Predictable Data-driven Resource Management: an Implementation using Autoware on Autonomous Platforms

Soroush Bateni  
Department of Computer Science  
University of Texas at Dallas  
soroush@utdallas.edu

Cong Liu  
Department of Computer Science  
University of Texas at Dallas  
cong@utdallas.edu

Abstract—Autonomous embedded systems (AES) are becoming prominent in many application domains such as self-driving cars. However, the conflict between the rather limited memory space in such systems and the data intensive nature of the workloads creates hard challenges on data and memory management, which may easily cause unpredictability in outputting autonomous control decisions. In this paper, we target data-driven AES featuring the integrated architecture by establishing a data-centric system model inspired by Heijunka, a mature production leveling methodology developed by Toyota. Based on this new model, we develop ResCue which contains a dynamic data scheduler and a flexible memory reservation scheme to ensure both temporal and spatial data availability, which shall guarantee predictability in generating outputs in terms of both meeting deadlines and minimizing jitters. We implement and extensively evaluate ResCue under various settings using a popular end-to-end self-driving software Autoware on top of the AES-specific NVIDIA AGX Xavier SoC. Results show that ResCue never misses a deadline and yields a maximum jitter of merely 834 microseconds, while incurring rather small overhead. Moreover, ResCue is able to noticeably reduce memory consumption compared to vanilla Autoware.

I. INTRODUCTION

Thanks to recent advancements in machine learning techniques and embedded hardware, autonomous embedded systems (AES) are becoming prominent in many application domains such as self-driving cars. AES are expected to be self-reliant in decision making and achieve full autonomy without user interference, typically achieved through analyzing the large amount of real-time environmental data collected through sensors. Consider an autonomous vehicle for example, which could collect 1GB of data a second and 4000 GB of data per day [1]. Such data are moved in real-time to a computation module typically driven by deep neural networks (DNN) that would transform this data into machine-readable labels (called “perception”) and make decisions based on identified parameters from the environment (called “decision & planning”). This decision is then propagated to an output mechanism, such as actuation modules for handling, gas, and break operations.

The Data Problem. AES are mostly driven by accelerator-enabled multicore platforms, such as NVIDIA’s GPU-based hardware, which is unfortunately restricted by its memory space. The limited memory space may become a worse problem given that the integrated architecture is becoming dominant in the architectural design for AES hardware, due to its programmability and the advantages in size, weight, and power (SWaP) compared to its discrete counterparts [2], [3]. For instance, NVIDIA, Intel, and Tesla are adopting the integrated architecture in their latest AES platforms [4], [5]. Under the integrated architecture, all computing units including CPU, GPU, or any other types of accelerators share the same RAM space. For instance, the latest AES-specific NVIDIA AGX Xavier SoC features eight NVIDIA “Carmel” cores, a 512-core Volta GPU, and two deep learning accelerators (DLA), all sharing a 16GB RAM. Memory could be a severe bottleneck given the data-intensive workloads seen in most AES. For example, for one of the most prominent end-to-end autonomous driving software Autoware [6], the program code for all modules excluding the DNN-driven perception module (i.e., YOLO [7]) can require 16 GB of memory space alone to operate [8]. The entire memory demand from all modules could require a memory space up to 32 GB [9], which far exceeds the maximum memory availability. The conflict between rather limited memory space and data-intensive (thus memory-demanding) workloads creates hard challenges on data and memory management, which may easily cause unpredictability in outputting autonomous control decisions—the most important issue currently faced by AES.

To understand in detail how memory bottleneck can cause unpredictability, imagine Fig. 1, a scenario depicting a partial task set of module execution of Autoware that consists of the...
Perception and the Planning module. In Fig. 1, the input sensing data is received periodically (e.g., one image frame every 33ms from a 30 frame-per-second camera). Upon receiving the input sensing data, a complex task set is released to process the data. For simplicity reasons, only the Perception and Planning abstract modules are shown (we detail a more granular set of Autoware modules in our evaluation in Sec. IV). In the first period between input intervals in Fig. 1, the task set is immediately released because the memory space is available for the code of the Perception and subsequently the Planning module.

However, due to other tasks being released in-between, the memory could have become full when the second input sensing data arrives. Even though the Perception task of the second task set has been technically released at $I_2$, the actual task ready time happens much later because the required computing code needs to be loaded into memory first.\(^1\) In this situation, the system has failed to provide spatial data availability for the program code of Perception. Moreover, once the code data for the Perception module has finished loading into memory, the disk-to-memory transfer channel could become busy due to the case where some other tasks' data transfer requests are scheduled earlier than this specific Planning module, as seen in the figure. Therefore, the Planning task can start execution much later than expected, and will ultimately miss the deadline at $O_2$. In this case, the system has failed to provide temporal data availability, where the data transfer request of a task needs to be scheduled on time for meeting the output predictability requirements.

This example intuitively illustrates the problem where given predictable, periodic input sensing data, rather unpredictable, sporadic output decisions could be made, due to the above-highlighted temporal and spatial data availability issues. Unfortunately, unpredictable (e.g., sporadic instead of periodic) output decisions are generally classified as unexpected behavior pattern in the system [10], [11] which is unacceptable for AES due to potential severe safety concerns, as shown in many previous works from both academia and industry [10], [12]–[15]. For example, autonomous vehicles are required to output control decisions in a periodic manner (the period ranges from 30ms – 300ms depending on different standards [16]–[18]) to ensure safety-critical decision making and to become certifiable [18].

A set of works have been conducted surrounding this problem of ensuring input-output predictability in the general real-time and cyber-physical systems domain [19]–[26]. Unfortunately, these works mostly focused on execution and computation scheduling or enhancing I/O operation efficiency in order to improve predictability and meet deadlines, yet ignoring the critical data and memory availability issues. Some techniques adopted in the industry [27], including Autoware, have relied on operating systems with dynamic memory management techniques to handle limited memory availability, yet these techniques are not predictable [28], [29]. On the other hand, existing memory reservation techniques [30] targeting at timing predictability may not be applicable to this problem context because they do not consider the issue of ensuring predictable outputs intervals given an unpredictable task release due to data and memory unavailability, particularly under the integrated architecture.

**Contributions.** To solve this problem, we found that most existing system models cannot adequately model IO-centric AES, with the exception being system models that are based on the Logical Execution Time paradigm (LET) [19], [20], [31]. LET assumes a scenario where data input and output are periodic with periodic task releases in between. These system models depict the underlying problem that predictability in AES ultimately translates to a constant and predictable output reaction. However, LET and their derivatives are not data-centric approaches, and thus, make the false assumption that periodic input/output would result in periodic task releases. Coping with this scenario using traditional compute-only schedulers can result in overly complicated methods, and no guarantee on predictability. As we shall discuss in Sec. III-D, the runtime computation scheduler can be simple as long as data is scheduled reliably to prevent delayed release times due to delayed data transfer and memory unavailability.

