공학석사학위논문

Guaranteeing Safety Despite Physical Errors in Cyber-Physical Systems

CPS에서의 물리 에러를 방지하는 안전 보장 메커니즘

2023년 2월

서울대학교 대학원
컴퓨터공학부
한 종 우

Guaranteeing Safety Despite Physical Errors in Cyber-Physical Systems

CPS에서의 물리 에러를 방지하는 안전 보장 메커니즘

지도교수 이 창 건

이 논문을 공학석사 학위논문으로 제출함

2022년 11월

서울대학교 대학원

컴퓨터공학부

한 종 우

한종우의 공학석사 학위논문을 인준함

2022년 12월

위 원 장	하 순 회	(인)
부위원장	이 창 건	(인)
위　　원	김 태 현	(인)

Abstract

Guaranteeing Safety Despite Physical Errors in Cyber-Physical Systems

Jongwoo Han
Department of Computer Science and Engineering
The Graduate School
Seoul National University

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 critical failure in physical error cases.

keywords : Physical Error, Self-Looping Module, Time Wall, Safety Backup

Student Number : 2021-22896

Contents

List of Figures

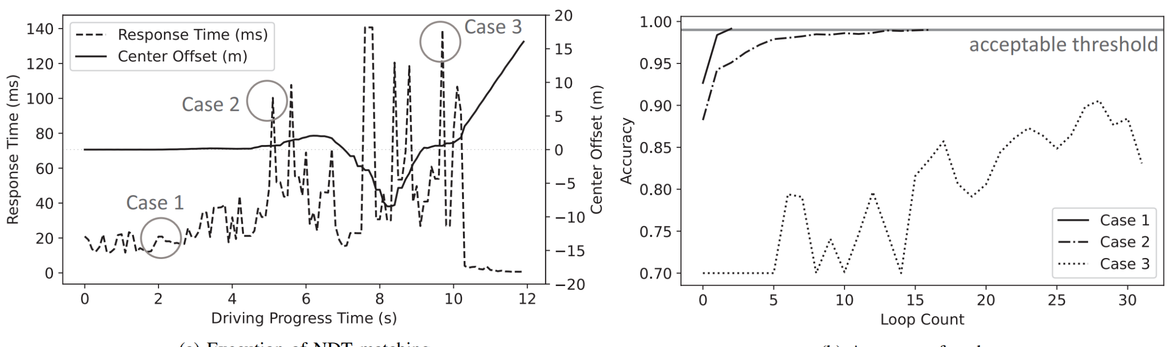
List of Tables

1 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".

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).



(a) Execution of NDT matching

(b) Accuracy of each case

Figure 1: Cases of NDT Matching

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 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.

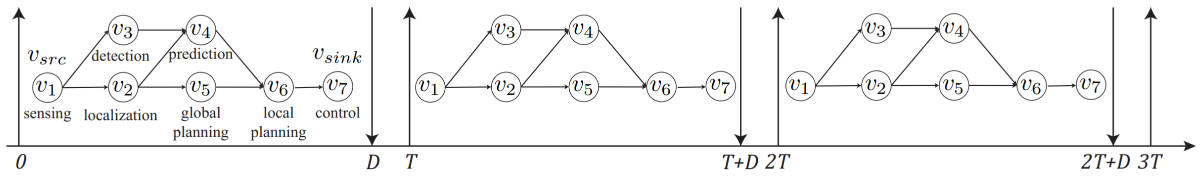
The rest of the paper is organized as follows: Section 2 presents related work. In Section 3, we define the task and resource models and present a motivation example. Section 4 describes our proposed safety guarantee mechanism against physical errors. Then, Section 5 explains a simple budget analysis based on classic bound. Section 6 explains an advanced budget analysis based on CPC method. Section 7 extends the classic bound analysis to model with multiple self-looping nodes. In Section 8, we report our experiment results. Finally, Section 9 concludes the paper and explains future work.

2 Related Work

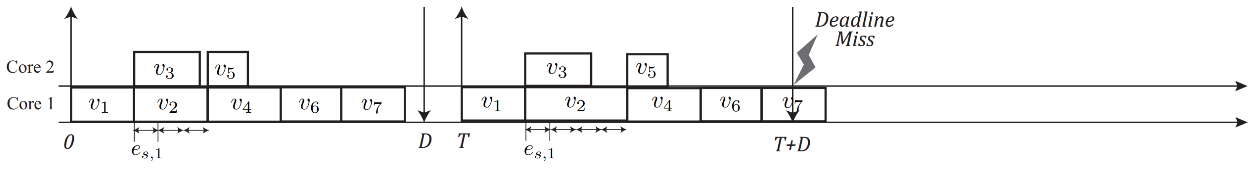
There have been lots of researches on DAG task scheduling on multicore processors. Their objective is reducing the makespan and tightening the worst-case analytical bound [9, 7, 10, 8, 11, 12, 13, 14, 15]. However, all of them assume fixed WCET for every node in the DAG. Thus, they cannot be directly 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.



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



(b) An example schedule on two processors

Figure 2: Autonomous Driving Task Example

3 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 $\mathcal{G}$ 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 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 a given DAG $\mathcal{G} = (V, E)$, we define its total workload $W(V)$ as the sum of WCETs of all the nodes in V, 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 critical nodes, while other nodes are non-critical nodes.

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 scheduling. In other words, we assume that a fixed priority is assigned to each node, and when a processor becomes idle, the highest priority node out of all the ready nodes starts executing on the processor. We allow preemption in Section 5 but disallow preemption for analysis in Section 6 since based analysis needs non-preemptive scheduling. Fig. 2(b) shows an example of such a 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.

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 that makes the inner-loop useless. Extension to multiple self-looping nodes will be explained in Section 7.

4 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 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 $\mathcal{G}_{dangle}$. Note that the lane-keeping module can also fail in more harsh conditions as no lane exists on the road. We can make a level of safety and perform emergency stops when the lane-keeping module fails. For better understanding, this

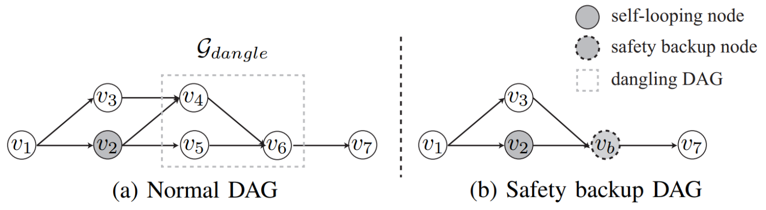


Figure 3: Normal DAG and Safety Backup DAG

paper only introduces the idea just mentioned for the case where v_b fails and assumes that v_b does not fail for all physical situations.

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,

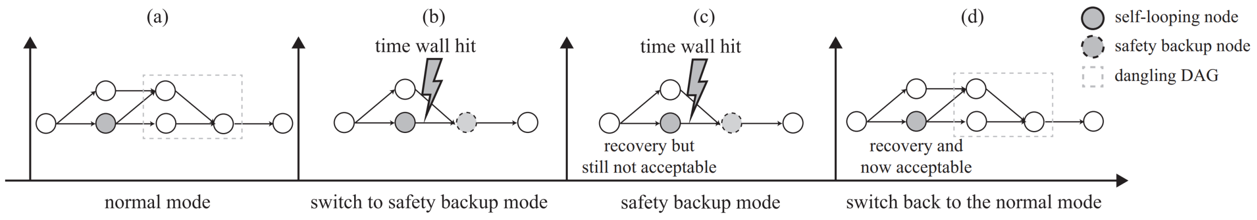


Figure 4: Normal Case and Physical Error Case

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.

