Mitigating Pessimism for Guaranteeing Safety Despite Physical Errors in CPS's

1st Jihwan Kim
Dept. of Computer Science
Seoul National University
Seoul, Korea
jhkim@rubis.snu.ac.kr

2nd Dongmin Shin
Dept. of Computer Science
Seoul National University
Seoul, Korea
dmshin@rubis.snu.ac.kr

3rd Chang-Gun Lee
Dept. of Computer Science
Seoul National University
Seoul, Korea
cglee@rubis.snu.ac.kr

***Abstract*— In this paper, we improve the handling of 'physical error', which is situation where an autonomous mobile CPS fails to accurately pinpoint a location within an acceptable level of accuracy. Analysis in the existing method has a pessimism that can lead to situations that are judged to be physical error even though it is not. Within this paper, we introduce the Interval-Occupancy Model, a novel scheduling and analysis approach designed to alleviate the inherent pessimism observed in the prior method. Our contribution seeks to enhance the discernment of genuine physical errors, ensuring more precise and accurate error identification within autonomous mobile CPS frameworks.**

Keywords— self-looping node, physical error, ideal budget, scheduling model, Interval-Occupancy Model

I. INTRODUCTION

The latest autonomous driving systems employ a real-time system architecture where multiple tasks are interdependent in their execution[1]. This structure is typically represented as a Directed Acyclic Graph (DAG). Among these tasks, some exhibit the characteristic of improving performance with repeated execution. A prime example of this is the NDT matching in the localization phase of autonomous driving systems, where the current position is calculated from sensor data within a map[2][3]. In the paper [4], these time-responsive performance node is defined as a self-looping node, and a situation where a self-looping node fails to reach the target accuracy within a given deadline in a real-time system environment is defined as a physical error. [4] introduced a method to manage physical error situations by imposing a maximum time budget for self-looping nodes: If this budget is exceeded, a safety backup module is activated instead of the self-looping module. The safety backup module is an algorithm with bounded execution time, albeit with lower performance compared to a self-looping node, thus resulting in a reduced quality of outcomes. Therefore, it is necessary to allocate the maximum budget to the self-looping node to avoid the situation where the safe backup module is executed as much as possible.

However, the budget analysis method for self-looping node presented in the paper [4] contains significant pessimism[6]. For example, even given a sufficient number of processors, a budget calculated based on classic bound based analysis in [4] will not allocate the ideal size budget for self-looping node. This pessimistic analysis can lead to under-budgeting of self-looping nodes and misinterpretation of normal situations as physical failure situations. The crucial reason for this pessimism is that the analysis method used in [4] is a general one that does not take into account the situation where the self-looping node needs to be given the maximum budget. In this paper, we solve this problem with a completely novel scheduling model, Interval-Occupancy Model and analysis method

The Interval-Occupancy Model demonstrates the ideal budget by assuming that all other nodes capable of parallel execution run simultaneously on separate cores to eliminate interference. This assumption hinges on the availability of a sufficient number of cores. In this model, we can readily calculate the ideal budget by considering the nodes dependent on the self-looping nodes and their respective deadlines. However, in real-world conditions, an infinite number of cores is not at our disposal, we analysis schedulability of this model by calculating minimum core count required to allocate the ideal budget. For this purpose, we introduce a metric called 'occupancy', which represents the proportion of cores occupied by the execution of each node during a certain time interval. The number of cores required is determined based on the interval at which the sum of occupancies across all nodes peaks over the entire period. Each node is optimized to run at low occupancy for as long as possible to minimize the maximum sum of occupancy and thus the number of cores required.

As a result, we showed that by adding this model and analysis method specialized for self-looping nodes to the existing method allows more budget to be provided under the same conditions as before.

II. TASK AND RESOURCE MODEL

In this paper, we consider a system of periodic single DAG task $\tau = \{T, D, G = (V, E)\}$. Let T be the period of the task and D be the deadline, which means that the DAG is executed every T periods and must end its execution before D. The structure of a DAG task is represented by a graph $G = (V, E)$. G contains nodes represented by $V = \{v_1, \dots, v_n\}$, and nodes have dependencies where one node's execution must follow another node's execution denoted by $E \subseteq (V \times V)$. This workload model is based on prior research [4]. Additionally, we define the set of nodes with no preceding nodes as V_{src} and the set of nodes with no succeding nodes as V_{sink}. We assume that the WCET(Worst Case Execution Time) is given for all nodes v_i in V except self-looping nodes, which is denoted by e_i. A self-looping node v_s is a node that has a variable execution time and produces more accurate results with longer execution time. We consider a computing hardware platform with m identical processors to execute a single DAG job containing only one self-looping node. To determine which node should execute on the m processors when more than m nodes are ready while satisfying all the priority constraints, we

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. RS-2023-00220985, No. 2023R1A2C3003007).