To transform LET to a data-driven model, we turn into an established production-leveling methodology, collectively called Heijunka. Heijunka, developed mainly by Toyota [32], is a group of techniques that can ensure predictable output in the parts-driven car manufacturing process. We apply similar techniques to program code (acting as parts) and its supplier (disk) to propose a new data-centric system model (Sec. III-B). Based on this model, we establish that for ensuring predictable outputs, there are both temporal and spatial requirements for data availability. Through leveraging and extending the underlying queuing theory methodology used in Heijunka, we develop a practical data-driven resource management framework, ResCue, for the integrated architecture, including a dynamic data scheduler that can load and unload program code to enable predictable execution (providing temporal data availability), and a flexible memory reservation scheme that makes intertwined decisions with the data scheduler to ensure minimal memory reservation requirement (providing spatial data availability).

Our contributions are summarized as follows:

- A new data-centric system model based on established concepts in real-time systems, production leveling, and queuing theory that can account for memory uncertainties as well as computation, and would significantly simplify scheduling decisions.
- A methodology to calculate static data allotment for AES.
- A dynamic data scheduler to ensure predictable temporal data availability under integrated architecture.
- A flexible memory reservation scheme that together with the data scheduler can provide predictable spatial data availability, which ultimately can ensure predictability in

\(^1\text{We ignore the memory requirement due to sensing data herein because it is trivial compared to the computing code data. For example, YOLO's input images are typically less than 200KBs, while the program code including weights exceeds 4GB.}\)
terms of 1) meeting output deadlines, and 2) reducing jitter such that the outputs are produced at (or close to) a constant rate.

**Implementation and evaluation.** We fully implement ResCue on a state-of-the-art embedded hardware directly aimed at autonomous systems, the NVIDIA AGX Xavier [33], used in the newest upcoming self-driving platforms [5]. We evaluate the comprehensive, open-source end-to-end self-driving software Autoware [6] in terms of predictability and overhead under various practical settings. Experimental results show that ResCue never misses the deadline (while the vanilla Autoware does), improves the response time by 49% on average, and yields a maximum jitter of merely 834 microseconds. Moreover, ResCue reduces the memory requirement by 473 MBs compared to the vanilla Autoware. We also find that the overhead of ResCue is minimal compared to the savings both in terms of memory consumption and latency.

II. BACKGROUND

In this section, we lay out key ideas behind our design and system model, as well as necessary background information. AES have various sensors which capture environmental data in a periodic manner. For instance, autonomous vehicles use cameras and lidars to capture data in a periodic manner. Such systems are also expected to generate an output control decision at constant intervals in a predictable manner. For example, autonomous vehicles are expected to output steering decision for every 300ms [18]. Unfortunately, as mentioned earlier, due to both temporal and spatial data availability issues, the actual task release (which can only happen when the required data is in memory) could be highly unpredictable (i.e., sporadic). Thus, the underlying model considered in this paper is of generating output at a predictable constant interval in a potentially unpredictable task release system due to data and memory management.

We introduce two notions from queuing theory that aim to enable predictable decision-making in the presence of unpredictable job release. We then provide a description for ROS and the integrated architecture as the state-of-the-art AES workloads and hardware architecture. As we shall discuss in Sec. III-D, integrated architectures provide a non-trivial challenge to the application of ResCue.

**Heijunka.** Heijunka (or known as production leveling) is a manufacturing technique first effectively developed and implemented by Toyota [32]. Heijunka is quite complex and involves many techniques. However, to the extent that it applies to real-time systems, Heijunka would ensure a predictable and constant output interval despite an unpredictable manufacturing pipeline and stochastic customer demand. Through modeling data-driven AES using Heijunka, we are able to leverage and extend two techniques of Heijunka in our system design: 1) a reserve of resources to ensure predictable outputs, and 2) scheduling certain actions to ensure smoothness of production.

Fig. 2(a) shows a production system with its various parameters. Three suppliers of parts are shown as A, B, and C. The output rate of each supplier are designated as variables $\alpha, \beta, \gamma$. Given the constant output required during the day, Heijunka is responsible for assigning various rates to each variable. For example, suppose each car requires 3 parts from A, 1 part from B and 2 parts from C. If the constant flow is to produce 100 cars a day, we need $\alpha = 300, \beta = 100, and \gamma = 200$ per day. Moreover, these parts need to be stored as they come from suppliers before the manufacturing pipeline (shown as Build) can process them. Thus, a buffer, or a parts bin is required to act as a dam for the flow of parts from suppliers. The size of the buffer is shown as $B$. Calculating the value of $B$ requires additional knowledge about queuing theory, which we will discuss in the next section. After the manufacturing, there will be sometime before the car can be delivered to customers. Thus, an additional inventory buffer of size $B'$ is required to store customer cars. Finally, the production will consume some amount of time, shown as $W$. $W$ plays a crucial role in determining $B$, as we shall see throughout our design in Sec. III.

We shall use Heijunka to present a new type of system model for our studied autonomous systems in Sec. III-B, to solve the static buffer requirement for that model in Sec. III-C, and in Sec. III-D to help set supplying rate from the disk and to choose a reservation window. Next, we present the most widely used queuing theory technique to deduce the size of $B$ (the parts bin) and $B'$ (the inventory), called Little’s theorem.

**Little’s Theorem.** Fig. 2(b) depicts a simplified version of network transmission across a transit channel. This channel could be anything from Ethernet cable to wireless signals. The transit channel acts as a pipeline to send a message. However, the network standard always requires network channels to agree on a bitrate and thus send data at a constant rate. Since the sender will sporadically release packages, a simple queue is required to hold packages until they can be transmitted.

On the other side, the receiver has to process packets, and thus, a queue is needed to hold the packets until the receiver is ready to process more packets. The similarity of this model to Heijunka is not unfounded. In fact, Heijunka techniques face many similar challenges and these challenges are usually solved by using queuing theory. The major question for the model of Fig. 2(b) is the size of the buffer $B$. A very large $B$ would be a waste of resources and a $B$ that is too small can result in loss of network packets. We use the fundamental Little’s Theorem in queuing theory to solve for the problem.

In the long term, the average number of packets in the system, $L$, can be calculated as the product of the average
arrival rate $\lambda$ and the average time $W$ spent in the system [34]:

$$L = \lambda W.$$  \hfill (1)

In order to deduce a reasonable queue size, it is sufficient to calculate $B = L \cdot (\text{Packet Size})$. The same technique could be used to calculate $B'$. The other challenge is calculating the average arrival rate $\lambda$. We use task release time sampling to calculate $\lambda$, and use the worst-case execution time to calculate the value of $W$. We shall use Little’s Theorem to calculate the static data allotment for our Heijunka inspired model in Sec. III-C.

