A^2: Towards Accelerator Level Parallelism for Autonomous Micromobility Systems

LINGYU SUN, Shanghai Jiao Tong University, Shanghai, China
XIAOFENG HOU, Shanghai Jiao Tong University, Shanghai, China
CHAO LI, Shanghai Jiao Tong University, Shanghai, China
JIACHENG LIU, The Chinese University of Hong Kong, Hong Kong, Hong Kong
XINKAI WANG, Shanghai Jiao Tong University, Shanghai, China
QUAN CHEN, Shanghai Jiao Tong University, Shanghai, China
MINYI GUO, Shanghai Jiao Tong University, Shanghai, China

Autonomous micromobility systems (AMS) such as low-speed minicabs and robots are thriving. In AMS, multiple Deep Neural Networks execute in parallel on heterogeneous AI accelerators. An emerging paradigm called Accelerator Level Parallelism (ALP) suggests managing accelerators holistically. However, there lacks a specialized and practical solution populating ALP for an AMS, where the varying real-time requirements under different working scenarios bring an opportunity to dynamically tradeoff between latency and efficiency. Furthermore, accelerator heterogeneity introduces enormous configuration space, and the shared-memory architecture results in dynamic bandwidth interference.

In this article, we propose A^2, a novel AMS resource manager optimizing energy and memory space efficiency under variable latency constraints. We gain insight from prior *Learn&Control* scheme to design an *Analyze&Adapt* scheme specialized for heterogeneous AI accelerators under shared-memory architecture. It features analyzing the system thoroughly offline to support two-step adaptation online. We build a prototype of A^2 and evaluate it on a commercial edge platform. We show that A^2 achieves 32.8% improvements in power and 13.8% in memory compared with control-based methods. As for timeliness enhancement, A^2 reduces the deadline violation rate by 9.2 percentage points (12.8% $\longrightarrow$ 3.6%) on average compared to directly porting *Learn&Control* methods.

CCS Concepts: • **Hardware** → **Platform power issues**; • **Computer systems organization** → **Robotic autonomy**; *System on a chip;*

Additional Key Words and Phrases: Resource management, autonomous system, accelerators

This work is supported by the National Key R&D Program of China (No. 2022YFB4501702) and the National Natural Science Foundation of China (No. 62122053).

Authors' Contact Information: Lingyu Sun, Shanghai Jiao Tong University, Shanghai, China; e-mail: sunlingyu@sjtu.edu.cn; Xiaofeng Hou, Shanghai Jiao Tong University, Shanghai, China; e-mail: hou-xf@cs.sjtu.edu.cn; Chao Li (Corresponding author), Shanghai Jiao Tong University, Shanghai, China; e-mail: lichao@cs.sjtu.edu.cn; Jiacheng Liu, The Chinese University of Hong Kong, Hong Kong, Hong Kong; e-mail: jiachengliu@cuhk.edu.hk; Xinkai Wang, Shanghai Jiao Tong University, Shanghai, China; e-mail: unbreakablewxk@sjtu.edu.cn; Quan Chen, Shanghai Jiao Tong University, Shanghai, Shanghai, China; e-mail: chen-quan@cs.sjtu.edu.cn; Minyi Guo, Shanghai Jiao Tong University, Shanghai, Shanghai, China; e-mail: guo-my@cs.sjtu.edu.cn.

ACM 1544-3973/2024/11-ART86
https://doi.org/10.1145/3688611

ACM Reference Format:
Lingyu Sun, Xiaofeng Hou, Chao Li, Jiacheng Liu, Xinkai Wang, Quan Chen, and Minyi Guo. 2024. A^2: Towards Accelerator Level Parallelism for Autonomous Micromobility Systems. *ACM Trans. Arch. Code Optim.* 21, 4, Article 86 (November 2024), 20 pages. https://doi.org/10.1145/3688611

1 Introduction

Embedded systems are moving towards full automation. In particular, **autonomous micromobility systems** (**AMS**), including self-driving minicabs, delivery robots, and drones [34, 60] are thriving. AMS features limited cost (\$1000–\$5000 [4]) and low speed (<30 km/h [60]). AMS is commonly supported by **system-on-chip** (**SoC**) containing CPU and AI accelerators, typically including an integrated GPU and 1–2 NPUs [2, 6, 50, 60]. GPU often has higher performance while NPU achieves better power efficiency [37, 43, 47]. They are commonly connected to a shared memory. As DNNs gradually become the primary workload on such SoCs, **Accelerator Level Parallelism** (**ALP**) [26, 44] suggests holistically regulating these heterogeneous AI accelerators.

An AMS typically executes multiple DNNs in parallel for perception. A fixed group of DNNs arrive periodically and share a common latency constraint [39, 60]. The constraint can vary as the AMS working scenario changes [25, 61]. This brings a resource management opportunity to save power by scheduling a larger portion of DNNs to energy-efficient but low-performance NPUs instead of GPUs when the latency constraint relaxes. Accelerator frequency adjustment can also be applied together for a more fine-grained power-performance tradeoff.

However, there are several AMS-specific challenges for resource management. Firstly, there is a large configuration space when finding suitable configurations under varying latency constraints. Specifically, a system configuration in this article includes DNN-accelerator mapping and the running frequency of each accelerator. This design space grows exponentially with the DNN number. Secondly, there is dynamic bandwidth contention among accelerators and CPU, resulting from SoC shared-memory architecture. Contention between GPU and NPUs makes their DNN execution latency coupled and complicates scheduling decisions. Contention between the CPU and accelerators introduces dynamic interference and could slow down DNN execution unexpectedly. Thirdly, DNN scheduling also affects memory space usage, which is precious on an embedded SoC. For a DNN to be executable on an accelerator, an accelerator-specific engine file has to be loaded into memory. The engine file is essentially a binary that describes the computational operations when executing a DNN and stores the weight parameters of the DNN. Therefore, a scheduling decision has to be power efficient while not loading redundant engines to use too much memory space.

Prior works focusing on resource management of edge systems can be roughly divided into three categories. **(1)** Learning-based ones [16, 21, 31, 32, 35, 55] handle system complexity well but fail to handle dynamic interference. **(2)** Control-based ones [11, 13, 28, 40, 53, 54] react fast to dynamic interference but are most suitable for simple systems. **(3)** Learn&Control ones [19, 20, 45] employ a learner to extract suitable configuration candidates and a controller to switch among them. Such a combination works well when a configuration switch incurs little overhead. However, this is not the case for AI accelerators when switching scheduling configurations, which involves loading new engines. A configuration switch then could not be performed frequently and timely, which leads to latency constraint violation.

Therefore, we propose *A*^{2} following *Analyze&Adapt* scheme, populating ALP for an AMS with heterogeneous AI accelerators. It extends *Learn&Control* scheme for better real-time performance and consists of an offline analyzer and an online adaptor. The analyzer not only learns for suitable configuration candidates but also profiles system latency characteristics for online fine-tuning. The adaptor not only periodically switches among pre-learned configurations but also continuously tweaks the frequency of a selected configuration.

The offline analyzer handles large configuration space. It tries to minimize power while keeping memory space usage small. It first solves a multi-dimensional optimization problem to extract Pareto optimal configurations and cherry-picks one best configuration candidate for each possible latency requirement. It then profiles the system to record necessary information supporting online configuration tweaking. This is realized by profiling workload latency under all frequency settings in various system statuses.

The online adaptor deals with dynamic bandwidth contention. It prioritizes real-time performance by tweaking frequency while trying to adhere to pre-learned configurations in a best-effort manner. It periodically switches among pre-learned configurations when the latency constraint or dynamic contention changes a lot. Within the switching period, the adaptor fine-tunes the frequency setting of the selected configuration in a lightweight manner in case of transient contention fluctuation or spikes.

We build a prototype of A^2 and evaluate it thoroughly on a commercial edge platform, Nvidia Jetson Xavier NX. We demonstrate that A^2 offers nearly optimal power consumption and memory usage while ensuring timeliness under various latency constraints in multiple setups.

The contributions of this article include three parts:

— **Analysis**: We analyze the power-performance-memory implications of heterogeneous AI accelerators on a popular commodity platform. Specifically, we first reveal a DNN scheduling opportunity for a power-performance tradeoff since the GPU is less power efficient but has higher performance than the NPU. We then identify that the scheduling decision also affects memory usage due to engine loading. Finally, we show that memory bandwidth contention among GPU-NPU-CPU causes performance degradation on DNN execution.

— **Design**: We introduce a novel resource management scheme of *Analyze&Adapt* populating ALP for AMS. It extends the traditional Learn&Control scheme for better real-time responsiveness to variable scenarios. During offline analysis, it searches for optimal scheduling configuration candidates lying on the Pareto optimal boundary of power-performance-memory space. During online adaption, it switches among the candidates infrequently and tweaks the candidate with fine-grained frequency control to avoid configuration switch overhead.

— **Evaluation**: We implement and evaluate A^2 on a commodity edge platform and thoroughly demonstrate its effectiveness, efficiency, and sensitivity. Experiment results show that A^2 achieves 32.8% power improvements and 13.8% memory improvements on average compared to heuristic-based control methods. It also outperforms the current *Learn&Control* manager greatly by reducing the average deadline violation rate from 12.8% to 3.6%, and the deadline violation extent of 99th tail latency is less than 5ms.