- **Safety backup mode**: In the safety backup mode, the self-looping node v_s performs a recovery action[1]. If the recovery action cannot regain 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

[1]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 acceptable accuracy can be regained while the safety backup node is backing up for minimal safety.

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. We can reduce the problem of optimizing the budget assigned to the self-looping node v_s to the problem of determining the largest value we can assign to the WCET parameter of a single node in a given DAG task. We can formally specify the problem as below:

Definition 1 (The MAX-WCET problem)**.** *Given a DAG $\mathcal{G} = (V, E)$, a vertex $v_s \in V$, a number of processors M, and a deadline D, determine the largest value that can be assigned to e_s such that the WCRT bound for $\mathcal{G}$ upon M identical processors does not exceed D.*

As mentioned in the Section 2, there are many studies that find the WCRT bound when all WCETs are fixed. But there is no analysis that finds the maximum WCET when the deadline and remaining nodes' WCET are given like the Max-WCET Problem. Therefore, this paper proposes time budget analyses based on WCRT bound analysis. Section 5 presents an algorithm based on classic bound. Section 6 presents an algorithm based on CPC bound that mitigates pessimism.

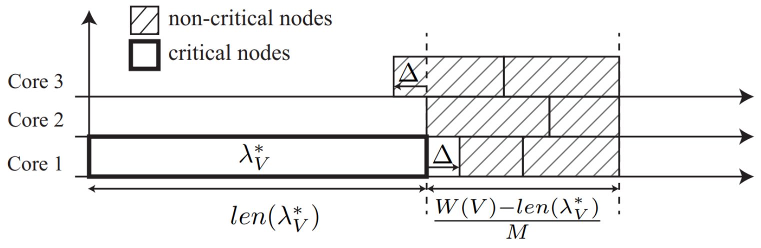


Figure 5: Intuition of Classic Bound

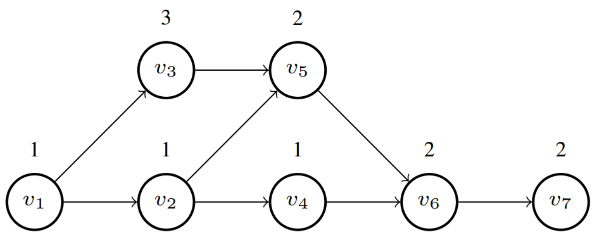
5 Classic Bound based Budget Analysis

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 follows:

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.



v_i	$est(v_i)$	$ltc(v_i)$
v_1	0	9
v_2	1	6
v_3	1	6
v_4	2	4
v_5	4	4
v_6	6	2
v_7	8	0

Some Useful Definitions

$est(v_i)$: earliest start time for v_i
$ltc(v_i)$: least time to completion for v_i

Figure 6: Terminology for graphs

5.1 Some Useful Computations on DAGs

Let us first briefly review some basic graph algorithms, beginning with discussing how the workload $W(V)$ and length of critical path $len(\lambda^*_V)$ may be computed efficiently for any DAG $\mathcal{G} = (V, E)$, in time $O(|V| + |E|)$. It is evident that $W(V)$ can be computed in $\Theta(|V|)$ time by simply summing the WCETs of all the nodes. Below we briefly describe a linear-time algorithm for computing $len(\lambda^*_V)$.

1. Obtain a topological ordering[2] of the nodes in V in $\Theta(|V| + |E|)$ time [21].

2. Compute $est : V \to \mathbb{N}$ (for ***earliest start time***) by setting $est(v_{src}) \leftarrow 0$ for each source node v_{src} and iterating over the remaining nodes in topological order as follows:

$$est(v_i) = \max_{\{v_j \mid (v_j, v_i) \in E\}} \{est(v_j) + e_i\}$$

   (For an example, see Figure 6.)

[2]Recall that a *topological ordering* of the nodes of a DAG is an arrangement of the DAG's nodes in a list, such that each DAG edge is from an earlier node in the list to a later one.

3. Determine $ltc : V \to \mathbb{N}$ (for ***least time to completion***) by setting $ltc(v_{sink}) \leftarrow 0$ for each sink node v_{sink} and iterating over the remaining nodes in reverse topological order:

$$ltc(v_i) = \max_{\{v_j \mid (v_i, v_j) \in E\}} \{ltc(v_j) + e_i\}$$

4. The $len(\lambda_V^*)$ is then computed using either of the following:

$$\begin{aligned} len(\lambda_V^*) &= \max_{\text{all sinks } v_i} \{est(v_i) + e_i\} \\ &= \max_{\text{all sources } v_i} \{ltc(v_i) + e_i\} \end{aligned}$$

We can now state a fairly obvious relationship that can be used to determine whether any node lies on the critical path or not for any $v_o \in V$, the maximum cumulative WCET of any path *that includes node* v_o is computed as follows:

$$L(v_o) = est(v_o) + e_o + ltc(v_o) \tag{2}$$

If $L(v_o) = len(\lambda_V^*)$, we can determine v_o lies on the critical path.

5.2 Simple Solution

With this classic response time bound $R(V)$, our goal is to find the maximum possible time budget e_s for the self-looping node v_s to meet the deadline D. Recall that $len(\lambda_V^*)$ should be the longest path. However, in the case of our system model including a self-looping node v_s, v_s may or may not lie on a critical path depending on the loop count. If we initially set e_s to 0 and calculated, three possible cases arise.

- Case 1) Critical path includes v_s when $e_s = 0$

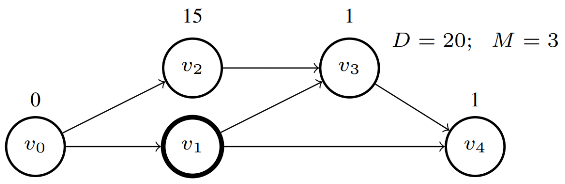


Figure 7: Case where a self-looping node can not become a critical path

- Case 2) When $e_s = 0$, v_s does not lie on a critical path, but if e_s is allocated as much as possible, v_s belongs to a critical path
- Case 3) No matter how many budgets are assigned to e_s, v_s can not become a critical path

For Case 1, the total workload $W(V)$ and the critical path length $len(\lambda_V^*)$ can be rewritten as follows:

$$W(V) = e_s + \sum_{v_j \in (V - \{v_s\})} e_j, \tag{3}$$

$$len(\lambda_V^*) = e_s + \sum_{v_j \in (\lambda_V^* - \{v_s\})} e_j. \tag{4}$$

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:

$$R(V) = e_s + \sum_{v_j \in (\lambda_V^* - \{v_s\})} e_j + \frac{\sum_{v_j \in (V - \lambda_V^*)} e_j}{M}. \tag{5}$$

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:

$$e_s = D - \sum_{v_j \in (\lambda_V^* - \{v_s\})} e_j - \frac{\sum_{v_j \in (V - \lambda_V^*)} e_j}{M}. \tag{6}$$