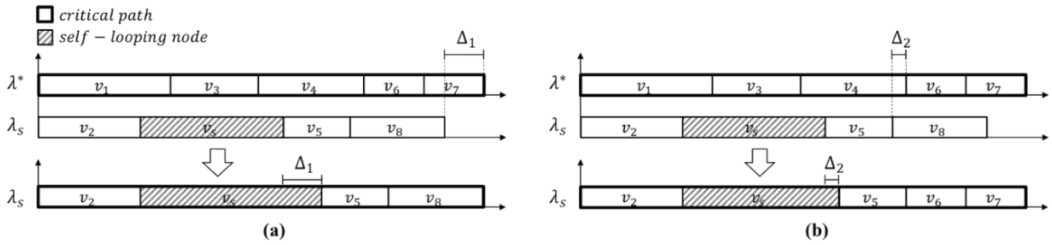

Figure 1. (a) no dependencies between any nodes in $\boldsymbol{\lambda_s}$ and $\boldsymbol{\lambda^*}$
(b) dependency exists between nodes in $\boldsymbol{\lambda_s}$ and $\boldsymbol{\lambda^*}$, $(\boldsymbol{v_5}, \boldsymbol{v_6}) \in \boldsymbol{E}$ in this case

assume fixed-priority non-preemptive scheduling, i.e., each node is assigned a fixed priority as in [5] and when a processor becomes idle, the node with the highest priority among all the ready nodes starts executing on the processor. Once a node starts executing, it will continue to execute until the end without being preempted, even if a node with a higher priority becomes ready.

III. Interval-Occupancy Model and Analysis Method

A. Calculate Ideal Budget

To compute the upper bound of the self-looping node budget, we need to consider the paths that contain the self-looping node. Path λ is an ordered set of nodes $\{v_{src}, \dots, v_{sink}\}$ representing a sequence of nodes. All neighboring nodes in path have a dependency, which is denoted by $\forall v_i, v_{i+1} \in \lambda, (v_i, v_{i+1}) \in E$. The path with the longest length of all paths is called the critical path λ^* (there can be multiple critical paths), and the nodes it contains are called critical nodes, and the others are called non-critical nodes. First, we show that a self-looping node is always in critical path.

Theorem 1. *There is always a critical path that contains a self-looping node.*

Proof. Assuming that there is no critical path contains a self-looping node, let λ_s be the path that contains a self-looping node. Intuitively, self-looping nodes have variable execution time, and longer execution time means better performance, so if they are not on a critical path, they can be given additional execution time to get on a critical path. However, there can be dependency between nodes, so let's look at two cases.

a) If there is no dependency issue between nodes in λ_s and λ^* as in Figure 1(a), we can simply increase the budget of the self-looping node by Δ_1 so that the length of λ_s is the same as the length of λ^*,. Thus, λ_s becomes the longest path (one of them) and becomes a critical path.

b) If dependency issue exists between nodes in λ_s and λ^*, simply increasing the budget by Δ_1 will violate the dependency. This is the case when there is node executes after the self-looping node on path λ_s must precede the node on the critical path λ^*, $(v_5, v_6) \in E$ for example in Figure 1(b). In this case, increase the budget as much as possible without violating the dependency. Through this change, the path before the node with a dependency problem on the path λ_s and the path after the node with a dependency problem on the path λ^* are combined to create the new longest path path (one of them) and becomes a critical path. □

Thus, we can see that there must be a critical path that contains a self-looping node. With this theorem, the ideal budget of a self-looping node e_s can be found simply by subtracting the WCET of all nodes except the self-looping node from the critical path containing the self-looping node at deadline.

$$e_s = D - \sum_{v_i \in (\lambda^* - v_s)} e(v_i)$$

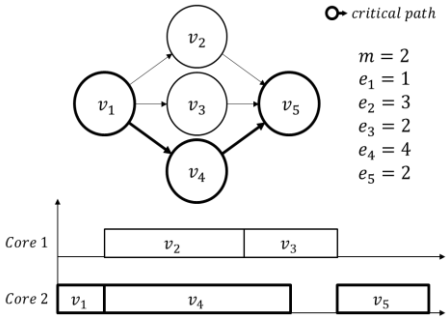

Figure 2. Interference by non-critical nodes when the number of cores is insufficient