2 Background and Related Works

2.1 AMS Software and Hardware Overview

The software workloads of AMS consist of two parts: workloads for acting autonomously and workloads for advanced functionalities, as shown in Figure 1. Workloads for autonomy contain three stages [14, 39, 60]: sensing, perception, and decision making. The sensing stage collects raw data and synchronizes their timestamps [60]. The perception stage uses various DNNs to perform object detection, semantic segmentation, and so on. It also employs some CPU algorithms to locate itself. The decision-making stage summarizes all extracted information and generates signals for system control. An AMS also offers advanced functionalities, including high-res video recording, face detection, or speech recognition. In this article, we focus on resource management of DNNs within the *perception stage* since it can account for more than 94% of computation [39].

The DNNs involved in the perception stage have three key characteristics: First, their number and type are fixed during run-time determined by sensor number [58], which is typically

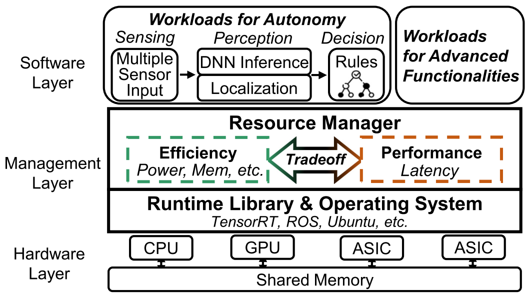

Fig. 1. AMS architecture.

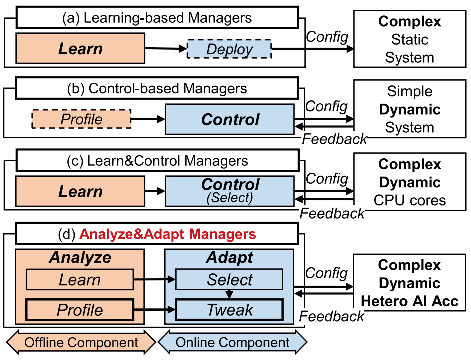

Fig. 2. A classification of edge resource managers.

10–20 [50]. Second, the fixed collection of DNNs executes periodically under a common latency constraint, which changes dynamically depending on the vehicle speed and the environment complexity [15, 25, 61, 62]. The constraint is usually several hundred milliseconds [11, 60]. Third, there is no data dependency among DNNs since each DNN is responsible for perceiving independent sensor inputs [58]. Admittedly, some recently proposed perception methods suggest integrating multi-sensor data through a single fusion-based DNN [24, 27, 48]. Nevertheless, a fusion-based DNN is composed of multiple independent backbone sub-models and one head sub-model. The backbones' intermediate results, or extracted features, are aggregated as the input of the head sub-model. Since the backbone sub-models condense high-resolution raw inputs into small-scale features, they account for most of the execution time of a fusion-based DNN [49, 56]. The backbone sub-models can use different DNN models depending on sensor modality and importance [38, 42]. Therefore, the identified three key characteristics of AMS perception DNNs are also applicable to the backbone sub-models in a fusion-based architecture.

AMS is commonly supported by SoCs carrying GPU and NPU for DNN inference and following the shared-memory architecture [12, 41, 46]. The main memory capacity is relatively small and needs to be used conservatively [12, 30]. Since CPU and accelerators are all connected to the same memory controller, they also contend for memory bandwidth [57]. In this article, we focus on the management of heterogeneous AI accelerators since they are relatively independent of the CPU. The memory bandwidth usage from the CPU side is considered a source of dynamic interference.

2.2 Classification of Edge Resource Managers

Autonomous micromobility systems are gaining popularity in both academia [9, 29, 34, 40, 50, 60] and industry [2, 6]. Most importantly, Shi-Chieh et al. present their architectural implications [39]. This work lies in the context of designing resource managers for DNN execution. The resource manager interacts with system runtime and OS to balance between efficiency and performance. Prior work in this context mainly considers using GPU alone [11, 53] or using both CPU and GPU [31, 40, 55]. Some works discuss heterogeneous accelerators yet overly focus on pre-silicon design [32, 34, 36, 57]. This article fills in the gap of managing AMS with heterogeneous AI accelerators on a commercial platform.

Learning-based managers are designed for complex static systems (Figure 2(a)). It leverages machine learning or optimization for design space exploration offline, generating a single output configuration to be deployed. Most managers designed for heterogeneous systems fall under this category. Sheng-Chun et al. design an offline DSE method mapping DNNs to sub-accelerators [32]. They ignore memory bandwidth contention caused by co-running CPU tasks and have no real-time consideration. Woosung et al. propose an offline optimization approach to schedule multiple DNNs on CPU and GPU in layer granularity [31]. Their approach involves **Worst-Case Execution Time** (**WCET**) analysis like [55]. This can be overly conservative when the interference level is low. Pablo et al. discussed a scheduling framework called POAS (Predict, Optimize, Adapt, and Schedule) to achieve ALP [44]. However, it is evaluated for matrix multiplication, and the implemented scheduler is purely static.

Control-based managers are designed for simple dynamic systems (Figure 2(b)). It relies on formal control theory or heuristics to react to run-time variation. The general pipeline involves monitoring, tuning, and receiving feedback. Soroush et al. suggest optimizing accuracy and power simultaneously while offering predictable latency for multiple DNNs on one GPU [11]. It solely relies on frequency tuning using LAG analysis and cannot handle the complexity of heterogeneity. Liangkai et al. designed a framework to coordinate multiple DNNs on one GPU [40]. They study many DNNs and claim the existence of a large latency variation of DNNs. However, DNNs used in this article are “proposal-free” models that are widely used in the industry. They have very stable latency.

There are also *Learn&Control* resource managers (Figure 2(c)) designed for a heterogeneous CPU. It learns several suitable configuration candidates and selects among them with an online controller. The learning process can be offline, semi-offline with online updates, or even purely online. This scheme is suitable for complex dynamic systems. Nikita et al. first propose the *Learn&Control* model while ignoring the emerging ALP trend and the importance of management timeliness [45]. Bryan et al. suggest using online reinforcement learning for a Learning Classifier Table to guide a feedback controller tuning frequency for a many-core system [20]. They target irregularly arriving tasks with a work queue, which is different from AMS perception. *we argue that traditional Learn&Control has deficits in real-time when a configuration switch incurs overhead and cannot be performed frequently enough. This is the case for heterogeneous AI accelerators.* Correspondingly, we offer a novel scheme named *Analyze&Adapt* in Figure 2(d), which features tweaking a selected configuration for better timeliness.

3 Motivation

3.1 Understanding Accelerator Specialties

Understanding the specialty and weakness of accelerators is critical for achieving efficient ALP. We conduct our experiments on Jetson Xavier NX, a commercial system-on-module equipped with 1 GPU and 2 ASIC **Deep Learning Accelerators** (**DLA**) (detailed in Section 5). The frequency of GPU can only be set to 12 fixed levels, while the frequency of DLA can be set to arbitrary values. Their minimal frequencies are both at about 300 MHz, and their maximal frequencies are both at about 1100 MHz. For a clear comparison, we evenly select 12 frequency levels among all available ones for DLA. In summary, the frequency levels for GPU in MHz include 306, 408, 510, 599, 752, 803, 854, 905, 956, 1007, 1058, and 1109. The selected frequency levels for DLA in MHz include 320, 384, 448, 512, 576, 640, 704, 768, 832, 896, 960, and 1024. To demonstrate the power-performance characteristics, we record the latency and power consumption of YOLO-v3, a widely used DNN for object detection, on each accelerator under every frequency level and present the results in Figure 3.

From the power perspective, GPU consumes 2x–4x more power than DLA in all frequency settings. But from the performance perspective, the execution latency on DLA is also 1.6x–2.5x higher than GPU. Combined with detailed findings on other ASIC NPUs [37, 43, 47], we believe such power-performance trade-off for GPU and NPU on edge is not uncommon. On the one hand, GPUs are more flexible in accelerating most parallel tasks, so manufacturers would be eager to equip a relatively powerful GPU as the main accelerator for versatility. On the other hand,

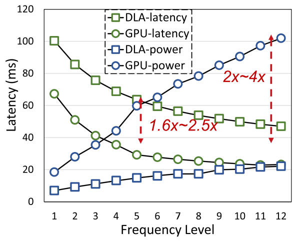

Fig. 3. Accelerator Specialties. GPU excels in performance, while DLA excels in power.

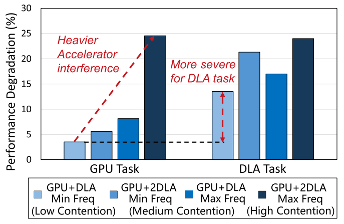

Fig. 4. The impact of inner memory contention. The mutual influence between GPU and DLA is studied.

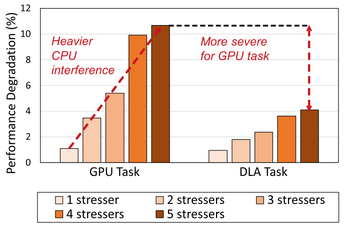

Fig. 5. The impact of outer memory contention. The influence of CPU on GPU or DLA is studied.

NPU accelerators can only run limited functionalities but achieve better power efficiency. The manufacturers would tend to allocate a small percentage of the die area to them, thus weakening their performance. For example, the GPU die area of NVIDIA Xavier is about twice as large as its NPU [1]. Another example is the higher-end NVIDIA Orin. Its GPU provides up to 170 Sparse TOPS (Tera Operations Per Second) of INT8 Tensor compute, while its two NPUs could only provide 75 TOPS in total [2]. This trade-off resembles the big. LITTLE architecture of ARM CPUs.

