Partitioned Scheduling and Parallelism Assignment for Real-Time DNN Inference Tasks on Multi-TPU

[1]Binqi Sun, [2]Tomasz Kloda, [3]Chu-ge Wu, [1]Marco Caccamo
[1]TUM School of Engineering and Design, Technical University of Munich [2]LAAS-CNRS, Université de Toulouse, INSA
[3]School of Automation, Beijing Institute of Technology
binqi.sun@tum.de,tkloda@laas.fr,wucg@bit.edu.cn,mcaccamo@tum.de

ABSTRACT

Pipelining on Edge Tensor Processing Units (TPUs) optimizes the deep neural network (DNN) inference by breaking it down into multiple stages processed concurrently on multiple accelerators. Such DNN inference tasks can be modeled as sporadic *non-preemptive gangs* with execution times that vary with their parallelism levels. This paper proposes a *strict partitioning* strategy for deploying DNN inferences in real-time systems. The strategy determines tasks' parallelism levels and assigns tasks to disjoint processor partitions. Configuring the tasks in the same partition with a uniform parallelism level avoids scheduling anomalies and enables schedulability verification using well-understood uniprocessor analyses. Evaluation using real-world Edge TPU benchmarks demonstrated that the proposed method achieves a higher schedulability ratio than state-of-the-art gang scheduling techniques.

1 INTRODUCTION

Pipeline parallelism can improve the performance of deep neural network (DNN) inference on embedded machine-learning co-processors by partitioning the layers of a model into multiple stages that can be processed in parallel. Edge Tensor Processing Unit (TPU), a custom ASIC designed by Google for accelerating DNN inferences on edge devices, harnesses pipelining in a twofold way. Firstly, the parallel execution of multiple segments increases the processing throughput. Secondly, spreading large DNN models across multiple TPUs results in a larger share of the model parameters stored in their internal memories [10] and, thus, faster execution speed [19].

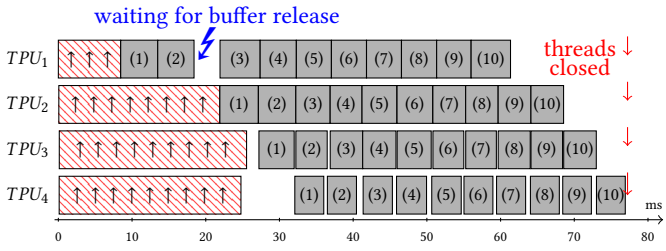


Figure 1: Pipelined execution of ten frames on four TPUs.

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06
https://doi.org/10.1145/3649329.3655979

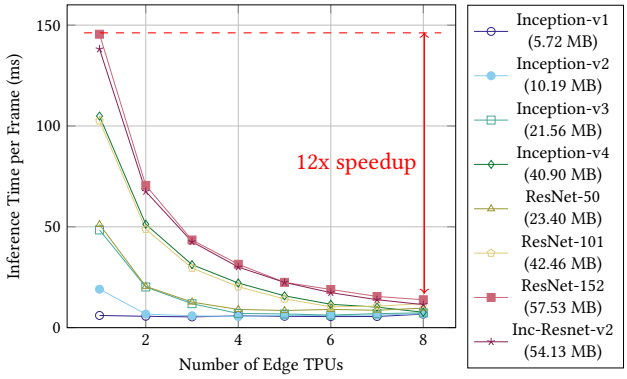


Figure 2: Multi-TPU DNN inference benchmarks executed on ASUS AI Accelerator CRL-G18U-P3D with 8 TPUs.

Fig. 1 shows the real trace of processing ten input frames by an image recognition network `inception-v3` on four TPUs. The model is divided into four segments so each TPU can run a segment in parallel. Before the inference starts, the model parameters are loaded to TPUs' on-chip memory (red boxes with ↑), which incurs an overhead ranging from 8 to 25 ms. Subsequentially, the frames are processed (gray boxes) in a pipeline: when one TPU finishes processing a frame, it saves the intermediate result to the input buffer of the next TPU and starts processing a new frame.

Integrating DNN accelerators in real-time systems, where tasks are subject to timing constraints and must be executed at a specific rate or in response to recurrent events, needs a scheduling model that leverages their parallelism and limits the reprogramming overhead (*e.g.*, DNN parameter loading). In this paper, we argue that both measures can be achieved using *non-preemptive gang* (NPG) scheduling. A NPG task spawns threads that execute in parallel and uninterruptedly (avoid reprogramming) on distinct processing units. All threads of the same job, for the sake of pipeline parallelism, start and end synchronously[1]. The gang tasks also need the parallelism level to be set. Fig. 2 shows the benchmarking results of the DNN inference time *w.r.t* the number of TPUs used by the network. Having multiple versions of the same model with different parallelism levels is not desirable due to the memory footprint (*e.g.*, 57.53 MB for each ResNet-152 version), and thus, in practice, only one parallelism level selected at design time is used.

