knowledge, this is the first work considering issues related to budget exhaustion and precedence constraints in the context of DNN executed by pools of threads.

The budget overrun technique was first proposed by Ghalazie and Baker[30] in the context of aperiodic task scheduling with shared resources, and later analyzed by Davis and Burns[31] under fixed-priority scheduling and by Behnam et al[32,33] under EDF.

Davis and Burns[31] proposed two different approaches for budget overrun, *with* and *without* payback: in the first one if a reservation overruns by x units of time, its following instance is replenished by $Q - x$ units of budget, whereas, in the second approach, reservations are always replenished by Q. Due to its advantages in terms of system schedulability,[31,33] this paper considers a budget overrun mechanism with a payback policy.

Other techniques have been designed to reduce the delay caused by reservations with exhausted budget holding the lock of a shared resource. For instance, Lamastra et al[34] and De Niz et al[35] proposed budget inheritance mechanisms (also called *proxy execution*) for uniprocessor systems, which allow the reservation managing a task blocked on a shared resource to inherit the budget of another reservation that is waiting for the same resource. This mechanism has been later extended to multiprocessor systems, for example, by Faggioli et al.[36] Differently, the SIRAP[37] protocol performs a budget check every time a lock on a shared resource is requested: if the budget is enough to complete the critical section the lock is given to the requesting reservation, otherwise the access is granted only at the next budget replenishment. The BROE[38,39] protocol performs a similar budget check at each lock request, but, if the current budget is not enough to complete the execution of the critical section, the budget is replenished to the maximum value and the deadline is postponed proportionally.

A detailed discussion about which of these mechanisms best suits to the specific case of deep neural networks scheduled by pools of threads under reservation-based scheduling is postponed to Section 5.2.

4 | PROPOSED INFRASTRUCTURE

This work considers the case in which DNN tasks are executed together with other real-time control tasks on the same multicore platform. Real-time tasks are subject to firm real-time constraints (ie, no deadline miss should occur) and, to improve their execution predictability, they are statically allocated to cores. DNN tasks are considered as computational activities that generate a soft real-time workload: this is because TensorFlow is a very complex software that has not been designed to be predictable nor to guarantee temporal requirements. Therefore, the system is assumed to be able to tolerate deadline misses of DNN tasks, for example, if a deadline of a DNN task τ is violated, control tasks can use results computed by previous instances of τ or take decisions that are independent of the results produced by τ (such as triggering a backup control logic).

Nevertheless, DNN tasks may be requested to meet quality-of-service requirements, therefore they cannot be simply relegated as low-priority workload with respect to the real-time control tasks. For this reason, this work considers the more general case in which DNN tasks can generate temporal interference to real-time control tasks. Unfortunately, predicting the temporal interference generated by each DNN thread under the TensorFlow scheduler (described in Section 2.1) is very challenging, as the random dispatching and the stealing mechanism makes the load managed by each thread very difficult to be quantified. Again, this issue is due to the fact that the TensorFlow scheduler and its inference engine are not designed to be predictable.

A possible solution could consist of radically modifying TensorFlow and designing a suitable partitioning strategy for each elementary node, possibly integrating a timing analysis model to optimize the partitioning as a function of the workload running in the system. However, after a deep inspection of the TensorFlow code, it emerged that this solution would require an enormous effort, which can also introduce incompatibility issues with some class of usage of TensorFlow.

Furthermore, due to the large number of elementary nodes generated by state-of-the-art DNNs, and their specific characteristics, new models and optimization algorithms would also be required to efficiently support a predictable execution of DNN tasks.

Conversely, the solution proposed in this paper aims at introducing simpler but effective modifications to TensorFlow and leveraging the resource reservation mechanism offered by `SCHED_DEADLINE` to control the temporal interference generated by the TensorFlow threads. Specifically, each TensorFlow thread is protected by a reservation server with budget and period chosen at design time. Furthermore, to leverage the benefits offered by partitioned scheduling, with respect to the pitfalls and the lower predictability provided by global schedulers,[40-42] each TensorFlow thread is statically allocated to one of the available cores. Unfortunately, due to the presence of precedence constraints in the parallel workload originated by TensorFlow, the timing protection of the TensorFlow threads via resource reservation can originate performance degradation: this problem is addressed in Section 5, where modifications to both the TensorFlow scheduler and `SCHED_DEADLINE` are presented. Furthermore, when statically allocating TensorFlow threads to cores, it is also possible to improve the TensorFlow scheduler by modifying the work stealing policy in such a way that privileges data locality: this aspect is addressed in Section 6.

5 | TIMING ISOLATION FOR TENSORFLOW THREADS

5.1 | Motivational example

This section reports examples to illustrate the possible performance degradation introduced by a resource reservation mechanism that protects the threads of TensorFlow. Consider the directed graph illustrated in Figure 3 as an example of graph of elementary nodes created by Eigen.

First, consider the example depicted in inset (A) of Figure 4, which targets a dual-core processor computing platform where each core serves the execution of a thread of the intra-op pool. Figure 4A shows a possible execution trace for the considered graph, also showing the reservation budget as a function of time. As it can be noted, at time 7, the budget of thread th_2 exhausts, thus leaving the execution of node v_4 incomplete. Due to precedence constraints, the graph execution cannot proceed until v_4 is completed, hence it resumes its execution only when the budget of th_2 is recharged. In this case, the TensorFlow scheduler is not aware of the fact that a thread is suspended due to budget exhaustion, and the work stealing mechanism does not help to make progress, it only steals work that did not start executing.

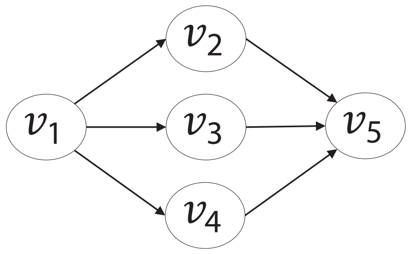


FIGURE 3 Example of a parallel task graph

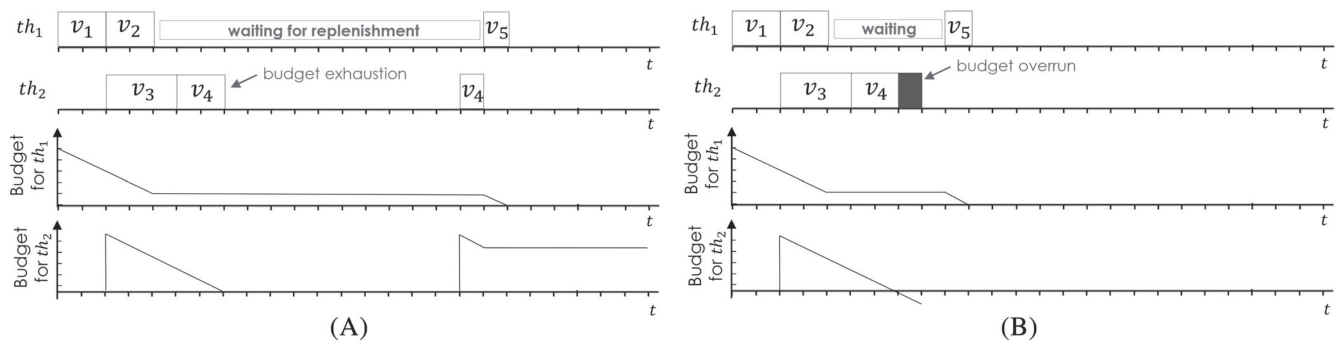


FIGURE 4 Example of a schedule of the parallel task shown in Figure 3 and the corresponding budget evolution with and without overrun [Colour figure can be viewed at wileyonlinelibrary.com]