In summary, accelerator specialties raise management complexity but also bring power-saving opportunities. When the latency constraint is relaxed, the NPUs' specialty manifests itself as the ability to execute DNNs more efficiently. Contrarily, GPU's specialty stands out to execute DNNs faster.

3.2 Understanding Memory Challenges

3.2.1 Memory Capacity Challenges. Since the main memory is shared among multiple processors, it is critical to save precious memory capacity to accommodate more advanced functionalities in AMS [30]. Take Xavier NX as an example, the total memory capacity is 8 GB with only 6 GB available for run-time management. The memory usage of studied DNNs is shown in Table 1, and their engines occupy about 200–300 MB. The engine of a DNN is essentially a binary that describes the computational operations when executing a DNN and stores the weight parameters of the DNN. To schedule a DNN on a particular accelerator, its corresponding engine for that accelerator must be loaded into memory before execution.

Loading all engines for each accelerator brings better scheduling flexibility but also increases memory usage. If we pre-load the engines of all three types of DNNs in Table 1 on all accelerators (3×3=9 engines) for flexible and warm online scheduling, 50% (3 GB) of the memory capacity would be occupied for engines. Considering that CPU tasks also need memory space to hold data, including high-res maps, the memory footprint of DNNs is too large. However, if we sacrifice flexibility and only pre-load a single type of DNN for specific hardware (1×3=3 engines), the memory footprint could be reduced to only 900 MB, saving up to 70% of memory. But in such a case, the DNN-Accelerator mapping is fixed. Execution latency and power consumption may be suboptimal when the working scenario changes.

3.2.2 Memory Bandwidth Challenges. Executing DNNs concurrently on GPU and DLAs could result in memory bandwidth contention and slow down all of them compared to running separately. We denote such memory contention between heterogeneous accelerators as ***Inner Memory Contention (IMC)***. We denote it as "inner" since the contention happens within the DNN accelerator subsystem. Meanwhile, CPU tasks for localization and other advanced functionalities often run simultaneously with DNN inference tasks in AMS. We denote the memory contention between CPU and accelerators as ***Outer Memory Contention (OMC)***. We denote it as "outer" since the

Table 1. Summary of Representative DNNs

DNN	YOLOv3-416	Resnet101	VGG16
Functionality	Object detection	Backbone net for many tasks	
Number of layers	106	101	16
Mem Usage (GPU)	299 MB	199 MB	631 MB
Mem Usage (DLA)	266 MB	146 MB	486 MB

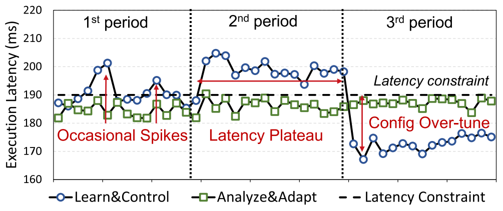

Fig. 6. Weakness of current Learn&Control schemes.

contention is caused by the CPU, which is outside the DNN accelerator subsystem. The optimization of CPU tasks is heavily studied and CPU tasks are merely considered as a source of dynamic interference to the accelerator subsystem in this article.

To thoroughly analyze the memory bandwidth challenges, we first study the IMC by co-running DNNs on different accelerator combinations, as shown in Figure 4. For GPU tasks (the first bar group), we fix the GPU frequency at the maximal and execute YOLO-v3 on it. We introduce IMC by simultaneously deploying independent YOLO-v3 on 1 or 2 DLAs with minimal or maximal DLA frequency. The bars report the performance degradation of YOLO-v3 compared to running on the GPU alone (23.2 ms). Similarly, for DLA tasks (the second bar group), we fix the frequency of DLAs at the maximal and execute YOLO-v3 on 1 or 2 DLAs while deploying independent YOLO-v3 on GPU with minimal or maximal GPU frequency. The bars report the performance degradation of YOLO-v3 compared to running on 1 DLA alone (47.1 ms). Note that the frequencies of the two DLAs have to be the same due to driver limitation. For clarity, the co-running setups are roughly divided into low, medium, and high contention levels. Overall, more co-running accelerators and higher co-running accelerator frequency can both lead to higher IMC and cause performance degradation. The *GPU+2DLA Min Freq* setup may cause higher degradation than *GPU+DLA Max Freq* setup since it has more co-running accelerators, although with lower co-running accelerator frequency. Therefore, finding suitable configurations for multiple DNNs becomes difficult since their total latency cannot be obtained by studying each accelerator separately.

Then, we study the OMC by running different numbers of memory bandwidth stressors to mimic CPU workload interference. Each memory stressor executes the memrate test of stress-ng tool [3] with a reading rate of 5 Gb/s and a writing rate of 1 Gb/s, consuming around 5% of memory bandwidth. We dispatch 1–5 stressors to create different levels of OMC. The resulting performance degradation is shown in Figure 5. The setup of two bar groups is the same as Figure 4, which fixes the frequency of GPU or DLA at the maximal and executes YOLO-v3. Both accelerators suffer from more severe performance degradation as the memory bandwidth usage of the CPU workload increases. The management scope of the resource manager in this article does not include the CPU. The CPU workload is not the performance bottleneck for autonomy, and CPU management has already been well-studied. Memory bandwidth usage from the CPU side is considered a source of dynamic interference in the DNN accelerator subsystem. Such run-time variation has to be managed to avoid unexpected deadline violations.

In summary, memory management challenges arise since the considerable demand on memory capacity and bandwidth for DNN inference conflicts greatly with the limited hardware resource of AMS. On-demand engine loading and interference awareness should be carefully and dynamically considered.

3.3 Learn&Control: Limitations

Current *Learn&Control* schemes are equipped with a simple online controller which periodically consults a learned reference table to decide the most proper configuration as system dynamics fluctuates [19, 20, 45]. We port such a scheme to heterogeneous AI accelerators for minimizing power consumption under the 190 ms latency constraint, as shown in Figure 6. This corresponds to detecting objects about 5m away from the AMS [60]. Each data point represents executing 5 *YOLO-v3* and 7 *Resnet101* in total under a particular configuration. A configuration includes DNN-accelerator mapping and frequency setting. The *Learn&Control* can handle IMC. Since IMC is static for any given configuration, it can be implicitly factored in during offline reference table learning. However, it has limitations when dealing with OMC, which depends on CPU workload and can be very dynamic during runtime.

In the first period, we shortly invoke the memory stressor twice to produce transient OMC spikes. When the second period begins, we permanently launch the memory stressor to produce a steady OMC increase. Such a procedure is employed to demonstrate the OMC pattern of CPU tasks in a controlled manner: Depending on the working scenario, the co-running CPU task can be different and cause large and steady OMC change [14, 59]. Meanwhile, sporadic user requests like speech recognition on the CPU can also cause transient bandwidth fluctuation. During the first period of Figure 6, the *Learn&Control* execution latency exceeds the latency constraints due to OMC spikes. During the second period, the controller requires much time to react and load new engines, which results in the latency plateau. At the start of the third period, a newly optimal configuration is finally applied while *Learn&Control* schemes over-tune the system and cause wasted latency slack. This over-tune leads to 0.3 w power waste (10% of total dynamic power, 3 w).

The limitation of *Learn&Control* manager lies in configuration switching overhead. A configuration switch consists of two steps. The CPU first loads DNN engines for the new destination accelerator into memory, while the acceleration subsystem still follows the old configuration. New scheduling and frequency settings are immediately applied once loading finishes, whereupon useless engines are cleaned up. Such configuration switches cannot happen frequently for two reasons. First, loading and cleaning typically take seconds, according to our experiment and prior works [10], which is significantly longer than common latency constraints of several hundred milliseconds. Second, memory space usage is high during configuration switch, and frequently doing so cancels out the benefit of on-demand loading. *In summary, it is critical to design a novel management scheme with superior timeliness under dynamic interference.*

4 System Design

4.1 Overview of A^2

Figure 7 shows the design overview of A^2. It bridges software and hardware by integrating an offline analyzer and an online adaptor.

When an AMS manufacturer finalizes the DNNs used for perception and the SoC model, the analyzer component plays its role. Its primary duty is to learn the most efficient configurations under all possible working conditions (i.e., different latency constraints). The analyzer achieves this by learning Pareto optimal configurations with distinct trade-offs between multiple efficiency objectives under various latency constraints. The secondary duty of the analyzer is to profile system latency sensitivity towards configuration fine-tuning. Such information complements the discrete searched configurations by understanding how their execution latency reacts toward frequency tweaking.

After an AMS is put into operation, the adaptor component regulates it to ensure timeliness while trying to attain maximal efficiency. It is capable of reacting to dynamic interference from the CPU side. Periodically, a new base configuration is selected if latency constraint changes or CPU interference level experiences a large steady change. Meanwhile, the frequency setting of the selected configuration gets tweaked continuously in case of transient memory fluctuation.

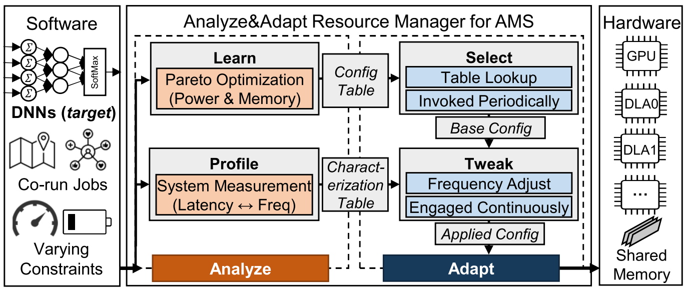

Fig. 7. A^2 design overview. The two gray rectangles in the middle are intermediate data structures.