In what follows, we propose a new partitioning and parallelism configuration strategy, compatible with standard fixed-priority schedulers, for deploying real-time DNN inferences on multi-TPU accelerators. This paper makes the following contributions:

[1]The current multi-TPU runtime library `libcoral` starts and ends all the threads on all the TPUs in the pipeline synchronously without allowing preemption.

- We present DNN inference benchmarks on multi-TPU accelerators and model the multi-TPU DNN inferences as NPG tasks.
- We propose an NPG strict partitioning strategy for scheduling DNN tasks on multi-TPU and a strict partitioning heuristic to determine the processor partitioning and task assignment.
- We compare the schedulability performance of the proposed strict partitioning strategy with two state-of-the-art global NPG response time analyses and federated scheduling.

The remaining of the paper is organized as follows. Section 2 covers the related work, and Section 3 introduces the task and platform model. Section 4 opens with motivation examples and then presents the NPG strict partitioning strategy. Section 5 proposes a strict partitioning heuristic. Section 6 contains evaluations of the proposed strategy, and Section 7 concludes the paper.

2 RELATED WORK

Partitioned scheduling of sequential tasks. Partitioned schedulers incur low runtime overheads but require solving a task-to-processor bin-packing problem. Since the partitioning problem is NP-hard in the strong sense [7], approximation methods were proposed for *sequential* preemptive [3] and non-preemptive [8, 20] tasks. However, these methods cannot be directly applied to the NPG tasks considered in this paper, as additional decisions (*i.e.*, processor partitioning and task parallelism configuration) need to be made.

Global gang scheduling. There have been several major works on gang scheduling, especially for preemptive systems [1, 5, 11]. For non-preemptive systems, Nelissen et al. [15] propose a response time analysis under the global fixed-priority policy for moldable gang tasks with a periodic time-triggered activation model. In this work, we consider an event-triggered task model (sporadic) that can react more promptly to online events. Such an activation model is also used in [12, 17], which propose response time analyses for the global non-preemptive fixed-priority scheduling of rigid gang tasks. We compare the schedulability performance of these works with our proposed strategy in Section 6.

Partitioning of parallel tasks. The idea of strict partitioning was first introduced in [16], which proposes a first-fit strict partitioning heuristic with the assumption that task parallelisms are fixed in advance. The NPG strict partitioning heuristic proposed in this paper extends the previous work by supporting processor partitioning, task assignment, and parallelism selection simultaneously. Ueter et al. [18] introduce the concept of *stationary gang* assignment, where rigid gang tasks are statically mapped to processors statically. However, in contrast to strict partitioning, the stationary gang assignment allows processors to be shared among tasks with different parallelism levels. As the paper considers *fully preemptive* scheduling, the analysis of priority inversion is not presented, and the method cannot be applied to NPG tasks. *Federated scheduling* [14] assigns dedicated processors to heavy utilization tasks and uses a global scheduler for the remaining light utilization tasks. By contrast, strict partitioning only uses uniprocessor schedulers, and each processor partition can be shared by multiple tasks. We will compare the performance of the proposed NPG strict partitioning method with a first-fit strict partitioning and a federated scheduling heuristic in Section 6.

3 TASK AND PLATFORM MODEL

We consider a multiprocessor platform Π comprised of M identical processors. A task set τ is comprised of n independent sporadic NPG tasks executing on Π. Each task $\tau_i \in \tau$ ($1 \leq i \leq n$) gives rise to an infinite sequence of jobs with consecutive jobs' invocations (arrivals) separated by at least T_i time units (*i.e., sporadic* task). We use J_i to denote a job of task τ_i. Job J_i released at time (arrival time) r_i has an absolute deadline r_i+D_i and must complete its execution by that time where $D_i \leq T_i$ (*i.e.,* constrained deadlines). Each job of task τ_i executes simultaneously on m processors for at most $C_{i,m}$ time units ($1 \leq m \leq M$) without interruption. As a result, a NPG task can be characterized by a 3-tuple $\tau_i = (C_i, T_i, D_i)$, where C_i is a vector of M elements and its m-th element $C_{i,m}$ denotes τ_i's worst-case execution time (WCET) with a degree of parallelism of m. Without loss of generality, we assume that all the above parameters are non-negative integers. We also assume that the tasks do not self-suspend and no task can be blocked by another task other than due to contention on processors. We define the utilization of a task τ_i as $U_{i,m} = C_{i,m} \cdot m/T_i$, and the reference task set utilization as the sum of task utilization with a parallelism level of 1 (*i.e.*, $U = \sum_{i=1}^{n} U_{i,1}$).

4 NPG STRICT PARTITIONING

We introduce our approach for scheduling a set of NPG tasks upon a set of identical processing units. We will first present its main concepts step by step with three motivation examples in Section 4.1, and then propose its formal definition in Section 4.2. We assume that the jobs are scheduled by a non-preemptive fixed-priority (NP-FP) scheduler and indexed in decreasing priority order (*i.e.,* job J_1 has the highest priority). For ease of presentation, we show the motivation examples with aperiodic jobs, but the approach can be generalized to periodic tasks without loss of generality.