Algorithm 1 Classic Bound Based Budget Analysis

```
Input: $G = (V, E), M, D$
Output: $e_s$
1: Calculate $L(v_i)$ for every nodes and $len(\lambda_V^*)$
2: if $L(v_s) == len(\lambda_V^*)$ then
3:     Assign $e_s$ by Eq. (6)
4: else
5:     Assign the smaller of the results of Eq. (6) and Eq. (7) to $e_s$
6: end if
```

We can not directly apply Eq. (6) for Case 3. For example, if we assume that v_s always lies on a critical path in Fig. 7, λ_V^* will be $\{v_0, v_1, v_3, v_4\}$ and e_s is calculated as $e_s = 20 - 2 - 15/3 = \mathbf{13}$. However, if $e_s = 13$, WCRT bound is calculated as $R(V) = 17 + 13/3 = \mathbf{21.3}$ which exceeds the specified deadline of 20. For Case 3, $W(V)$ is same as Eq. (3) but $len(\lambda_V^*)$ is length of the longest path, not like Eq. (4). Hence, e_s is calculated as follows.

$$e_s = M \times D - (M-1) \times len(\lambda_V^*) - \sum_{v_j \in (V - \{v_s\})} e_j. \tag{7}$$

With Eq. (7), e_s is calculated as 9 and WCRT bound is calculated as $R(V) = 17 + 9/3 = \mathbf{20}$ which meets deadline.

For Case 2, Eq. (6) gives a safe bound, and its budget is larger than one from Eq. (7). So we can assign the budget calculated from Eq. (6). The detailed algorithm is as follows.

Note that λ_V^* is the longest path **includes** v_s when using Eq. (6). By applying this budget analysis 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.

5.3 Solution with LP

We now briefly describe our alternative method for solving the Max-WCET Problem (Definition 1) by first converting it to a linear program (LP) and then solving the resulting LP via an LP solver. This approach is, in general, computationally less efficient than the algorithm presented in Section 5.2, but it will prove useful to us later in this paper (in Section 7.3) when we generalize the model to allow for multiple self-looping nodes.

Given DAG $\mathcal{G} = (V, E)$ in which the WCET e_i is a variable for a particular specified $v_s \in V$ and given as a constant for the other nodes $v \in V$, we will

- Define two variables W and L to represent the total workload and critical path length of the DAG, and $|V|$ additional variables to represent the earliest start times ($est(v)$) of the nodes $v \in V$.
- For each source node $v \in V$, set $est(v) = 0$
- Write a constraint for each edge $(v_i, v_j) \in E$:

$$est(v_j) \geq est(v_i) + e_i \tag{8}$$

- Write a constraint representing W, the total workload of DAG G, as the sum of the WCETs of all the nodes in V:

$$W = \sum_{v_i \in V} e_i \tag{9}$$

- For each sink node $v_{sink} \in V$, write a constraint representing L, critical path length of DAG G, according to Eq.(2):

$$L \geq est(v_{sink}) + e_{sink} \tag{10}$$

- Write a constraint that the WCRT bound of Eq. (1) should not exceed the deadline parameter D:

$$D \geq \frac{W}{m} + L \times \left(1 - \frac{1}{m}\right) \tag{11}$$

- Set the objective function to be $\big($**maximize** $v_s\big)$

Although it is fairly obvious that this LP does indeed solve the MAX-WCET problem, we briefly explain the reasoning for the sake of completeness. Constraint 11 bounds the values that may be assigned to the variables W and L; since L is thus bounded from above, Constraint 10 bounds the value of $est(v)$ for each sink node v, which, therefore, via multiple instantiations of Constraint 8, bound the value of $est(v)$ for every node v. This in turn bounds the value that can be assigned to e_s, implying that e_s will be assigned the largest value that it can get without the DAG's makespan bound according to Expression 1 exceeding D.

We now illustrate the construction of the LP via an example. For the DAG depicted in Figure 7, let us use the steps described above to write an LP representation of the problem of determining the largest WCET that may be assigned to the node v_1.

- Variables: We would have (i) a variable e_1; (ii) variables W and L; and (iii) variables $est(v_o)$, $est(v_1)$, $est(v_2)$, $est(v_3)$, and $est(v_4)$.

- Constraints:

  $est(v_o) = 0$ (v_o is the sole source node)

  $est(v_2) \geq est(v_o) + 0$ (Constraint for edge (v_o, v_2))

  $est(v_1) \geq est(v_o) + 0$ (Constraint for edge (v_1, v_2))

  $est(v_3) \geq est(v_2) + 15$ (Constraint for edge (v_3, v_2))

  $est(v_3) \geq est(v_1) + e_1$ (Constraint for edge (v_3, v_1))

  $est(v_4) \geq est(v_3) + 1$ (Constraint for edge (v_3, v_4))

  $est(v_4) \geq est(v_1) + e_1$ (Constraint for edge (v_1, v_4))

  $W = 0 + e_1 + 15 + 1 + 1$ (Total workload)

  $L \geq est(v_4) + 1$ (Critical Path Length; v_4 is the sole sink node)

  $20 \geq \frac{W}{3} + L \times \left(1 - \frac{1}{3}\right)$ (Deadline satisfaction)

- Objective function: **maximize** e_1

It may be verified that the optimal solution sets

$$est(v_o) = est(v_1) = est(v_2) = 0, est(v_3) = 15, est(v_4) = 16$$

so that $L = 17$, and $W = 20$ and $e_1 =$ **9** which is indeed the desired value (as we have seen in Section 5.2). □

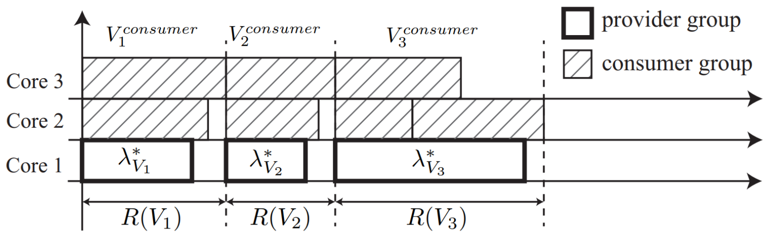


Figure 8: Abstraction of CPC Method

6 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 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]. Since CPC assumes non-preemptive fixed-priority scheduling, we do not allow preemption in CPC based budget analysis. It means once a node starts executing, it continues to the end without being preempted, even if a higher-priority node becomes ready. 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. 8. 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.