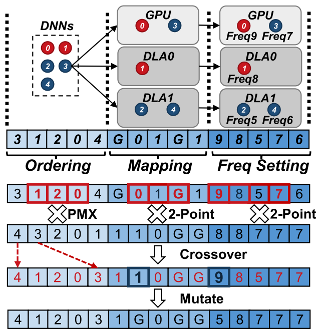

Fig. 8. Illustration of our gene design and **genetic algorithm** (**GA**).

4.2 Offline Analyzer Component

4.2.1 Learning Optimal Configuration Candidates. As stated in Section 2.1, workloads of AMS are usually fixed and recurrent with flexible latency constraints. Current learning models are complex and not suitable for AMS [19, 20, 45] since their assumption is that tasks arrive irregularly and are pushed into a task queue. We choose to customize the GA to solve a Pareto optimization problem and then cherry-pick final configuration candidates further. GA is an effective **Design Space Exploration** (**DSE**) approach with one main deficiency of being time-consuming. However, this does not matter since the optimization happens purely offline, which exempts the necessity of using overcomplicated algorithms. The optimization is performed only once for a fixed group of DNNs and a specific SoC board.

Specifically, we design our algorithm based on NSGA-II [17], a popular multi-objective optimization algorithm. In NSGA-II, each configuration is encoded into a gene as Figure 8. A configuration's gene contains both *scheduling* and *frequency setting* of all DNNs. We further split the scheduling part into *ordering* (deployment order) and *mapping* (DNN-Accelerator mapping) as in [32]. DNN with a smaller ordering number deploys earlier on GPU (*G*), DLA0 (*0*), or DLA1 (*1*) with a specific frequency level. The power, memory, and latency value of each explored configuration is obtained by executing it multiple times on the real machine and taking the average.

Take five DNNs as an example. Initially, a random set of genes is sampled to become the initial population, and the initial genes are sorted by dominating relationship and crowding distance to select *elite genes*. A gene dominates another when it is better on all objectives, and crowding distance prefers sparsity [17]. Elite genes crossover with each other (in red squares) and then mutate (in black squares) to produce the next population. We apply formal **Partially Mapped Crossover** (**PMX**) [23] on the ordering part and simple 2-Point Crossover [51] on mapping and frequency parts. With the genetic learning algorithm, configurations lying on the Pareto boundary in power-performance-memory space are obtained.

Further cherry-picking is needed after obtaining the 3-dimensional Pareto boundary. Firstly, we require only *one best power-memory configuration for each possible latency constraint.* This best configuration has to achieve both low power and small memory usage. Secondly, the frequency settings of any picked best configuration must be *elastic* enough to leave space for the online tweaker to boost frequency and reduce latency when necessary.

We design an algorithm to cherry-pick the configuration with the following properties in order: having acceptable worst-case latency, near-minimal power usage, and minimal memory. We divide the Pareto boundary in 3D space into latency bins of 10 milliseconds wide, which produces a 2D Pareto boundary slice in each bin. One best configuration in the slice is selected. According to

ALGORITHM 1: Configurations Cherry-picking Algorithm

```
Require: $C_{all}$ ▷ All configs explored
         $P_{th}$ ▷ Power threshold beyond minimum
         $M_{th}$ ▷ Memory cost per saved watt threshold
Ensure: $C_{sel}$ ▷ Best configs selected
1: for each latency bin $lat$ do
2:     // First filter configs by worst-case latency
3:     $C_{fltr1} \leftarrow c$ in $C_{all}[lat]$ if $WorstLat(c) < lat$
4:     // Then filter configs by near-minimal power
5:     $P_{min} \leftarrow Min([Power(c)$ for $c$ in $C_{fltr1}])$
6:     $C_{fltr2} \leftarrow c$ in $C_{fltr1}$ if $Power(c) < P_{min} + P_{th}$
7:     // Finally select config with minimal memory
8:     $C_{cand} \leftarrow Argmin([Mem(c)$ for $c$ in $C_{fltr2}])$
9:     // See if Memory cost per saved watt is worthy
10:    if $Mem(C_{cand}) > Mem(C_{sel}[lastLat])$
       and $\frac{Mem(C_{cand})-Mem(C_{sel}[lastLat])}{Power(C_{sel}[lastLat])-Power(C_{cand})} > M_{th}$ then
11:        $C_{sel}[lat] \leftarrow C_{sel}[lastLat]$
12:    else
13:        $C_{sel}[lat] \leftarrow C_{cand}$
14:    end if
15:    $lastLat = lat$
16: end for
```

our experiments, the 2D Pareto boundary slices in power-memory space are generally not smooth, such that conventional intuitive methods preferring "turning points" would not work. Therefore, we designed a concise algorithm detailed in Algorithm 1. For each latency bin (Line 1), we first filter all the configurations according to their worst-case latency (Line 3). The worst-case latency is obtained by overwriting a configuration's frequency settings to maximum and executing it under a pre-defined worst-case outer memory bandwidth level. Then, the selected configurations are further filtered based on power usage (lines 4–6). Finally, a candidate with minimal memory usage is selected (Line 8). However, an inconsistent situation may happen due to insufficient exploration or randomness during learning. This candidate may consume much more memory compared to the previously picked one in the last latency bin with marginal power saving. To avoid such inconsistency, we allow memory usage growth only if the subsequent memory increase per saved watt is less than M_{th} (Line 10). Otherwise, we reuse the selected configuration of the previous latency bin instead (Line 11). Eventually, the C_{sel} of each possible latency constraint bin is saved as a reference table for online lookup. Several hyper-parameters including P_{th} and M_{th} can be user-defined, bringing flexibility to customize for different trade-off considerations.

4.2.2 Profiling Latency Sensitivity Characteristics. Since we will tweak the frequency setting of a selected configuration for better real-time ability, there has to be extra information regarding its latency sensitivity toward frequency tuning. Therefore, the analyzer also characterizes system latency sensitivity under various running conditions through profiling. We choose to collect the latency of each DNN type running on any accelerator (GPU or DLA) under each frequency setting while taking *IMC* and *OMC* into consideration. The information is organized into a nested-hash-table data structure, *characterization table*, for lightweight online lookup.

In more detail, we build a characterization table for each DNN type involved in the AMS perception stage. Although there can be many independent DNNs in total, as mentioned in Section 2.1,

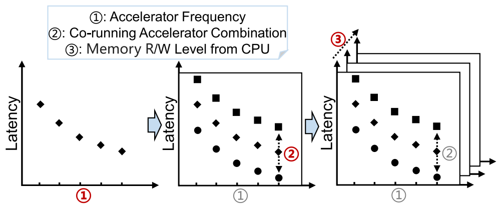

Fig. 9. The construction process of the characterization table corresponding to three system status.

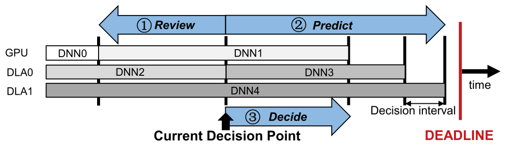

Fig. 10. Illustration of multiple DNN execution and the three steps (review, predict, and decide) during tweaking at each decision boundary point.

most of them would be independent instances of the same DNN type, like YOLO for object detection and Resnet for image classification. Therefore, the number of characterization tables can be kept small. In our implementation, we consider convolution-based DNN architectures for image processing. We execute each DNN type on our target Jetson board and measure its latency-frequency curve at all possible accelerator frequency levels (see the leftmost part of Figure 9). The characterization table contains multiple such curves under several typical GPU-DLA-CPU co-running setups to guide frequency fine-tuning in different system status (Section 4.3.2). Its building procedure is illustrated as follows.

First, to incorporate IMC, we measure the latency-frequency curve under several typical accelerator co-running setups. Suppose our target DNN runs on the GPU. We profile accelerator co-running setups, including GPU running alone, with 1 DLA or 2 DLAs. To reduce the amount of profiling, the co-running DLAs execute YOLO at either the highest or the lowest frequency. (Note that the frequency of the DLAs has to be the same.) A sketch of the resulting curves is shown in the middle part of Figure 9. During runtime, when the co-running accelerator is not at the highest or lowest frequency, we can interpolate the corresponding latency-frequency curve. During runtime, when the co-running accelerator is not executing YOLO, we make a compromise by still using the profiled latency. Evaluation in Section 6 shows that the overall deadline satisfaction is still good.

Second, to incorporate OMC, we repeat the first step under several typical CPU memory bandwidth usage levels. In our implementation, we run 0–5 memory stressors introduced in Section 3.2.2 to create six stable OMC levels. This results in the final three-dimensional table shown in the rightmost part of Figure 9. The characterization table is platform-specific and has to be rebuilt when using new hardware platforms. However, the rebuilding only takes 1–2 hours for each DNN type, which is small and easily affordable during the offline phase.