4.1 Motivation examples

4.1.1 Avoid unbounded priority inversion. We recap a phenomenon in global NPG scheduling called *2-D blocking* [6].

Motivation Example 4.1. Consider eight aperiodic jobs J_{1-8} that execute non-preemptively on two identical processors. J_1 must run on two processors simultaneously (*i.e.,* it can be scheduled when there are at least two idle processors, and no higher-priority job is pending), while all other jobs can run on any single processor. J_3 and J_8 have execution times of 2 and all other jobs of 4. A priority inversion occurs when a higher-priority job is released during the execution of a job with lower priority and waits for its completion. For instance, all lower-priority jobs can be released one clock tick before J_1. A work-conserving scheduler always tries to keep the processors busy, and whenever a processor becomes idle, a pending job that can run on idle processors is scheduled instantaneously. While it results in good resource utilization, it can also lead to long priority inversions. As shown in Fig. 3, when the start and finishing times of old and new "thin" jobs overlap, the new "thin" jobs can get ahead of a pending "thick" job (*i.e.,* J_1 cannot start as long as two processors are busy).

To avoid unbounded priority inversion, one can assign the same parallelism to all jobs so the priority inversion will be bounded by the duration of the longest lower-priority job (*e.g.,* in the above

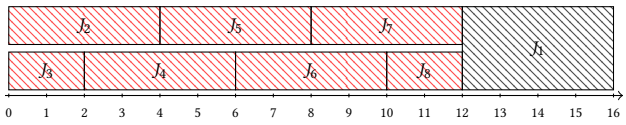
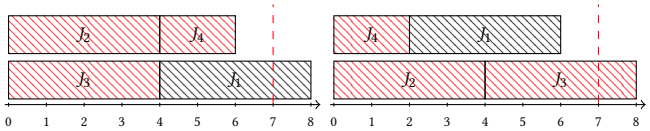


Figure 3: 2-D blocking in global NPG scheduling.

example, all jobs can be set to run on two processors). However, different jobs may prefer different parallelism, and such an approach might lead to processor underutilization. Therefore, we propose to divide the set of jobs into different subsets, where each job has the same degree of parallelism, and assign each subset of same-parallelism jobs to an exclusive subset of processors. That is, the jobs with different parallelism levels do not run on the same processor.

4.1.2 Advantages of partitioned scheduling. Now, the question is how to schedule a set of same-parallelism jobs on a set of processors. Two main approaches are used: *partitioned* (*i.e.,* tasks are statically allocated to processors) and *global* (*i.e.,* tasks can execute on any preassigned processor) scheduling, which are not comparable in terms of schedulability (*i.e.,* there are tasks schedulable with global but cannot be scheduled with any partitioned approach and conversely) [13]. In this work, we apply partitioned scheduling as it allows better control of low-priority blocking (see the example below). Moreover, partitioned scheduling allows us to use the exact uniprocessor non-preemptive response time analysis [4].

Motivation Example 4.2. Consider four jobs J_{1-4}, with execution times of 4 for J_{1-3} and 2 for J_4. The first job has a deadline fixed to 7, and other jobs have no deadline. Fig. 4 illustrates (a) global and (b) partitioned scheduling policies. In this example, by assigning J_1 and J_4 to the same processor (*i.e.,* partitioned scheduling), the blocking suffered by the most urgent job can go down from 4 to 2.

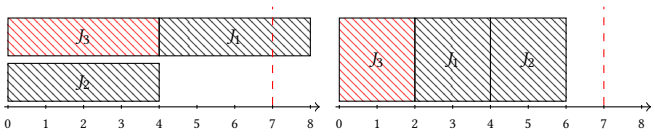


(a) Global scheduling. **(b) Partitioned scheduling.**

Figure 4: Priority inversion in non-preemptive scheduling.

4.1.3 Benefit of large partitions. The above considerations might suggest that the more partitions we create, the less low-priority job blocking we can have. However, forming fewer large partitions can also be beneficial.

Motivation Example 4.3. Consider three aperiodic jobs J_{1-3} that execute non-preemptively on two identical processors. All three jobs have the same degree of parallelism and execution times: 4 when running on a single processor and 2 when running concurrently on two processors. J_1 and J_2 must complete by their deadline fixed to 7, and J_3 has no deadline. Fig. 5 illustrates the critical instant for the first two jobs when (a) J_1 and J_3 are allocated to the first processor, and J_2 to the second one, and (b) all three jobs are allocated to two processors. In (a), there is no schedulable assignment for jobs partitioned across different processors, while in (b), all jobs are schedulable with at least one additional slack time. Processor utilization remains the same in both cases (*i.e.,* 12 units of processing time in total) with no speedup from the higher degree of parallelism, but the J_3's lower-priority (LP) blocking becomes less of a factor when spread across two processors.



(a) More small partitions. **(b) Fewer large partitions.**

Figure 5: Reducing LP blocking by large partitions.