To ensure the ideal budget is allocated, Conditions regarding the number of cores are required. Figure 2 shows a simple five-node DAG task to illustrate the consequences of insufficient number of core. The sum of the WCETs of the nodes in the critical path $\lambda^* = \{v_1, v_4, v_5\}$ amounts 7. However, as cores are limited to $m = 2$, non-critical nodes can cause interference, resulting in a prolonged execution time to 8. Therefore, to ensure that the response time of the DAG task does not exceed the length of the critical path due to interference of non-critical nodes, it is necessary to assume a sufficiently large number of cores. In most situations, we have a limited number of cores for scheduling, so we need to check how many cores are actually used for scheduling after finding the ideal budget under the above assumptions.

We present a novel model that uses the ideal budget obtained above to schedule the remaining nodes on the least number of cores. The goal of the model is to split the execution of a node over as long a period of time as possible to reduce the number of cores required.

B. Occupancy

Occupancy $O(v, i)$ is the core concept of the Interval-Occupancy Model. After specifying an artificial release time and deadline for a node, it indicates how much of the core will be occupied by the node if it runs throughout the interval. The concept of "running throughout the interval" is illustrated in Figure 3. Let v_a's release time be $t^r_{v_a}$, deadline be $t^d_{v_a}$. Rather than simply executing from $t^r_{v_a}$ to $\left(t^r_{v_a} + e_a\right)$, it is considered to be executed by occupying only $O(v_a, i_a)$ portion of cores from $t^r_{v_a}$ to $t^d_{v_a}$ (i_a refers to the target time interval of v_a, the specific concept will be explained later). The formula can be represented as follows:

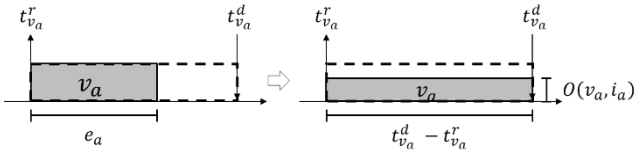

Figure 3. Running node v_a throughout the interval

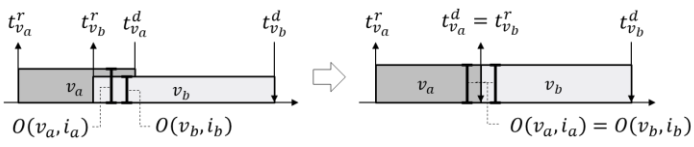

Figure 4. Resolving dependency issue

$$O(v_a, i_a) = \frac{e_a}{t^d_{v_a} - t^r_{v_a}}$$

So the amount of execution of this node on the core is $(t^d_{v_a} - t^r_{v_a}) \times O(v_a, i_a) = e_a$, which is the same as before. To determine the minimum number of cores required for scheduling, sum Occupancy of all nodes that are executable within a specific interval and find the smallest integer greater than this sum. This process can be applied to the entire interval, and the highest value among these minimum core requirements indicates the necessary core count for scheduling this DAG task.

C. Determine artificial release time, deadline

To calculate the occupancy, we need to find the artificial release time, deadline of each node. Since our goal is to distribute the occupancy as much as possible to minimize the sum of occupancy in each interval, the release time should be as early as possible and the deadline as late as possible. We define lst (for latest start time of node) and eft (for earliest finish time of node) to calculate this considering the dependency of the node.

a) $lst : V \to \mathbb{N}$

If $v \in V_{src}$, the release time is 0 because it can be started immediately when the task releases. If a predecessor node of v exists, v can be executed after the execution of the preceding node has finished, so for all predecessor nodes, the largest value of the sum of the lst of each predecessor node and e becomes the lst of v. This definition is expressed by the following formula:

$$lst(v) = \begin{cases} 0, & if\ v \in V_{src} \\ \max\limits_{\{u|(u,v)\in E\}} \{lst(u) + e_u\}, & otherwize \end{cases}$$

b) $eft : V \to \mathbb{N}$

eft is a concept that is symmetrical to lst. If $v \in V_{sink}$, deadline is the same as D (i. e. , $eft = 0$). If there is a successor node to v, v must finish earlier than D by the sum of e and eft of successor node. This definition is expressed by the following formula:

$$eft(v) = \begin{cases} 0, & if\ v \in V_{sink} \\ \max\limits_{\{u|(v,u)\in E\}} \{eft(u) + e_u\}, & otherwize \end{cases}$$

Intuitively, we can understand lst as the length of the critical path of a subgraph consisting of nodes that must be executed before node v, and eft as the length of the critical path of a subgraph consisting of nodes that can be executed only after node v is executed. Since we have assumed a sufficient number of cores, there is no interference from non-critical node of that subgraph. Therefore, the maximum window of time that a node v can run is between $lst(v)$ and $D - eft(v)$. Based on this analysis, t^r_v and t^d_v can be set to lst and $(D - eft)$, respectively.

