2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS) | 978-1-6654-9998-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/RTAS54340.2022.00009

Guaranteeing Safety Despite Physical Errors in Cyber-Physical Systems

1st Jongwoo Han
Dept. of Computer Science
Seoul National University
Seoul, Korea
jwhan@rubis.snu.ac.kr

2nd Seonghyeon Park
Dept. of Computer Science
Seoul National University
Seoul, Korea
seonghyeonpark@rubis.snu.ac.kr

3rd Haejoo Jeon
Dept. of Computer Science
Seoul National University
Seoul, Korea
haejoojeon@rubis.snu.ac.kr

4th Chang-Gun Lee
Dept. of Computer Science
Seoul National University
Seoul, Korea
cglee@rubis.snu.ac.kr

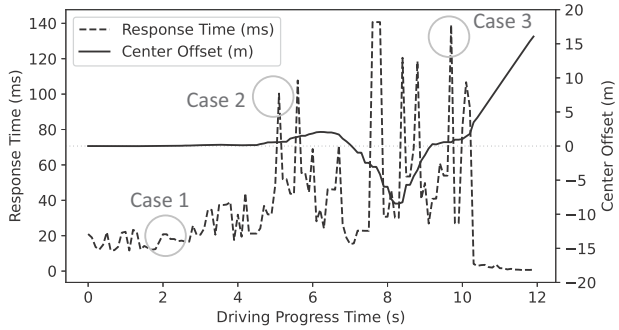
***Abstract*—This paper considers a cyber-physical system with a so-called "self-looping" node that repeats the inner-loop for physical situation awareness, i.e., more loops for more harsh physical situations. Regarding such a self-looping node, we observe the existence of physical errors that make the looping useless and eventually cause a critical failure. To prevent such a critical failure despite a physical error, this paper proposes a novel mechanism by introducing "time wall" and "safety backup". The time wall limits the time budget for the self-looping node so as to switch to the safety backup while still meeting the deadline to prevent critical failure despite physical errors. Our experiments through both simulation and actual implementation show that the proposed mechanism gives a comparable accuracy with the existing methods in normal cases while completely preventing the critical failure in physical error cases.**

***Index Terms*—Physical Error, Self-Looping Module, Time Wall, Safety Backup**

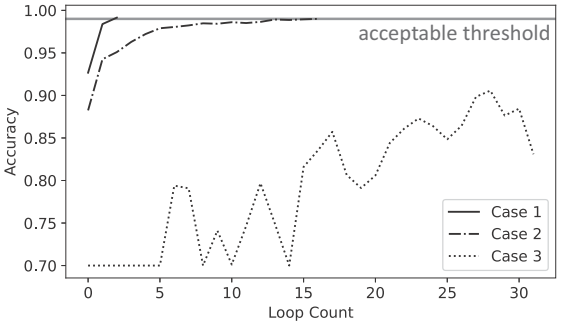
I. Introduction

Most recent cyber-physical systems such as autonomous driving systems include a complex computational module for physical situation awareness. The NDT (Normal Distribution Transform) matching module in Autoware [1], [2] (i.e., an open-source autonomous driving SW based on ROS) is a typical example. The module uses the current snapshot image sensed from the LiDAR sensor and tries to match it to the pre-built 3D point cloud map to localize the car's current position in the map. The module is programmed to repeat the matching several times with an inner-loop to find the best possible matching, that is, to find the accurate physical location of the car. Thus, we call such a module with an inner-loop a "self-looping" module. The ICP (Iterative Closest Point) algorithm for object tracking is also such an example [3], [4].

Such a self-looping module tends to improve its accuracy by increasing the looping count, similar to the concept of imprecise computation [5], [6]. However, one fundamental difference is that there exist cases where the accuracy never improves even though we increase the looping count. Such cases happen because it is impossible to cover all possible physical scenarios in the design phase of the self-looping module. If we encounter a physical scenario that is not well covered in the inner-loop design, even if the self-looping module repeats the inner-loop many times, the accuracy cannot improve above the acceptable threshold, which case we call a "physical error".



(a) Execution of NDT matching



(b) Accuracy of each case

Fig. 1. Cases of NDT Matching

For the example self-looping module of NDT matching, the dashed line in Fig. 1(a) shows the fluctuation of the actually measured execution times for its 120 periodic instances while autonomously driving the car from 0 sec to 12 sec. Such fluctuation is because the loop count necessary for reaching the acceptable accuracy varies depending on the physical situation as shown in Fig. 1(b) for Cases 1 and 2 of Fig. 1(a).

This work was supported in part by IITP grants funded by the Korean government (MSIT) (No.2015-0-00209, SW Starlab and No.2021-0-02068, AI Innovation Hub) and also in part by AI Institute at Seoul National University (AIIS).

978-1-6654-9998-9/22/$31.00 ©2022 IEEE
DOI 10.1109/RTAS54340.2022.00009

However, for some physical situations like Case 3 of Fig. 1, the self-looping module never reaches the acceptable accuracy—*physical error* and hits the time limit with an unacceptable accuracy. As a result, the localization by NDT matching fails, and in turn, the car starts moving far beyond the center of the lane as shown by the center offset, i.e., the car's position from the lane center, denoted by the solid line in Fig. 1(a).

This is an example case where a physical error happens because of an unexpected physical scenario that is not properly covered in the inner-loop design. The more serious problem is that the self-looping module never notices the physical error, and hence it continues the looping and may eventually violate a critical deadline.

A simple-minded solution for this is to set a maximum loop count. However, we do not know how to determine the maximum loop count. More seriously, what if the accuracy is still not acceptable even after the maximum loop count due to a physical error?

In order to tackle this challenge, this paper proposes a novel mechanism that can guarantee the minimal safety of a cyber-physical system despite physical errors using the notions of "time wall" and "safety backup". Intuitively speaking, for a self-looping module, we pre-compute the maximum possible time budget, i.e., "time wall", and allow the looping only within the time wall. If the self-looping module can achieve an acceptable accuracy within the time wall—normal case, the subsequent computing modules normally execute and finally actuate the physical system before the deadline. Otherwise—physical error case, a "safety backup" module executes within the deadline providing only minimal safety despite the physical error while giving up the advanced feature of the original self-looping module. For the Autoware example, if the NDT matching module achieves an acceptable accuracy within the assigned time wall, the subsequent modules make the car follow the optimal path from the car's current position to the final destination. Otherwise, a vision-based lane-keeping module, i.e., a safety backup, comes in and actuates the steering angle to simply keep the center of the lane while giving up the localization-based path following until the NDT matching module regains the acceptable accuracy.

For the proposed mechanism, we have to pre-compute the time wall, i.e., the time budget for a self-looping module, such that (1) all the subsequent nodes in the DAG can be completed before the deadline in the normal case and (2) the safety backup node and its subsequent nodes can be completed before the deadline in the physical error case. We propose two ways of computing the time wall, one for a pessimistic but simple budget analysis based on the classic response time bound analysis [7] and the other for a less pessimistic but more complex budget analysis based on the CPC (Concurrent Provider/Consumer) method [8].

Our experimental study by simulation with a synthetic workload shows that our approach completely prevents critical failures despite physical errors. Also, our actual implementation with Autoware shows that our approach safely keeps the car inside the driving lane even when the original Autoware makes the car cross over the lane boundary due to physical errors.

The rest of the paper is organized as follows: Section II presents related work. In Section III, we define the task and resource models and present a motivation example. Section IV describes our proposed safety guarantee mechanism against physical errors. Then, Section V explains a simple budget analysis based on classic bound. Section VI explains an advanced budget analysis based on CPC method. In Section VII, we reports our experiment results. Section VIII extends our approach to a more general model. Finally, Section IX concludes the paper.