**Autonomous Embedded Systems (AES).** Practical AES have various goals and architectural details. However, most have a similar flow: Sensing $\rightarrow$ Perception $\rightarrow$ Decision & Planning $\rightarrow$ Actuation. In this paper, we focus on a pervasive kind of autonomous systems seen in practice, namely autonomous driving. These systems heavily rely on complicated Perception, Decision, and Planning modules (see Sec. IV) to make output control decisions.

AES are expected to be heavily data-driven. Thus, communication and coordination between modules is key in AES [35]. To help guide the communication between these components, system software such as Robot Operating System (ROS) has been developed to manage data coming from and going to various modules. However, ROS does not provide any guarantee towards predictability in the system when managing data and memory. Recent research has focused a lot on making the concept of task set real-time [36] (e.g., real-time computation scheduling for ROS [37]). However, to the best of our knowledge, no research in this area of AES and general cyber-physical systems has focused on periodicity of these data-driven systems: the presence of data is essential to provide any predictability guarantee. Even if computations can be technically released and scheduled on time, \(^2\) without the presence of data in accessible memory, there can be no guarantee on timeliness or predictability, no matter how rigorous the theoretical analysis or how effective the system software is.

**Integrated Architecture.** The data and memory management problem is exacerbated by the integrated architecture which has become a dominant architectural feature seen in most AES platforms. Examples include NVIDIA’s series of driving-specific platforms [2], Intel’s OpenVINO integrated GPU platform [5], and Tesla’s newly released AI chip [38]. These platforms are interesting from the data and memory management perspective because all computing units share a unified and limited memory space. The interesting question thus would be how a unified memory space affects memory reservation requirements and what type of data scheduling should be done to enable predictable computing?

**Logical Execution Time (LET).** The constant input/output interval is not unfounded in real-time systems. This model is similar to LET [31]. LET forces the system to interact with its environment at constant intervals through periodic IO. We subscribe to the argument presented in [19], [20], where a predictable output at constant intervals is critical for CPS and AES, because of the predictability requirement and the reliance on IO operations and memory management.

The periodic input of LET is also useful for realistically modeling AES. In AES workloads, a task set is released when an input becomes periodically available. The job of the task scheduler would then be to ensure that the task set can meet the fixed deadline, which is the output interval. This simple change of view can dramatically simplify the task scheduler because the scheduler would know beforehand when input arrives and output is expected before any task execution. For example, in an autonomous car, a camera module directly writes images to memory at 33ms intervals (input) and the gas pedal actuator module will read a specific region of memory every 33ms (output). This approach realistically resembles how Direct Memory Access (DMA) or similar approaches are used to perform periodic IO operations in modern AES. Moreover, the complexity of handling IO can be relegated to another system process, making the scheduler much more simplified and predictable. However, in all existing LET models, an input event is synonymous with the release of the corresponding task set. As we will discuss in Sec. III, this is a major flaw of LET models because memory limitations can cause a delay between input availability and the release of the task set.

### III. Design

#### A. Overview

In this section, we lay out an overview of our design, which heavily relies on the system model presented in Sec. III-B. This system model is an extension of LET to account for the more realistic data-centric sporadic task model.

As was mentioned in Sec. II, the periodic IO model of LET is useful for realistically modeling AES. However, to the best of our knowledge, existing LET models only consider a periodic computation task release, which assumes that the task set is released as soon as the input interval is over (some work such as [20] have considered aperiodic input events, but the computation task set is nonetheless released as soon as the input event is triggered). This assumption mainly is the consequence of ignoring the data requirements of the task set. Even though the input can be periodically released, the tasks in the corresponding task set cannot, unless the code for those tasks is also available. As we shall expand on in Sec. III-D, a limited memory pool (prevalent in most AES) can prevent the timely release of the tasks, resulting in sporadic task release. The most naive solution to this problem would be to use a more complicated task scheduler that can manage sporadic task releases on a best-effort basis. However, the simplicity of the scheduler is one of the key benefits of using LET and the predictability is a key concern in AES.

To solve this, we found astonishing similarity in Heijunka. According to Heijunka, the input and output can be periodic,
but the tasks in the corresponding task set can be sporadic as long as the supply line (program code) is considered in addition to the building pipeline (computation). In other words, smartly considering the movement of data from disk to main memory can mitigate uncertainties due to memory limitations. The predictability is then achieved under Heijunka by controlling the supply rate, and introducing a buffer.

We make our case to merge LET and Heijunka into a new system model that is two dimensional (in terms of memory and computation) in Sec. III-B. However, this does not inherently improve predictability unless supply rate of program code can be carefully managed and a buffer with a calculated size is present. There are two approaches to this problem. First, we show that assuming no limitations on the buffer size, we can mitigate the need to adjust the supply rate. Intuitively, if an infinite amount of memory is present, all data and program code can be in memory at all times. Under this condition, the analysis of existing LET models are correct. We calculate the buffer size required for predictability in Sec. III-C. This approach also has the benefit of being static, since the supply of program code does not need to be adjusted. Nonetheless, the calculated buffer size is often unfeasible for integrated architecture.

To solve the issues under the more realistic limited memory pool, which is generally more feasible for AES, we propose ResCue, a dynamic memory manager (scheduler and reservation) depicted in Alg. 1. More specifically, we design a dynamic data scheduler in Sec. III-D-a, which would load the program code on demand as soon as the input is released. However, doing so naively could result in deadline misses, because under some circumstances, the code would become available too late, violating temporal data availability. We quantify this lateness in terms of Last Chance Release time (line 4), the latest time by which the code should be available for each task of the task set. Analogous to the methodology of Heijunka, we use the Last Chance Release time to calculate at what rate program code is needed (similar to calculating at what rate parts are needed) in line 5. Our scheduler would then use this rate to calculate when to start the code transfer (line 6) and to calculate if the system is schedulable (line 7). We provide an analysis to show under what circumstances the system could be schedulable even if the task release time is sporadic and prove that our method can guarantee meeting deadlines if certain conditions are met. Nonetheless, according to the methodology of Heijunka, maintaining the supply rate alone is not enough. There need to be a reserve of memory (parts bin) to accommodate for incoming program code (parts). Intuitively, there should be enough memory space to hold some amount of program code, ensuring spatial data availability. Using the same supply rate, we calculate the absolute minimum amount of memory reservation based on the principles of Heijunka in Sec. III-D-b, and provide a runtime algorithm that maintains two reserved slots to also utilize memory pooling benefits [39] (line 8). Finally, we account for multiple instances of the task set present by choosing one with the earliest deadline (lines 2 and 3).