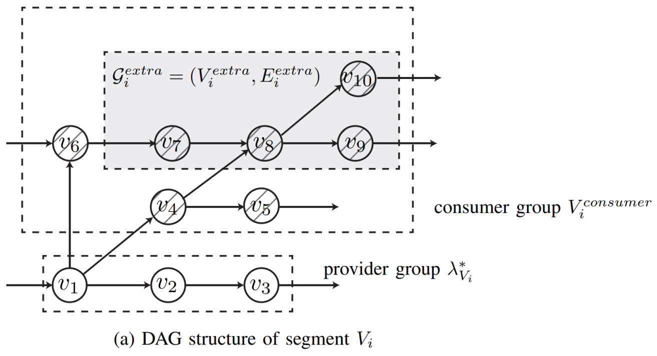
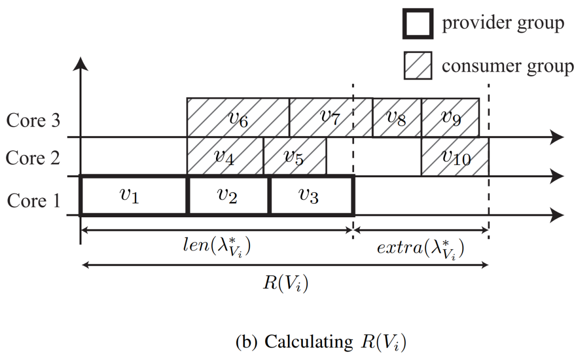

$\mathcal{G}_i^{extra} = (V_i^{extra}, E_i^{extra})$
v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9 v_{10}
consumer group $V_i^{consumer}$
provider group $\lambda^*_{V_i}$
(a) DAG structure of segment V_i


Figure 9: Calculating Response Time $R(V_i)$ of One Segment V_i

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:

$$R(V) = \sum_{i=1}^{n} R(V_i). \tag{12}$$

In order to explain how the CPC method computes $R(V_i)$, let us use the example segment in Fig. 9 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:

$$R(V_i) = len(\lambda^*_{V_i}) + extra(\lambda^*_{V_i}). \tag{13}$$

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. 9(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 $extra(\lambda^*_{V_i})$. We denote the set of such nodes by V_i^{extra}. In the Fig. 9, $V_i^{extra} = \{v_7, v_8, v_9, v_{10}\}$. 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. 9, v_7, v_8, v_9 and v_{10} and their associated edges denoted by E_i^{extra}[3]. As a result, the CPC method computes $extra(\lambda^*_{V_i})$ as follows:

$$extra(\lambda^*_{V_i}) = len(\lambda^*_{V_i^{extra}}) + \frac{W(V_i^{extra}) - len(\lambda^*_{V_i^{extra}})}{M}. \tag{14}$$

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. (14). 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

[3]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].

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 Section 6.1, 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 Section 6.2, we perform a binary search to enlarge the initial budget under the deadline constraint maximally.

6.1 Initial Budget Calculation

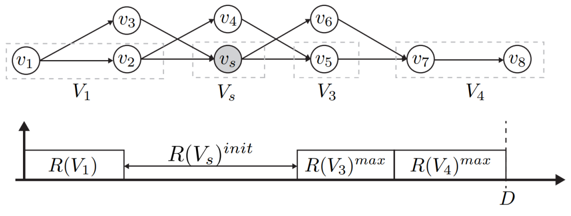
In order to address the dependency of undetermined e_s of the self-looping node v_s and its subsequent segment's response time bounds, we use an upper limit of e_s to compute its subsequent segments' response time bounds conservatively. Such an upper limit of e_s can be given by the fact that the length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

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

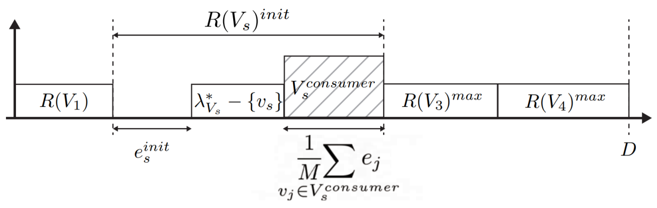
This is a necessary condition for the DAG $\mathcal{G} = (V, E)$ to be completed before D. Using this necessary condition, an upper limit of e_s denoted by e_s^{max} can be given as follows:

$$e_s^{max} = D - \sum_{v_j \in \lambda_V^* - \{v_s\}} e_j. \tag{15}$$

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. 10(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



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



(b) Calculating e_s^{init}

Figure 10: Calculating Initial Budget e_s^{init}

constraint can be given as follows as depicted in Fig. 10(a):

$$R(V_s)^{init} = D - \sum_{i<s} R(V_i) - \sum_{i>s} R(V_i)^{max}. \tag{16}$$

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. (6) to the segment V_s as depicted in Fig. 10(b):

$$e_s^{init} = R(V_s)^{init} - \sum_{v_j \in \lambda^*_{V_s} - \{v_s\}} e_j - \frac{\sum_{v_j \in V_s^{consumer}} e_j}{M}. \tag{17}$$

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.

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. (16). Also, due to Eq. (17), 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. □

6.2 Binary Search for Finding the Optimal Budget

The initial budget e_s^{init} by Eq. (17) 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:

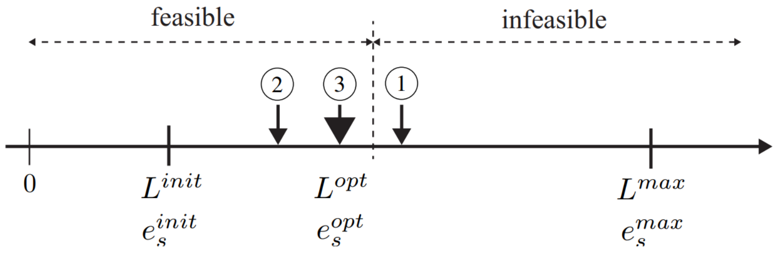


Figure 11: Binary Search for Finding the Optimal Budget

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

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. 11. Algorithm 2 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 use the CPC method, i.e., Eqs. (12), (13) and (14), 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}

Algorithm 2 Optimal Budget Selection Algorithm

```
Input: $\{V_i : 1 \leq i \leq n, V_i = \lambda^*_{V_i} \cup V_i^{consumer}\}, D$
Output: $e_s^{opt}$
    $L^{low} = \max\{0, L^{init}\}$
 2: $L^{high} = L^{max}$
    while $L^{low} < L^{high}$ do
 4:     $L^{mid} = \lfloor (L^{high} + L^{low} + 1)/2 \rfloor$
        $e_s = L^{mid} \times e_{s,1}$
 6:     calculate $R(V_i)(1 \leq i \leq n)$
        $R(V) = \sum_{i=1}^{n} R(V_i)$
 8:     if $R(V) > D$ then
            $L^{high} = L^{mid} - 1$
10:     else
            $L^{low} = L^{mid}$
12:     end if
    end while
14: return $L^{low} \times e_{s,1}$
```

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 the 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 2 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 2 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.

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

1. *CPC method is used for a 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 2, 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. $\square$

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.

7 Generalize to Multiple Self-looping Nodes

In Section 7.1, we examine a different example that brings to light some ambiguities that must be resolved in order to formulate the multiple self-looping nodes problem formally. In Section 7.2, we suggest one reasonable resolution to these ambiguities and propose a formal model accordingly. In Section 7.3, we extend the classical analysis to the multiple self-looping node problem as defined by our proposed formal model.