II. Related Work

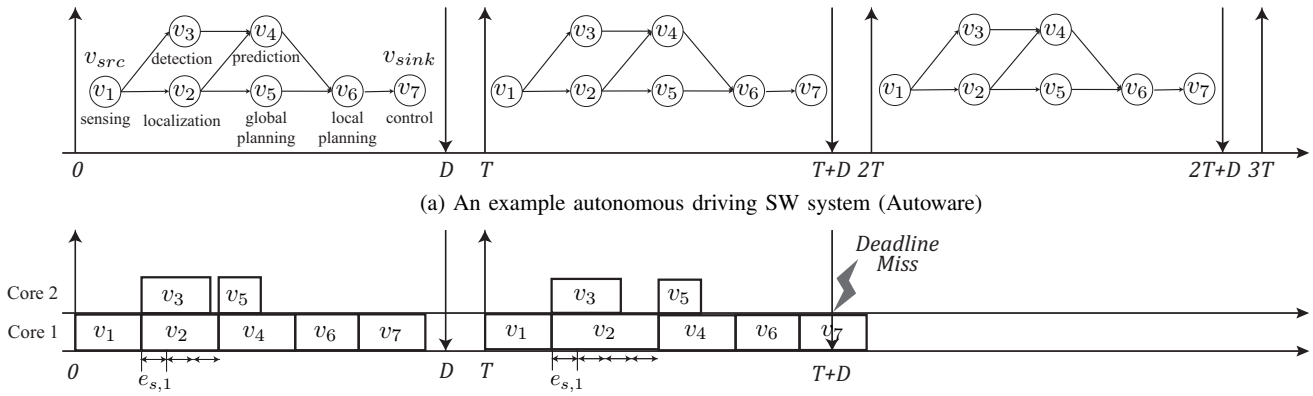
There have been lots of researches on DAG task scheduling on multicore processors. Their objective is reducing the makespan and tightening the worst-case analytical bound [7]–[15]. However, all of them assume fixed WCET for every node in the DAG. Thus, they cannot be directly applied to a DAG task with a self-looping node whose execution time largely varies depending on the loop count for different physical situations.

The self-looping node is similar to the concept of imprecise computation [5], [6] where the computational accuracy improves along with the invested time and hence the objective is to maximize the overall accuracy within time constraints. However, imprecise computation does not consider a physical error which makes continuing the computation useless.

To address the physical error, our proposed idea of switching to a safety backup is similar to the concept of the simplex algorithm [16], [17]. The simplex algorithm mathematically expresses system state space as an n-dimensional ellipsoid. The more complex the control algorithm is, the better the control performance is, but it is vulnerable to errors, i.e., the ellipsoid becomes smaller. Therefore, the fault can be prevented by switching to a simpler control algorithm with only basic control performance but more stable with a larger ellipsoid. However, for general software, it is impossible to envelop all physical situations with mathematical ellipsoids. Therefore, the simplex algorithm cannot be a concrete solution for physical errors in general cyber-physical systems.

III. Task and Resource Model

We consider a system with a single DAG task $\tau = \{T, D, \mathcal{G} = (V, E)\}$ that periodically executes a DAG as in Fig. 2. T is the period of the task and D is the task's relative deadline, which means that every instance of τ released at every period T should complete the execution of the DAG $\mathcal{G}$ before D. The DAG structure is defined as $\mathcal{G} = (V, E)$, where $V = \{v_1, \ldots, v_n\}$ is a set of nodes and $E \subseteq (V \times V)$ is a set of directed edges. Each node v_i represents a computational module and a directed edge (v_i, v_j) from v_i to v_j represents the precedence constraint meaning that the computation module v_i should be completed before starting v_j. This task model well represents a ROS application like Autoware [2], [18], [19]. Without loss of generality, we assume that the DAG



(a) An example autonomous driving SW system (Autoware)

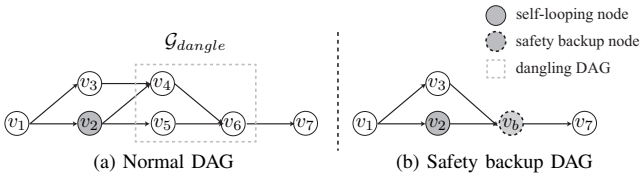
(b) An example schedule on two processors

Fig. 2. Autonomous Driving Task Example

$\mathcal{G}$ has exactly one source node v_{src} and sink node v_{sink}. Fig. 2(a) shows a simplified view of Autoware, where the given DAG has ROS nodes, i.e., $v_1 = v_{src}$ for sensing the surrounding environment with LiDAR, v_2 for NDT matching based localization, v_3 for surrounding object detection, v_4 for objects' motion prediction, v_5 for global path planning, v_6 for local path planning to follow the waypoints of the global path avoiding collisions with other objects, and $v_7 = v_{sink}$ for finally actuating the car along with the local path planning. Such DAG should be periodically executed to drive the car safely.

For every node v_i in $\mathcal{G} = (V, E)$, we assume the fixed WCET denoted by e_i except one node v_s called a *self-looping node*. The self-looping node v_s has a largely varying execution time depending on the looping count necessary for accurate awareness of varying physical situations. In the above Autoware example of Fig. 2(a), the NDT matching node v_2 is a self-looping node that repeats the matching of the current LiDAR image to the pre-built 3D point cloud map until the acceptable matching accuracy can be achieved for accurately localizing the car's current position in the map. For such a self-looping node v_s, we define the execution time for one loop as $e_{s,1}$. Therefore, when v_s repeats the inner-loop L times for the acceptable accuracy, its WCET is modeled as $e_s = L \times e_{s,1}$. The self-looping node tends to achieve better accuracy by increasing the loop count but "not always". When it encounters a physical situation that is not well covered at the design time of its inner-loop, which we call *physical error*, the accuracy does not improve by repeating the inner-loop as shown in Case 3 of Fig. 1. Note that it is totally unpredictable when the self-looping node encounters a physical error.

For executing such a single DAG task with a self-looping node, we assume a computing hardware platform with M identical processors. In order to determine which nodes should be executed on the M processors when more than M nodes are concurrently ready while satisfying all of their precedence constraints, we assume fixed-priority non-preemptive scheduling. In other words, we assume that a fixed priority is assigned to each node as in [8] and, when a processor becomes idle, the highest priority node out of all the ready nodes starts executing on the processor. Once a node starts executing, it continues to the end without being preempted even if a higher priority node becomes ready. Fig. 2(b) shows an example of such schedule when $M = 2$. The first instance shows a case where the self-looping node v_2 repeats the inner-loop three times, and hence the DAG completes meeting the deadline. On the other hand, the second instance shows a case where v_2 repeats the inner-loop four times and eventually misses the deadline.



(a) Normal DAG (b) Safety backup DAG

Fig. 3. Normal DAG and Safety Backup DAG

For this task and resource model, our problem is how to execute the DAG meeting every deadline while guaranteeing minimal safety even when the self-looping node encounters a physical error which makes the inner-loop useless. Extension to multiple DAG tasks with multiple self-looping nodes will be briefly sketched in Section VIII.

IV. Proposed Safety Guarantee Mechanism Against Physical Errors

Our proposed mechanism for guaranteeing minimal safety despite physical errors uses notions of "time wall" and "safety backup". The time wall is the amount of time budget allowed for the self-looping node. If the self-looping node finishes with an acceptable accuracy before hitting the time wall, the subsequent nodes normally execute. Otherwise, it may be due to a physical error and hence a "safety backup" executes to guarantee minimal safety.

For this, the safety backup DAG is formally defined as follows: For the case where the self-looping node v_s fails to achieve an acceptable accuracy before hitting the time wall, a safety backup node v_b is introduced. The safety backup node v_b is designed with the objective of guaranteeing only minimal

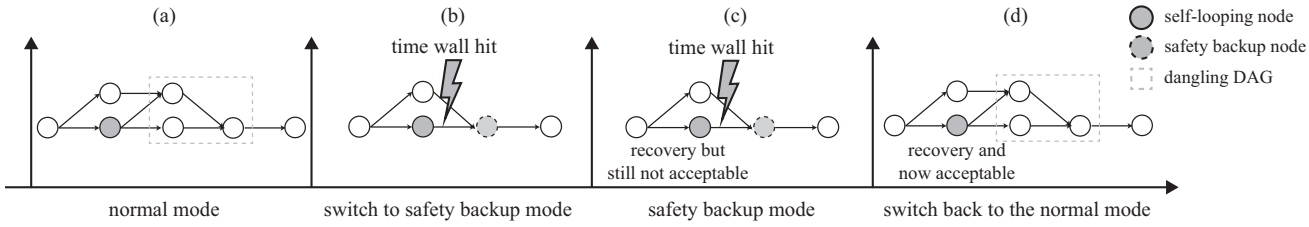


Fig. 4. Normal Case and Physcial Error Case

safety and hence has much simpler logic that can be guaranteed to work in a broader spectrum of physical situations. For the above Autoware example in Fig. 2(a), the lane-keeping module can be an example safety backup node for the self-looping node, i.e., NDT matching (v_2), and its subsequent nodes (v_4, v_5, v_6) for the localization-based path following. When the NDT matching fails in accurately localizing the car's position, the lane-keeping module can continue driving only forward while keeping the car at the center of the lane with a vision-based lane detection algorithm [20]. As we can understand in this example, the safety backup node replaces the roles of not only the self-looping node v_s itself but also some of its subsequent nodes. The sub-DAG consisting of such subsequent nodes replaced by the safety backup node is called a *dangling DAG* of v_s and denoted by $\mathcal{G}_{dangle}$. For the example normal DAG of Autoware in Fig. 3(a), the sub-DAG consisting of v_4, v_5, v_6 represented by the dashed box is the dangling DAG of the NDT matching node v_2. This dangling DAG $\mathcal{G}_{dangle}$ will be replaced by the safety backup node v_b. Thus, the safety backup v_b has the same predecessors and successors as the dangling DAG $\mathcal{G}_{dangle}$. The safety backup node v_b does not use the data from v_s but may still use the data from other predecessors. In addition, v_b should produce the same format data compatible with the non-replaced successors. As a result, the safety backup DAG is formed from the original normal DAG by replacing the dangling DAG $\mathcal{G}_{dangle}$ with the safety backup node v_b. Fig. 3(b) shows the safety backup DAG for the normal DAG of Autoware example in Fig. 3(a).