4.3 Online Adaptor Component

4.3.1 Selecting Base Configuration Periodically. Selector switches the base configuration in response to two circumstances: 1) Latency constraint varies due to changes in the working scenario; 2) Outer memory interference would experience a large and steady change due to the launch of a new long-term workload. The selection process follows the typical feedback control procedures [18] and is triggered periodically. To decouple the management of accelerators from the CPU, we design the selector to be unaware of the launch of a CPU workload. It conservatively computes the 90th tail latency slowdown during the last period as the estimation of the current outer contention level. It then decides whether to switch configurations by looking up the reference table generated by the cherry-picking algorithm in Section 4.2.1. The base configuration selected contains both DNN-accelerator mapping and accelerator frequency setting. Since alternating DNN-accelerator mapping incurs much overhead, the selection can only happen infrequently. Rapid changes or bursts are handled by the tweaker in the next section.

4.3.2 Tweaking Configuration Frequency Continuously. Tweaker fine-tunes frequency setting of a base configuration in case of transient outer memory interference. Different from fine-grained per-layer **dynamic voltage and frequency scaling** (**DVFS**) [13] or coarse-grained per-execution DVFS [22, 63], we design a medium-grained DVFS mechanism. More specifically, we find that per-layer DVFS suffers from non-negligible overheads due to its unnecessary fine granularity. On Jetson Xavier, tuning hardware frequency takes ∼1 ms, 10× larger than the execution time of one DNN layer. On the contrary, the per-execution DVFS approach only boosts execution once for all DNNs, which could be too coarse-grained to handle short-term interference.

We extend LAG analysis [11, 13, 52], widely used in real-time systems, to the heterogeneous hardware scenarios. In LAG analysis, *LAG* describes how far ahead or behind a task is compared to the deadline. It is defined as the difference between the predicted task finish time and task deadline. A controller could then boost or slow down the system according to the *LAG* value at any time. The most critical step in LAG analysis lies in finish time prediction. A^2 makes boost and slowdown decisions at the end of each DNN inference task. As shown in Figure 10, the end of each DNN inference task divides the whole execution period into multiple decision intervals. At each interval boundary, the progressive mechanism takes three steps to decide a proper adjustment for the next interval.

In the first step, the latency of the last execution interval is reviewed and compared to the characterization table to infer the current level of OMC. The review window contains the latest three decision intervals and takes their average OMC level as the inferred current one. This helps reflect the most recent OMC status. Since most intervals would not witness a full DNN execution (e.g., DNN2), we could not directly compare a measured interval to data stored in the characterization table. This calls for the maintenance of the progress of each DNN at each decision boundary. The progress of DNNs that have not yet been finished is updated according to the inferred OMC level. In more detail, at the “current decision point” shown in Figure 10, we first derive the current contention level utilizing the characterization table. For a given co-running accelerator combination and accelerator frequency, the characterization table provides a one-to-one mapping between DNN latency and OMC level. Denote the length of the last decision interval as t_{last} and the progress of DNN2 at the last decision point as p_{DNN2}. We compare $t_{last}/(1-p_{DNN2})$ to the closest latency of DNN2 stored in the characterization table and take the corresponding OMC level as the inferred current level. Then, we can use the inferred OMC level to update the progress of DNN1 and DNN4. Take DNN4 as an example, denote its latency at the inferred OMC level stored in the characterization table as t_{DNN4}, then we can update p_{DNN4} such that $p_{DNN4} += t_{last}/t_{DNN4}$.

In the second step, the remaining execution time is predicted to decide whether to boost, slow down, or do nothing. The predict window covers all remaining execution time in the current multi-DNN execution period. It assumes the execution still follows the base configuration later and consults the characterization table to get the execution time of each future interval iteratively. All latency values of future decision intervals are accumulated to get the finish time prediction. We can then calculate *LAG* and tune frequency accordingly. Additionally, we find that there might be incorrigible deadline violations during each multi-DNN execution period. Later decision points in each period may be unable to reduce latency sufficiently even if it has boosted accelerator frequencies to the highest value when an unexpectedly high OMC spike appears. As a result, we have to tune accelerator frequencies to higher values than just enough at early decision points. Therefore, we multiply a *conservative factor*, which is larger than 1, to the predicted remaining execution time at each decision point. We calculate *LAG* using this amplified remaining execution time prediction to decide the accelerator frequencies before the next decision point. The initial value of the conservative factor at the start of each multi-DNN execution period is set as a fixed

Table 2. Experiment Platform Specification

Device	Nvidia Jetson Xavier NX
CPU	6-core Carmel ARM v8.2
GPU	384-core Volta GPU with 48 Tensor cores
Memory	8GB LPDDR4x shared mem (59.7 GB/s)
Accelerator	2x NVDLAs
System	Jetpack 4.6.2, Ubuntu 18.0.4, TensorRT 8.2

Table 3. Description of Evaluated Schemes

Scheme	**Descriptions**
Race-to-idle [33]	*Highest freq then idle, GPU+2DLA*
NeuOS [11]	*Fine-grained frequency tweaking only, GPU+1DLA*
CALOREE [45]	*Pareto learner + Periodic selector*
A^2	*Analyzer (learner+profiler) + Adaptor (selector+tweaker)*
A^2_power	*A^2 minimizing power*
A^2_mem	*A^2 minimizing mem*

hyperparameter. Its value drops to 1 linearly with time until the deadline of each period, which helps avoid over-compensation.

In the last step, a proper frequency for the next decision interval is decided. The decision window only includes the time until the next DNN finishes (a single decision interval). The corresponding frequency tuning register in the Linux */sys* directory is modified to apply the frequency change. We implement the progressive tweaker to execute asynchronously with workloads in a separate thread. So, the management logic will never affect the critical path of system execution.

5 Evaluation Methodology

5.1 Platform and Benchmarks

We implement A^2 on a commercial edge platform for AMS, NVIDIA Jetson Xavier NX, detailed in Table 2. This platform originally provides nine hierarchical power modes with 10/15/20 w power, and we choose the maximum one *MODE_20W_6CORE* as the basic power mode, overwriting its default frequency settings. Both GPU and DLAs have 12 optional frequency levels. An onboard power meter (*TI INA3221x*) updated at 5 ms intervals captures power readings with default smoothing mode disabled. All power consumption values are dynamic power of acceleration subsystem, obtained by subtracting static power reading from *5V_IN* power rail readings. System static power is about 4.2 watts when the CPU is idle.

Due to the unnecessary complexity of commercial systems, we mimic the perception stage of AMS with a mix of YOLOv3-416 and Resnet101 (detailed in Table 1) to evaluate A^2 (memory usage of VGG16 is too huge to adopt). Two different setups are evaluated: 12 DNNs (5 YOLO and 7 Resnet) and 16 DNNs (10 YOLO and 6 Resnet). Since DLA does not support FP32, we both use FP16 on GPU and DLA for fairness. We generate DNN engines with *TensorRT* instead of *Pytorch* or *Tensorflow* since it is specialized for high-performance inference and supports DLA better. For compactness, we evenly divide all latency constraints tested into tight/mid/loose ranges. For 12(16) DNN setups, the minimal latency constraint is set to 150(240) ms, and each range contains 5(7) latency constraints with 10 ms intervals. The result obtained for each range is an averaged value of 5 or 7 latency constraints.

5.2 A^2 Design Specification

Our implementation takes about 1 k LoCs in C++ and 2 k LoCs in Python. The engines for accelerators are compiled in advance for deployment. The offline Analyzer is implemented in Python, while the online Adaptor is in C++ for fast execution. During online execution, we additionally create stress-ng memory stressors on the CPU as a synthetic interference generator. We use synthetic generators for better controllability. The generator has both steady and transient variations to simulate the real CPU workloads of an AMS.

In more detail, we designed the following process to simulate a real AMS working condition in a controlled manner. Starting from the idle state, the number of CPU memory stressors is first increased every 30 s until five and then decreased until zero. Some additional CPU memory stressors are randomly generated to simulate spikes. In the meantime, DNN inference tasks run

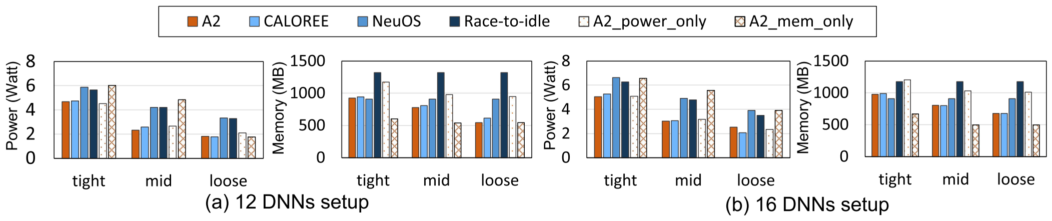

Fig. 11. A comparison of power and memory optimization effectiveness. (Lower is better.)

periodically on accelerators, and a resource manager adjusts configuration dynamically. For A^2 and *Learn&Control* managers, a configuration switch happens every 45 s. We choose 45 s to avoid the selector working in lockstep with the increase(decrease) of memory stressors. Otherwise, the selector would behave very poorly since its contention level estimation would always be wrong.

The hyper-parameters in the GA, including iteration and population size, are set to 80. In Algorithm 1, P_{th} is set to 0.5, and M_{th} is set to 300. The initial value of the conservative factor is set to 1.2 and will linearly decay to 1.