7.1 Ambiguity in multiple self-looping nodes

If multiple self-looping nodes are specified in a single DAG, then for each, we need also to specify a corresponding dangling DAG (of nodes that do not need to execute upon it flagging a physical error) and a corresponding backup node (that does). Suppose that nodes v_2, v_4, and v_5 in Fig. 2 (reproduced in Fig. 12, bottom right) were all designated to be self-looping nodes, with the following corresponding dangling DAGs and backup nodes:

self-looping node	dangling DAG	back-up node
v_2	$\{v_4, v_5, v_6\}$	$v_{b,2}$
v_4	$\{v_6\}$	$v_{b,4}$
v_5	$\{v_6\}$	$v_{b,5}$

Upon any invocation of this problem instance, different combinations of the self-looping nodes may flag physical errors; hence, several different failure scenarios are possible during any given invocation of the instance:

- No physical errors are flagged (see Fig. 12(a)).

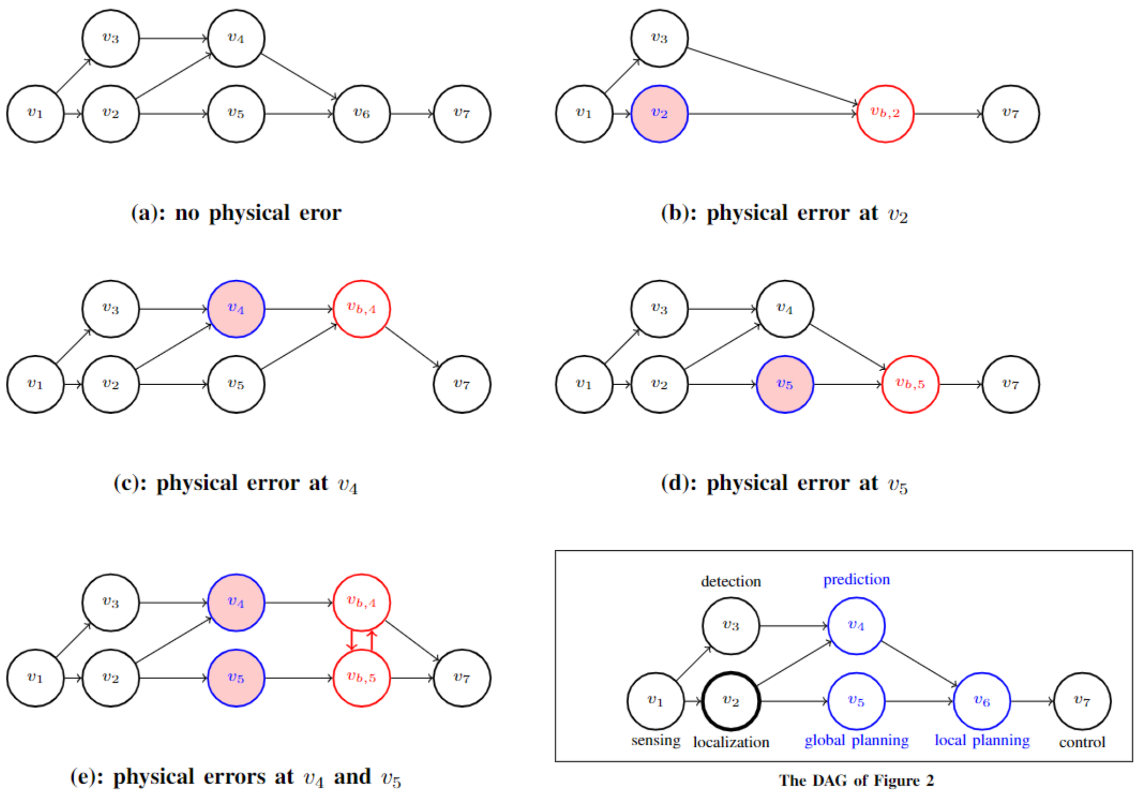

(a): no physical eror
(b): physical error at v_2
(c): physical error at v_4
(d): physical error at v_5
(e): physical errors at v_4 and v_5
detection
prediction
sensing
localization
global planning
local planning
control
The DAG of Figure 2

Figure 12: Ambiguity of Multiple self-looping nodes

- Node v_2 flags a physical error; consequently the self-looping nodes v_4 and v_5, both belonging to the dangling DAG for v_2, do not execute (see Fig. 12(b)).
- One of v_4 or v_5 flags a physical error (see Figs. 12(c)-(d)).
- Nodes v_4 and v_5 both flag physical errors.

This last failure scenario presents a dilemma. As described in Section 3, all incoming edges to the dangling DAG for v_4 (the dangling DAG for v_5, respectively) should now be incident on the backup node $v_{b,4}$ (the backup node $v_{b,5}$, resp.). But this implies that the edge (v_5, v_6) be redirected to $v_{b,4}$, and the edge (v_4, v_6) be redirected to $v_{b,5}$. However, v_4 and v_5 have both failed, and hence the outputs that they were expected to generate are instead generated by $v_{b,4}$ and $v_{b,5}$ respectively; this implies that the redirected edges **form a cycle** between the backup nodes $v_{b,4}$ and $v_{b,5}$! – see Fig. 12(e).

7.2 Multiple Self-Looping Nodes: A Formal Model

We propose a formal model for instances with multiple self-looping nodes that essentially defines away the problem identified in Section 7.1 above, by requiring the system developer to explicitly specify the system behavior that is desired in the event of multiple self-looping nodes flagging physical errors during the same invocation. That is, the system developer is required to specify different **modes** corresponding to combinations of self-looping nodes that may flag physical errors during the same invocation. To do so, the system developer must specify the following.

- $\mathcal{G} = (V, E)$ is a DAG as described in Section 3.
- $V_{self} \subset V$ is *the set of self-looping nodes*.
- A number of *modes* is defined. Each mode is specified as a 3-tuple $(S, V_{<dangle,S>}, v_{b,S})$, where
  1. $S \subset V_{self}$ is a subset of the set of self-looping nodes;
  2. $V_{<dangle,S>} \subset (V \setminus S)$, with the additional constraint that each node in $V_{<dangle,S>}$ is an immediate or transitive successor to each node in S; and
  3. $v_{b,S}$ is a different job (i.e., not in V), with an associated specified WCET $c(v_{b,S})$.

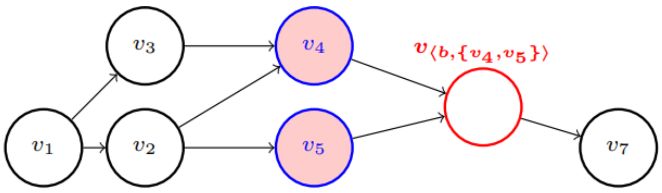


Figure 13: Resolving Ambiguity in Multiple Self-looping nodes

This mode $(S, V_{<dangle,S>}, v_{b,S})$ is defined to occur in an invocation when all the nodes in S flag physical errors during that invocation. In the event of this mode occurring, none of the nodes in $V_{<dangle,S>}$ need to execute; instead, the single job $v_{b,S}$ executes. All incoming edges to nodes in $V_{<dangle,S>}$ become incoming edges to $v_{b,S}$, and all outgoing edges from nodes in $V_{<dangle,S>}$ become outgoing edges from $v_{b,S}$.

(We point out that the non-occurrence of any physical error during an invocation may also be conveniently represented as a mode $(\emptyset, \emptyset, v_{dud})$ where v_{dud} is a dummy node whose WCET is zero.)