Throughout the rest of this section, we assume a task set \( \Delta \) exists with a period \( I \) (equal to the input period). However, the tasks \( T_j \in \Delta \) are sporadic. Moreover, \( T_j \) depends on \( T_i \) if \( j \geq i \). In other words, all tasks in the task set are executed in a chained fashion (next task can be executed as soon as the previous task finishes). Once all tasks in the task set finish executing once (e.g. \( T_n \in \Delta \) is finished), the task set is considered finished.

### B. System Model

In this section, we show how Heijunka can be applied to model an AES in three general steps, as illustrated in Fig. 3(a). On the left and right side of Fig. 3(a), there are two separate reserved memory spaces for real-time tasks, acting as a queue. In step 1, the task codes (acting as the parts) are ready on the disk (which is acting as the supplier). As evident in the figure, tasks 1, 2, 3 and 4 could fit in the reserved memory space (akin to parts bin), while tasks 5, 6, and 7 have to be on disk. This situation can arise if program code or data for \( T_{5-7} \) have not yet been loaded to memory or if memory swapping was enforced by the operating system because of a memory overflow. Similarly, in Heijunka, the parts bin can become full, thus cutting off the supply of parts.

In step 2 of Fig. 3(a), the tasks currently residing in the memory are fed to a compute engine, which could be a CPU, a GPU, or a DL unit. This compute engine translates to the manufacturing pipeline (named Build in Fig. 2). The output is then stored into memory for subsequent tasks (similar to inventory) to use. Finally, step 3 showcases the timeline of task releases for \( T_1, T_2, T_3, \) and \( T_4 \), shown as arrows, which are released in a sporadic manner due to data availability. However, for mission-critical systems such as autonomous vehicles that require predictability, a decision is expected at constant intervals. This is different from the traditional sporadic task models in which, a sporadic task release is expected to finish by a relative deadline to that release. This problem is similar to the constant car manufacturing rate expected in Heijunka.

---

**Algorithm 1 ResCue**

Require: \( \{ \Delta \} \) \( \triangleright \) List of all instances of task set \( \Delta \).

Require: \( O \) \( \triangleright \) The output interval (periodic for all instances of the task set).

Require: \( I \) \( \triangleright \) The input interval (periodic for all instances of the task set).

1: function \( \text{ResCue}(\Delta, O, \{ I \} ) \)
2:    if Multiple instances of \( \Delta \) exists then
3:        Choose one with the earliest deadline as winner \( \Delta^w \)
4:    Calculate Last Chance Release time for each \( T_i \in \Delta^w \)
5:    Calculate supply rate \( \gamma \)
6:    Calculate \( S \), the necessary start time to transfer code
7:    if System is schedulable based on \( \gamma \), LCR, and \( S \) then
8:        Reserve two minimum required memory spaces based on \( \gamma \)
9:        Start transferring code at \( S \)
10:   Switch to compute scheduler

---


the temporal requirement, since the memory for task $T_{i+1}$ is copied to memory at time 0 regardless of the release time. The only remaining concept that would be interesting is, how much memory do we exactly need for the task set $\Delta$ to satisfy the spatial requirement?

**Reserving memory for $\Delta$.** For an allotment of sufficient memory to ensure predictability for the chain task set $\Delta$, we use worst-case memory consumption (WCMC) [40], [41] for each individual task $T_i \in \Delta$, denoted by $M_{T_i}$. The static allotment of memory space can be simply calculated as:

$$A = \sum_{T \in \Delta} M_{T_i}. \quad (2)$$

However, Eqn. 2 is lacking crucial details. First and foremost, the task set $\Delta$ will not be the only task present in the system. Other instances of the task set will be released periodically between the output intervals if the input period is smaller than the output period. Thus, the calculation of $A$ might be too optimistic. Second, there could be different types of memory present in the system. For example, certain deep learning accelerators have their separate memory space [42], which need to be accounted for in a general solution. Third, Eqn. 2 is at the same time rather pessimistic. The calculations for $A$ demands that before the first task in task set $\Delta$ is released, all the required memory for all $T \in \Delta$ must be reserved. Next, we address these three problems.

First and foremost, how do we account for multiple instances of the task set? Imagine that beforehand it was known that at most $L$ instance of the task set will be present in the system. We can modify Eqn. 2 to be:

$$A = L \sum_{T \in \Delta} M_{T_i}. \quad (3)$$

Guessing the value of $L$ is not trivial due to unpredictable sporadic task releases, in which case it is impossible to predict the number of concurrent instances of the task set beforehand. However, as we discussed in Sec. II, using Little’s Theorem, $L$ can be calculated as $L = \lambda W$, in which $W$ is the worst-case execution time of task set $\Delta$ calculated as $W = \sum_{T \in \Delta} W_T$. Replacing everything in Eqn. 3, we get:

$$A = \lambda \sum_{T \in \Delta} W_T \sum_{T \in \Delta} M_{T_i}. \quad (4)$$

$\lambda$ can be calculated by sampling the system over time. For example, if a task set is initiated every 24ms, then $\lambda = 1/24$.

For the second issue of Eqn. 2, we account for multiple memory types by dividing the task set $\Delta$ into $\Delta_{CPU}$ for CPU, $\Delta_{GPU}$ for GPU, and $\Delta_{ACC}$ for accelerator and calculating $A_{CPU}$, $A_{GPU}$, and $A_{ACC}$ from Eqn. 4.

**D. Dynamic Data Scheduling and Memory Reservation under the Integrated Architecture**

For the limited memory space under an integrated architecture, reserving the entire memory requirement statically (at time 0) is unacceptable. For example, on the AGX Xavier, running Autoware and with the output period equal to input
period set to 30ms, the calculated value for $A$ can be up to 6.6GB for CPU ($A_{CPU}$) in our testings. Moreover, Autoware relies on GPU for running some sub-components of the perception module, which leads to a value of 4GB for $A_{GPU}$. As it becomes obvious, $4GB + 6.6GB = 10.6GB$ which is already over 60% of the maximum memory availability. This is also given only one instance of the perception module.

Thus, we design a dynamic data scheduler and a flexible memory reservation scheme for resolving the temporal and spatial requirements for predictable outputs in Sec. III-C.