4.2 Our approach – strict partitioning

Based on the above observations, we propose a new scheduling strategy called *NPG strict partitioning*. Its main idea is to divide the processors and tasks into disjoint subsets (partitions) and only allow tasks with the same parallelism to execute on each partition. Given a set of processors Π and NPG tasks τ, two *offline* decisions need to be made for strict partitioning:

- *Processor Partitioning* divides the set of processors Π into a set of *disjoint* processor partitions $\rho = \{\rho_j^m \subseteq \Pi, \forall m, j\}$, where j is the index of processor partition ρ_j^m with size $|\rho_j^m| = m$. Formally, $\rho_{j_1}^{m_1} \cap \rho_{j_2}^{m_2} = \emptyset, \forall \rho_{j_1}^{m_1} \neq \rho_{j_2}^{m_2}$ and $\Pi = \bigcup_{\forall m,j} \rho_j^m$.
- *Task Assignment* is a mapping from each processor partition $\rho_j^m \in \rho$ to a set of NPG tasks $\tau(\rho_j^m) \subseteq \tau$. We denote $\tau(\rho) = \{\tau(\rho_j^m), \forall \rho_j^m \in \rho\}$. Each task is assigned to one and only one processor partition, thus $\tau(\rho_{j_1}^{m_1}) \cap \tau(\rho_{j_2}^{m_2}) = \emptyset, \forall \rho_{j_1}^{m_1} \neq \rho_{j_2}^{m_2}$ and $\tau = \bigcup_{\forall m,j} \tau(\rho_j^m)$. All the tasks in $\tau(\rho_j^m)$ are configured to have the same parallelism level m.

After a valid offline partitioning, the tasks assigned to each processor partition are scheduled by an *online* NP-FP scheduler. Since each task in $\tau(\rho_j^m)$ has a parallelism level of m, it requires to occupy all the m processors in ρ_j^m to start its execution. This implies that at most one task in $\tau(\rho_j^m)$ can execute on ρ_j^m at each time instant. Therefore, the online scheduler of each partition boils down to a uniprocessor NP-FP scheduler, and its schedulability can be verified by a uniprocessor NP-FP schedulability test.

In this paper, we adopt the exact NP-FP response time analysis [4] as our uniprocessor schedulability test. The analysis verifies the schedulability of each task in the task set. For each task τ_i, it calculates the latest starting time of the l-th job to check whether the job can start C_i time units before its deadline. The job's latest starting time is given by the smallest positive integer satisfying the following equation:

$$s_{i,l} = \max_{\tau_j \in lp(i)} C_j + (l-1) \cdot C_i + \sum_{\tau_j \in hp(i)} \left\lceil \frac{s_{i,l}}{T_j} \right\rceil \cdot C_j, \qquad (1)$$

where $hp(i)$ and $lp(i)$ denote the tasks with higher- and lower-priority than τ_i, respectively, and the analysis checks all the jobs of τ_i within the *busy-period*. Details of the analysis can be found in [4].

The advantages of strict partitioning include the reduction of the lower-priority blocking and a less pessimistic schedulability test (*i.e.,* the exact test in [4]). The disadvantage is the complexity of the underlying design optimization problem to efficiently find an optimal processor partitioning and task assignment, for which we propose a heuristic in Section 5.

Algorithm 1: Top-level Design of NPG-SP*

Input: τ: a NPG task set to be scheduled;
M: number of available processors;
Output: ρ: processor partitions; $\tau(\rho)$: task assignment;
$sched$: schedulability of τ on M processors;

1 $\overline{\tau} \leftarrow \tau$;
2 $\rho \leftarrow$ Initialize M processor partitions with size 1;
3 $\tau(\rho) \leftarrow$ Initialize task assignment as empty sets;
4 **while** $|\rho| \geq 1$ **do**
5 $\overline{\tau} \leftarrow BestVolumeBinPack(\rho, \tau(\rho), \overline{\tau})$;
6 **if** $\overline{\tau} = \emptyset$ **then**
7 $sched \leftarrow True$; **break**;
8 **if** $|\rho| = 1$ **then**
9 $sched \leftarrow False$; **break**;
10 **else**
11 $\rho \leftarrow MergePartitions(\rho, \tau(\rho), \overline{\tau})$;
12 **return** $\rho, \tau(\rho), sched$;

5 A NPG STRICT PARTITIONING HEURISTIC

5.1 Overview

Now, we propose a heuristic NPG-SP* to assist the NPG strict partitioning strategy in making processor partitioning and task assignment decisions. Without loss of generality, we assume that task priorities are assigned by *Deadline Monotonic*.

The top-level design of NPG-SP* is presented in Algorithm 1. At first, we initialize M processor partitions of size one with empty task assignments. Then, in each iteration of lines 4-11, the task assignment is made by a bin-packing heuristic called *BestVolumeBinPack*, which takes into account the influence of the parallelism level on the task utilization and performs a local search if a task cannot be assigned to any existing processor partition. Based on the bin-packing results, the algorithm returns a success if all the tasks have been assigned to the existing partitions. Otherwise, it will invoke the *MergePartitions* component to explore new processor partitions with different parallelism levels. The key idea of *MergePartitions* is to merge the two most underutilized partitions into one large partition. It offers an opportunity to improve resource utilization and thus make more tasks schedulable. The algorithm terminates and returns a failure if there is only one partition of size M and still remain unassigned tasks.