The motivating example of Section 7.1 is conveniently represented using the notation introduced above in this section, with V_s, the set of self-looping nodes, equal to $\{v_2, v_4, v_5\}$, and five modes:

1. Mode $(\emptyset, \emptyset, v_{dud})$ – this is the mode that is depicted in Fig. 12 (a).
2. Mode $(\{v_2\}, \{v_4, v_5, v_6\}, v_{b,2})$ – depicted in Fig. 12 (b).
3. Mode $(\{v_4\}, \{v_6\}, v_{b,4})$ – depicted in Fig. 12 (c).
4. Mode $(\{v_5\}, \{v_6\}, v_{b,5})$ – depicted in Fig. 12 (d).
5. Mode $(\{v_4, v_5\}, \{v_6\}, v_{b,\{v_4,v_5\}})$ – this mode is depicted in Fig. 13.

The last mode above is particularly noteworthy – as we saw earlier (Fig. 12 (e)), the behavior of this example system is ambiguously specified when both v_4 and v_5 signal physical errors, without the use of the model we are proposing in this section.

7.3 Multiple self-looping nodes: Computing WCETs

In Sections 5 and 6 we discussed the algorithm for computing the WCET for the self-looping node when there is only one self-looping node present in a DAG. In this section, we will represent the problem of determining the WCETs of multiple self-looping nodes in a single DAG as a Linear Program that generalizes the one in Section 5.3.

Let us suppose that k distinct modes (including $(\emptyset, \emptyset, v_{\text{dud}})$, corresponding to the occurrence of no physical errors) are specified for the instance under consideration. As we saw from our example in Section 7.1 (also see Figs. 12 and 13), the occurrence of each of the k modes may require a different DAG to be executed. We now extend the algorithm presented in Section 5.3 to represent the schedulability conditions for all these DAGs as a single linear program. In this linear program, there will be one variable for each self-looping node to represent its WCET. Then for each of the k DAGs corresponding to the k modes, we will separately

- Have variables to denote its workload W and its critical path length L, and for each node v in this DAG, its earliest start time $est(v)$
- Write constraints upon these variables, as well as the (global) variables representing the WCETs of the self-looping nodes, to denote the constraints on earliest start times imposed by the edges in the DAG (analogous to Eq.(8) for the case of a single self-looping node).

- Write constraints analogous to Eqs. (9), (10), and (11), to represent respectively the workload, the critical path length, and the requirement that the deadline be met.

For example, the instance considered in Section 7.1, could be represented with five modes, the DAGs corresponding to which are depicted in Fig. 12 (a)–(d) and Fig. 13. Note that the DAGs in Fig. 12 (a), Figs. 12 (c)–(d) and Fig. 13 each have 7 nodes and 8 edges, while the DAG in Fig. 12 (b) has 5 nodes and 5 edges.

In representing this instance as an LP, we would therefore have 3 variables to represent the WCETs of the self-looping nodes, $\underline{\text{plus}}$ $(7 + 5 + 7 + 7 + 7 =)$ 33 variables for the $est(\cdot)$'s, $\underline{\text{plus}}$ $(2 \times 5 =)$ 10 W, L variables, for a total of **46** variables.

We can similarly count the number of constraints: there is one per source node (for a total of 5 across all the DAGs) $\underline{\text{plus}}$ $(8 + 5 + 8 + 8 + 8 =)$ 37 instantiations of Eq. (8) corresponding to the edges $\underline{\text{plus}}$ one per DAG (for a total of 5) instantiation of Eq. (10) for each sink node $\underline{\text{plus}}$ 1 per DAG (for a total of 5) instantiation of Eq. (9) $\underline{\text{plus}}$ 1 per DAG (for a total of 5) instantiation of Eq. (11), for a total of **57** constraints.

It is fairly evident that the LP constructed as described above does indeed represent feasible solutions to the problem of assigning WCET values to the self-looping nodes in order to ensure that the WCRT bound of Eq. (1) for the DAG corresponding to each mode does not exceed D. It remains to specify an *objective function* that maximizes the WCET values that are so assigned. This step is necessarily dictated by semantic considerations: what form of optimization does the system designer desire? Generally speaking, a rich set of desiderata are expressible with linear objective functions. For instance, setting the objective function to be the sum of the variables representing the WCETs of the self-looping nodes would dimension the system maximally but may not assign values equitably across the variables. If the objective is to maximize the value assigned to each of them, then one would introduce an additional variable B, add one constraint per self-looping node that its WCET be $\geq B$, and set maximizing B to be the objective function. (One could similarly express a constraint that the WCET of one self-looping node should be twice as large as that of another by making requiring that one of these variables be $\geq 2B$, and so on.)

8 Evaluation

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

8.1 Simulation with Synthetic DAG Workload

In order to show the effectiveness of our proposed mechanism for various DAG workloads, this section conducts a 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 $\frac{e_{avg}\times N}{\rho\times M}$.

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

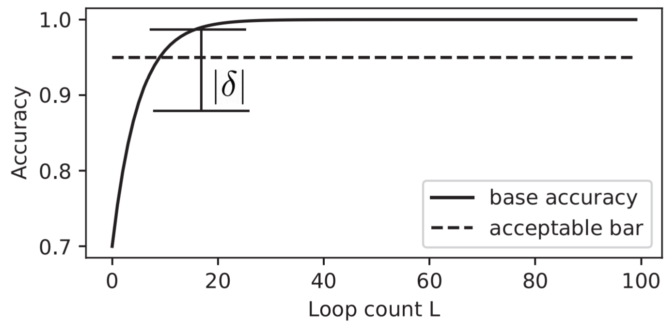


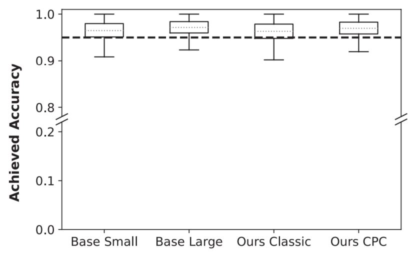
Figure 14: Self-looping Node Model in Synthetic Workload

$\mathcal{G}_{dangle}$'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.

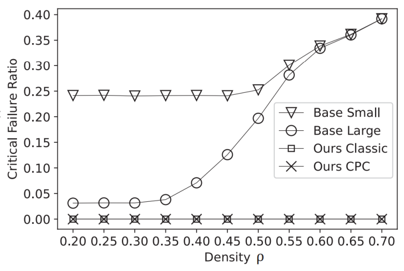
The self-looping node v_s is characterized by the following accuracy function $A(L)$:

$$A(L) = 1 - e^{-L/5+ln0.3} - |\delta|. \tag{18}$$

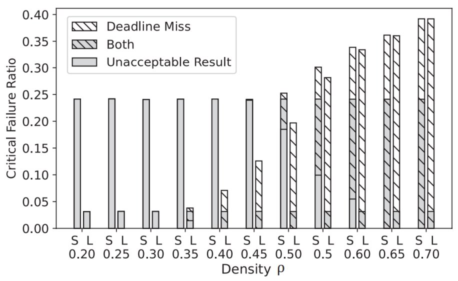
This accuracy function is well illustrated in Fig. 14. In this function, the $1 - e^{-L/5+ln0.3}$ part represents the base accuracy that increases as increasing the loop count L. In addition to this base accuracy, we subtract the $|\delta|$ part to model randomly happening physical errors. δ follows 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

Figure 15: 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 a safety backup. The time wall is based on the classic bound based budget analysis in Section 5.
- Ours CPC: Our proposed mechanism with a safety backup. The time wall is based on the CPC based budget analysis in Section 6.