Fig. 4(a) depicts an example scenario, with input interval $I_1$ releasing the task set $\Delta_1 = \{T_1^1, T_2^1, T_3^1\}$. The corresponding output interval (deadline) is marked as $O_1$. In this example, due to an untimely data transfer (marked as $|M_{T_1^1}|$), task $T_1^1$ is released late, resulting in $T_2^1$ being released late and thus, making task $T_3^1$ and the whole task set $\Delta_1$ to miss the output interval. We formally define what it means for a task to be released late in our dynamic data transfer scheduler by defining a last chance release time (depicted as $l_i^1$ for $\Delta_1$), in which we define the latest possible release time for each task of a task set in order to meet the deadline. As we shall see in Sec. III-D-a, $l_i^1$ can help with scheduling the memory transfers to ensure periodic output.

Fig. 4(b) depicts the spatial problem given limited memory, where due to our dynamic data transfer scheduler (as we shall discuss shortly), tasks $T_1^2$ and $T_2^2$ were released on time (at time instance S). However, the memory has become full by $T_3^2$, and the whole task set $\Delta_2$ cannot happen in time, leading to output interval miss. Thus, there should also be a reservation scheme that can collaboratively work with our data transfer scheduler to avoid such problems. We present our dynamic memory reservation module in Sec. III-D-b.

a) Dynamic Data Scheduler.: In the example of Fig. 4(a), the deadline miss can be avoided by carefully scheduling the data transfer so that the task codes data can be transferred on time to memory and always become available when needed. Before formally providing our solution, we first address the scenario where multiple instances of the task set (each corresponding to an input/output) are present because the input interval can be smaller than the output interval. In such a scenario, the runtime execution scheduler would prioritize the task set that has the best chance of generating an output by the deadline (which is the task set with the earliest release time). We indicate the winner task set as $\Delta_w$.

Going back to the original problem, if the system is to meet the output interval, the final task $T_n^w$ of the winner task set $\Delta_w$ must start executing at most at time instance $O - W_{T_n^w}$, in which $O$ is the next output interval and $W_{T_n^w}$ is the worst-case execution time of task $n$. In other words, the task $n$ should be released with enough headroom for execution in order to meet the output interval. This assumption is sufficient since it also implies that previous tasks in the task set must have executed early enough to leave enough room for $T_n^w$’s execution. Next, we formally define this concept.

Definition 1 (Last Chance Release Time). We call the time instance $l_i^w = D - W_{T_i}$ the Last Chance Release time (LCR) of task $T_i$ for task set $w$, task $i$, and absolute deadline $D$. For example, $l_n^w$ for task set $w$ indicates that the last task must be released by $O - W_{T_n}$ if the task set is to output a decision by the output interval.

Calculating LCR can be iteratively applied to all tasks in the task set, thus calculating an LCR set $LCR^w = \{l_i^1, ..., l_n^w\}$, as is depicted in Alg. 2. The first LCR is calculated for task $T_n$ (line 9) after determining the winner task set $\Delta_w$ (line 5) and the deadline (initially set to output interval $O$). Next, the deadline is updated to be $O - W_{T_n^w}$ (line 10). Thus the next iteration will calculate the last chance release time for task $n-1$, iteratively going to task 1. The output of Alg. 2 (line 12) is a set $LCR = \{l_i^1, ..., l_n^w\}$ of last chance release times for all tasks in the winner task set $\Delta_w$. Fig. 5(a) depicts an example of applying Alg. 2 to the example of Fig. 4. Since there is no other task set in the system, $\Delta_1$ is the winner task set ($\Delta_w$). Moreover, LCR is calculated for the three tasks of task set $\Delta_1$ as is depicted in the figure.

Subsequently, with the knowledge of the LCR, we can calculate when the memory for task $T_i^w$ should be reserved and the data transfer is initiated. Imagine a task $T_i^w$ will be executed at time instance $l_i^w$ at the latest. Also, imagine a worst-case memory consumption of $M_{T_i^w}$ for the task. With a transfer rate of $r_D$, which is hardware dependent, the data

Algorithm 2 Last Chance Release Time.

Require: $\{\Delta\}$ \hspace{1em} \triangleright \hspace{1em} List of all task sets.
Require: $O$ \hspace{1em} \triangleright \hspace{1em} The next output interval.
Require: $l_i^\Delta$ \hspace{1em} \triangleright \hspace{1em} List of input intervals for each task set in $\{\Delta\}$.
Require: $\{W_i^\Delta \mid \forall T_i \in \{\Delta\}\}$ \hspace{1em} \triangleright \hspace{1em} WCET for all tasks in the system.

1: function $WINNER(\Delta, \{I\})$
2: \hspace{1em} Choose $\Delta_w \in \{\Delta\}$ s.t. $I_i = \min I \in \{I\}$
3: \hspace{1em} return $\Delta_i$ as $\Delta_w$
4: function $LAST\_CHANCE\_RELEASE\_TIME(\Delta, O, \{I\})$
5: \hspace{1em} $\Delta_w = WINNER(\Delta, \{I\})$ \triangleright \hspace{1em} Winner for the current output period.
6: \hspace{1em} $\text{CURR\_TASK} = |\Delta_w|$ \hspace{1em} \triangleright \hspace{1em} Set current task to last task (n) in $\Delta_w$.
7: \hspace{1em} $\text{DEADLINE} = O$
8: \hspace{1em} while $\text{CURR\_TASK} > 0$ do
9: \hspace{2em} $\text{CURR\_TASK} = \text{DEADLINE} - W_{\text{CURR\_TASK}}$
10: \hspace{1em} $\text{DEADLINE} = \text{DEADLINE} - W_{\text{CURR\_TASK}}$ \hspace{1em} \triangleright \hspace{1em} Adjust deadline.
11: \hspace{2em} $\text{CURR\_TASK} = \text{CURR\_TASK} - 1$ \hspace{1em} \triangleright \hspace{1em} Switch to previous task.
12: \hspace{1em} return $\{l_i^w\}$ as LCR
transfer takes at most $M_i^w/r_D$. Thus, the data transfer must be initiated at most at $l_i^w - M_i^w/r_D$ so that the data is ready by the last chance release time of $T_i^w$. Fig. 5(a) shows this scenario, where the data transfer for task $T_1^1$ starts exactly at $l_1^1 - M_1^1/r_D$, which allows for $T_1^1$ to be released at $l_1^1$.

However, data transfer (copy) from disk to memory cannot happen in parallel, which is a major issue for our design. Imagine the example of Fig. 5(a), in which task $T_2^1$ is to be released after $T_1^1$. If we initiate the data transfer of task $T_1^1$ at time instance $l_1 - M_1^1/r_D$, task $T_2^1$ and $T_3^1$ will miss their last chance release time. This is due to the fact that the data transfer for task $T_2^1$ must be initiated at $l_2 - M_2^1/r_D$ at the latest, which is inside the window of data transfer for $T_1^1$.