Next, we explain the *BestVolumeBinPack* and *MergePartitions* components in detail and analyze their time complexities.

5.2 Details of the heuristic

The BestVolumeBinPack component assigns each task to the processor partition that results in the best resource efficiency (*i.e.*, the least $U_{i,m}$) and sets its parallelism as the corresponding partition size. Specifically, the procedure (Algorithm 2) assigns tasks to partitions in the decreasing order of task priorities. For each task τ_i, the partitions $\rho_j^m \in \rho$ are sorted in the ascending order of the corresponding task utilization $U_{i,m}$. The partitions with the same volume m result in the same task utilization and are sorted according to their index j (tie-breaking). For each partition, we check the feasibility of accepting task τ_i (see Equation 1). If the task set is schedulable after accepting τ_i, we assign it to this partition and continue

Algorithm 2: The *BestVolumeBinPack* Component

Input: τ: a set of NPG tasks to be scheduled;
ρ: processor partitions; $\tau(\rho)$: task assignment;
Output: $\overline{\tau}$: unassigned task set;

1 $\overline{\tau} \leftarrow \emptyset$;
2 Sort tasks by priority (from higher to lower);
3 **foreach** $\tau_i \in \tau$ **do**
4 $sched \leftarrow False$;
5 Sort the partitions in the ascending order of $U_{i,m}$;
6 **foreach** $\rho_j^m \in \rho$ **do**
7 **if** $\tau(\rho_j^m) \cup \{\tau_i\}$ *is schedulable* **then**
8 $\tau(\rho_j^m) \leftarrow \tau(\rho_j^m) \cup \{\tau_i\}$;
9 $sched \leftarrow True$;
10 **break**;
11 **if** $sched = False$ **then**
12 $sched \leftarrow LocalSearch(\rho, \tau(\rho), \tau_i)$;
13 **if** $sched = False$ **then**
14 $\overline{\tau} \leftarrow \overline{\tau} \cup \{\tau_i\}$;
15 **return** $\overline{\tau}$;

the assignment of the next task. If task τ_i cannot be schedulable with any of the existing partitions, the algorithm will invoke a *LocalSearch* procedure for an intensification. The *LocalSearch* tries to move one of the already-assigned tasks to another partition to spare space for the non-schedulable task τ_i. The details of the *LocalSearch* procedure are given in Algorithm 3.

Algorithm 3: The *LocalSearch* Procedure

Input: ρ: processor partitions; $\tau(\rho)$: task assignment;
τ_k: task to be assigned;
Output: $sched$: schedulability of the input task;

1 **foreach** $\rho_j^m \in \rho$ **do**
2 **foreach** $\tau_i \in \tau(\rho_j^m)$ **do**
3 **if** $\tau(\rho_j^m) \setminus \{\tau_i\} \cup \{\tau_k\}$ *is schedulable* **then**
4 **foreach** $\rho_{j'}^{m'} \in \rho \setminus \{\rho_j^m\}$ **do**
5 **if** $\tau(\rho_{j'}^{m'}) \cup \{\tau_i\}$ *is schedulable* **then**
6 $\tau(\rho_{j'}^{m'}) \leftarrow \tau(\rho_{j'}^{m'}) \cup \{\tau_i\}$;
7 $\tau(\rho_j^m) \leftarrow \tau(\rho_j^m) \setminus \{\tau_i\} \cup \{\tau_k\}$;
8 **return** $True$;
9 **return** $False$;

The MergePartitions component aims to explore new processor partitions based on the current task assignment. It selects the two partitions ($\rho_{j_1}^{m_1}$ and $\rho_{j_2}^{m_2}$) with the least processor utilization and merges them into a large partition. The size of the merged partition m' is the sum of the two previous partitions (*i.e.*, $m' = m_1 + m_2$). The tasks that were previously assigned to those partitions will be unloaded and added to the unassigned task set, which will be assigned to the new partitions in the next iteration. The intuition behind this is to eliminate processor underutilization and explore different parallelism levels for the unassigned tasks. The detailed procedures are given in Algorithm 4.

Algorithm 4: The *MergePartitions* Component

Input: ρ: proc. partitions; $\tau(\rho)$: task assign.; $\overline{\tau}$: unassigned tasks;
Output: ρ': merged processor partitions;
$\rho_{j_1}^{m_1}, \rho_{j_2}^{m_2} \leftarrow$ the two partitions with the least task utilization;
$\overline{\tau} \leftarrow \overline{\tau} \cup \tau(\rho_{j_1}^{m_1}) \cup \tau(\rho_{j_2}^{m_2})$;
$\rho' \leftarrow \rho \setminus \{\rho_{j_1}^{m_1}\} \setminus \{\rho_{j_2}^{m_2}\} \cup \{\rho_{j_1}^{m_1} \cup \rho_{j_2}^{m_2}\}$;
return ρ';