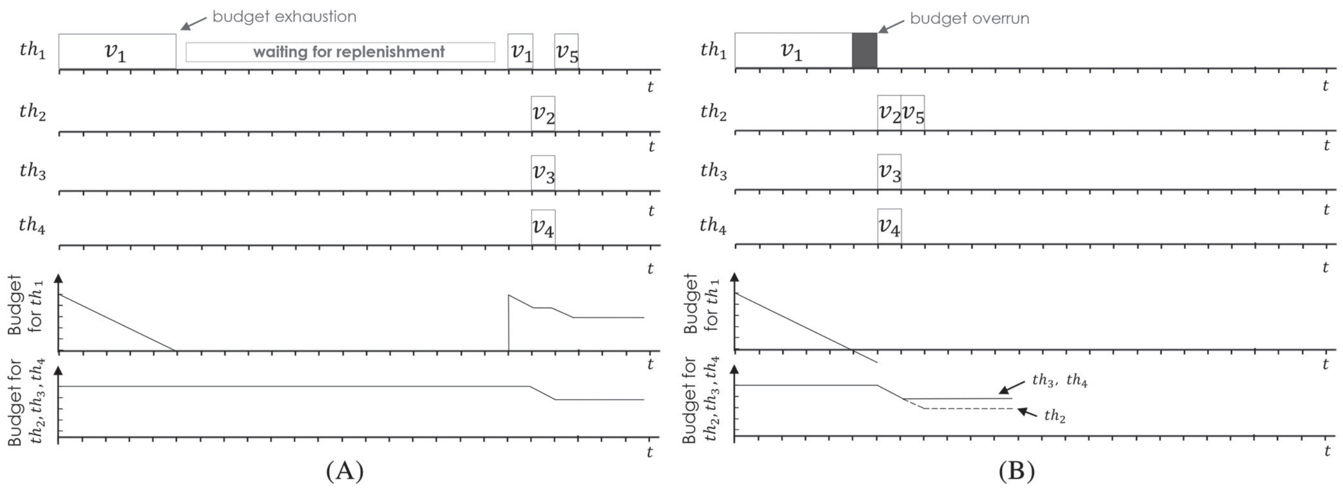


FIGURE 5 A second example of a schedule of the parallel task shown in Figure 3 and the corresponding budget evolution with and without overrun [Colour figure can be viewed at wileyonlinelibrary.com]

In general, this phenomenon originates a performance degradation, as multiple threads can be blocked by a single thread suspended for budget exhaustion, although there exists budget available to make progress in the execution of the graph.

A second example is shown in inset (A) of Figure 5, which targets a four-core processor platform, where again each core serves the execution of a thread of the intra-op pool. In this case, the budget exhaustion of th_1 prevents nodes v_2, v_3, and v_4 from being spawned, thus leaving the other threads (which all have available budget for executing) idle.

5.2 | Proposed solution

This section discusses about possible solutions to solve this problem. For instance, we may let the elementary nodes that remain incomplete due to budget exhaustion (v_4 in the example) to be served by another thread of the pool. However, this would require a migration of a C++ function that already started to execute, which is largely not supported by the TensorFlow scheduler (that runs in user space), as discussed in Section 2.

Killing the execution of a node and restarting it from scratch in another thread is also not practically viable, as it would pose additional constraints on the implementation of the operations performed by the nodes, which can make them not compatible with the new scheduler. For instance, some elementary nodes also create new nodes: a kill-and-restart behavior could hence generate duplicates.

Another alternative consists in allowing a suspended thread to inherit the budget from another reservation (possibly allocated in other cores). This solution is the one adopted by the so called proxy execution mechanism (see Section 3 for a more detailed discussion). However, this method is not supported by `SCHED_DEADLINE`, since it requires nontrivial efforts to be implemented, as noted in a recent discussion in the Linux Kernel Mailing List (LKML)[43] and by Parri et al.[44] Furthermore, the extension of proxy execution to handle the additional blocking due to precedence constraints and budget exhaustion is not straightforward. Indeed, in the case of shared resources, a reservation that exhausted its budget inside a critical section can inherit the budget of a dependent reservation that is blocked waiting for the mutex protecting the same resource. As it emerges from the examples in Figure 5A, it is difficult to determine the thread from which th_1 should inherit the budget, since none of the others have a task actually blocked by th_1 yet and, due to the work stealing mechanism, it is hard to predict which threads will actually run the nodes spawned by v_1. Conversely, the adoption of the SIRAP and BROE protocols (recalled in Section 3) would give rise to the need of checking whether the budget is sufficient to execute each node, hence requiring Tensorflow to know a priori the worst-case execution times of all the nodes, which are often not available when executing complex DNNs.

For the reasons explained above, this paper considers a simpler but effective solution, consisting in allowing budget overruns.[31] That is, if the budget is exhausted when a thread of the pool is serving a node, the execution of that node is continued until it is completed. In this way, the progress of the graph is not affected by the reservation mechanism. This case is illustrated in Figures 4B and 5B, where it is possible to observe that the graph has a shorter completion time with respect to the case without overrun illustrated in Figures 4A and 5A, respectively. For instance, Figure 5B shows how allowing budget overruns enables v_2, v_3, and v_4 to be spawned by v_1 before the next replenishment. In this way, th_2, th_3, and th_4 do not remain idle while waiting for the next replenishment time of th_1. Note that this solution does not prevent pending work in a suspended thread to be served by other threads (via work-stealing), as it is always due to nodes that did not start executing.

It is important to observe that an overrun violates the budget allocated to a thread. Nevertheless, as reported in Section 5.4, it is still possible to provide analytical guarantees for the timing constraints of real-time tasks executing in parallel with DNNs. Furthermore, it is worth noting that the longest amount of violation in a thread is equal to the execution time of the largest node served by it: fortunately (as reported in Section 2.1) a profiling of state-of-the-art DNNs revealed that *the duration of elementary nodes is typically very limited* (in the order of a few microseconds on a modern processor), hence the impact of this issue is also limited. This key observation motivated the adoption of a modified reservation scheduler that supports budget overruns by receiving signals from the TensorFlow scheduler to notify the start and the completion of nodes. Clearly, to guarantee a bounded overrun independently of the behavior of TensorFlow, the scheduler must also enforce a maximum overrun time, for example, to protect the system from a faulty node that takes a very large time to execute, or to penalize Tensorflow threads only in the rare cases in which large nodes are executed.

To the best of our knowledge this is the first work exploiting such techniques for improving the performance of DNNs (and parallel task graphs in general) under reservation-based scheduling.

5.3 | Implementation

In Linux, the budgeting mechanism for threads is implemented at the level of the operating system within the `SCHED_DEADLINE` scheduling class. After configuring the budgets and periods for the reservation servers (see Section 2.2), threads are subject to resource reservation and periodically de-scheduled by the operating system without receiving specific signals about budget exhaustions. Therefore, new system calls need to be implemented to disable and reenable the budgeting mechanism to allow budget overruns. For this reason, budget overruns could also introduce considerable additional run-time overhead. In principle, for each elementary node, the code that controls the work queues of the thread pools is required to invoke the operating system twice: one time to disable the budgeting mechanism before starting the execution of the node and another time to reenable it when the node is completed. To reduce such an overhead, this approach can be generalized by executing a certain number k of elementary nodes between the two system calls. That is, after notifying the kernel to disable the budgeting mechanism, multiple elementary nodes are executed before notifying the kernel to reenable it. As the number k increases, the run-time overhead decreases at the cost of increasing the overrun. The resulting logic is shown in the pseudocode reported in Algorithm 1, which is representative of the behavior of the code that controls the work queue in each thread of the pools created by TensorFlow.

Cyclically, each thread first tries to retrieve a function to execute from its own ready queue (line 6). If that queue is empty, it checks whether it is possible to retrieve workload from the queue of another thread (line 8). In this case, it disables the budget exhaustion, increments a variable c used for counting the number of elementary nodes consecutively executed with budget exhaustion disabled, and execute the node (lines 13:17). When the variable c is equal to k, the budget exhaustion is re-enabled (line 20).