5.3 Points of Comparison

We compare A^2 to five baselines as shown in Table 3. *Race-to-idle* [33] is simple yet proven to be effective in many scenarios. It evenly distributes DNNs to accelerators at the highest frequency. Once the execution of all DNNs is finished, the accelerators become idle until the latency constraint is reached. Control-based manager resembles *NeuOS* [11], which employs LAG analysis to tune frequency under fixed scheduling for all latency constraints. This fixed scheduling configuration is chosen to use 1 GPU and 1 DLA in a load-balanced way under the highest frequency since this setup shows the maximal achievable latency range when adjusting frequency. For Learn&Control schemes, we choose to transport *CALOREE* [45], which learns suitable configuration candidates offline and uses feedback control online to select among them. We integrate the learner and selector in A^2 as the *CALOREE* manager. Also, we compare two variants of A^2. During the cherry-picking stage in Section 4.2.1, A^2*_power*(A^2*_mem*) simply chooses a configuration with minimal power(memory) regardless of memory(power) usage. They are used to examine *"what if the trade-off is completely ignored"*. A purely Learning-based scheme is not compared since it is doomed to have poor real-time performance.

6 Evaluation Results

6.1 Overall Effectiveness

We evaluate five important effectiveness indicators: average power consumption (*Power*) and memory usage (*Memory*) for efficiency; deadline violation rate (*ddl_vio_rate*) and 99th-percentile (P99) of absolute deadline violation extent (*ddl_vio_extent*) for timeliness.

Figure 11 shows the power and memory optimization. In both 12 and 16 DNN setups, power consumption and memory usage of A^2 and *CALOREE* are both roughly the lowest among the four main baselines. Compared to *NeuOS*, the power consumption of A^2 is 32.8% less, and the memory footprint is 13.8% smaller on average. The large memory usage in the "loose" setup and high power consumption of *NeuOS* is caused by its inability to adjust DNN mapping dynamically. The other baseline method, *Race-to-idle*, never exceeds latency constraint but performs poorly in terms of both power and memory. Thus, we argue that blindly using GPU with mid/loose latency constraint as *Race-to-idle* does wastes power and memory. Although *CALOREE* achieves comparable performance and efficiency with A^2, we show that it falls behind in the timeliness guarantee.

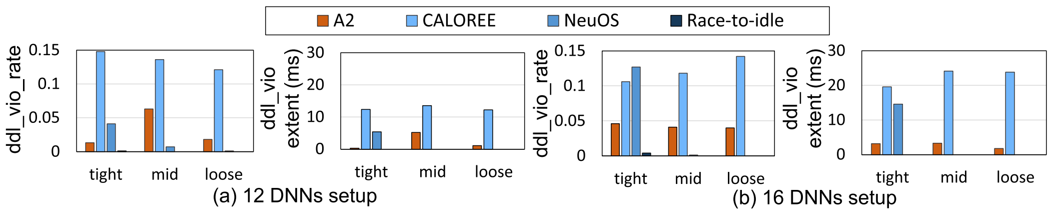

Fig. 12. A comparison of deadline violation rate and extent. (Lower is better.) The configurations used are the same as Figure 11. Although *Race-to-idle* has almost no violation, it behaves poorly in efficiency optimization.

Figure 12 shows the timeliness guarantee comparison. With specialized tweaker, A^2 violates deadline less than 5% in most setups (3.6% on average), and the p99 value of deadline violation extent is almost always less than 5 ms. As for comparison, the deadline violation rate of *CALOREE* can be up to 15%, and the violation p99 value can be 10–25 ms higher than latency constraints. This implies that vanilla Learn&Control methods are ineffective under spikes and rapid changes. The deadline violation rate of *NeuOS* in the "tight" range is high due to its inability to enable more accelerators. This shows that merely tuning frequency leads to reduced schedulability.

Finally, both A^2*_power* and A^2*_mem* perform well on one aspect, sacrificing the other. The memory usage of A^2*_power* is the second largest among all evaluated schemes in the 12 DNN setup, while the power consumption of A^2*_mem* is the highest one in 16 DNN setups. Some may notice that the power consumption of A^2*_power* is sometimes slightly higher than A^2. The reason is that bars shown in Figure 11 are averaged values under several adjacent latency constraints. In summary, completely ignoring trade-offs brings very skewed results.

6.2 Case Study

We further conduct a case study to demonstrate the effectiveness of A^2 under a more realistic runtime condition setup. As mentioned in Section 4, A^2 can handle run-time dynamicity in various executing scenarios. For example, an AMS may execute tasks in an aperiodic or irregular manner for environment adaptation or human interaction. We assume that all such unpredictable tasks are executed on the CPU or hardware accelerators aside from the GPU and NPUs (or DLAs on Jetson). These co-running workloads bring run-time performance interference to the fixed set of DNNs, which is addressed by A^2. The capability of adapting to unpredictable DNNs for AMS perception is left for future work. In this case study, we select several popular real-world co-running workloads used in AMS and deploy them together with the A^2-managed DNNs. We show that A^2 can adapt well under the unpredictable performance interference by these irregular co-running tasks.

We select the following three co-running workloads popular in AMS. *NDT localization* (NDT) [5] performs high-resolution localization using point-cloud maps. It is multi-threaded and executes on the CPU. ***Dense Optical Flow*** (**DOF**) [7] detects fast-moving objects leveraging the frame difference of image sequences. It executes on a specialized accelerator named PVA (Programmable Vision Accelerator) on Jetson. ***Temporal Noise Reduction*** (**TNR**) [8] performs denoising on high-resolution video. It executes on a specialized accelerator named **VIC** (**Video Image Compositor**) on Jetson. All of them are heavy-weight tasks introducing non-negligible performance degradation to the DNNs. We execute them using the sample input data fragment provided by their respective developers. Their execution time is 126, 45, and 8 s, respectively. To maintain reproducibility while creating an irregular co-running combination, we deploy the three tasks periodically but at distinct periods of 200, 80, and 30 s in a total evaluation duration of 400 s. Note that A^2 does not assume the periodicity of the co-running tasks and handles them as black boxes. We follow the 12 DNN

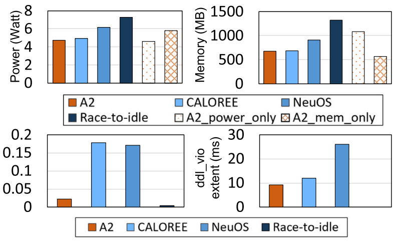

Fig. 13. Case study results with OMC from real-world non-predictable workloads.

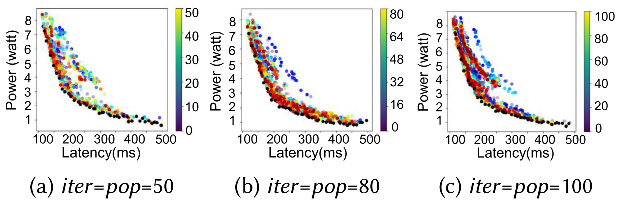

Fig. 14. Pareto boundaries in power-performance space. The color bar denotes the iteration number.

setup. To thoroughly test the adaptation capability of A^2, we also vary the latency constraint of the DNNs in the meantime. The latency constraint is set to 150 ms initially and increases by 50 ms every 100 s. All other evaluation setups are the same as Section 2.2.

The evaluation results are shown in Figure 13. We measure the same effectiveness indicators as Section 6.1. The power consumption and memory space usage of A^2 are both the lowest compared to the baselines. The deadline violation rate of A^2 is merely 2.2%, and the p99 value of deadline violation extent is less than 10 ms. This proves the effectiveness of A^2 under a more realistic runtime condition setup with irregular co-running tasks.

6.3 Detailed Analysis

6.3.1 Sensitivity of Learning Hyper-parameters. We customized a GA, NSGA-II, to find the 3D Pareto boundary in Section 4.2.1. The hyper-parameters used especially iterations (*iter*) and population size (*pop*), are important for fast and sufficient configuration space exploration. Figure 14 shows the projection of every explored configuration onto power-performance space under three different hyper-parameter settings. Bluish points are explored in early iterations, while reddish ones are explored later. Configurations lying on the 2D Pareto boundary are colored black.

Under all three setups, the GA produces a Pareto boundary with a similar good shape. This implies the learner is not sensitive to the two hyper-parameters. We choose 80 in all experiments since it yields slightly better results, balancing between exploration sufficiency and execution time. For an A^2 user, we recommend first selecting "pop" to any value between 50–100 and then incrementally increasing "iter" without restart. Plotting exploration status can help in deciding when to stop based on the quality of the current results.

6.3.2 Sensitivity of Conservative Factor. In Section 4.3.2, we introduced a linearly decaying *"conservative factor"* to make conservative decisions during the early execution. In this part, we investigate the influence of its initial value. A large initial value may lead to high power usage, while a small one can result in deadline violation. In Figure 15, we set the initial value to 1/1.2/1.5 and plot the deadline violation rate and p99 of absolute deadline violation in 12 DNNs setup.

Not incorporating a "conservative factor" (setting to 1) leads to large deadline violation rates, which go up to 17% in the mid case. Increasing it to 1.2 reduces the deadline violation rate greatly with negligible increase in power consumption. Further increasing it to 1.5 yields better p99 violation but raises power consumption slightly by 0.2 watts. Therefore, the conservative factor is important, and setting it to 1.2–1.5 would be good enough depending on the expected deadline satisfaction requirement. We choose 1.2 in all experiments.

6.3.3 Zoomed-in Analysis on Tweaking Effect. We show the effectiveness of A^2 by presenting detailed latency curves under tight and loose latency constraints in Figure 16 (mid constraints are shown in Figure 6). The experiment setup is the same as 3.3.