5.3 Time complexity

The overall time complexity of NPG-SP* depends on its two main components *BestVolumeBinPack* and *MergePartitions*. In *BestVolumeBinPack*, for each task, we check its schedulability on each partition (lines 6-10 of Algorithm 2), taking a time complexity of $O(M\Phi)$, where $O(\Phi)$ is the complexity of the uniprocessor schedulability test. If the task is not schedulable on any of the partitions, the local search will be applied, which takes a time complexity of $O(nM\Phi)$, as it performs a schedulability check for each already-assigned task in the worst case. Therefore, the total time complexity of the *BestVolumeBinPack* component is $O(n^2M\Phi)$. The *MergePartitions* component takes $O(M)$ to select the partitions with the least utilization and $O(n)$ to move the tasks from the previous partitions to the unassigned task set. Therefore, the time complexity of each iteration in Algorithm 1 is bounded by $O(n^2M\Phi)$. Since there are at most M iterations, the total complexity of NPG-SP* is $O(n^2M^2\Phi)$. The complexity of the exact NP-FP response time analysis [4] used in this paper is $\Phi = n^2 \cdot \max_{i,m} C_{i,m}/(1-U_{limit})$, where $U_{limit} = 0.99$ is the maximal allowable utilization of any single partition.

6 PERFORMANCE EVALUATION

We evaluate the performance of the proposed NPG strict partitioning on real-world multi-TPU benchmarks by comparing it with state-of-the-art NPG scheduling techniques.

6.1 Experimental setup.

Edge TPU benchmarking. We benchmark 8 representative DNNs in computer vision applications on multi-TPU accelerators. The tested DNN architectures are listed in Fig. 2. The hardware used in the experiment is ASUS AI Accelerator card CRL-G18U-P3D[2], which integrates 8 Google Edge TPUs with PCIe connection. For each DNN, we vary the input image size from 100×100 to 700×700 with a step of 100 pixels for both height and width. Thus, we have in total 56 configurations of the tested DNNs with different input sizes. Furthermore, we compile[3] each DNN configuration with different parallelism levels varied from 1 to 8 and execute them on the TPU pipelines with the corresponding volume (*i.e.*, the number of TPUs in the pipeline equals the DNN parallelism). We repeat each execution 1,000 times and record the WCET, ranging from 3 ms to 343 ms. Fig. 2 shows a subset of the benchmarking results with input size 300×300 for all the tested DNNs.

Task set generation. We generate random DNN task sets based on the above benchmarking results for the schedulability performance evaluation. We fix the number of processors $M = 8$ according to our hardware configuration while varying the task set size $n \in \{8, 16\}$ and task set utilization $U \in [0.1, 8]$ with a step of 0.1. Additionally, we consider three subsets of the DNN benchmarks depending on the range of their WCETs when executing on single processor: $C_{i,1} \in [3, 50]$ ms (min. 20 FPS), $C_{i,1} \in [3, 100]$ ms (min. 10 FPS), and $C_{i,1} \in [3, 343]$ ms (min. 3 FPS). For each combination of the above parameters, we generate 10,000 task sets as follows. First, we randomly sample a number of n tasks from the DNN benchmark task set. Second, we use a standard tool DRS [9] to generate each task utilization $U_{i,1}$ under the given target reference utilization U such that $\sum_{i=1}^{n} U_{i,1} = U$. Third, we calculate the task period as $T_i = C_{i,1}/U_{i,1}$ and set $D_i = T_i$. In this way, we generated a total of 4,800,000 task sets for the evaluation.

Comparison algorithms. We evaluate the proposed NPG strict partitioning strategy by comparing the achieved schedulability ratio with the following algorithms when scheduling the same task sets.

- State-of-the-art global NPG response time analyses: RTSS'22 [12] and RTAS'23 [17];
- Federated scheduling [14] extended to gang tasks: FedGang;
- Strict partitioning heuristics: NPG-SP* (Algorithm 1) and a baseline strict partitioning heuristic SP-UFF based on *uniform* processor partitioning and *first-fit* bin-packing.

For RTSS'22 and RTAS'23, each task τ_i's parallelism m_i is set as the value leading to the minimum utilization (*i.e.*, $m_i = \arg\min_m U_{i,m}$). FedGang divides the task set into two subsets: (i) parallel tasks τ^p with $U_{i,1} > 1$ and (ii) sequential tasks τ^s with $U_{i,1} \leq 1$. Each task in τ^p is assigned exclusively the minimum number of processors such that its utilization is not greater than 1, *i.e.*, $m_i = \min\{m | U_{i,m} \leq 1\}$. The tasks in τ^s are configured as sequential tasks and scheduled on the remaining $M - \sum_{\tau_i \in \tau^p} m_i$ processors using the global NP-FP policy. SP-UFF first divides the processors into uniform partitions (*i.e.*, all the partitions have the same size m) and then applies first-fit bin-packing to decide task assignment as in [8]. In our experiments, we invoke SP-UFF on every possible uniform partitioning (*i.e.*, $m = 1, 2, 4, 8$) and return schedulable if it succeeds on any of them.