At the Linux level, the `sched_setattr` system call has been modified by introducing an additional parameter to notify the disabling and enabling of the budget exhaustion. On the kernel side, the logic to react to such notifications has been implemented as follows: (i) when a budget exhaustion occurs, it takes effect only if the thread did not notify the kernel to disable it; (ii) independently of the notifications sent by the served thread, the kernel enforces a maximum budget overrun of δ time units, where δ is a configuration parameter of the kernel; (iii) when a thread notifies to reenable the budgeting mechanism, if the budget of the server is negative (ie, it was overrunning) then it is descheduled as for a regular budget exhaustion.

Algorithm 1. Pseudocode of the code that controls the work queues of the thread pools

```
procedure WorkerLoop($k$)
    $c \leftarrow 0$
    while DNN inference not terminated do
        $f \leftarrow$ NULL
        if work_queue is not empty then
            $f \leftarrow$ Pop_front(work_queue)
        else
            $f \leftarrow$ Work_stealing()
        end if
        if $f ==$ NULL then
            Suspend thread waiting for work
        else
            if $c == 0$ then
                Disable budget exhaustion
            end if
            $c \leftarrow c + 1$
            Execute $f$
            if $c == k$ then
                $c \leftarrow 0$
                Enable budget exhaustion
            end if
        end if
    end while
end procedure
```

5.4 | Guaranteeing the schedulability of real-time tasks

Thanks to Hard Constant Bandwidth Server (H-CBS) implemented by `SCHED_DEADLINE`, the proposed infrastructure allows providing analytical guarantees for the timing constraints of hard real-time tasks while isolating them from DNNs. In this way, both real-time tasks and DNNs are allocated to a reservation server with a period P and a budget Q.

For the sake of simplicity, this section assumes that task deadlines are equal to the corresponding period, and that real-time tasks execute synchronously with the corresponding reservations.[1]. Furthermore, we assume that real-time tasks do not self-suspend and do not share resources. Given a core p under analysis, the set of reservation servers assigned to real-time tasks and allocated to core p is denoted as $\mathcal{R}^{\mathrm{RT}}$, and the set of reservations assigned to DNNs and allocated to core p is denoted as $\mathcal{R}^{\mathrm{DNN}}$. For notational convenience, we omit the dependency of sets $\mathcal{R}^{\mathrm{RT}}$ and $\mathcal{R}^{\mathrm{DNN}}$ from the core p, and we refer for the entire section to a single core p under analysis.

Reservation servers scheduled under partitioned-EDF can be analyzed by means of the processor demand criterion[45] extended to handle *budget overrun with payback*.[33] According to this approach, the workload of a H-CBS reservation server can be modeled with a *demand-bound function* $dbf_i(t)$:

$$dbf_i(t) = \left\lceil \frac{t}{P_i} \right\rceil \cdot Q_i + O_i(t), \tag{1}$$

where $O_i(t)$ is a term related to the effect of the budget overrun. In Reference [33] $O(t)$ is defined as:

$$O_i(t) = \begin{cases} H & \text{if } t \geq P_i \\ 0 & \text{otherwise,} \end{cases} \tag{2}$$

[1]Both assumptions can easily be released: deadlines smaller than periods can naturally be handled by the processor-demand criterion[45] whereas tasks executing asynchronously with respect their reservation can be analyzed using approaches based on supply-bound functions.[46,47]

where H is the maximum budget overrun.

In this paper, the analysis proposed in Reference [33] is extended to consider the peculiarities of the proposed software infrastructure. First, note that only reservations $r_j \in \mathcal{R}^{\mathrm{DNN}}$ can overrun, that is, $\forall r_i \in \mathcal{R}^{\mathrm{RT}} \Rightarrow O_i(t) = 0$. Consequently, the demand bound function of a real-time task τ_i can simply be expressed as $\mathrm{dbf}_i^{\mathrm{RT}}(t) = \left\lceil \frac{t}{P_i} \right\rceil \cdot Q_i$. Conversely, reservations $r_j \in \mathcal{R}^{\mathrm{DNN}}$ can overrun, and the maximum budget overrun value depends on the parameter k reported in Algorithm 1.

The maximum overrun H reported in Equation (2) is bounded by the minimum of (i) the duration of the k longest nodes of the deep neural network, and (ii) the maximum budget overrun δ configured in the kernel, that is,

$$H_i(k) = \min\left(\delta, \sum_{c=1}^{k} e_{i,c}\right), \tag{3}$$

where $E = \{e_{i,1}, e_{i,2}, \dots\}$ is a sequence that contains the worst-case execution times of each node of the jth DNN, ordered by decreasing values. The demand bound function $\mathrm{dbf}_i^{\mathrm{DNN}}(t)$ of each reservation associated with a DNN thread is then expressed as in Equation (1).

With the definition of $\mathrm{dbf}_i^{\mathrm{RT}}(t)$ and $\mathrm{dbf}_i^{\mathrm{DNN}}(t)$ in place, each real-time task is guaranteed to complete within its deadline if:

$$\forall t \in \mathcal{D}, \sum_{r_i \in \mathcal{R}^{\mathrm{DNN}}} \mathrm{dbf}_i^{\mathrm{DNN}}(t) + \sum_{r_i \in \mathcal{R}^{\mathrm{RT}}} dbf_i^{\mathrm{RT}}(t) \ \leq \ t, \tag{4}$$

where $\mathcal{D} = \cup_{r_i \in \mathcal{R}^{\mathrm{DNN}} \cup \mathcal{R}^{\mathrm{RT}}} \{t = fP_i : t < L^* \wedge f \in \mathbb{N}_{\geq 0}\}$ and L^* is the length of the analysis interval, which can be computed[2] as in Reference [48].

To check the schedulability of all the real-time tasks allocated in the system, the test needs to be performed for each core $p = 1, \dots, m$.

6 | IMPROVING THE TENSORFLOW SCHEDULER

As emerged by profiling the InceptionV3 neural network (see Section 2.1), state-of-the-art DNNs are characterized by a memory-intensive workload. Nevertheless, the current TensorFlow scheduler employs a randomized work stealing to perform load balancing, taking decisions that may easily disrupt data locality, hence increasing the delays due to cache misses or communications between remote memories. These facts motivated us to modify the TensorFlow scheduler to improve the timing performance of DNN tasks by taking scheduling decisions that privilege data locality.

Under the proposed infrastructure, which statically allocates the TensorFlow threads to the available cores, it is possible to realize an improved work stealing policy by properly modifying the Tensorflow scheduler. The key idea consists of privileging work stealing from the work queues of threads that execute on "close" processors in the memory hierarchy, for example, a thread tends to privilege work stealing from the thread running upon the processor with which it shares the highest level of cache.

To this end, the TensorFlow scheduler has been enhanced by introducing the concept of *prioritized CPU groups*, which is represented by an ordered list CG of N^G sets of groups of cores. Formally, the i-th element of the list is a set $\{G_1^i, \dots, G_{N_i}^i\}$, where each element G_j^i represents a group of cores such that $\cap_{j=i}^{N_i} G_j^i = \emptyset$ (ie, a core can belong to at most one group), and $\cup_{j=i}^{N_i} G_j^i = C$, with C being the set of all cores on which the threads execute. By definition, the last set includes all the available cores, that is, $G_1^{N^G} = C$ and $N_{N^G} = 1$.

Leveraging CPU groups, TensorFlow has been modified to integrate a *localized work stealing* strategy, which is reported in Algorithm 2. Consider a thread running on core c that intends to perform work stealing. It explores the ordered list CG from $i = 1$ to $i = N^G$, and, for each priority level i, it tries to steal work from the threads associated to the cores into the group to which c belongs (lines 2 and 3). If this step succeeds, then the algorithm terminates (line 7); otherwise, if none of them has pending work to be stolen, then the algorithm advances the index i moving to the next set of groups. Like the original work stealing approach, this algorithm fails when no thread has work to be stolen (line 11).

[2]The only modification required consists in inflating the budget of each reservation $r_j \in \mathcal{R}^{\mathrm{DNN}}$ of $H_j(k)$ units of time to account for the overrun.