Note that even in the safety backup DAG, the self-looping node v_s keeps executing, which is necessary for the recovery effort to regain the accuracy of the self-looping node and roll back to the normal DAG.

With such defined normal DAG and safety backup DAG, our proposed mechanism switches back and forth between the normal mode and safety backup mode as follows:

- **Normal mode**: If the self-looping node finishes before hitting the time wall, the normal DAG continues executing to the end. The first instance of Fig. 4 shows such a normal mode execution.
- **Switch to safety backup mode**: If the self-looping node hits the time wall with an unacceptable accuracy, we give up the dangling DAG $\mathcal{G}_{dangle}$ and execute the safety backup node v_b instead, which is the switch to the safety backup DAG. The second instance of Fig. 4 shows such a switch.

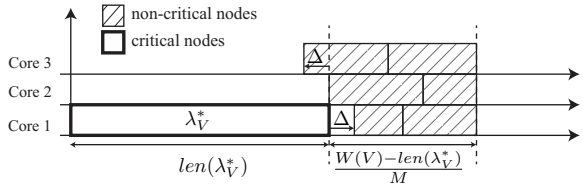


Fig. 5. Intuition of Classic Bound

- **Safety backup mode**: In the safety backup mode, the self-looping node v_s performs a recovery action . If the recovery action cannot regain the acceptable accuracy before hitting the time wall, the safety backup node v_b and its subsequent nodes in the safety backup DAG execute to provide only minimal safety while giving up the features of the normal DAG. The third instance of Fig. 4 shows such a safety backup mode execution.
- **Switch back to the normal mode**: After several periods of the safety backup mode, the self-looping node's recovery action can regain acceptable accuracy. In that case, we roll back to the normal mode by executing the dangling DAG $\mathcal{G}_{dangle}$ and its subsequent nodes to the end, which is the switch back to the normal DAG. The fourth instance of Fig. 4 shows such a switch back to normal.

For this mechanism to successfully work, the remaining issue is how to determine the time budget for the self-looping node such that both normal and safety backup DAGs can be successfully scheduled on M identical processors before the deadline D.

V. Classic Bound based Budget Analysis

Our first method to calculate the time budget for the self-looping node is based on the classic WCRT (Worst-Case Response Time) bound [7]. The classic WCRT bound gives an upper bound of the WCRT for any work-conserving scheduling of all the nodes of the given DAG on M identical processors. The classic WCRT bound can be intuitively explained as

For the NDT matching example, we may use the information from the low-quality GPS as the initial pose and repeat the inner-loop to find the acceptable matching. It may take several retries over multiple task periods until the acceptable accuracy can be regained while the safety backup node is backing up for the minimal safety.

follows: For a given DAG $\mathcal{G} = (V, E)$, we define its total workload $W(V)$ as the sum of WCETs of all the nodes, i.e., $W(V) = \sum_{v_j \in V} e_j$. We also define a path λ as an ordered set $\{v_{src}, \ldots, v_{sink}\}$ which is a sequence of nodes from the starting node v_{src} to the ending node v_{sink} such that $(v_k, v_{k+1}) \in E, \forall v_k \in \lambda - \{v_{sink}\}$. The length of a path λ is defined as $len(\lambda) = \sum_{v_j \in \lambda} e_j$. Out of all the paths in V, the longest one is defined as the critical path λ_V^*. The nodes in the critical path are called the critical nodes, while other nodes are non-critical nodes. With these definitions, the classic WCRT bound $R(V)$ is given by Eq. (1).

$$R(V) = len(\lambda_V^*) + \frac{W(V) - len(\lambda_V^*)}{M}. \tag{1}$$

This equation most pessimistically considers no overlap between the critical nodes and the non-critical nodes by sequentially adding the critical path length $len(\lambda_V^*)$ and the length for executing non-critical nodes with the M cores, i.e., $\frac{W(V)-len(\lambda_V^*)}{M}$, as illustrated in Fig. 5. $\frac{W(V)-len(\lambda_V^*)}{M}$ is an upper bound of the extra delay beyond $len(\lambda_V^*)$ by all the non-critical nodes. This is because even if a non-critical path λ is longer than $\frac{W(V)-len(\lambda_V^*)}{M}$ by an amount of Δ as marked by the left-arrow Δ in the figure, it makes the same size hole marked by the right-arrow Δ. Also, such Δ cannot be larger than the critical path length $len(\lambda_V^*)$. Therefore, Eq. (1) gives a safe upper bound of the response time.

With this classic response time bound $R(V)$, since our goal is to find the maximum possible time budget e_s for the self-looping node v_s to meet the deadline D, we define the critical path λ_V^* as the longest path out of all the paths including v_s. With such defined λ_V^*, the total workload $W(V)$ and the critical path length $len(\lambda_V^*)$ can be rewritten as follows:

$$W(V) = e_s + \sum_{v_j \in (V - \{v_s\})} e_j,$$
$$len(\lambda_V^*) = e_s + \sum_{v_j \in (\lambda_V^* - \{v_s\})} e_j.$$

With these $W(V)$ and $len(\lambda_V^*)$, the classic WCRT bound in Eq. (1) can be rewritten as a function of e_s as follows:

$$R(V) = e_s + \sum_{v_j \in (\lambda_V^* - \{v_s\})} e_j + \frac{\sum_{v_j \in (V - \lambda_V^*)} e_j}{M}. \tag{2}$$

Since this WCRT bound $R(V)$ should be less than or equal to D, the maximum possible time budget e_s for v_s is given as follows:

$$e_s = D - \sum_{v_j \in (\lambda_V^* - \{v_s\})} e_j - \frac{\sum_{v_j \in (V - \lambda_V^*)} e_j}{M}. \tag{3}$$

By applying this budget analysis in Eq. (3) to both cases of normal DAG in Fig. 3(a) and safety backup DAG in Fig. 3(b), we can obtain two budgets denoted by e_s^{norm} and e_s^{backup}. Taking the minimum of these two, i.e., $e_s = \min\{e_s^{norm}, e_s^{backup}\}$, we can finally determine the time budget e_s for v_s that can meet the deadline D in both normal and safety backup modes.

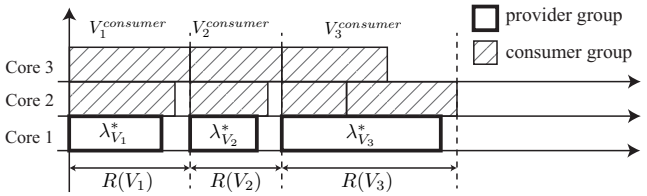


Fig. 6. Abstraction of CPC Method

VI. CPC BASED BUDGET ANALYSIS

Since the classic response time bound in Eq. (1) is pessimistic, the budget for the self-looping node obtained in the previous section, i.e., Eq. (3), is quite limited. In order to give the self-looping node as much budget as possible, in this section, we propose a more advanced budget analysis based on the CPC (Concurrent Provider/Consumer) method [8]. The CPC method mitigates the pessimism of the classic bound by considering the possible concurrent executions among critical nodes and non-critical nodes. For this, the CPC method partitions the critical path λ_V^* into a sequence of segments $\lambda_{V_1}^*, \lambda_{V_2}^*, \cdots, \lambda_{V_n}^*$ as in Fig. 6. Each $\lambda_{V_i}^*$ is a subset of the critical nodes in the critical path and is called a "provider group" that occupies one core for its own execution while providing $M - 1$ cores to a "consumer group" denoted by $V_i^{consumer}$, that is, a subset of the non-critical nodes that can be concurrently executed with the provider group. We denote the union of the provider and consumer groups of each segment by $V_i = \lambda_{V_i}^* \cup V_i^{consumer}$. We also denote the segment including v_s as V_s. Note that subscript s of v_s and V_s may have different values, but for the notational simplicity we use the same s wherever there is no confusion. For the details of constructing provider and consumer groups, the interested readers are referred to [8].