6.2 Comparison results

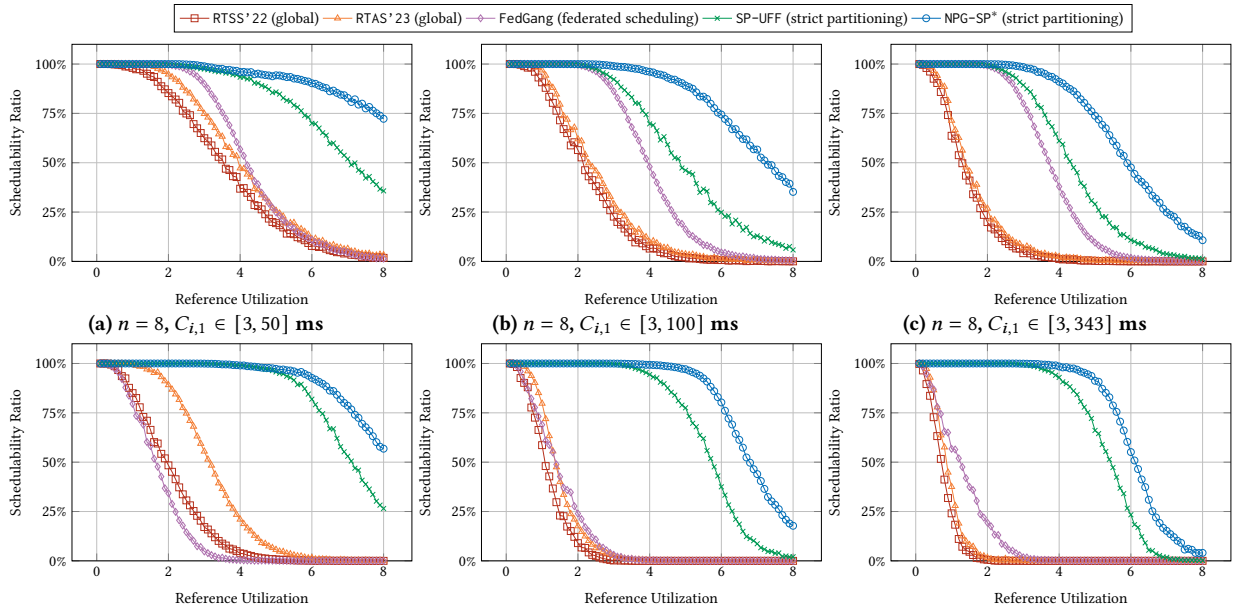
The schedulability comparison results are presented in Fig. 6.

Overall comparison. The strict partitioning approaches (NPG-SP* and SP-UFF) achieve a higher schedulability ratio than global NPG scheduling (RTSS'22 and RTAS'23) and FedGang. In particular, the global NPG response time analyses have a very low schedulability ratio for high utilization task sets (*e.g.*, $U > 6$), while the strict partitioning heuristics can schedule task sets even for $U = 8$.[4] Moreover, NPG-SP* outperforms SP-UFF by up to 50.11% more task sets deemed schedulable, which demonstrates the effectiveness of the proposed strict partitioning heuristic.

Effect of the task set size. The schedulability ratio of RTSS'22 and RTAS'23 is lower on larger task sets. This is expected because global NPG scheduling suffers from the unbounded priority inversion as discussed in Section 4, and the 2-D blocking increases with the task set size. FedGang has a similar trend because it assigns dedicated processors to tasks that require more than one processor, and the processors can easily become insufficient as the task number grows.

[2] https://iot.asus.com/products/AI-accelerator/AI-Accelerator-PCIe-Card/
[3] https://coral.ai/docs/edgetpu/compiler/

[4] Recall that the reference task set utilization is defined as the sum of task utilization with a parallelism level of 1 (*i.e.*, $U = \sum_{i=1}^{n} U_{i,1}$).



(a) $n = 8$, $C_{i,1} \in [3, 50]$ **ms** **(b)** $n = 8$, $C_{i,1} \in [3, 100]$ **ms** **(c)** $n = 8$, $C_{i,1} \in [3, 343]$ **ms**

(d) $n = 16$, $C_{i,1} \in [3, 50]$ **ms** **(e)** $n = 16$, $C_{i,1} \in [3, 100]$ **ms** **(f)** $n = 16$, $C_{i,1} \in [3, 343]$ **ms**

Figure 6: Comparison of schedulability ratios on Multi-TPU DNN inference benchmarks.

In contrast, the strict partitioning approaches achieve a higher schedulability ratio on larger task sets for $U \leq 6$, although there is a steeper drop than on smaller task sets when $6 \leq U \leq 8$.

Effect of the WCET range. All comparison algorithms achieve a lower schedulability ratio for a larger WCET range. This is expected since the larger the WCET range, the more likely a task suffers from a larger blocking time from other lower-priority tasks. Nonetheless, NPG-SP* achieves almost 100% schedulability ratio for $U \leq 3$ and $U \leq 4$ for all WCET ranges on $n = 8$ and $n = 16$, respectively.