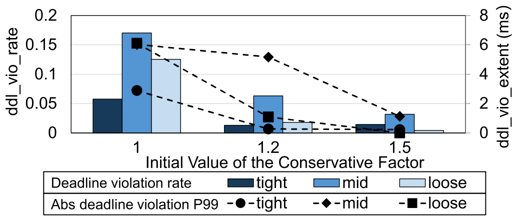

Fig. 15. Real-time performance comparison of different conservative factors.

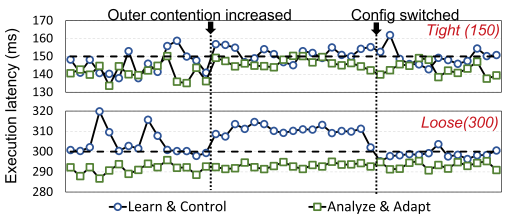

Fig. 16. Latency curves under 150 and 300 ms constraints.

A^2 does not suffer from severe deadline violation in all three cases. The deadline violation extent in loose case may seem to be more severe (20 ms) than in tight case (10 ms), but the violation extent in percentage are similar at ~7% (also for Figure 6). The latency curve of A^2 does not show spikes when a configuration switch is triggered. During a configuration switch, new engines are first loaded by the CPU, and useless engines are flushed. This could cause extra memory bandwidth. However, the online tweaking scheme deals with such extra contention as well as handling the two transient spikes before outer contention increases.

6.3.4 Overhead Analysis. The online tweaker must be lightweight. In our implementation, each decision-making process takes <1 ms, which is small compared to the typical DNN execution time of 20–50 ms. The decision-making logic executes asynchronously with DNN workloads and is not on the critical path. The overhead of the online selector and tweaker on CPU utilization is also negligible. The execution time of the GA ranges between 1–8 hours, depending on hyper-parameter selection, and characterization table profiling takes about 1–2 hours.

7 Conclusions

Emerging AMS raise new challenges for achieving ALP to optimize system efficiency and guarantee management timeliness. In this work, we propose A^2, a well-crafted resource manager covering both workload analysis and real-time adaption. It achieves 32.8% improvements in power and 13.8% in memory compared with control-based methods. It also reduces the deadline violation rate by 9.2 percentage points on average compared to directly porting *Learn&Control* methods. We expect that our design will have a transformative impact on AMS with heterogeneous AI accelerators.

Acknowledgments

We sincerely thank all the anonymous reviewers for their valuable comments. The corresponding author is Chao Li.

References

[1] 2019. NVIDIA Tegra Xavier. Retrieved from https://en.wikichip.org/wiki/nvidia/tegra/xavier
[2] 2022. Retrieved from https://www.nvidia.cn/self-driving-cars/drive-platform/hardware/
[3] 2022. Stress-ng Test Tool. Retrieved from https://github.com/ColinIanKing/stress-ng/tree/a71aad422b1b934981d10e7b3be866b2744f19c7
[4] 2023. Europe's 6-wheeled Delivery Robots Begin Invasion of US Campuses. Retrieved from https://sifted.eu/articles/starship-robot-delivery/
[5] 2023. NDT Localization. Retrieved from https://github.com/koide3/hdl_localization
[6] 2024. Mobileye EyeQ. Retrieved from https://www.mobileye.com/technology/eyeq-chip/
[7] 2024. VPI Dense Optical Flow. Retrieved from https://docs.nvidia.com/vpi/sample_optflow_dense.html
[8] 2024. VPI Temporal Noise Reduction. Retrieved from https://docs.nvidia.com/vpi/sample_tnr.html

[9] Aporva Amarnath, Subhankar Pal, Hiwot Kassa, Augusto Vega, Alper Buyuktosunoglu, Hubertus Franke, John-David Wellman, Ronald Dreslinski, and Pradip Bose. 2022. HetSched: Quality-of-mission aware scheduling for autonomous vehicle SoCs. arXiv:2203.13396. Retrieved from https://arxiv.org/abs/2203.13396

[10] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. 2020. PipeSwitch: Fast pipelined context switching for deep learning applications. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation.* 499–514.

[11] Soroush Bateni and Cong Liu. 2020. NeuOS: A latency-predictable multi-dimensional optimization framework for DNN-driven autonomous systems. In *Proceedings of the 2020 USENIX Annual Technical Conference.* 371–385.

[12] Soroush Bateni, Zhendong Wang, Yuankun Zhu, Yang Hu, and Cong Liu. 2020. Co-optimizing performance and memory footprint via integrated CPU/GPU memory management, an implementation on autonomous driving platform. In *Proceedings of the 2020 IEEE Real-Time and Embedded Technology and Applications Symposium.* IEEE, 310–323.

[13] Soroush Bateni, Husheng Zhou, Yuankun Zhu, and Cong Liu. 2018. Predjoule: A timing-predictable energy optimization framework for deep neural networks. In *Proceedings of the 2018 IEEE Real-Time Systems Symposium.* IEEE, 107–118.

[14] Behzad Boroujerdian, Hasan Genc, Srivatsan Krishnan, Wenzhi Cui, Aleksandra Faust, and Vijay Reddi. 2018. Mavbench: Micro aerial vehicle benchmarking. In *Proceedings of the 2018 51st annual IEEE/ACM International Symposium on Microarchitecture.* IEEE, 894–907.

[15] Shuo Cheng, Liang Li, Xiang Chen, Jian Wu, and Hong-da Wang. 2020. Model-predictive-control-based path tracking controller of autonomous vehicle considering parametric uncertainties and velocity-varying. *IEEE Transactions on Industrial Electronics* 68, 9 (2020), 8698–8707.

[16] Ismet Dagli, Alexander Cieslewicz, Jedidiah McClurg, and Mehmet E. Belviranli. 2022. AxoNN: Energy-aware execution of neural network inference on multi-accelerator heterogeneous SoCs. In *Proceedings of the 59th ACM/IEEE Design Automation Conference.* 1069–1074.

[17] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (2002), 182–197.

[18] Kalyanmoy Deb, Udaya Bhaskara Rao N, and Sindhya Karthik. 2007. Dynamic multi-objective optimization and decision-making using modified NSGA-II: A case study on hydro-thermal power scheduling. In *Proceedings of the International Conference on Evolutionary Multi-criterion Optimization.* Springer, 803–817.

[19] Yi Ding, Nikita Mishra, and Henry Hoffmann. 2019. Generative and multi-phase learning for computer systems optimization. In *Proceedings of the 46th International Symposium on Computer Architecture.* 39–52.

[20] Bryan Donyanavard, Tiago Mück, Amir M. Rahmani, Nikil Dutt, Armin Sadighi, Florian Maurer, and Andreas Herkersdorf. 2019. Sosa: Self-optimizing learning with self-adaptive control for hierarchical system-on-chip management. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture.* 685–698.

[21] Sandeep D'souza and Ragunathan Rajkumar. 2018. CycleTandem: Energy-saving scheduling for real-time systems with hardware accelerators. In *Proceedings of the 2018 IEEE Real-Time Systems Symposium.* IEEE, 94–106.

[22] Anne Farrell and Henry Hoffmann. 2016. MEANTIME: Achieving both minimal energy and timeliness with approximate computing. In *Proceedings of the 2016 USENIX Annual Technical Conference.* 421–435.

[23] David E. Goldberg and Robert Lingle. 1985. Alleles, loci, and the traveling salesman problem. In *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, Vol. 154. Lawrence Erlbaum Hillsdale, NJ, 154–159.

[24] Adam W Harley, Zhaoyuan Fang, Jie Li, Rares Ambrus, and Katerina Fragkiadaki. 2023. Simple-bev: What really matters for multi-sensor bev perception? In *Proceedings of the 2023 IEEE International Conference on Robotics and Automation.* IEEE, 2759–2765.

[25] Seonyeong Heo, Sungjun Cho, Youngsok Kim, and Hanjun Kim. 2020. Real-time object detection system with multipath neural networks. In *Proceedings of the 2020 IEEE Real-Time and Embedded Technology and Applications Symposium.* IEEE, 174–187.

[26] Mark D. Hill and Vijay Janapa Reddi. 2021. Accelerator-level parallelism. *Communications of the ACM* 64, 12 (2021), 36–38.

[27] Junjie Huang, Guan Huang, Zheng Zhu, Yun Ye, and Dalong Du. 2022. Bevdet: High-performance multi-camera 3d object detection in bird-eye-view. arXiv:2112.11790. Retrieved from https://arxiv.org/abs/2112.11790

[28] Connor Imes, David HK Kim, Martina Maggio, and Henry Hoffmann. 2015. POET: A portable approach to minimizing energy under soft real-time constraints. In *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium.* IEEE, 75–86.

[29] Wonseok Jang, Hansaem Jeong, Kyungtae Kang, Nikil Dutt, and Jong-Chan Kim. 2020. R-TOD: Real-time object detector with minimized end-to-end delay for autonomous driving. In *Proceedings of the 2020 IEEE Real-Time Systems Symposium.* IEEE, 191–204.

[30] Mingoo Ji, Saehanseul Yi, Changjin Koo, Sol Ahn, Dongjoo Seo, Nikil Dutt, and Jong-Chan Kim. 2022. Demand layering for real-time DNN inference with minimized memory usage. In *Proceedings of the 2022 IEEE Real-Time Systems Symposium*. IEEE, 291–304.