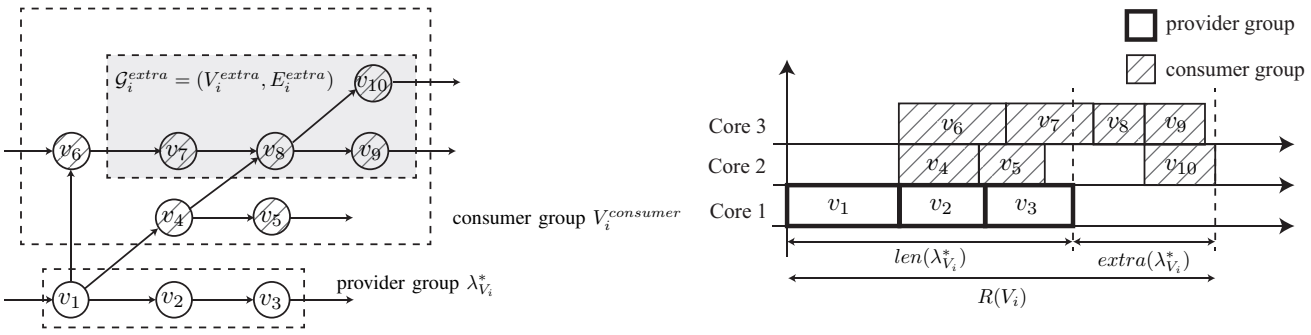
With this partition of the entire DAG into a sequence of segments, the WCRT bound $R(V)$ is given as the sum of the WCRT bound $R(V_i)$ of each segment V_i as follows:

$$R(V) = \sum_{i=1}^{n} R(V_i). \tag{4}$$

In order to explain how the CPC method computes $R(V_i)$, let us use the example segment in Fig. 7 where v_1, v_2, v_3 is the provider group $\lambda_{V_i}^*$, i.e., a part of the critical path λ_V^*, and $v_4, v_5, v_6, v_7, v_8, v_9, v_{10}$ is the consumer group $V_i^{consumer}$. The CPC method computes $R(V_i)$ as follows:

$$R(V_i) = len(\lambda_{V_i}^*) + extra(\lambda_{V_i}^*). \tag{5}$$

where $len(\lambda_{V_i}^*)$ is the provider group's length and $extra(\lambda_{V_i}^*)$ is the delay beyond $len(\lambda_{V_i}^*)$ by some of the consumer group nodes. More specifically, the CPC method computes the "finish time bounds" for all the nodes in the DAG and considers the consumer group nodes with finish time bounds earlier than or equal to $len(\lambda_{V_i}^*)$ can be executed in parallel with the provider group. In the Fig. 7(b), v_4, v_5, and v_6 are such nodes. Only the consumer group nodes whose finish time bounds are later than $len(\lambda_{V_i}^*)$ can make the extra delay $extra(\lambda_{V_i}^*)$.



(a) DAG structure of segment V_i

(b) Calculating $R(V_i)$

Fig. 7. Calculating Response Time $R(V_i)$ of One Segment V_i

We denote the set of such nodes by V_i^{extra}. In the Fig. 7, $V_i^{extra} = \{v_7, v_8, v_9, v_{10}\}$. Now, $extra(\lambda_{V_i}^*)$ can be computed by applying the classic bound, i.e., Eq. (1), to the sub-DAG $\mathcal{G}_i^{extra}$ formed by V_i^{extra}, e.g., in Fig. 7, v_7, v_8, v_9 and v_{10} and their associated edges denoted by E_i^{extra} . As a result, the CPC method computes $extra(\lambda_{V_i}^*)$ as follows:

$$extra(\lambda_{V_i}^*) = len(\lambda_{V_i^{extra}}^*) + \frac{W(V_i^{extra}) - len(\lambda_{V_i^{extra}}^*)}{M}. \tag{6}$$

Unlike the classic bound, we cannot directly apply the CPC method to our problem due to the dependency between the time budget of the self-looping node e_s and the CPC method's $extra(\lambda_{V_i}^*)$ formula in Eq. (6). Recall that the CPC method computes the finish time bounds to form the sub-DAG $\mathcal{G}_i^{extra}$ for computing $extra(\lambda_{V_i}^*)$. However, due to the self-looping node v_s whose time budget e_s is not determined yet, the finish time bounds of v_s's descendant nodes become non-deterministic. Therefore, for the segment containing v_s and all the subsequent segments, their extra delays $extra(\lambda_{V_i}^*)$s and also the WCRT bounds $R(V_i)$s cannot be computed until resolving the non-determinism.

We tackle this challenge by leveraging the sustainability of the CPC method, which says that the WCRT bound $R(V_i)$ computed assuming a larger execution time of any node v is also a safe upper bound of the response time for all the cases of a shorter execution time of v. For this, in Subsection VI-A, we first compute the initial budget for the self-looping node by applying the CPC formular assuming an upper limit of e_s when computing the response time bounds of v_s's subsequent segments. Then, in Subsection VI-B, we perform a binary search to maximally enlarge the initial budget under the deadline constraint.

When forming the sub-DAG that can make the extra delay $extra(\lambda_{V_i}^*)$, the CPC method considers a heuristically assigned priority of the nodes to further mitigate the pessimism of the original classic bound [8].

A. Initial Budget Calculation

In order to address the dependency of undetermined e_s of the self-looping node v_s and its subsequent segment's response time bounds, we use an upper limit of e_s to conservatively compute its subsequent segments' response time bounds. Such an upper limit of e_s can be given by the fact that the length of the critical path, i.e., $len(\lambda_V^*)$ should be less than or equal to D:

$$\sum_{v_j \in \lambda_V^*} e_j \leq D.$$

This is a necessary condition for the DAG $\mathcal{G} = (V, E)$ to be completed before D. Using this necessary condition, an upper limit of e_s denoted by e_s^{max} can be given as follows:

$$e_s^{max} = D - \sum_{v_j \in \lambda_V^* - \{v_s\}} e_j. \tag{7}$$

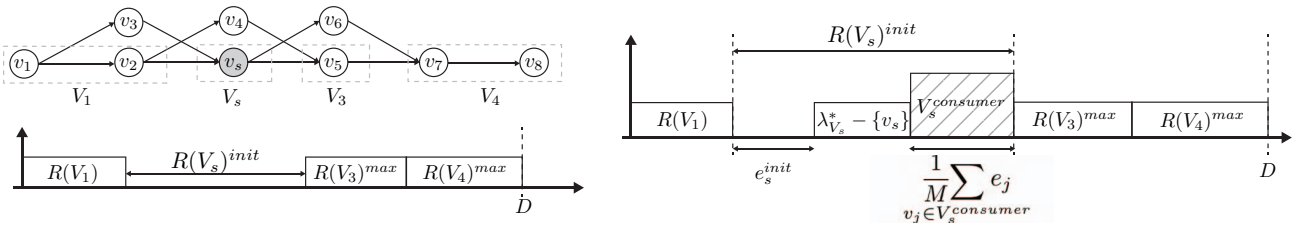
Assuming this e_s^{max} for v_s, the CPC method can now conservatively compute the finish time bounds for all of v_s's descendant nodes and in turn compute the subsequent segment's response time bounds denoted by $R(V_i)^{max}(i > s)$ as depicted in Fig. 8(a). With these conservative $R(V_i)^{max}(i > s)$ values, the initial time budget denoted by $R(V_s)^{init}$ that can be given to v_s's segment V_s under the deadline D constraint can be given as follows as depicted in Fig. 8(a):

$$R(V_s)^{init} = D - \sum_{i<s} R(V_i) - \sum_{i>s} R(V_i)^{max}. \tag{8}$$

Using this conservatively computed initial budget $R(V_s)^{init}$ for v_s's segment, now the initial budget e_s^{init} for v_s itself can be computed as follows by pessimistically applying the classic bound budget analysis, i.e., Eq. (3) to the segment V_s as depicted in Fig. 8(b):

$$e_s^{init} = R(V_s)^{init} - \sum_{v_j \in \lambda_{V_s}^* - \{v_s\}} e_j - \frac{\sum_{v_j \in V_s^{consumer}} e_j}{M}. \tag{9}$$

Regarding this computed e_s^{init}, we can claim that if the self-looping node executes shorter than e_s^{init}, the DAG $\mathcal{G} = (V, E)$ can be completed before D as formally stated in the following lemmas and theorem.