Algorithm 2. Pseudocode for localized stealing

```
procedure LocalizedStealing($c$)
    for $i = 1, \dots, N^G$ do
        cpu_group ← $\{G_j^i : c \in G_j^i \wedge j \in [1, N^G]\}$
        for each core $k \in$ cpu_group do
            $f$ ← Pop_back(work_queue($k$))
            if $f \neq$ NULL then
                return $f$
            end if
        end for
    end for
    return NULL
end procedure
```

The configuration of CPU groups is architecture-dependent and must reflect a priority list of cores from which it is convenient to steal work, given the memory hierarchy of the underlying platform. For instance, for a eight-core platform with (i) private per-core L1 caches, (ii) two shared L2 caches for the first and second four cores, respectively, and (iii) a common shared L3 cache, a natural configuration of CPU groups would be composed of two sets: priority $i = 1$, $\{G_1^1 = \{0, 1, 2, 3\}, G_2^1 = \{4, 5, 6, 7\}\}$, and priority $i = 2$, $G_1^2 = \{0, 1, 2, 3, 4, 5, 6, 7\}$[3]. This configuration prioritizes stealing between processors that share the same L2 cache. In the realized implementation, CPU groups can be configured for each TensorFlow `Session` object.

7 | EXPERIMENTAL RESULTS

This section reports an experimental study that has been conducted for evaluating the performance improvement that can be achieved by adopting the proposed infrastructure. The experimental study is divided in two parts both targeting two state-of-the-art DNN, namely MobileNetV2[49] and InceptionV3.[11] In the former, DNNs have been executed in isolation to test the effectiveness of the proposed techniques in improving the response times, while varying reservation budgets and periods. The latter focuses on a case-study consisting of the aforementioned DNNs and a set of real-time tasks selected from the TACLe benchmark.[50]

The experiments have been carried out on an 8-core Intel i7 machine running at 3.5GHz running Ubuntu Linux 16.04.4 LTS, and the proposed approaches have been implemented by modifying Tensorflow version 1.5.

For each DNN, the *inter-op* and *intra-op* thread pools have been configured with 8 threads each for both DNNs, pinning the ith thread of each pool to the ith CPU. For simplicity, all reservations have been assigned the same period P. A preliminary experimental study we conducted showed that the observed response time are less sensitive to the amount of budget assigned to threads in the *inter-op* pool; hence, we set a budget of $Q = 0.1 \cdot P$ in all the tested configurations. All the threads in the *intra-op* pool have been provided with the same budget, which is reported in the caption below each chart. Periods have been set to 2 seconds for both DNN activities. Note that the results in different charts are not directly comparable as they correspond to different runs.

Localized stealing CPU groups have been configured by grouping the cores according to the shared levels of caches of the adopted i7 processor, and resulted as follow: priority $i = 1$, $\{\{0, 1\}, \{2, 3\}, \{4, 5\}, \{6, 7\}\}$, and priority $i = 2$, $G_1^2 = \{0, 1, 2, 3, 4, 5, 6, 7\}$. In each configuration, the system has been run for 30 minutes, collecting the response times.

7.1 | Exploring period and budget assignments for DNN threads

In the first part of the experimental study, each DNN has been run in isolation while varying the period and budget assigned to each reservation. Figure 6 shows the cumulative distribution function (CDF) of the collected response times

[3]The numbers in the sets indicate the core identifiers.

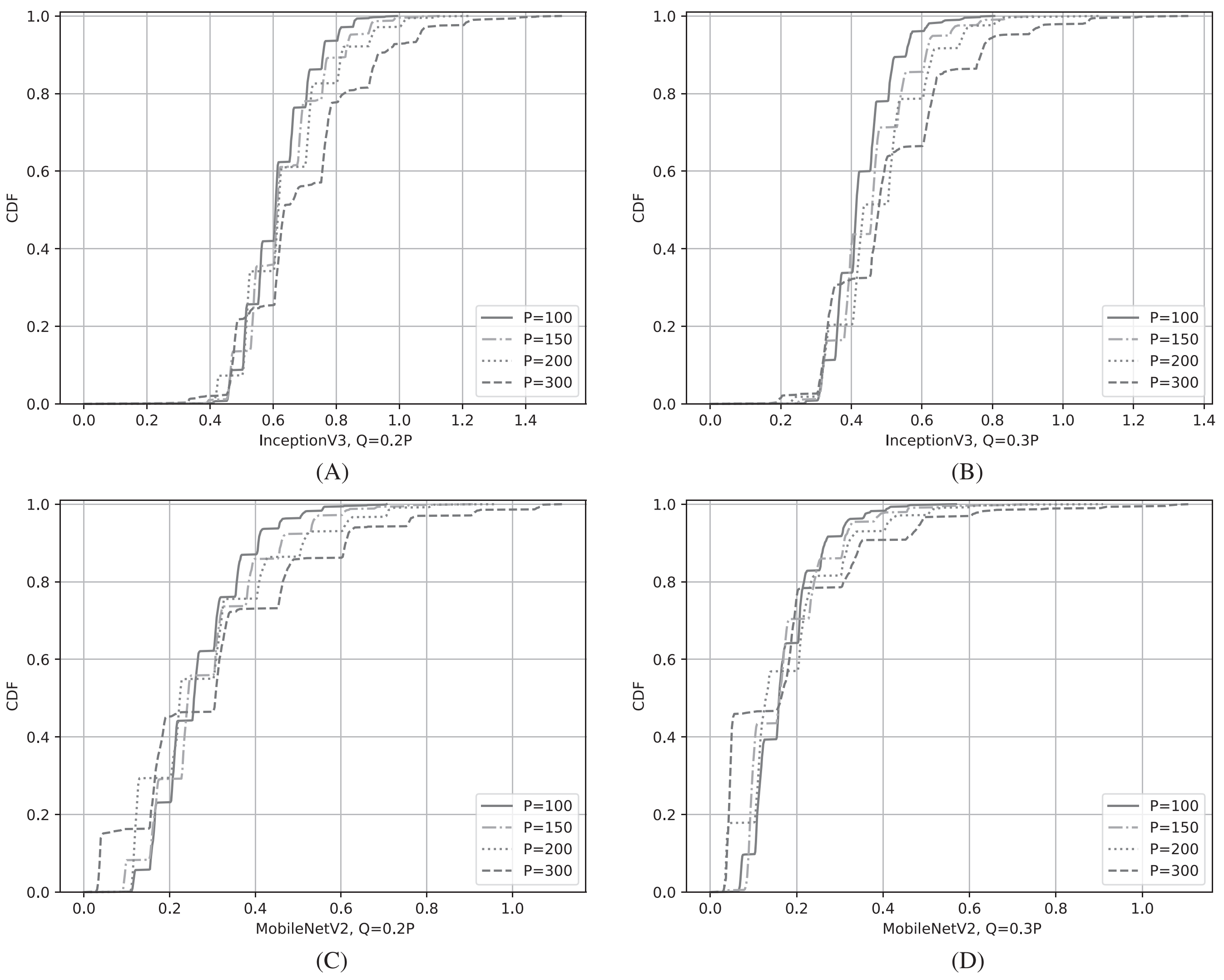


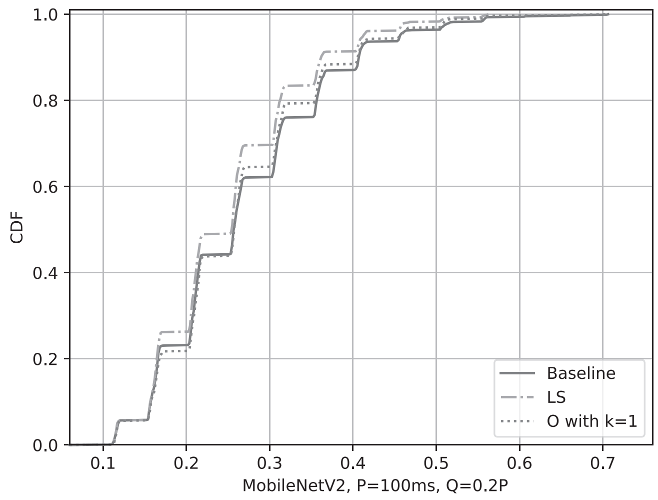
FIGURE 6 Cumulative distribution function of InceptionV3 (insets (A) and (B)) and MobileNetV2 (insets (C) and (D)) when the reservation period *P* varies. The budget assignment for threads in the intra-op pool is reported in the caption below each chart [Colour figure can be viewed at wileyonlinelibrary.com]

(normalized to the reservation periods) when the baseline approach is adopted (ie, `SCHED_DEADLINE` without budget overrun and localized stealing), showing the dependency from the reservation period. Figure 6 shows that a smaller period provides a faster response time, due to a more frequent budget provisioning, targeting InceptionV3 (insets (A) and (B)) and MobileNetV2 (insets (C) and (D)).