7 CONCLUSION AND FUTURE WORK

In this paper, we propose a new scheduling strategy called *NPG strict partitioning* for real-time DNN inferences running on multi-TPU accelerators. We propose a simple yet effective heuristic to solve the underlying design optimization problem to determine an appropriate partitioning of DNN tasks and TPUs. Extensive experiments on DNN benchmarks, obtained by profiling DNN execution with different input sizes and parallelism on multi-TPU accelerators, demonstrated that the proposed strict partitioning strategy can achieve a much higher schedulability ratio than the state-of-the-art global NPG response time analyses and federated scheduling.

Future work includes the scheduling of moldable gang tasks and the co-scheduling of DNN tasks on heterogeneous accelerators [2].

ACKNOWLEDGMENTS

Marco Caccamo was supported by an Alexander von Humboldt Professorship endowed by the German Federal Ministry of Education and Research. Chu-ge Wu was supported by the National Natural Science Foundation of China under Grant 62203049.

REFERENCES

[1] Waqar Ali and Heechul Yun. 2019. RT-Gang: Real-time gang scheduling framework for safety-critical systems. In *RTAS*. 143–155.

[2] Shuangshuang Chang, Jinghao Sun, Zhenyu Liu, Xufeng Zhao, and Qingxu Deng. 2022. Response time analysis of parallel tasks on accelerator-based heterogeneous platforms. *Journal of Systems Architecture* 126 (2022), 102484.

[3] Jian-Jia Chen. 2016. Partitioned multiprocessor fixed-priority scheduling of sporadic real-time tasks. In *ECRTS*. 251–261.

[4] Robert I. Davis, Alan Burns, Reinder J. Bril, and Johan J. Lukkien. 2007. Controller Area Network (CAN) Schedulability Analysis: Refuted, Revisited and Revised. *Real-Time Systems* 35, 3 (2007), 239–272.

[5] Zheng Dong and Cong Liu. 2017. Analysis Techniques for Supporting Hard Real-Time Sporadic Gang Task Systems. In *RTSS*. 128–138.

[6] Zheng Dong and Cong Liu. 2022. A Utilization-based Test for Non-preemptive Gang Tasks on Multiprocessors. In *RTSS*. 105–117.

[7] Pontus Ekberg and Sanjoy Baruah. 2021. Partitioned Scheduling of Recurrent Real-Time Tasks. In *RTSS*. 356–367.

[8] Nathan Fisher and Sanjoy Baruah. 2006. The partitioned multiprocessor scheduling of non-preemptive sporadic task systems. In *RTNS*. 99–108.

[9] David Griffin, Iain Bate, and Robert I Davis. 2020. Generating utilization vectors for the systematic evaluation of schedulability tests. In *RTSS*. 76–88.

[10] Changhun Han, Hoon Sung Chwa, Kilho Lee, and Sangeun Oh. 2023. SPET: Transparent SRAM Allocation and Model Partitioning for Real-time DNN Tasks on Edge TPU. In *DAC*. 1–6.

[11] Shinpei Kato and Yutaka Ishikawa. 2009. Gang EDF Scheduling of Parallel Task Systems. In *RTSS*. 459–468.

[12] Seongtae Lee, Nan Guan, and Jinkyu Lee. 2022. Design and timing guarantee for non-preemptive gang scheduling. In *RTSS*. 132–144.

[13] Joseph Y.-T. Leung and Jennifer Whitehead. 1982. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation* 2, 4 (1982), 237–250.

[14] Jing Li, Jian Jia Chen, Kunal Agrawal, Chenyang Lu, Chris Gill, and Abusayeed Saifullah. 2014. Analysis of Federated and Global Scheduling for Parallel Real-Time Tasks. In *ECRTS*. 85–96.

[15] Geoffrey Nelissen, Joan Marcè i Igual, and Mitra Nasri. 2022. Response-Time Analysis for Non-Preemptive Periodic Moldable Gang Tasks. In *ECRTS*.

[16] Binqi Sun, Tomasz Kloda, and Marco Caccamo. 2024. Strict Partitioning for Sporadic Rigid Gang Tasks. arXiv:2403.10726

[17] Binqi Sun, Tomasz Kloda, Jiyang Chen, Cen Lu, and Marco Caccamo. 2023. Schedulability Analysis of Non-preemptive Sporadic Gang Tasks on Hardware Accelerators. In *RTAS*. 147–160.

[18] Niklas Ueter, Mario Günzel, Georg von der Brüggen, and Jian-Jia Chen. 2021. Hard real-time stationary gang-scheduling. In *ECRTS*. 10:1–10:19.

[19] Jorge Villarrubia, Luis Costero, Francisco D. Igual, and Katzalin Olcoz. 2023. Improving inference time in multi-TPU systems with profiled model segmentation. In *PDP*. 84–91.

[20] Shuai Zhao, Nan Chen, Yinjie Fang, Zhao Li, and Wanli Chang. 2023. A Universal Method for Task Allocation on FP-FPS Multiprocessor Systems with Spin Locks. In *DAC*. 1–6.