(a) Calculating $R(V_s)^{init}$

(b) Calculating e_s^{init}

Fig. 8. Calculating Initial Budget e_s^{init}

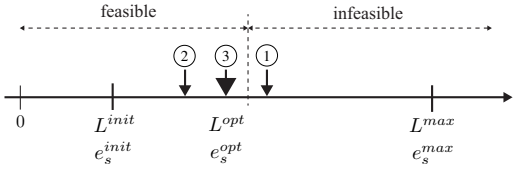


Fig. 9. Binary Search for Finding the Optimal Budget

Lemma 1. ***(Sustainability of the CPC method)*** *If any node v_j of a segment V_i executes less than its WCET, V_i's response time is not greater than $R(V_i)$. Using the same rationale, the lemma still holds if v_j's finish time becomes earlier just like v_j's execution time becomes shorter.*

Proof. This sustainability of the CPC method is proven in [8]. □

Lemma 2. *If the actual execution time v_s is shorter than e_s^{max}, the actual response times of all V_s's subsequent segments $V_i (i > s)$ is not greater than $R(V_i)^{max}$.*

Proof. If the actual execution time of v_s is shorter than e_s^{max}, the finish time bound of every v_s's descendent node, say v_j, becomes earlier or the same compared with the one computed assuming e_s^{max}. Thus, the actual response time of the segment $V_i (i > s)$ containing v_j is not greater than $R(V_i)^{max}$ due to Lemma 1. □

Theorem 1. *If the self-looping node v_s executes shorter than e_s^{init}, the DAG $\mathcal{G} = (V, E)$ can be completed before D.*

Proof. Due to Lemma 2, for all cases of $e_s \leq e_s^{init} \leq e_s^{max}$, the response times of all the V_s's subsequent segments are not greater than $R(V_i)^{max} (i > s)$. Thus, the time budget that the v_s's segment V_s can have within the deadline D is larger than or equal to $R(V_s)^{init}$ by Eq. (8). Also, due to Eq. (9), if the self-looping node v_s executes shorter than e_s^{init}, the v_s's segment V_s takes shorter than $R(V_s)^{init}$. Therefore, the DAG can be completed before D. □

B. Binary Search for Finding the Optimal Budget

The initial budget e_s^{init} by Eq. (9) is feasible to meet the deadline D as stated in Theorem 1 but unnecessarily limited since $R(V_i)^{max} (i > s)$ for the v_s's subsequent segments are pessimistically large assuming e_s^{max}. Therefore, there can exist a larger but still feasible budget e_s in between e_s^{init} and e_s^{max}.

In this subsection, we propose a binary search algorithm to find an optimal budget e_s^{opt} in between e_s^{init} and e_s^{max}. Using e_s^{init} and e_s^{max}, we can compute their corresponding loop counts L^{init} and L^{max}, respectively, as follows:

$$L^{init} = \left\lfloor \frac{e_s^{init}}{e_{s,1}} \right\rfloor,$$
$$L^{max} = \left\lfloor \frac{e_s^{max}}{e_{s,1}} \right\rfloor.$$

Within the integer space of $[\max\{0, L^{init}\}, L^{max}]$, our algorithm conducts a binary search to find the largest feasible loop count L as depicted in Fig. 9. Algorithm 1 formally states this binary search algorithm. Lines 1 and 2 initially set $L^{low} = \max\{0, L^{init}\}$ and $L^{high} = L^{max}$. The **while** loop from Line 3 to Line 13 actually conducts the binary search while $L^{low} < L^{high}$. In each iteration of the **while** loop, Line 4 sets L^{mid} as $\lfloor (L^{high} + L^{low} + 1)/2 \rfloor$. Line 5 computes the corresponding execution time e_s for the self-looping node for L^{mid}. Using e_s, Line 6 and 7 uses the CPC method, i.e., Eqs. (4), (5) and (6), to compute each segment's response time bound $R(V_i)$ and in turn the overall response time bound $R(V)$. If such computed $R(V)$ is greater than D as in Line 8, it means L^{mid} and corresponding e_s are infeasibly large. Thus, we shrink L^{high} to $L^{mid} - 1$ in Line 9 to check with a smaller L^{mid} in the next iteration. Otherwise, i.e., $R(V) < D$, it means L^{mid} and its corresponding e_s are feasible but too small. Thus, we enlarge L^{low} to L^{mid} in Line 11 to check with a larger L^{mid} in the next iteration. After completing this binary search, L^{low} indicates the largest feasible loop count, and hence we return $L^{low} \times e_{s,1}$ for e_s^{opt}. If $L^{low} \geq 1$, we can use the returned e_s^{opt} value as largest feasible time budget for v_s. Otherwise, we cannot assign a budget to v_s since DAG is not feasible even when $e_s = 0$.

According to [8], the CPC method has a time complexity of $O((|V| + |E|)^2)$. Therefore, our binary search algorithm Algorithm 1 can be performed with a time complexity of $O(logN \times (|V| + |E|)^2)$ where N is $L^{max} - \max\{0, L^{init}\}$.

Theorem 2 says that Algorithm 1 finds the optimal time budget e_s for the self-looping node v_s with the condition that the CPC method is used for the feasibility check and all the nodes v_i actually take the WCET e_i.

Algorithm 1 Optimal Budget Selection Algorithm

Input: $\{V_i : 1 \leq i \leq n, V_i = \lambda^*_{V_i} \cup V_i^{consumer}\}, D$
Output: e_s^{opt}

$L^{low} = \max\{0, L^{init}\}$
$L^{high} = L^{max}$
while $L^{low} < L^{high}$ **do**
  $L^{mid} = \lfloor (L^{high} + L^{low} + 1)/2 \rfloor$
  $e_s = L^{mid} \times e_{s,1}$
  calculate $R(V_i)(1 \leq i \leq n)$
  $R(V) = \sum_{i=1}^{n} R(V_i)$
  if $R(V) > D$ **then**
    $L^{high} = L^{mid} - 1$
  else
    $L^{low} = L^{mid}$
  end if
end while
return $L^{low} \times e_{s,1}$

Theorem 2. *The time budget obtained through Algorithm 1 is optimal under the following condition.*

1) *CPC method is used for feasibility check.*
2) *All nodes v_i actually take the WCET e_i.*

Proof. Using $e_s^{opt} = L \times e_{s,1}$ obtained by Algorithm 1, the response time $R(V)$ calculated by CPC method is less than or equal to D. However, for $e_s = (L+1) \times e_{s,1}$, the computed response time bound $R(V)$ by the CPC method is greater than D. Therefore, e_s^{opt} is the largest feasible time budget for v_s that makes the CPC based response time bound $R(V)$ shorter than or equal to D when all other nodes v_is actually take the WCET e_is. □

As in the classic bound based budget analysis, we apply this CPC based budget analysis for both normal DAG in Fig. 3(a) and safety backup DAG in Fig. 3(b) to compute their corresponding budgets e_s^{norm} and e_s^{backup} and take the minimum of them to finally determine the time budget e_s for the self-looping node v_s to meet the deadline D in both normal and safety backup modes.

VII. Evaluation

This section evaluates the proposed mechanism through both simulation and actual implementation.

A. Simulation with Synthetic DAG Workload

In order to show the effectiveness of our proposed mechanism for various DAG workloads, this section conducts simulation with randomly generated synthetic tasks assuming $M = 4$ identical cores.

A synthetic task is randomly generated as follows: (1) For a DAG $\mathcal{G}$, the node number N is randomly chosen from uniform(30, 50) and the DAG depth is randomly chosen from uniform(5,8). One node is included in the first layer and another node is included in the last layer to make a single source and a single sink. Then, all other nodes are randomly distributed to the remaining layers. Edges are randomly created to connect a pair of nodes such that every node has at least one path from the source and at least one path to the sink. One randomly chosen node is marked as a self-looping node v_s and its single loop execution time $e_{s,1}$ is 8 ms. All other nodes' execution times e_is are randomly chosen from uniform(20 ms, 60 ms) with the average e_{avg} of 40 ms. (2) The period T and the deadline D are assumed to be the same. Thus, the task's density ρ on $M = 4$ cores is represented by $\frac{e_{avg} \times N}{T \times M} = \frac{e_{avg} \times N}{D \times M}$. To make a synthetic task with a specific density ρ, $T = D$ is determined as $\frac{e_{avg} \times N}{\rho \times M}$.