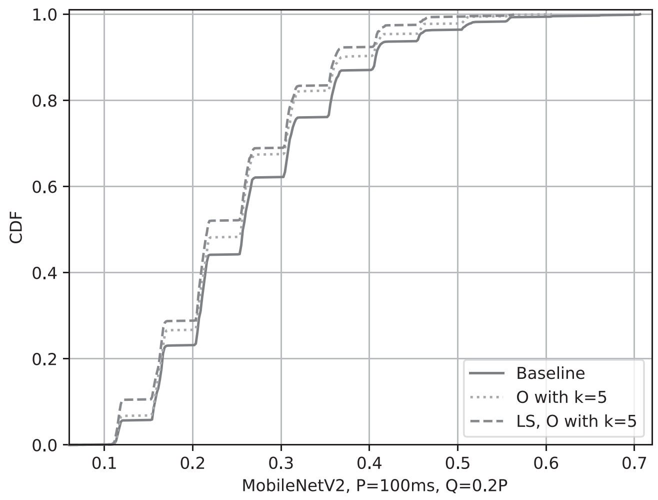
Figure 7 shows the improvement that can be obtained by adopting the proposed techniques. Budget overrun and localized stealing have been tested under different configurations by varying parameter k of Algorithm 1 in the set $\{1, 2, 3, 4, 5\}$. The maximum duration of any budget overrun δ has been set to 1 ms.

Figure 7A targets MobileNetV2 and illustrates that adopting localized stealing and budget overrun improves both average-case and worst-observed response times. Insets (B) and (E) target the same DNN with the same budget and period configuration, showing that setting $k = 5$ provides a bigger improvement. The improvement comes at the cost of having a higher overrun, which is anyway bounded by the maximum budget overrun δ. Figure 7C, D and F targets InceptionV3, showing similar results.

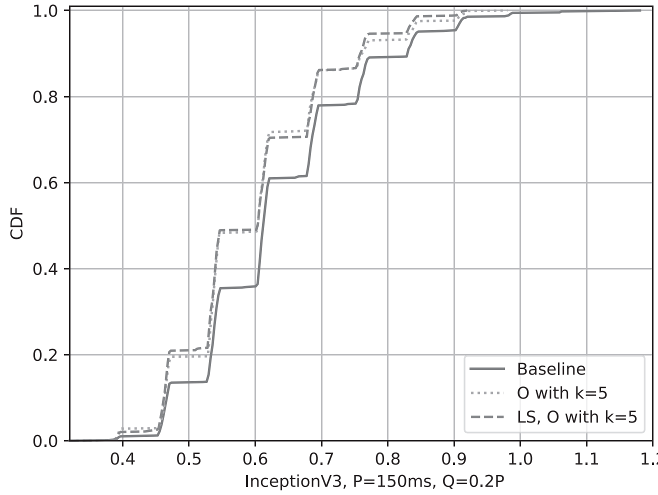
Next, we show the performance of the proposed approaches when executing the DNNs concurrently with a set of real-time tasks.



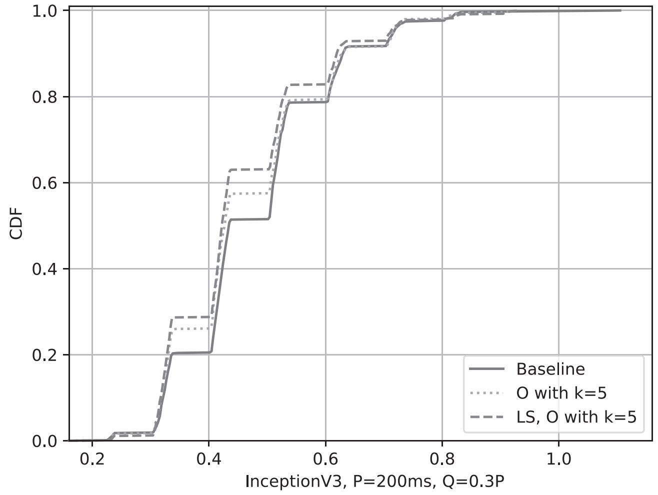
(A)



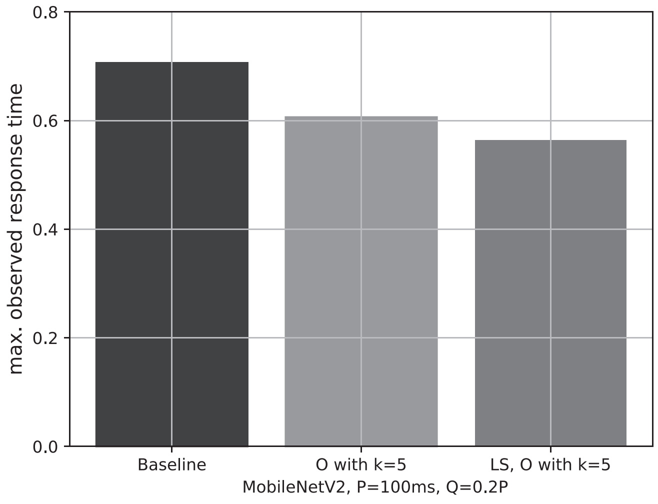
(B)



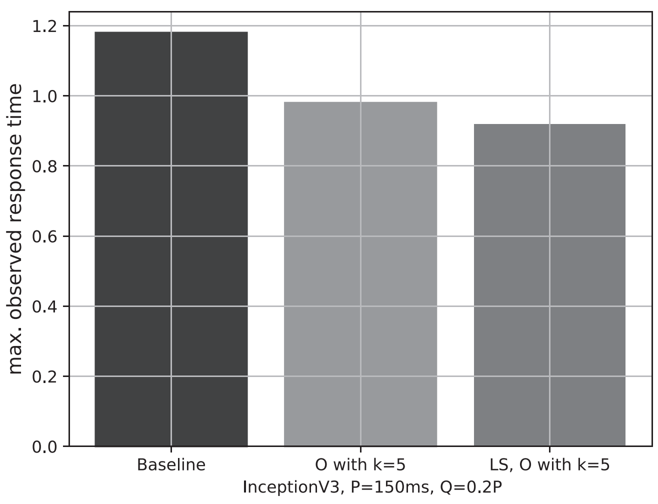
(C)



(D)



(E)



(F)

FIGURE 7 Cumulative distribution functions and maximum observed response times of InceptionV3 (insets (A), (B), and (E)) and MobileNetV2 (insets (C), (D), and (F)) when localized stealing (denoted as "*LS*") and budget overrun (denoted as "*O*") are adopted. The label "*LS, O*" denotes the case in which localized stealing and budget overrun are adopted in conjunction. The budget and period is reported in the caption below each chart [Colour figure can be viewed at wileyonlinelibrary.com]