To resolve this, we leverage a technique of Heijunka again, in which a supplier rate for disk is calculated in such a way that can avoid overlapping the use of the transfer channel. We calculate the supply rate of the disk for each output period as:

$$\gamma^w = \min\{|l_i^w - l_i^w| \mid \forall l_i^w \in LCR_i^w\},$$

in which $|l_{i+1}^w - l_i^w|$ is the distance between LCR of task $i$ and LCR of the immediate next dependent task $i+1$. In other words, imagine that the loading of data for task $i$ has just finished by $l_i^w$. In this situation, how long does the system have to load the data for the next task $i+1$? The min function calculates the smallest value of this interval so that the rate for the current output period is set according to the worst-case scenario. $\gamma^w$ indicates that a data transfer must be scheduled at least every $\gamma^w$, in order to avoid missing deadlines. For the example of Fig. 5(b), $\gamma^1$ is the distance $|l_2^1 - l_1^1|$.

However, when should the initial data transfer for the task set $\Delta^w$ start to avoid overlapping the data transfers? We can calculate the total time to transfer all the data for task set $\Delta^w$ as:

$$\sum_{T \in \Delta^w} \frac{M_T}{r_D}.$$ 

Given $|\Delta^w|$, the number of tasks in the task set, the starting time $S$ can be calculated as:

$$S = l_1^w - \left(\frac{\sum_{T \in \Delta^w} M_T}{r_D}\right) + |\Delta^w| \cdot \gamma^w.$$ 

In the example of Fig. 5(b), starting the data transfer at $S$ has resulted in all tasks meeting the deadline.

Next, we prove that given $S$ and the LCR set, the system can guarantee meeting the deadline:

**Theorem 1:** As long as $S > I$, there can be a guarantee that an output can be given at $O$.

**Proof.** Before presenting the proof, we define the concept of channel utilization.

**Definition 2 (Channel Utilization).** Imagine that $M$ units of data has been transferred for a time period of $t_2 - t_1$; $t_2 > t_1$. With a disk transfer rate $r_D$, we calculate the Channel Utilization as:

$$CU = \frac{M}{(t_2 - t_1) \cdot r_D}.$$ \hspace{1cm} (8)

A channel utilization (CU) of 1 means that we transferred maximum amount of data possible for time period of $t_2 - t_1$.

The time instance $S$ is the starting point of our algorithm for data transfer. For the first task $T_1^w$ to be released at or before $l_1^w$, the data must have been transferred fully beforehand. Thus, the amount of data transferred in the period of $l_1^w - S$ must equal or be more than $M_1$, the worst-case memory consumption of task 1. Since we intend our algorithm to maximize channel utilization, we assume a CU of 1, and get:

$$1 = \frac{M_1}{(t_2 - S) \cdot r_D}.$$ \hspace{1cm} (9)

By rearranging Eq. 9, we get:

$$(t_2 - S) \cdot r_D = M_1 \rightarrow t_2 = \frac{M_1 + S \cdot r_D}{r_D}.$$ \hspace{1cm} (10)

Thus, given the maximum channel utilization, the time-instance where data transfer for task 1 finishes is calculated as $t_2$. We can continue by using $t_2$ as the starting point for data transfer of task $T_2^w$ and calculate $t_3$, the finishing time for data transfer for task 2:

$$t_3 = \frac{M_2 + t_2 \cdot r_D}{r_D}.$$ \hspace{1cm} (11)

Replacing Eq. 10 in Eq. 11, we get:

$$t_3 = \frac{M_2 + (M_1 + S \cdot r_D) \cdot r_D}{r_D} = \frac{M_2 + M_1 + S \cdot r_D}{r_D}.$$ \hspace{1cm} (12)

Continuing the same trend, for the final task $n$, the finishing time for data transfer, $t_{n+1}$, can be calculated as:

$$t_{n+1} = \frac{M_n + t_{n-1} \cdot r_D}{r_D}.$$ \hspace{1cm} (13)

Recursively expanding the value of $t_{n-1}$ in Eq. 13, we get:

$$t_{n+1} = \frac{(\sum M_T) + S \cdot r_D}{r_D} = \frac{\sum M_T}{r_D} + S.$$ \hspace{1cm} (14)

We assumed the value of $S$ to be derived from Eq. 7.

\footnote{The phrase $|\Delta^w|$ in Eq. 7 should be $(|\Delta^w| - 1)$ because the phrase $|l_{i+1}^w - l_i^w|$ can only be calculated up to $l_{n-1}^w$. This is also true for the $\sum$, min, and max function on $|l_{i+1}^w - l_i^w|$. We omit this detail for clarity reasons, but should be the assumption throughout the rest of the paper.}
Replacing that in Equ. 13, we get:
\[
t_{n+1} = \left( \sum_{i} M_{i} r_{D} \right) \frac{1}{M_{n}} + l_{1}^{w} - \left( \frac{\sum_{i} M_{i} \Delta^{w}}{r_{D}} \right) + |\Delta^{w}| \cdot \gamma^{w}.
\]
(15)

Since \( \gamma^{w} \leq |l_{1}^{w} - l_{1}^{w}|, \forall T_{i}^{w} \in \Delta^{w} \) by definition, we get:
\[
t_{n+1} = l_{1}^{w} + |\Delta^{w}| \cdot \gamma^{w} \leq l_{1}^{w} + \sum_{T_{i} \in \Delta^{w}} |l_{i+1}^{w} - l_{i}^{w}|.
\]
(16)

However, \( l_{1}^{w} + \sum_{T_{i} \in \Delta^{w}} |l_{i+1}^{w} - l_{i}^{w}| = l_{n}^{w} \). Thus, we get:
\[
t_{n+1} \leq l_{n}^{w}.
\]
(17)

The same procedure can be recursively applied until we get \( t_{2} \leq l_{1}^{w} \), which we omit due to lack of space. This result indicates that the data transfer given a known \( r_{D} \) and a maximum channel utilization, starting from \( S \) can always finish memory transfer for all tasks in \( \Delta^{w} \) before their respective last chance release times.

b) **Dynamic Memory Reservation Scheme:** Given the previously mentioned data scheduler, we update our allotment technique given in Sec. III-C for an integrated architecture. The proposed data transfer scheduler can guarantee that the data can be present before the last chance release time. However, if the memory is already full, it is impossible to transfer any data. Thus, a minimum memory reservation is still required. Imagine the example presented in Fig. 4(b), where the transfer for task set \( \Delta^{1} \) has started at time instance \( S \). However, as is evident in the figure, \( T_{3}^{1} \) and subsequently \( \Delta^{1} \) will miss the output interval. This is due to the fact that the memory has become full when the transfer of data for \( T_{3}^{1} \) was supposed to start, delaying the data transfer and subsequently the task release considerably. Thus, there is still a need for memory reservation even if a dynamic disk scheduler is present.