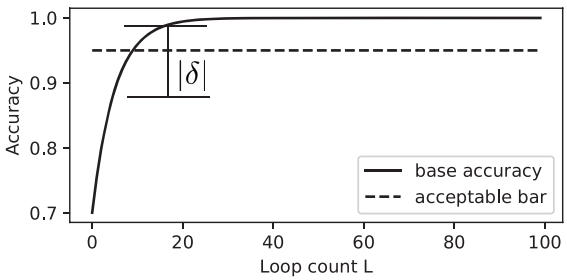


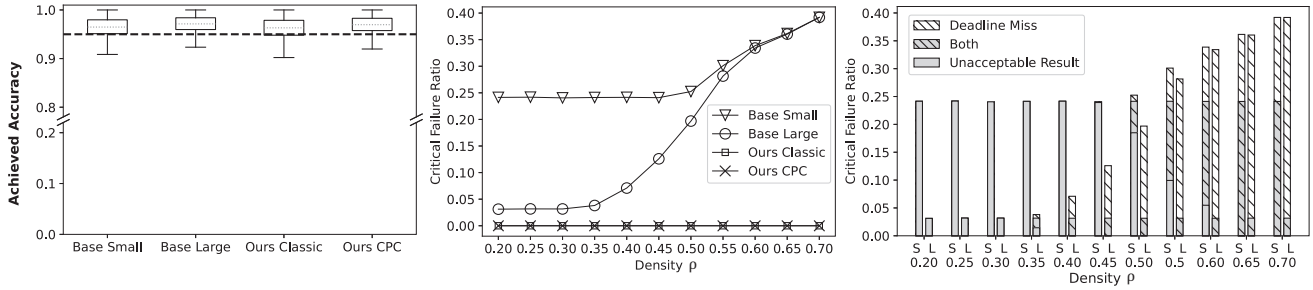
Fig. 10. Self-looping Node Model in Synthetic Workload

In the experiment, we use only feasible DAGs meaning that they can be scheduled on $M = 4$ cores before D by the fixed-priority non-preemptive scheduling of CPC [8] when the self-looping node's loop count L is 1. For such a feasible DAG $\mathcal{G}$, we choose a set of v_s's descendent nodes as a dangling DAG $\mathcal{G}_{dangle}$ such that $\mathcal{G}_{dangle}$'s workload is 20% of the total workload of $\mathcal{G}$. For the backup node v_b that replaces $\mathcal{G}_{dangle}$, its execution time e_b is set to a half of $\mathcal{G}_{dangle}$'s workload.

The self-looping node v_s is characterized by the following accuracy function $A(L)$:

$$A(L) = 1 - e^{-L/5 + ln0.3} - |\delta|. \tag{10}$$

This accuracy function is well illustrated in Fig. 10. In this function, the $1 - e^{-L/5+ln0.3}$ part represents the base accuracy that increases as increasing the loop count L. In addition to this base accuracy, we subtract the $|\delta|$ part to model randomly happening physical errors. δ follows the normal distribution $N(0, \sigma)$ and hence our experiment controls the probability of physical errors with σ, that is, a large σ models a harsh physical situation with a high probability of physical errors. In our experiment, for each loop of the self-looping node, we increase L by one and then use the above $A(L)$ function to generate the accuracy value. When the value becomes higher than the acceptable bar, i.e., 0.95 in our experiment, the self-looping node completes and produces the result to its descent nodes. Alternatively, the self-looping node may stop regardless of the accuracy when the loop count L reaches the loop count limit.



(a) Accuracy (ρ = 0.4, σ = 1.0)

(b) Critical failure ratio v.s. density ρ (σ = 1.0)

(c) Failure reason for baseline approaches

Fig. 11. Simulation Result for Synthetic Workload

With this setting, we compare the following four methods:

- Base Small: The basic DAG execution without a safety backup. The self-looping node has a small loop count limit of 50.
- Base Large: The same as Base Small except that the self-looping node has a large loop count limit of 100.
- Ours Classic: Our proposed mechanism with safety backup. The time wall is based on the classic bound based budget analysis in Section V.
- Ours CPC: Our proposed mechanism with safety backup. The time wall is based on the CPC based budget analysis in Section VI.

With these four methods, we simulate each randomly generated DAG $\mathcal{G}$'s execution on $M = 4$ cores for 100 periodic instances. The following results are statistics for 10,000 DAGs.

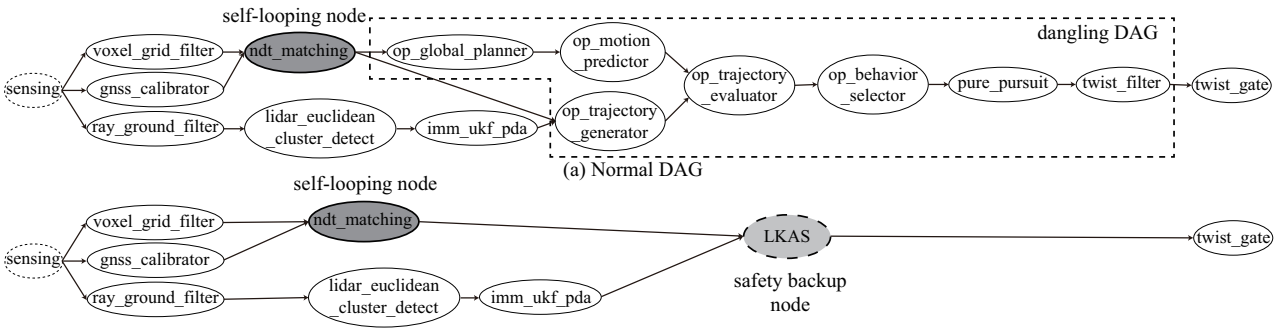
Fig. 11(a) shows the statistics (i.e., mean, quartiles, minimum and maximum value) of the achieved accuracy by the above four methods when the DAG's density ρ is 0.4 and standard deviation σ for modeling physical errors is 1.0. For both Base Small and Base Large, the mean of the achieved accuracy is higher than the acceptable bar 0.95 (i.e., dashed line in Fig. 11(a)). Obviously, Base Large shows higher achieved accuracy than Base Small since the former runs the self-looping node with a larger loop count limit. However, one thing we have to note is that, even for Base Large, there exist cases where the achieved accuracy is lower than the acceptable bar, which can lead to the critical failure if there is no safety backup. Ours Classic achieves relatively low accuracy since the time budget assigned to the self-looping node is quite limited due to the pessimism of the classic response time bound. On the other hand, Ours CPC achieves higher accuracy, which is comparable with Base Large. Moreover, even when the achieved accuracy is lower than the acceptable bar, Ours Classic and Ours CPC have a safety backup that prevents critical failure, which is not the case for Base Small and Base Large.

The critical failure can happen (1) when the achieved accuracy is lower than the acceptable bar or (2) when the DAG execution misses the deadline. Fig. 11(b) compares the critical failure ratios by the above four methods as increasing the DAG's density ρ when $\sigma = 1.0$. Base Small shows a non-negligible critical failure ratio even when the DAG's density ρ is small. This is because there is a non-negligible probability that the accuracy is lower than the acceptable bar due to a small limit of loop count. On the other hand, Base Large shows a very small (even if non-zero) critical failure ratio thanks to a large loop count limit when ρ is small. However, as increasing ρ, the large limit of loop count is likely to make the deadline miss and hence the critical failure ratio also increases. For a better understanding of the reasons for the critical failures, Fig. 11(c) reports the breakdown of the reasons for the critical failures of Base Small and Base Large.

Unlike Base Small and Base Large, Ours Classic and Ours CPC show zero critical failure ratio. This is because (1) they allow the self-looping node's execution within the time wall to guarantee the deadline in all cases and (2) even if such achieved accuracy by the self-looping node is lower than the acceptable bar, the safety backup can always backup within the deadline.