TABLE 1 Real-time tasks selected from the TaCLe benchmark

Name	Description	Budgets (ms)	Periods (ms)	CPU
cjpeg_transupp	JPEG routines	20	110	0
mpeg2	Motion estimation	40	110	1
susan	Image processing	40	110	2
pm	Pattern match	25	110	3
st	Statistics	30	80	4
gsm_dec	GSM decoder	30	80	5
gsm_enc	GSM encoder	30	80	6
ammunition	Arithmetics	100	300	7

7.2 | Case study

This section reports the results of running InceptionV3 and MobileNetV2 concurrently with a set of real-time tasks chosen from the TACLe benchmark[50] for being representative of the workload of an industrial autonomous system (summarized in Table 1). Unfortunately, the benchmark did not report periods and deadlines for the real-time tasks. Hence, periods have been set as follows: (i) all the computations related to pattern matching and image processing (ie, tasks *cjpeg_transupp*, *susan*, *mpeg*, *pm*) are triggered every 110 ms, which corresponds to a frame rate of 9 fps (ie, the same rate used in the vision-based defect inspection system proposed by Zhou et al[51]), (ii) statistical data (ie, *st* task) are collected every 80 ms, and sent through a communication network at the same rate (ie, *gsm_enc* and *gsm_dec* tasks), and (iii) the industrial control system performs mathematical computation (eg, related to a control algorithm used to move an object in correspondence of a camera for image acquisition) every 300 ms. Real-time tasks have been statically allocated to cores as reported in Table 1, and their deadlines have been set equal to their periods. Each real-time task is periodically released and protected by a `SCHED_DEADLINE` reservation. Budgets (reported in Table 1) have been configured according to a preliminary experimental study aimed at evaluating the execution requirement of each task, while the reservation periods have been set equal to task periods. As in Section 7.1, the DNNs periods and deadlines have been set to 2 seconds, the maximum length of any budget overrun δ has been set to 1 ms.

Figure 8 reports the CDF of InceptionV3 when executed concurrently with the real-time tasks reported in Table 1. Inset (A) shows that both average-case performance and worst-observed response times are improved with respect to the case in which baseline approach is adopted, that is, `SCHED_DEADLINE` without budget overrun and localized stealing, up to 24% and 23% by adopting the proposed techniques, respectively. Insets (B), (C), and (D) report the CDF of the real-time tasks, showing that their average-case and worst-case observed performance are not considerably affected by the proposed approach. Similar results have been obtained for the other real-time tasks, which always completed within the deadline. Figure 9A reports the CDF on MobileNetV2, when executed concurrently with the real-time tasks reported in Table 1. Finally, Figure 9B compares the improvement in the worst observed response time of InceptionV3 and MobileNetV2 (called I and M in the column caption, respectively), which amounts to 22% and 30%, respectively.

Overhead introduced by the proposed approach.

The most computationally demanding part of our approach consists of calling the `sched_setattr` system call, which is required to enable and disable budget overrun (lines 13 and 20 in Algorithm 1). We measured the time required to execute such a system call by using the `perf`[52] tool of Linux. The measurements have been performed over a sample of 400,000 calls to `sched_setattr`. Each call required 0.503 microseconds on average and up to 18 microseconds (maximum observed value).[4] Despite these are relatively small values, reducing the overhead related to this system call may be a desirable option in some cases because the `sched_setattr` system call may be called several times. As mentioned in Section 5.3, this is the reason for which Algorithm 1 takes an integer k as a parameter (remember that it denotes the

[4]Note that the measurements performed by the `perf` tool may include possible interference due to interrupts: to the best of our knowledge, this is very difficult to avoid by using standard measurement tools.

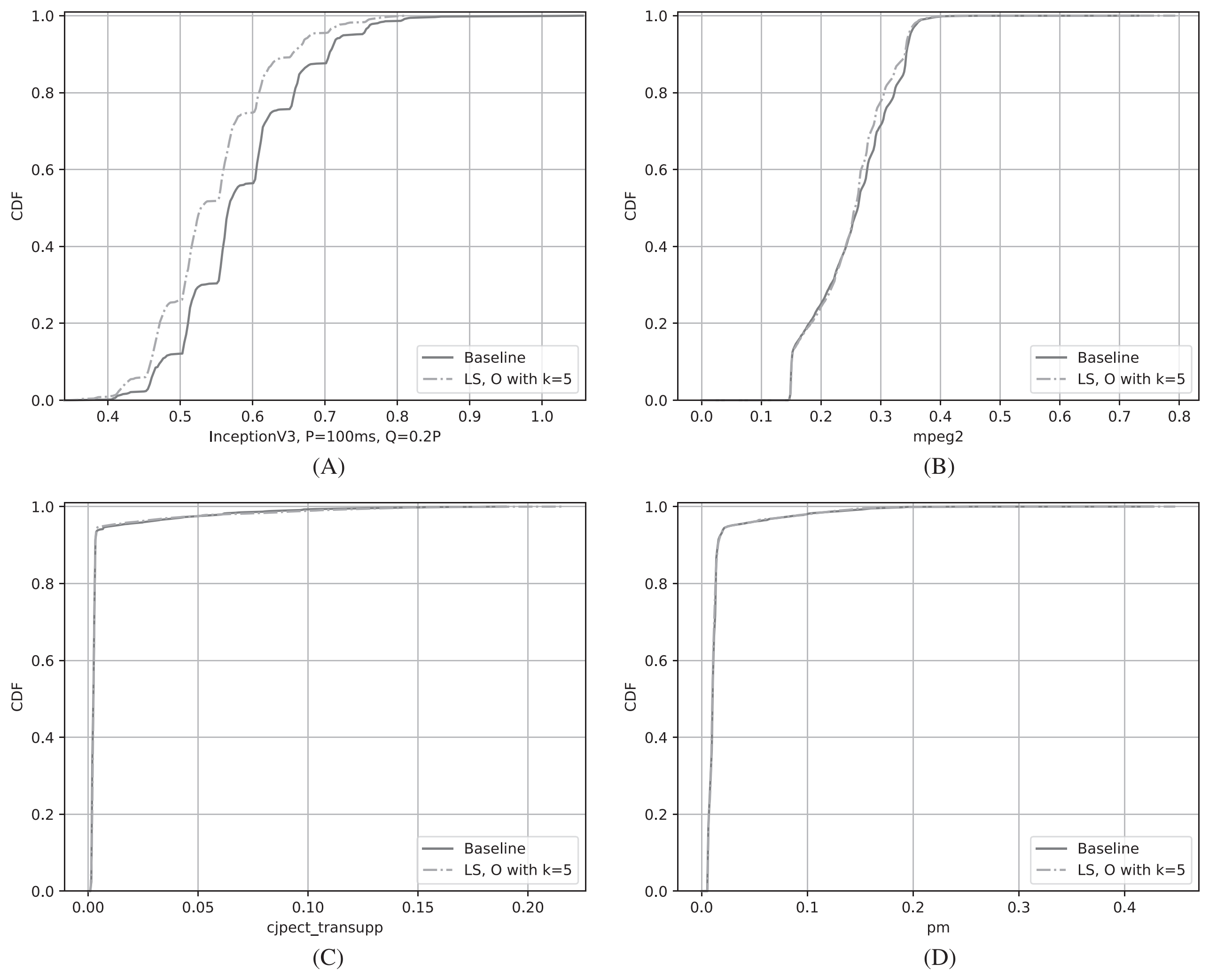


FIGURE 8 Cumulative distribution functions of InceptionV3 and real-time tasks when executed concurrently. The label "*LS, O*" denotes the case in which localized stealing and budget overrun are adopted in conjunction. The budget and period is reported in the caption below the graph [Colour figure can be viewed at wileyonlinelibrary.com]