[31] Woosung Kang, Kilho Lee, Jinkyu Lee, Insik Shin, and Hoon Sung Chwa. 2021. Lalarand: Flexible layer-by-layer CPU/GPU scheduling for real-time dnn tasks. In *Proceedings of the 2021 IEEE Real-Time Systems Symposium*. IEEE, 329–341.

[32] Sheng-Chun Kao and Tushar Krishna. 2022. MAGMA: An optimization framework for mapping multiple DNNs on multiple accelerator cores. In *Proceedings of the 2022 IEEE International Symposium on High-Performance Computer Architecture*. IEEE, 814–830.

[33] David HK Kim, Connor Imes, and Henry Hoffmann. 2015. Racing and pacing to idle: Theoretical and empirical analysis of energy optimization heuristics. In *Proceedings of the 2015 IEEE 3rd International Conference on Cyber-physical Systems, Networks, and Applications*. IEEE, 78–85.

[34] Srivatsan Krishnan, Zishen Wan, Kshitij Bhardwaj, Paul Whatmough, Aleksandra Faust, Sabrina Neuman, Gu-Yeon Wei, David Brooks, and Vijay Janapa Reddi. 2022. Automatic domain-specific SoC design for autonomous unmanned aerial vehicles. In *Proceedings of the 2022 55th IEEE/ACM International Symposium on Microarchitecture*. IEEE, 300–317.

[35] Hyoukjun Kwon, Liangzhen Lai, Michael Pellauer, Tushar Krishna, Yu-Hsin Chen and Vikas Chandra. 2020. Heterogeneous dataflow accelerators for Multi-DNN Workloads. arXiv:1909.07437. Retrieved from https://arxiv.org/abs/1909.07437

[36] Hyoukjun Kwon, Liangzhen Lai, Michael Pellauer, Tushar Krishna, Yu-Hsin Chen, and Vikas Chandra. 2021. Heterogeneous dataflow accelerators for multi-DNN workloads. In *Proceedings of the 2021 IEEE International Symposium on High-Performance Computer Architecture*. IEEE, 71–83.

[37] Yen-Lin Lee, Pei-Kuei Tsung, and Max Wu. 2018. Techology trend of edge AI. In *Proceedings of the 2018 International Symposium on VLSI Design, Automation and Test*. IEEE, 1–2.

[38] Tingting Liang, Hongwei Xie, Kaicheng Yu, Zhongyu Xia, Zhiwei Lin, YongtaoWang, Tao Tang, BingWang, and Zhi Tang. 2022. Bevfusion: A simple and robust lidar-camera fusion framework. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems*. 10421–10434.

[39] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E Haque, Lingjia Tang, and Jason Mars. 2018. The architectural implications of autonomous driving: Constraints and acceleration. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*. 751–766.

[40] Liangkai Liu, Zheng Dong, Yanzhi Wang, and Weisong Shi. 2022. Prophet: Realizing a predictable real-time perception pipeline for autonomous vehicles. In *Proceedings of the 2022 IEEE Real-Time Systems Symposium*. IEEE, 305–317.

[41] Shaoshan Liu, Jie Tang, Zhe Zhang, and Jean-Luc Gaudiot. 2017. Computer architectures for autonomous driving. *Computer* 50, 8 (2017), 18–25.

[42] Zhijian Liu, Haotian Tang, Alexander Amini, Xinyu Yang, Huizi Mao, Daniela L Rus, and Song Han. 2023. Bevfusion: Multi-task multi-sensor fusion with unified bird's-eye view representation. In *Proceedings of the 2023 IEEE International Conference on Robotics and Automation*. IEEE, 2774–2781.

[43] Raju Machupalli, Masum Hossain, and Mrinal Mandal. 2022. Review of ASIC accelerators for deep neural network. *Microprocessors and Microsystems* 89, 1 (2022), 104441.

[44] Pablo Antonio Martínez, Gregorio Bernabé, and Jose Manuel García. 2022. POAS: A high-performance scheduling framework for exploiting accelerator level parallelism. arXiv:2209.10245. Retrieved from https://arxiv.org/abs/2209.10245

[45] Nikita Mishra, Connor Imes, John D Lafferty, and Henry Hoffmann. 2018. Caloree: Learning control for predictable latency and low energy. *ACM SIGPLAN Notices* 53, 2 (2018), 184–198.

[46] Mohammad Alaul Haque Monil, Mehmet E. Belviranli, Seyong Lee, Jeffrey S. Vetter, and Allen D. Malony. 2020. MEPHESTO: Modeling energy-performance in heterogeneous SoCs and their trade-offs. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. 413–425.

[47] Eriko Nurvitadhi, David Sheffield, Jaewoong Sim, Asit Mishra, Ganesh Venkatesh, and Debbie Marr. 2016. Accelerating binarized neural networks: Comparison of FPGA, CPU, GPU, and ASIC. In *Proceedings of the 2016 International Conference on Field-Programmable Technology*. IEEE, 77–84.

[48] Jonah Philion and Sanja Fidler. 2020. Lift, splat, shoot: Encoding images from arbitrary camera rigs by implicitly unprojecting to 3d. In *Proceedings of the Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XIV 16*. Springer, 194–210.

[49] Aditya Prakash, Kashyap Chitta, and Andreas Geiger. 2021. Multi-modal fusion transformer for end-to-end autonomous driving. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 7077–7087.

[50] Yuqiong Qi, Yang Hu, Haibin Wu, Shen Li, Haiyu Mao, Xiaochun Ye, Dongrui Fan, and Ninghui Sun. 2021. Tackling variabilities in autonomous driving. arXiv:2104.10415. Retrieved from https://arxiv.org/abs/2104.10415

[51] William M Spears and Kenneth A De Jong. 1991. An analysis of multi-point crossover. In *Proceedings of the Foundations of Genetic Algorithms*. Vol. 1. Elsevier, 301–315.

[52] Ion Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy K. Baruah, Johannes E. Gehrke, and C. Greg Plaxton. 1996. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*. IEEE, 288–299.

[53] Chengcheng Wan, Muhammad Santriaji, Eri Rogers, Henry Hoffmann, Michael Maire, and Shan Lu. 2020. ALERT: Accurate learning for energy and timeliness. In *Proceedings of the 2020 USENIX Annual Technical Conference*. 353–369.

[54] Yidi Wang, Mohsen Karimi, Yecheng Xiang, and Hyoseung Kim. 2021. Balancing energy efficiency and real-time performance in GPU scheduling. In *Proceedings of the 2021 IEEE Real-Time Systems Symposium*. IEEE, 110–122.

[55] Yecheng Xiang and Hyoseung Kim. 2019. Pipelined data-parallel CPU/GPU scheduling for multi-DNN real-time inference. In *Proceedings of the 2019 IEEE Real-Time Systems Symposium*. IEEE, 392–405.

[56] Cheng Xu, Xiaofeng Hou, Jiacheng Liu, Chao Li, Tianhao Huang, Xiaozhi Zhu, Mo Niu, Lingyu Sun, Peng Tang, Tongqiao Xu, K.-T. Tim Cheng, and Minyi Guo. 2023. MMBench: Benchmarking end-to-end multi-modal DNNs and understanding their hardware-software implications. In *Proceedings of the 2023 IEEE International Symposium on Workload Characterization*. IEEE, 154–166.

[57] Yuanchao Xu, Mehmet Esat Belviranli, Xipeng Shen, and Jeffrey Vetter. 2021. PCCS: Processor-centric contention-aware slowdown model for heterogeneous system-on-chips. In *Proceedings of the MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1282–1295.

[58] Ming Yang, Shige Wang, Joshua Bakita, Thanh Vu, F. Donelson Smith, James H. Anderson, and Jan-Michael Frahm. 2019. Re-thinking CNN frameworks for time-sensitive autonomous-driving applications: Addressing an industrial challenge. In *Proceedings of the 2019 IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 305–317.

[59] Khalid Yousif, Alireza Bab-Hadiashar, and Reza Hoseinnezhad. 2015. An overview to visual odometry and visual SLAM: Applications to mobile robotics. *Intelligent Industrial Systems* 1, 4 (2015), 289–311.

[60] Bo Yu, Wei Hu, Leimeng Xu, Jie Tang, Shaoshan Liu, and Yuhao Zhu. 2020. Building the computing system for autonomous micromobility vehicles: Design constraints and architectural optimizations. In *Proceedings of the 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 1067–1081.

[61] Hengyu Zhao, Yubo Zhang, Pingfan Meng, Hui Shi, Li Erran Li, Tiancheng Lou, and Jishen Zhao. 2020. Driving scenario perception-aware computing system design in autonomous vehicles. In *Proceedings of the 2020 IEEE 38th International Conference on Computer Design*. IEEE, 88–95.

[62] Hengyu Zhao, Yubo Zhang, Pingfan Meng, Hui Shi, Li Erran Li, Tiancheng Lou, and Jishen Zhao. 2020. Safety score: A quantitative approach to guiding safety-aware autonomous vehicle computing system design. In *Proceedings of the 2020 IEEE Intelligent Vehicles Symposium*. IEEE, 1479–1485.

[63] Liang Zhou, Laxmi N Bhuyan, and KK Ramakrishnan. 2020. Gemini: Learning to manage CPU power for latency-critical search engines. In *Proceedings of the 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 637–349.

Received 7 February 2024; revised 24 June 2024; accepted 25 July 2024