First, let us update Equ. 3 based on our last chance release time and our earliest release time scheduler. Since our scheduler gives the highest priority to one task set, we can eliminate \( L \) altogether. Moreover, since \( |l_{i+1}^{w} - l_{i}^{w}| \) is the distance between the last chance release times of task \( i \) and \( i + 1 \), we can assume pessimistically that a maximum of \( |l_{i+1}^{w} - l_{i}^{w}| \cdot r_{D} \) can be moved between disk and memory. Moreover, before the first last chance release time \( l_{1}^{w} \), there will be an initial data transfer from time \( S \) to \( l_{1}^{w} \), with a maximum amount of \( (l_{1}^{w} - S) \cdot r_{D} \) of data movement. Thus, we get:
\[
A = (I_{1}^{w} - S) \cdot r_{D} + \sum_{T_{i} \in \Delta^{w}} |l_{i+1}^{w} - l_{i}^{w}| \cdot r_{D}.
\]
(18)

Since the memory is unified, there is no need to separate queues anymore, and \( \Delta^{w} \) includes tasks for all processors. However, there is still the issue of reserving large memory before execution. A combination of our earliest release time scheduler and Equ. 18 means that only tasks that belong to the task set with the earliest deadline will execute. However, other instances of the task set will still be released if the input period is smaller than the output period, which means Equ. 18 would still demand too much memory and will not leave enough space for sensor data to be stored at every input interval.

We solve this problem by narrowing the reserve window, as is depicted in Fig. 6(b) from the output period to \( |l_{i+1}^{w} - l_{i}^{w}| \). With such a narrow window, calculating the memory requirement for that window can be as simple as:
\[
A_{[i,i+1]} = |l_{i+1}^{w} - l_{i}^{w}| \cdot r_{D}.
\]
(19)

However, keeping track of each value of \( |l_{i+1}^{w} - l_{i}^{w}| \) and calculating \( A \) for each transfer window is both memory and compute intensive. Thus, we simply replace \( |l_{i+1}^{w} - l_{i}^{w}| \) with \( \gamma \), the minimum transfer window. Our reservation algorithm is depicted in the scenarios of Fig. 6 (value of \( \gamma \) is \( |l_{2}^{w} - l_{2}^{w}| \)). For our algorithm to operate, we should only reserve \( \gamma \cdot r_{D} \). However, keeping two reservation windows ensures a smoother operation because memory pooling techniques could be used [39] to provide a rolling reservation slot, bringing the total reservation to \( 2 \cdot \gamma \cdot r_{D} \). At the next reservation window, the contents of the first reserved memory space is cleared and remapped to be ready after the second window.

In the example of Fig. 6 Scenario 1, two windows of size \( \gamma \) have been reserved. Once \( T_{3}^{1} \) has completed its execution, the first reserved window is emptied, and assigned to be used for memory transfer operations that happen after the previous memory reservation window becomes full (Scenario 2 in Fig. 6).

IV. EVALUATION

A. Evaluation Methodology

**Platform.** We use the most advanced NVIDIA integrated architecture as of writing this paper, the AGX Xavier, with 8 NVIDIA "Carmel" cores, 16GBs of RAM, a 512-core Volta GPU and 2 DLA engines. We fully implement ResCue and integrate our implementation with Robot Operating System (ROS) [35], which is used by Autoware. ROS is a large set of tools and libraries that follow a modular publisher/subscriber model. Many AES software, including many self-driving car software rely on ROS for drivers, utility, communications, etc.

**Autoware.** We evaluate one of the most promising end-to-end self-driving software that is both open-source and free to use, the latest version of Autoware platform [6] as of writing this paper. We use the car configuration used to generate the data in [43] (running a subset of Autoware modules as necessary) and play back this data using the recorded ROSBAG. The
We implement ResCue with a step scheduler with the resolution of 1 microsecond for both execution and data transfer and integrate it with ROS. Our implementation enables the system designer to pin certain processes down as static memory in order to improve predictability. As we shall discuss shortly, our method is able to achieve substantial memory savings even without transferring this memory. Because ResCue is a system solution implementation, and has to be verified at runtime. All values measured are either presented as worst case, or presented in the form of a cumulative distribution function which also includes the worst case.

**Scenarios.** Throughout this section, we use three variations for output and input intervals to show the versatility of ResCue:

- **S1:** $I=42, O=300$ is where the output period is set to 300ms, the lowest threshold for human reaction time [46], [47]. Moreover, the input period is 42ms (i.e., 24 fps).
- **S2:** $I=33, O=200$ is where the output period is set to 200ms. We would like to make the output period as tight as possible to increase the guarantee on reaction time. As we shall discuss in Sec. IV-B, 200ms is a good middle ground between the tightest achievable deadline and 300ms, which is the human reaction time threshold. The input period is 33ms (i.e., 30 fps).
- **S3:** $I=171, O=171$ is where the output period is set as tight as possible, which we find to be 171ms for the combination of Autoware and ResCue. We test the variations in input period by setting $I = O$.

**B. Implementation and Evaluation Results**

We implement ResCue with a step scheduler with the resolution of 1 microsecond for both execution and data transfer and integrate it with ROS. Our implementation enables the system designer to pin certain processes down as static memory in order to improve predictability. As we shall discuss in Sec. IV-B, we pin down one Autoware module (out of 15) to enable tighter deadlines.

**Overview on Autoware.** The overview of Autoware components, along with their category, their Worst-Case Execution Time (WCET) and Worst-Case Memory Consumption (WCMC) is shown in columns 1-5 of Table I. The largest component both in terms of WCET and WCMC is ndt_matching, which is based on the Normal Distribution Transform [48] used for localizing LIDAR data. The data requirement for ndt_matching is 2065 MBs, which requires 350ms to be transferred using an $r_D = 5882$. In order to make a tighter deadline possible, we pin down the memory for this module. As we shall discuss shortly, our method is able to achieve substantial memory savings even without transferring this module. The WCET is however used in all our calculations. Without ndt_matching, the total baseline memory consumption of our sub-modules is 483.2MBs. The end-to-end WCET of the Autoware sub-components is 93.9ms (see Summary).

**Parameters.** Our runtime scheduler calculates the values of LCR based on WCET and WCMC of each component given a value for I and O. The results are shown in columns 6-8 of Table I. Based on the values of LCR, our scheduler calculates the required starting point $S$, shown in the first row of Parameters, which are 129.994ms, 29.994ms, and 0.994ms for each scenario respectively. The starting point indicates when the data transfer scheduler should start moving data.
from disk relative to I in order to meet the deadline. As is evident in the figure, the output period 171 almost leaves no room between the release time (I) and the starting time of data transfer, making 171 a tight deadline. We also calculate the value of $\gamma$, which is used to set the data transfer rate and the reservation window. The value of $\gamma$ is based on the distance between LCR of tasks, which are in turn based on the WCET, which is the same across scenarios. Thus, the value of $\gamma$ is 0.84 for the task set regardless of the values of O and I.