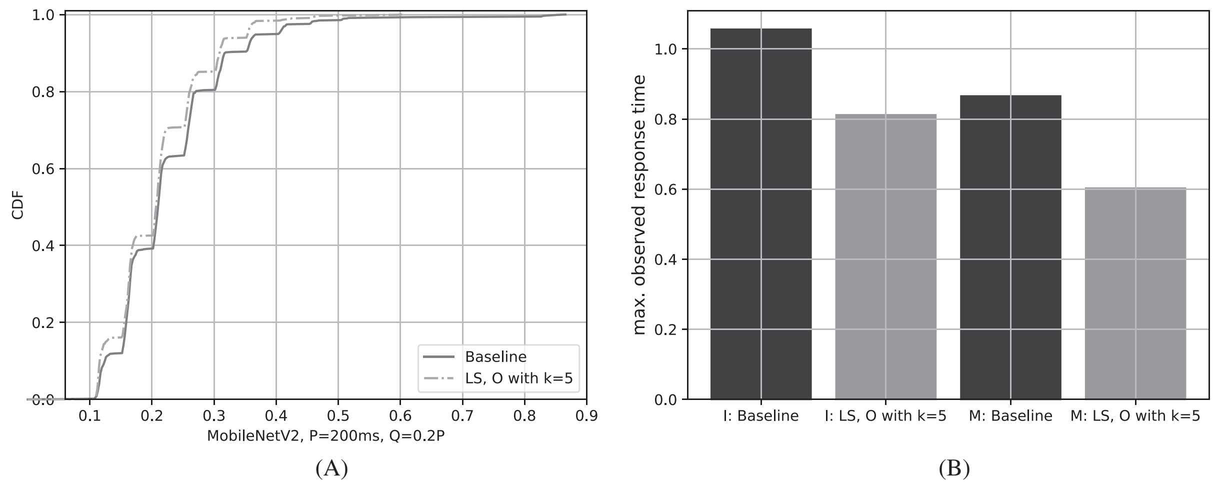


FIGURE 9 Inset (A) shows the cumulative distribution function of MobileNetV2 while inset (B) reports the maximum observed response times for InceptionV3, when each DNN executes concurrently with the real-time tasks. The label "*LS, O*" denotes the case in which localized stealing and budget overrun are adopted in conjuction [Colour figure can be viewed at wileyonlinelibrary.com]

number of nodes to execute between two system call invocations in the main loop of the algorithm). Given a DNN with n elementary nodes, the overhead introduced by the proposed approach due to the `sched_setattr` system call as a function of the parameter k can be estimated as

$$OV(k) = \left\lceil \frac{n}{k} \right\rceil \cdot x \cdot 2, \tag{5}$$

where x is the time required to perform a single call to `sched_setattr`. For example, as discussed in Section 2.1, the InceptionV3 DNN is composed of 34 000 nodes, that is, $n = 34\,000$. Considering the average execution time of `sched_setattr` of 0.5 microseconds, for $k = 1$ it follows that $OV(1) \simeq 34$ ms, while for $k = 5$ it holds $OV(5) \simeq 6.8$ ms. In the case study discussed above, these values imply an additional CPU utilization equal to 1.7% for $k = 1$ and 0.34% for $k = 5$. These measurements corroborate the choice of grouping the execution of multiple nodes between a pairs of calls to `sched_setattr` by controlling the parameter k in Algorithm 1 with the purpose of reducing the introduced overhead.

8 | CONCLUSIONS

This paper presented a scheduling infrastructure to support the execution of DNNs together with real-time tasks upon Linux. New mechanisms have been designed and implemented in TensorFlow and the Linux scheduler to ensure a proper temporal isolation of DNNs and improve data locality during the inference of DNNs. A case study based on a real implementation demonstrated improvements on both average-case (up to 24%) and longest-observed (up to 30%) response times of DNNs, with respect to the adoption of standard approaches. Possible future research directions target the design of predictable mechanisms for scheduling DNN on heterogeneous platforms, such as GPUs[29] and reprogrammable FPGAs,[53] and the extension of other protocols (eg, proxy execution[34,35] and BROE[38,39]) to handle budget exhaustion inside critical sections to parallel workloads scheduled by reservation servers.

ACKNOWLEDGEMENTS

The authors would like to thank Daniel Bristot De Oliveira and Luca Abeni for their advice in configuring and modifying the SCHED_DEADLINE scheduling class of Linux and in performing the overhead measurements. This work has been partially supported by the Department of Excellence in Robotics and Artificial Intelligence of Scuola Superiore Sant'Anna, Pisa, Italy.

ORCID

Daniel Casini https://orcid.org/0000-0003-4719-3631

REFERENCES

1. Chen C, Seff A, Kornhauser A, Xiao J. DeepDriving: learning affordance for direct perception in autonomous driving. Paper prresented at: Proceedings of the IEEE International Conference on Computer Vision (ICCV 2015). December 2-6, 2015; Araucano Park, Las Condes, Chile.
2. Lenz I, Lee H, Saxena A. Deep learning for detecting robotic grasps. *Int J Robot Res*. 2015;34(4-5):705-724.
3. Lenssen JE, Toma A, Seebold A, et al. Real-time low snr signal processing for nanoparticle analysis with deep neural networks. Paper presented at: Proceedings of the 11th International Joint Conference on Biomedical Engineering Systems and Technologies (BIOSTEC 2018). January 19-21, 2018; Funchal, Portugal.
4. Lin W, Ren X, Zhou T, Cheng X, Tong M. A novel robust algorithm for position and orientation detection based on cascaded deep neural network. *Neurocomputing*. 2018;308:138-146.
5. Russakovsky O, Deng J, Su H, et al. ImageNet large scale visual recognition challenge. *Int J Comput Vis (IJCV)*. 2015;115(3):211-252.
6. http://blog.paralleldots.com/data-science/must-read-path-breaking-papers-about-image-classification/.
7. Buttazzo GC. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. 3rd ed. New York, NY: Springer; 2011.
8. Abadi M, Agarwal A, Barham P, et al. TensorFlow: large-scale machine learning on heterogeneous distributed systems; 2015. http://download.tensorflow.org/paper/whitepaper2015.pdf.
9. Eigen Library. http://eigen.tuxfamily.org/index.php?title=Main_Page.
10. Min SJ, Iancu C, Yelick K. Hierarchical work stealing on manycore clusters. Paper presented at: Proceedings of the 5th Conference on Partitioned Global Address Space Programming Models (PGAS 2011); October 15-18, 2011; Tremont House, Galveston Island, TX.
11. Szegedy C, Vanhoucke V, Ioffe S, Shlens J, Wojna Z. Rethinking the inception architecture for computer vision. Paper presented at: Proceedings of the IEEE/CVF 29th Conference on Computer Vision and Pattern Recognition (CVPR 2016); June 26-July 1, 2016; Las Vegas, NV, United States.