B. Implementation

In this subsection, we implement our proposed mechanism based on Autoware [2]. The DAG structure of the original Autoware, which is the normal DAG, is shown in Fig. 12(a). The $voxel\_grid\_filter$ node filters LiDAR sensing data and $gnss\_calibrator$ calculates the car's rough pose and position based on GPS. They produce the resulting data to the $ndt\_matching$ node. The $ndt\_matching$ node is our self-looping node that loops the inner-loop to find more accurate localization using the data produced by its preceding nodes. Then, the $op\_global\_planner$ and $op\_trajectory\_generator$ nodes perform the global planning and generate the car's future trajectory to follow the global path planning. On the other hand, the ray_ground_filter, $lidar\_euclidean\_cluster\_detect$ and imm_ukf_pda nodes detect surrounding obstacles and then the $op\_motion\_predictor$ node predict the obstacles' future behavior relative to the car's current position. Considering the global planning based trajectory and obstacles' predicted behavior, the $op\_trajectory\_evaluator$ and $op\_behavior\_selector$ finds the local path to follow while avoiding collision with obstacles. Then, the $pure\_pursuit$,



(b) Safety backup DAG

Fig. 12. Autoware's Normal DAG and Safety Backup DAG in Our Implementation

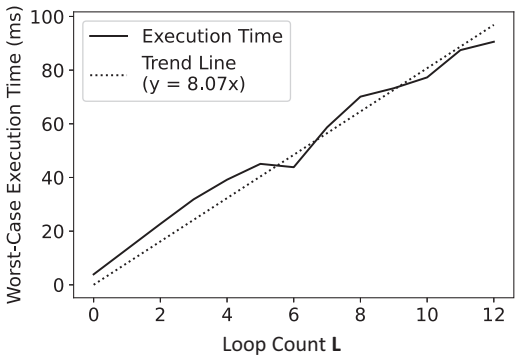


Fig. 13. $e_{s,1}$ of $ndt\_matching$

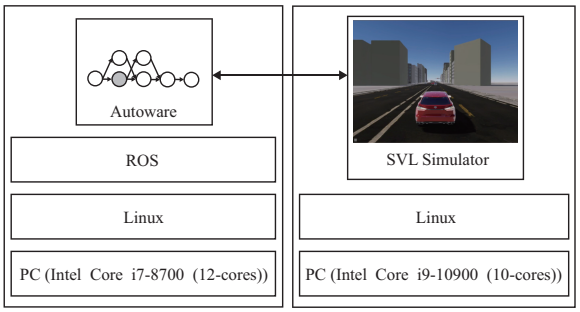


Fig. 14. Implementation Stack

$twist\_filter$, and $twist\_gate$ nodes eventually actuate the acceleration and steering angle to drive the car along the local path.

Fig. 12(b) shows our safety backup DAG for the case where the self-looping node, i.e., $ndt\_matching$, fails in achieving an acceptable accuracy before hitting the time wall. As a safety backup node, we use the $LKAS$ node that runs the Lane-Keeping Assistance System algorithm to find the driving lane in front of the car from the vision image and drive the car keeping the center of lane [20]. The $LKAS$ node replaces the dangling DAG, marked by the dashed box in the normal DAG in Fig. 12(a) until the $ndt\_matching$'s recovery action regains the acceptable accuracy.

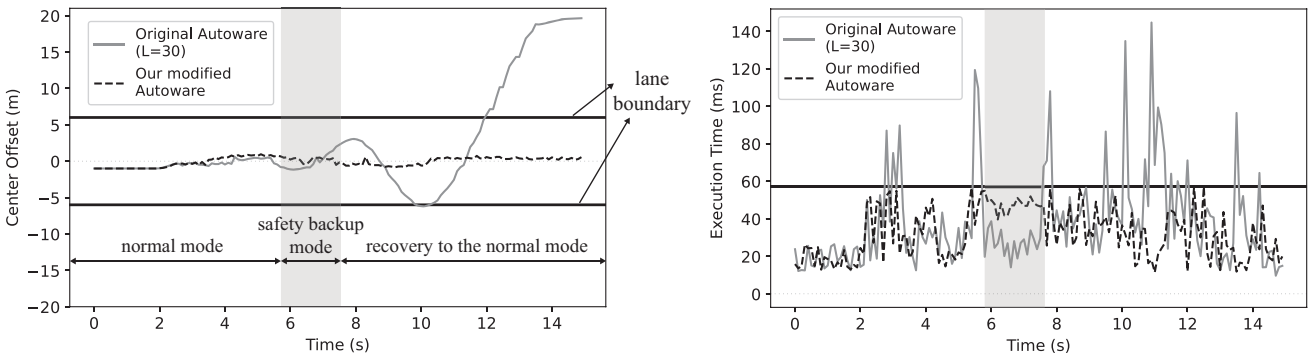
TABLE I
WCET OF NODES

Node name	WCET (ms)
$voxel\_grid\_filter$	0.60
$gnss\_calibrator$	0.28
ray_ground_filter	2.16
$ndt\_matching$	8.07 ($e_{s,1}$)
$lidar\_euclidean\_cluster\_detect$	17.05
$op\_global\_planner$	0.11
imm_ukf_pda	38.13
$op\_motion\_predictor$	5.90
$op\_trajectory\_generator$	1.02
$op\_trajectory\_evaluator$	2.97
$op\_behavior\_selector$	1.30
$pure\_pursuit$	0.90
$twist\_filter$	0.38
$twist\_gate$	0.41
$LKAS$	58.1

Table I shows the measured WCETs of all the nodes. For $ndt\_matching$, the measured WCET e_s can be modeled as a linear function of the loop count L as shown in Fig. 13, that is, $e_s = 8.07 \times L = e_{s,1} \times L$.

Fig. 14 shows our implementation stack where the left Linux PC executes the aforementioned Autoware DAG with $T = D = 125$ ms and the right Linux PC runs a real-time simulated car in the simulated physical environment provided by SVL [21].

Fig. 15 shows the our implementation results. Fig. 15(a) compares the car's center offsets for 14 sec driving by the original Autoware with NDT matching loop count limit of 30 and also by our modified Autoware applying the time wall and the safety backup. At time 5.7 sec, by the original Autoware, the car starts moving far from the lane center and eventually crosses over the lane boundary at 12 sec, which is the critical failure. This is due to a physical error, that makes the $ndt\_matching$ keep looping up to the limit but still resulting in unacceptable accuracy and also violating the deadline. To verify this, the solid line in Fig. 15(b) shows the measured execution time the $ndt\_matching$ of the original Autoware. We can observe occasional large values, especially



(a) Center offset

(b) Execution time of NDT matching

Fig. 15. Implementation Result

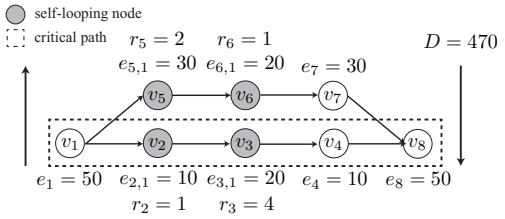


Fig. 16. Task with Multiple Self-Looping Nodes

at time 5.7 sec, which eventually make the $ndt\_matching$ completely lose the location tracking. On the other hand, by our modified Autoware, the car always keeps close to the lane center. Also, the execution time of $ndt\_matching$—dashed line in Fig. 15(b) is always below the time wall, i.e., 56 ms by the CPC based budget analysis. This is thanks to our time wall and safety backup mechanism. Whenever the $ndt\_matching$ hits the time wall with unacceptable accuracy, its output is ignored. Instead, the safety backup, i.e., LKAS, comes in and just keeps the car along the center of the lane until the $ndt\_matching$ rolls back. Fig. 15(a) shows such safety backup duration of [5.7 sec, 7.8 sec] marked as the gray area.

VIII. Idea Sketch for Extending to Multiple Self-Looping Nodes and Multiple DAG Tasks

In this section, we give sketched ideas for extending to multiple self-looping nodes and multiple DAG tasks.

For extending to multiple self-looping nodes, let us consider an example DAG task in Fig. 16 with two paths from v_1 to v_8 and four self-looping nodes, i.e., v_2 and v_3 on the bottom path and v_5 and v_6 on the top path. When applying our aforementioned budget analysis in Sections V and VI, a challenge is that we do not know which path is the critical path since it depends on the loop counts of the self-looping nodes. To address this challenge, we can assume that the nominal ratio of loop counts is given for typical physical situations. For Fig. 16, the nominal loop count ratio $r_2 : r_3 : r_5 : r_6$ of v_2, v_3, v_5, v_6 is given as $1 : 4 : 2 : 1$. Then, we can use a single value L to control their loop counts prorated to the nominal ratio. That is, their loop counts are given as $r_2 L$, $r_3 L$, $r_5 L$, and $r_6 L$. With this assumption, we can say that the path with the largest sum of $e_{s,1} r_s$ of the self-looping nodes becomes the critical one after a certain value of L. For the DAG task in Fig. 16, the bottom path becomes critical after $L = 2$ since $e_{2,1} r_2 + e_{3,1} r_3 = 90$ is larger than $e_{5,1} r_5 + e_{6,1} r_6 = 80$.