Assessing the memory reservation requirement. As we discussed in Sec. III-D, we reserve 2 windows of size $\gamma \cdot r_D$, which is 0.00083 $\cdot$ 5882 = 4.88. Thus, we reserve 2 $\cdot$ 4.88 = 9.76. The WCMC for the dynamic (unpinned) part of our Autoware subset modules was 483.2, making it a 48.9x (97.98%) improvement in memory consumption for the dynamic portion of Autoware.

Assessing predictability. We extensively execute our runtime scheduler by running it multiple times and record the response time. First and foremost, the jitter variation in response time is low, with a maximum jitter of merely 834 microseconds. Moreover, we report the deadline miss rate in the last row of Table I. As is evident from the table, ResCue never misses the deadline and can improve response time considerably compared to vanilla Autoware, improving by 49% on average.

Assessing the response time. As an added granularity, we measure the end-to-end response time of Autoware execution under ResCue given the three scenarios. The cumulative distribution function for the response time in ms is given in Fig. 9. The deadlines (output interval) for each scenario is depicted as dashed black lines in the figure. As is evident in the figure, ResCue can meet the deadline by a margin which is due to variations in program execution time and the current disk transfer rate (which is always faster than $r_D$). However, the variation in execution time is minimal.

Assessing the dynamic data transfer scheduler. We also would like to measure the effect of our data transfer scheduler on Autoware. For this reason, we disable this feature (and only use reserve) and record the response time in ms, as is depicted in Fig. 10. As is evident in the figure, the response time for Autoware becomes larger than the deadline when there is no data scheduling. This is due to the fact that tasks require on-demand data loading from disk, which introduces latencies. Thus, our dynamic data transfer scheduler is quite effective in reducing the response time and making the system predictable.

Assessing the dynamic memory reservation scheme. Similarly, we would also like to measure the efficacy and the necessity of memory reservation in order to meet deadlines. Fig. 11 shows the cumulative distribution function for the measured end-to-end response time of Autoware with no
reservation. As is evident in the figure, the task set completely misses the deadline by an even wider margin when compared to no scheduler. This is due to the fact that the memory can become full for an unbounded amount of time (since processes may never exit or clear the memory), whereas the utilization of the data transfer channel can be bounded by the size of the program. Our dynamic memory reservation is quite effective and required in order to meet the deadline.

**Comparison to vanilla Autoware.** Finally, we measure the end-to-end response time of Autoware with all features of ResCue disabled in Fig. 12. As is evident in the figure, Autoware will completely miss the deadline by a very wide margin, and can have latencies of up to 450ms, which is unacceptable in a self-driving car scenario. This large latency is due to the absence of temporal and spatial data availability considerations. The variations among scenarios are a result of the different input periods, which leads to a different number of task set releases during the output period. Based on this experiment, we conclude that the predictability and the response time of Autoware can be improved substantially by our methods, making ResCue very effective in meeting deadlines and improving the response time.

**Overhead.** Finally, we report the overhead of ResCue both in terms of cumulative memory consumption and cumulative execution time in Fig. 13 measured by executing ResCue multiple times. This overhead includes the overhead associated with memory operations. As is evident in Fig. 13(a), the overhead for computation is negligible compared to the 1 second interval (1000000 steps). Moreover, the reserve time takes the majority of our overhead due to memory operation latencies. The scheduler itself has negligible overhead (below 1ms). The same is true for memory consumption in Fig. 13(b) which includes the reserve window (technically used by task sets but reserved by us). The scheduler itself requires minimal memory for data keeping.

**V. RELATED WORK**

Many recent works have been proposed on predictability and performance estimation in autonomous systems and autonomous embedded systems. Many have focused on convolutional neural networks (CNN) [49]–[55]. Our approach covers general AES workloads, where CNNs can be one of the tasks in the task set. Others have emphasized scheduling in AES, especially when accelerators such as GPU are present [10], [12]–[15], [25], [56]–[66]. However, none to the best of our knowledge cover memory limitations due to program code in their system design. The authors in [67] offer a review on embedded software approaches and mention memory size (for program code) as one of the major sources of contention currently facing AES, but offer no recent papers on system-level solutions on how to mitigate memory constraints, except [68]. However, [68] is limited to path planning and only considers memory interference (i.e., contention on memory access and not program code). Our approach is general and we resolve memory contentions by serializing them in a predictable manner (by calculating LCR and S). None of the aforementioned work consider periodic input/output (or even consider input events and output events), which is crucial for AES tasks.

Considering periodic input/output (also called events) for execution in CPS is a relatively recent application of the abstract timing model introduced by Giotto [22], [31]. In such systems, input and output are considered as periodic, but computation is also generally considered to be periodic [23]–[25] (to simplify scheduling, which is desired in LET), and memory is mentioned as a concern but never addressed. Some works do consider sporadic input task release [20], [26], however, are not fundamentally program code data-aware and focus on runtime execution scheduling.

Memory bottleneck has been the subject of many recent real-time system papers [69]–[73]. However, these works have focused on memory access by the CPU (thus memory-cache data transfer), even when embedded systems and IO were concerned [70], [74] and some works base their reservation schemes on the transfer channel bandwidth between CPU cache and main memory [30], [75]. Our focus is mainly on transferring program code into the main memory dynamically from disk in a real-time manner and to reduce jitter. Moreover, these works do not consider the combination problem of special periodic case for input/output under the integrated architecture with sporadic task release due to data/memory unavailability. In the context of dependent tasks in AES, memory constraint is generally only considered when two tasks depend on each other, but the architecture is heterogenous and data needs to be moved between the main memory and secondary accelerator memories (precedence constraint) [76], [77], but program code is never considered.

Finally, one work [78] does consider spatio-temporal constraints on memory when memory contention exists, but does not consider input/output and only focus on GPU. Our approach covers all types of processors and is generalized using a more robust system model.

**VI. CONCLUSION**

In this paper, we present ResCue, a predictable data-driven resource management for autonomous systems featuring the integrated architecture. We leverage the Heijunka methodology to establish a data-centric system model, and further develop ResCue which contains a data scheduler and a memory reservation scheme to ensure both temporal and spatial data availability. The implementation and evaluation using Autoware on top of an AES platform prove the efficacy of ResCue.

![Figure 13: Overhead of ResCue in terms of average execution time and worst-case memory consumption.](image-url)