12. Lelli J, Scordino C, Abeni L, Faggioli D. Deadline scheduling in the linux kernel. *Softw Pract Exp*. 2016;46(6):821-839.
13. Biondi A, Melani A, Bertogna M. Hard constant bandwidth server: comprehensive formulation and critical scenarios. Paper presented at: Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014); June 18-20, 2014; Pisa, Italy.
14. de Oliveira DB, de Oliveira RS. Timing analysis of the PREEMPT_RT linux kernel. *Softw Pract Exp*. 2016;46(6):789-819.
15. de Oliveira DB, de Oliveira RS, Cucinotta T. A thread synchronization model for the PREEMPT_RT linux kernel. *J Syst Arch*. 2020;107. https://www.sciencedirect.com/science/article/abs/pii/S1383762120300230?via=ihub.
16. Albaqsami A, Hosseini MS, Bagherzadeh N. HTF-MPR: a heterogeneous tensorflow mapper targeting performance using genetic algorithms and gradient boosting regressors. Paper presented at: Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE 2018); March 19-23, 2018; Florence, Italy.
17. Zhou H, Bateni S, Liu C. S3DNN: supervised streaming and scheduling for GPU-accelerated real-time DNN workloads. Paper presented at: Proceedings of the 24th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2018); April 11-13, 2018; Porto, Portugal.
18. M. Yang SW, Bakita J, Vu T, Smith FD, Anderson JH, Frahm J. Re-thinking CNN frameworks for time-sensitive autonomous-driving applications: addressing an industrial challenge. Paper presented at: Proceedings of the 25th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2019); April 16-18, 2019; Montreal, QC, Canada.
19. Lane ND, Bhattacharya S, Georgiev P, et al. DeepX: a software accelerator for low-power deep learning inference on mobile devices. Paper presented at: Proceedings of the 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN 2016); April 11-14, 2016; Vienna, Austria.
20. Kang W, Chung J. DeepRT: predictable deep learning inference for cyber-physical systems. *Real-Time Syst*. 2019;55(1):106-135.
21. Bateni S, Zhou H, Zhu Y, Liu C. PredJoule: a timing-predictable energy optimization framework for deep neural networks. Paper presented at: Proceedings of the 39th IEEE Real-Time Systems Symposium (RTSS 2018); December 11-14, 2018; Nashville, TN.
22. Hong H, Oh H, Ha S. Hierarchical dataflow modeling of iterative applications. Paper presented at: Proceedings of the 54th Annual Design Automation Conference (DAC 2017); June 18-22, 2017; Austin, TX, USA.
23. Casini D, Biondi A, Buttazzo G. Analyzing parallel real-time tasks implemented with thread pools. Paper presented at: Proceedings of the 56th Annual Design Automation Conference (DAC 2019); June 2-6, 2019; Las Vegas, NV.
24. Bateni S, Liu C. ApNet: approximation-aware real-time neural network. Paper presented at: Proceedings of the 39th IEEE Real-Time Systems Symposium (RTSS 2018); December 11-14, 2018; Nashville, TN.
25. TensortRT. https://developer.nvidia.com/tensorrt.
26. Forsberg B, Marongiu A, Benini L. GPUguard: towards supporting a predictable execution model for heterogeneous SoC. Paper presented at: Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE 2017); March 27-31, 2017; Lausanne, Switzerland.
27. Capodieci N, Cavicchioli R, Valente P, Bertogna M. SiGAMMA: server based integrated GPU arbitration mechanism for memory accesses. Paper presented at: Proceedings of the 25th ACM International Conference on Real-Time Networks and Systems (RTNS 2017); October 4-6, 2017; Grenoble, France.
28. Ali W, Yun H. Protecting real-time GPU kernels on integrated CPU-GPU SoC platforms. Paper presented at: Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS 2018); July 3-6, 2018; Barcelona, Spain.
29. Capodieci N, Cavicchioli R, Bertogna M, Paramakuru A. Deadline-based scheduling for GPU with preemption support. Paper presented at: Proceedings of the 39th IEEE Real-Time Systems Symposium (RTSS 2018); December 11-14, 2018; Nashville, TN.
30. Ghazalie TM, Baker TP. Aperiodic servers in a deadline scheduling environment. *Real-Time Syst*. 1995;9(1):31-67.
31. Davis RI, Burns A. Resource sharing in hierarchical fixed priority pre-emptive systems. Paper presented at: Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS 2006); December 5-8, 2006; Rio de Janeiro, Brazil.
32. Behnam M, Shin I, Nolte T, Nolin M. Scheduling of semi-independent real-time components: overrun methods and resource holding times. Paper presented at: Proceedings of the 2008 IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2008); September 15-18, 2008; Hamburg, Germany.
33. Behnam M, Nolte T, Sjodin M, Shin I. Overrun methods and resource holding times for hierarchical scheduling of semi-independent real-time systems. *IEEE Trans Ind Inf*. 2010;6(1):93-104.
34. Lamastra G, Lipari G, Abeni L. A bandwidth inheritance algorithm for real-time task synchronization in open systems. Paper presented at: Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001); December 3-6, 2001; London, UK.
35. de Niz D, Abeni L, Saewong S, Rajkumar R. Resource sharing in reservation-based systems. Paper presented at: Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001); December 3-6, 2001; London, UK.
36. Faggioli D, Lipari G, Cucinotta T. The multiprocessor bandwidth inheritance protocol. Paper presented at: Proceedings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS 2010); July 6-9, 2010; Brussels, Belgium.
37. Behnam M, Shin I, Nolte T, Nolin M. SIRAP: a synchronization protocol for hierarchical resource sharing in real-time open systems. Paper presented at: Proceedings of the 7th ACM & IEEE International Conference on Embedded Software (EMSOFT 2007); September 30-October 3, 2007; Salzburg, Austria.
38. Bertogna M, Fisher N, Baruah S. Resource-sharing servers for open environments. *IEEE Trans Ind Inform*. 2009;5(3):202-219.
39. Biondi A, Buttazzo GC, Bertogna M. Schedulability analysis of hierarchical real-time systems under shared resources. *IEEE Trans Comput*. 2016;65(5):1593-1605.
40. Brandenburg B, Gül M. Global scheduling not required: simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations. Paper presented at: Proceedings of the 37th IEEE Real-Time Systems Symposium (RTSS 2016); November 29-December 2, 2016; Porto, Portugal.

41. Casini D, Biondi A, Buttazzo G. Semi-partitioned scheduling of dynamic real-time workload: a practical approach based on analysis-driven load balancing. Paper presented at: Proceedings of the 29th Euromicro Conference on Real-Time Systems (ECRTS 2017); June 27-30, 2017; Dubrovnik, Croatia.
42. Biondi A, Sun Y. On the Ineffectiveness of 1/m-based interference bounds in the analysis of global EDF and FIFO scheduling. *Real-Time Syst*. 2018;54(3):515-536.
43. Linux kernel mailing list: towards implementing proxy execution. https://lkml.org/lkml/2018/10/9/431.
44. Parri A, Marinoni M, Lelli J, Lipari G. An implementation of a multiprocessor bandwidth reservation mechanism for groups of tasks. Paper presented at: Proceedings of the 16th Real Time Linux Workshop (RTLWS 2014); October 12-13, 2014; Dusseldorf, Germany.
45. Baruah SK, Rosier LE, Howell RR. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Syst*. 1990;2(4):301-324.
46. Shin I, Lee I. Periodic resource model for compositional real-time guarantees. Paper presented at: Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003); December 3-5, 2003; Cancun, Mexico.
47. Casini D, Abeni L, Biondi A, Cucinotta T, Buttazzo G. Constant bandwidth servers with constrained deadlines. Paper presented at: Proceedings of the 25th ACM International Conference on Real-Time Networks and Systems (RTNS 2017); October 4-6, 2017; Grenoble, France.
48. Zhang F, Burns A. Schedulability analysis for real-time systems with EDF scheduling. *IEEE Trans Comput*. 2009;58(9):1250-1258.
49. Sandler M, Howard A, Zhu M, Zhmoginov A, Chen L. MobileNetV2: inverted residuals and linear bottlenecks. Paper presented at: Proceedings of the IEEE/CVF 31th Conference on Computer Vision and Pattern Recognition (CVPR 2018); June 18-23, 2018; Salt Lake City, UT.
50. Falk H, Altmeyer S, Hellinckx P, et al. TACLeBench: a benchmark collection to support worst-case execution time research. Paper presented at: Proceedings of the 16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016); July 5, 2016; Toulouse, France.
51. Zhou Q, Chen R, Huang B, Liu C, Yu J, Yu X. An automatic surface defect inspection system for automobiles using machine vision methods. *Sensors*. 2019;19(3):644.
52. Linux kernel profiling with perf. https://perf.wiki.kernel.org/index.php/Tutorial.
53. Biondi A, Balsini A, Pagani M, Rossi E, Marinoni M, Buttazzo G. A framework for supporting real-time applications on dynamic reconfigurable FPGAs. Paper presented at: Proceedings of the 37th IEEE Real-Time Systems Symposium (RTSS 2016); November 29-December 2, 2016; Porto, Portugal.

How to cite this article: Casini D, Biondi A, Buttazzo G. Timing isolation and improved scheduling of deep neural networks for real-time systems. *Softw Pract Exper*. 2020;50:1760–1777. https://doi.org/10.1002/spe.2840