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. 15(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. 15(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 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. 15(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. 15(c) reports the breakdown of the reasons for the critical failures of Base Small and Base Large.

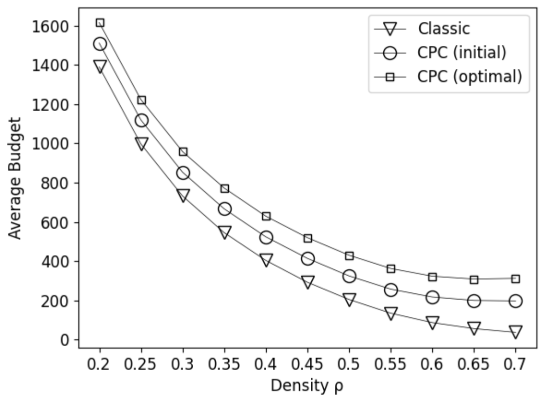


Figure 16: Average Budget for Classic based analysis and CPC based analysis

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 back up within the deadline.

In Fig. 15, Ours CPC uses an optimal budget e_s^{opt} through binary search in Section 6.2. It can allocate more than e_s^{init} but takes more time due to binary search. Therefore, we compare e_s^{init}, and e_s^{opt}, and the budget obtained through Classic bound based analysis. The experimental configurations are the same as in the above experiment, but we did not experiment with multiple instances since the budget value is always the same for the same DAG. Fig. 16 shows the average of the budget obtained by each method for 10000 DAGs according to density. CPC (initial) means e_s^{init} and CPC (optimal) means e_s^{opt}.

Basically, the larger the density, the shorter the deadline, so the budget tends to

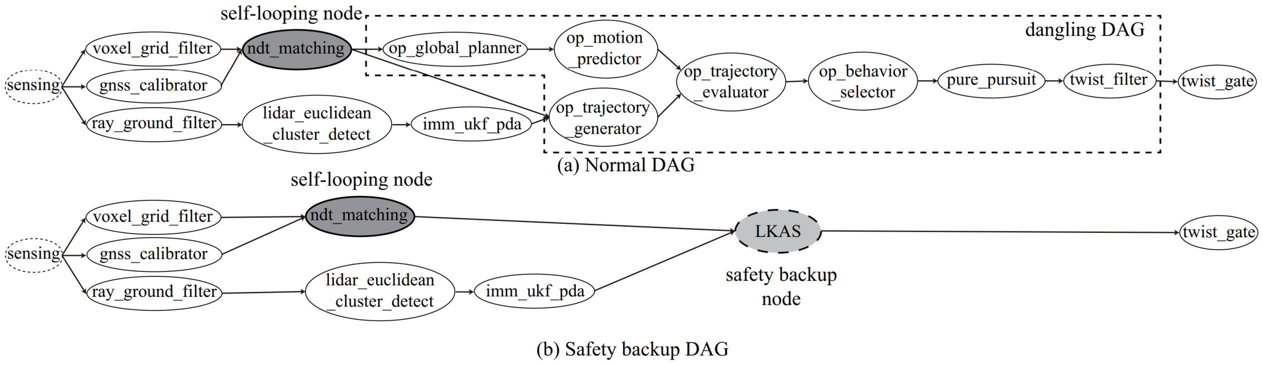


Figure 17: Autoware's Normal DAG and Safety Backup DAG in Our Implementation

calculate smaller. e_s^{opt} is larger than e_s^{init} since it calculates largest budget between e_s^{init} and e_s^{max}. Important point is that e_s^{init} is always larger than the budget obtained by classic bound based analysis. It means that even if e_s^{init} is conservatively obtained, it gives better results than the classic bound since CPC considers parallelism between critical and non-critical nodes. So we can justify the need for CPC based budget analysis.

8.2 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. 17(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

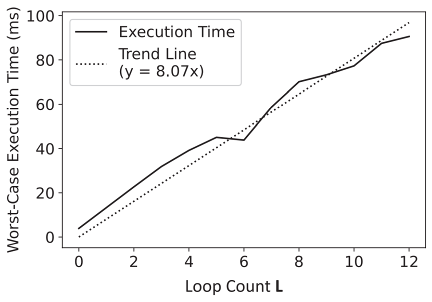
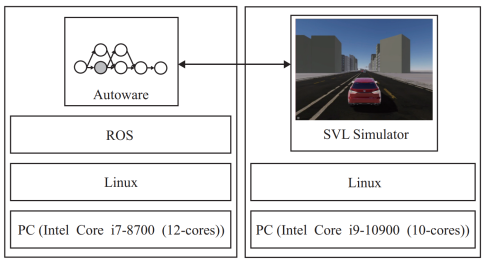


Figure 18: $e_{s,1}$ of $ndt\_matching$



Autoware
ROS
Linux
PC (Intel Core i7-8700 (12-cores))
SVL Simulator
Linux
PC (Intel Core i9-10900 (10-cores))

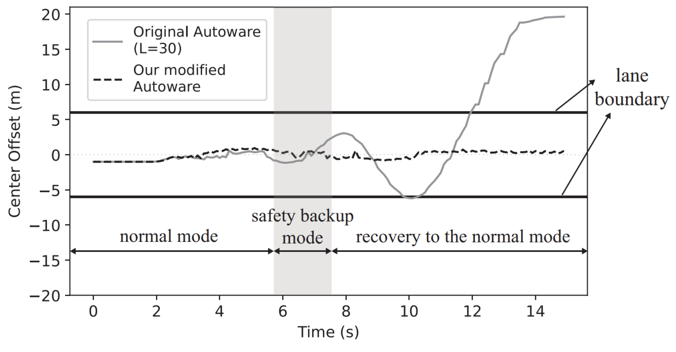
Figure 19: Implementation Stack

global path planning.

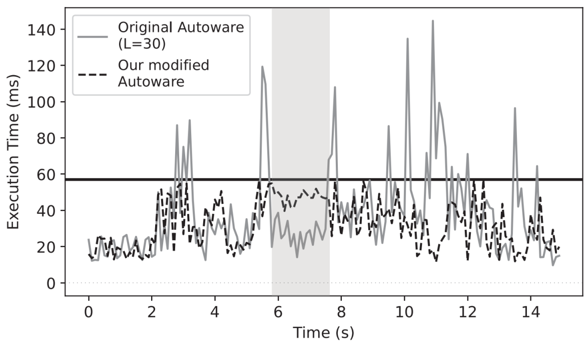
On the other hand, the *ray_ground_filter*, *lidar_euclidean_cluster_detect* and *imm_ukf_pda* nodes detect surrounding obstacles and then the *op_motion_predictor* node predict the obstacles' future behavior relative to the car's current position. Considering the global planning-based trajectory and 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*, *twist_filter*, and *twist_gate* nodes eventually actuate the acceleration and steering angle to drive the car along the local path.

Fig. 17(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. 17(a) until the *ndt_matching*'s recovery action regains the acceptable accuracy.