Once the critical path is determined, we can use the classic bound equation in Eq. (2) to model the response time bound $R(V)$ as a function of L. For our example in Fig. 16, $R(V) = 110 + 90L + (30 + 80L)/M$. Since the response time bound $R(V)$ should be less than or equal to the deadline D, the maximum L can be calculated as $(D - (110 + \frac{30}{M}))/(90 + \frac{80}{M})$. With such calculated L, now we can assign the time budget of each self-looping node v_s as $e_{s,1} r_s L$.

For CPC based budget analysis, recall that we should calculate three values without non-determinism: (i) e_s^{max} for every self-looping node as in Eq. (7), (ii) $R(V_s)^{init}$ for every self-looping node's segment as in Fig. 8(a) and Eq. (8), and (iii) e_s^{init} for every self-looping node as in Fig. 8(b) and Eq. (9). First, the upper limit e_s^{max} of e_s for every self-looping node v_s can be calculated using the condition that the critical path length should not be longer than the deadline D. For Fig. 16, the sum of e_2 and e_3 should be limited under 360, so that the critical path length should not be longer than the deadline $D = 470$. We can assign this limit 360 to e_2 and e_3 prorating to $e_{2,1} r_2 = 10$ and $e_{3,1} r_3 = 80$ and hence we set e_2^{max} = 40 and e_3^{max} = 320. In this case, L is 4. Using this $L = 4$, the upper limits for e_5 and e_6 are set as $e_5^{max} = e_{5,1} r_5 L = 240$ and $e_6^{max} = e_{6,1} r_6 L = 80$. Second, assuming these e_s^{max} values, we can obtain conservative $R(V_i)^{max}$s for all the subsequent segments $V_i (i > s)$. Then, $R(V_s)^{init}$ can be calculated by subtracting other segments' $R(V_i)^{max} (i > s)$s from the deadline D as in Fig. 8(a). Third, by using the above determined critical path, e.g., the bottom path in Fig. 16, we can construct the CPC structure and classify all the nodes into provider and consumer groups [8] for the segments including self-looping nodes. Then, we can obtain e_s^{init} similarly to the

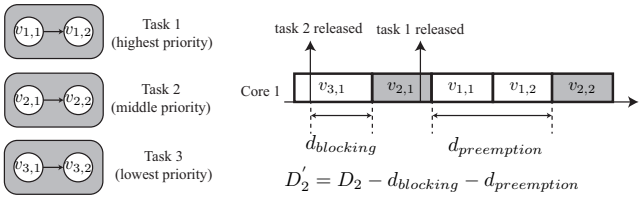


Fig. 17. Multiple DAG Tasks

illustration of Fig. 8(b) and Eq. (9). We can also find the optimal budget by binary search between e_s^{init} and e_s^{max} as depicted in Fig. 9.

For extending to multiple DAG tasks, we can assume a node-level non-preemptive multiple DAG scheduling introduced in [8]. For such scheduling, each DAG task has its own priority and it can experience both blocking delay by a node of a lower priority task and preemption delay by nodes of higher priority tasks. For example, in Fig. 17 assuming a single core, when task 2 is released, the node $v_{3,1}$ of task 3 is running and task 2 waits for $v_{3,1}$ finishes—blocking delay. Then, $v_{2,1}$ starts but task 1 is released during $v_{2,1}$'s execution. Thus, $v_{2,2}$ is delayed until task 1's nodes complete—preemption delay. The equations to compute the maximum blocking delay $d_{blocking}$ and the maximum preemption delay $d_{preemption}$ are given in [8]. Therefore, for each DAG task, by subtracting $d_{preemption}$ and $d_{preemption}$ from the original deadline D, we can compute the effective deadline D', e.g, $D_2' = D_2 - d_{blocking} - d_{preemption}$ in Fig. 17. Using this effective deadline D' instead of the original deadline, we can calculate the time budget of the self-looping node for an individual task with our proposed budget analysis.

IX. Conclusion

In this paper, we propose a novel mechanism for always guaranteeing the deadline while ensuring minimal safety despite physical errors in cyber-physical systems. Our proposed mechanism uses "time wall" and "safety backup" where the time wall bounds the time budget for the self-looping node so as to switch to the safety backup within the deadline to prevent the critical failure due to a physical error. Our experiments through both simulation and actual implementation show that our approach completely prevents critical failure despite physical errors. In future work, we plan to make the sketched ideas concrete for extending to multiple DAG tasks with multiple self-looping nodes.

References

[1] P. Biber and W. Straßer, "The normal distributions transform: A new approach to laser scan matching," in *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)(Cat. No. 03CH37453)*, vol. 3. IEEE, 2003, pp. 2743–2748.

[2] S. Kato *et al.*, "Autoware on board: Enabling autonomous vehicles with embedded systems," in *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 2018, pp. 287–296.

[3] P. J. Besl and N. D. McKay, "Method for registration of 3-d shapes," in *Sensor fusion IV: control paradigms and data structures*, vol. 1611. International Society for Optics and Photonics, 1992, pp. 586–606.

[4] S. Rusinkiewicz and M. Levoy, "Efficient variants of the icp algorithm," in *Proceedings third international conference on 3-D digital imaging and modeling*. IEEE, 2001, pp. 145–152.

[5] J. W. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung, "Imprecise computations," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 83–94, 1994.

[6] L. Mo, A. Kritikakou, and O. Sentieys, "Controllable qos for imprecise computation tasks on dvfs multicores with time and energy constraints," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 8, no. 4, pp. 708–721, 2018.

[7] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969.

[8] S. Zhao, X. Dai, I. Bate, A. Burns, and W. Chang, "Dag scheduling and analysis on multiprocessor systems: Exploitation of parallelism and dependency," in *2020 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2020, pp. 128–140.

[9] Y. Suzuki, T. Azumi, Nobuhiko, Nishio, and S. Kato, "Hlbs: Heterogeneous laxity-based scheduling algorithm for dag-based real-time computing," in *2016 IEEE 4th International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*. IEEE, 2016, pp. 83–88.

[10] Q. He, x. jiang, N. Guan, and Z. Guo, "Intra-task priority assignment in real-time scheduling of dag tasks on multi-cores," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 10, pp. 2283–2295, 2019.

[11] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. C. Buttazzo, "Response-time analysis of conditional dag tasks in multiprocessor systems," in *2015 27th Euromicro Conference on Real-Time Systems*. IEEE, 2015, pp. 211–221.

[12] Z. Dong and C. Liu, "An efficient utilization-based test for scheduling hard real-time sporadic dag task systems on multiprocessors," in *2019 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2019, pp. 181–193.

[13] J. Fonseca, G. Nelissen, and V. Nélis, "Improved response time analysis of sporadic dag tasks for global fp scheduling," in *Proceedings of the 25th international conference on real-time networks and systems*, 2017, pp. 28–37.

[14] F. Guan, J. Qiao, and Y. Han, "Dag-fluid: A real-time scheduling algorithm for dags," *IEEE Transactions on Computers*, vol. 70, no. 3, pp. 471–482, 2020.

[15] M. Qamhieh and S. Midonnet, "Schedulability analysis for directed acyclic graphs on multiprocessor systems at a subtask level," in *Ada-Europe International Conference on Reliable Software Technologies*. Springer, 2014, pp. 119–133.

[16] L. Sha, "Using simplicity to control complexity," *IEEE Software*, vol. 18, no. 4, pp. 20–28, 2001.

[17] S. Z. Bak, *Verifiable COTS-based cyber-physical systems*. University of Illinois at Urbana-Champaign, 2013.

[18] M. Quigley *et al.*, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.

[19] M. Verucchi, M. Theile, M. Caccamo, and M. Bertogna, "Latency-aware generation of single-rate dags from multi-rate task sets," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2020, pp. 226–238.

[20] M. R. Haque, M. M. Islam, K. S. Alam, H. Iqbal, and M. E. Shaik, "A computer vision based lane detection approach," *International Journal of Image, Graphics and Signal Processing*, vol. 12, no. 3, p. 27, 2019.

[21] G. Rong *et al.*, "Lgsvl simulator: A high fidelity simulator for autonomous driving," in *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 2020, pp. 1–6.