Table 1 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



(a) Center offset



(b) Execution time of NDT matching

Figure 20: Implementation Result

in Fig. 18, that is, $e_s = 8.07 \times L = e_{s,1} \times L$.

Fig. 19 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 [22].

Fig. 20 shows the our implementation results. Fig. 20(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. 20(b) shows the measured execution time the $ndt\_matching$ of the original Autoware. We can observe occasional large values, especially at time 5.7 sec, which eventually make the $ndt\_matching$ completely lose the location tracking. On the other hand, with our modified Autoware, the car always keeps close to the lane center. Also, the execution time of $ndt\_matching$—dashed line in Fig. 20(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. 20(a) shows such safety backup duration of [5.7 sec, 7.8 sec] marked as the gray area.

9 Conclusion

9.1 Summary

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 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.

9.2 Future Work

In this section, we give sketched ideas for extending to multiple DAG tasks. 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 delays by a node of a lower priority task and preemption delay by nodes of higher priority tasks. For example, in Fig. 21 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, $D'_2 = D_2 - d_{blocking} - d_{preemption}$

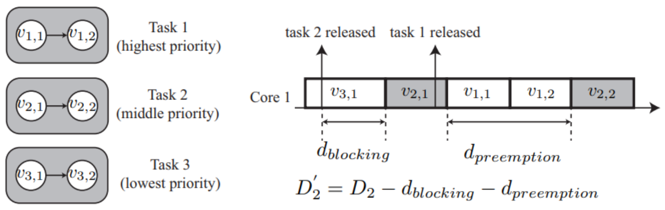


Figure 21: Multiple DAG Tasks

in Fig. 21. 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. Generally, the processor does not just run the DAGs we want. When target DAG is executed together with other processes, we expect that the above idea can be applied by considering each other process as a DAG. In future work, we plan to make this sketched idea concrete for extending to multiple DAG tasks.

References

[1] Peter Biber and Wolfgang 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)*, volume 3, pages 2743–2748. IEEE, 2003.

[2] Shinpei Kato, Shota Tokunaga, Yuya Maruyama, Seiya Maeda, Manato Hirabayashi, Yuki Kitsukawa, Abraham Monrroy, Tomohito Ando, Yusuke Fujii, and Takuya Azumi. Autoware on board: Enabling autonomous vehicles with embedded systems. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pages 287–296. IEEE, 2018.

[3] Paul J Besl and Neil D McKay. Method for registration of 3-d shapes. In *Sensor fusion IV: control paradigms and data structures*, volume 1611, pages 586–606. International Society for Optics and Photonics, 1992.

[4] Szymon Rusinkiewicz and Marc Levoy. Efficient variants of the icp algorithm. In *Proceedings third international conference on 3-D digital imaging and modeling*, pages 145–152. IEEE, 2001.

[5] Jane WS Liu, Wei-Kuan Shih, Kwei-Jay Lin, Riccardo Bettati, and Jen-Yao Chung. Imprecise computations. *Proceedings of the IEEE*, 82(1):83–94, 1994.

[6] Lei Mo, Angeliki Kritikakou, and Olivier 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*, 8(4):708–721, 2018.

[7] Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 17(2):416–429, 1969.

[8] Shuai Zhao, Xiaotian Dai, Iain Bate, Alan Burns, and Wanli Chang. Dag scheduling and analysis on multiprocessor systems: Exploitation of parallelism and dependency. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 128–140. IEEE, 2020.

[9] Yuhei Suzuki, Takuya Azumi, Nobuhiko, Nishio, and Shinpei 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)*, pages 83–88. IEEE, 2016.

[10] Qingqiang He, xu jiang, Nan Guan, and Zhishan Guo. Intra-task priority assignment in real-time scheduling of dag tasks on multi-cores. *IEEE Transactions on Parallel and Distributed Systems*, 30(10):2283–2295, 2019.

[11] Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio C Buttazzo. Response-time analysis of conditional dag tasks in multiprocessor systems. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 211–221. IEEE, 2015.

[12] Zheng Dong and Cong 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)*, pages 181–193. IEEE, 2019.

[13] José Fonseca, Geoffrey Nelissen, and Vincent 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, pages 28–37, 2017.

[14] Fei Guan, Jiaqing Qiao, and Yu Han. Dag-fluid: A real-time scheduling algorithm for dags. *IEEE Transactions on Computers*, 70(3):471–482, 2020.

[15] Manar Qamhieh and Serge Midonnet. Schedulability analysis for directed acyclic graphs on multiprocessor systems at a subtask level. In *Ada-Europe International Conference on Reliable Software Technologies*, pages 119–133. Springer, 2014.

[16] Lui Sha. Using simplicity to control complexity. *IEEE Software*, 18(4):20–28, 2001.

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

[18] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[19] Micaela Verucchi, Mirco Theile, Marco Caccamo, and Marko 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)*, pages 226–238. IEEE, 2020.

[20] Md Rezwanul Haque, Md Milon Islam, Kazi Saeed Alam, Hasib Iqbal, and Md Ebrahim Shaik. A computer vision based lane detection approach. *International Journal of Image, Graphics and Signal Processing*, 12(3):27, 2019.

[21] A. B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5:558–562, 1962.

[22] Guodong Rong, Byung Hyun Shin, Hadi Tabatabaee, Qiang Lu, Steve Lemke, Mārtiņš Možeiko, Eric Boise, Geehoon Uhm, Mark Gerow, Shalin Mehta, et al. Lgsvl simulator: A high fidelity simulator for autonomous driving. In *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*, pages 1–6. IEEE, 2020.

Table 1: 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

요약(국문초록)

본 논문은 물리 상황 인지를 위해 내부 루프를 반복하는 자기반복 모듈(self-looping module)이 있는 사이버 물리 시스템을 고려한다. 자기반복 모듈은 더 높은 정확도를 위해 내부 루프를 반복하지만, 설계 단계에서 고려되지 못한 물리 환경을 마주하게 되면 루프를 반복하더라도 목표 정확도에 도달하지 못하는 물리 에러(physical error) 상황이 발생할 수 있다. 문제는 현재 시스템의 경우 물리 에러 상황에서 자기반복 모듈이 에러를 인지하지 못하기 때문에 계속해서 루프를 반복하게 되고, 데드라인을 놓치는 등 시스템 자체의 치명적인 오류로 이어진다는 것이다. 본 논문에서는 물리 에러 상황에서도 최소한의 안전을 보장하기 위해 ”시간 장벽(time wall)”과 ”안전 백업(safety backup)”을 도입한 새로운 메커니즘을 제안한다. 시간 장벽은 자기반복 모듈의 최대 수행 시간으로, 자기반복 모듈이 시간 장벽만큼 실행했는데도 목표 정확도에 도달하지 못하면 안전 백업 모드로 전환한다. 본 논문은 시뮬레이션과 실제 자율주행 소프트웨어인 Autoware에 제안하는 메커니즘을 적용하여 제안하는 메커니즘이 치명적 오류를 완전히 방지하면서, 실제 자율주행 소프트웨어에도 적용 가능함을 보였다.

주요어 : 물리 에러, 자기반복 모듈, 시간 장벽, 안전 백업
학 번 : 2021-22896