However, the problem arises when we actually execute these nodes with occupancy from $lst(v)$ to $eft(v)$, which is the case in Figure 4. In Figure 4, v_a and v_b have a dependency but $t^d_{v_a}$ is later than $t^r_{v_b}$. Since v_b must be executed after v_a is ended, we need to set an appropriate border between $t^d_{v_a}$ and $t^r_{v_b}$, and set $t^d_{v_a}{}'$ and $t^r_{v_b}{}'$(where $t^d_{v_a}{}'$ and $t^r_{v_b}{}'$ represent the new values of $t^d_{v_a}$ and $t^r_{v_b}$) to that border. The location of border is the point where the Occupancy of each node is the same after adjusting $t^d_{v_a}$ and $t^r_{v_b}$. Since $O(v_a, i_a) = e_a / (t^d_{v_a} - t^r_{v_a})$ and $O(v_b, i_b) = e_b/(t^d_{v_b} - t^r_{v_b})$, border can be calculated by the following formula.

$$\forall (v_a, v_b) \in E,\ t^d_{v_a} > t^r_{v_b} \to t^d_{v_a}{}' = t^r_{v_b}{}' = \frac{t^d_{v_b} e_a + t^r_{v_a} e_b}{e_a + e_b}$$

Our objective is to minimize the total occupancy values across all sections. In this scenario, when the occupancy of one node increases, the occupancy of the other node decreases. Enforcing equal occupancy values for both nodes is a policy that minimizes the maximum value of the total occupancy sum.

D. Analysis Method

We have determined the artificial release time, deadline, and thus the occupancy of each node. Finally, we need to find the point in time where the sum of the occupancies of the nodes can execute that point is the highest, but since we can't check all of them for the entire T, we divide the period T into intervals. The interval $I = \{i_1, \ldots, i_n\}$ is defined by dividing T with $t^r_{v_j}, t^d_{v_j}\ \forall\ v_j \in V$. The occupancy of each node remains constant within the interval between the artificial release time and the deadline, so $O(v_j, i_n)$ does not change within each interval. Therefore, we only need to calculate the occupancy sum once per interval, which is expressed by the following formula.

$$Required\ number\ of\ cores = \left\lceil \max_{i_n \in I} \sum_{v_j \in V} O(v_j,\ i_n) \right\rceil$$

E. Scheduling Method

In this section, we introduce a priority assignment scheme for scheduling on m cores when the sum of the occupancy of nodes in interval i_n is not greater than m, and show that it is feasible.

First, Divide the workload of each node proportionally to the size of the interval and assign it to each interval, i.e., we distribute the workload so that it has the same occupancy in all intervals. In Figure 5, the interval with the largest

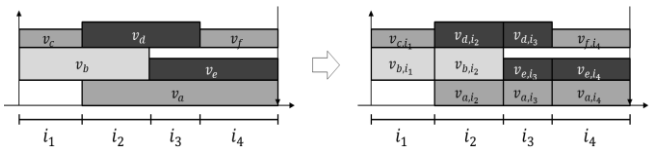

Figure 5. Slicing each node's workload by interval

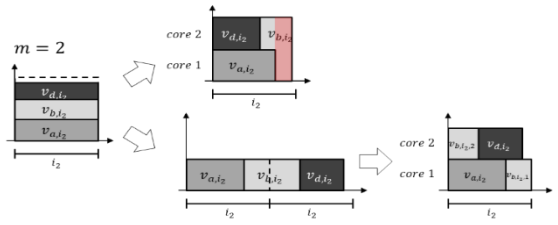

Figure 6. Scheduling in interval:
Concatenate all workload and slice by interval size

occupancy sum is i_2. The workload assigned to interval i_2 is $\{v_{d,i_2}, v_{b,i_2}, v_{a,i_2}\}$, and the number of cores required is $m = \lceil O(v_{d,i_2}, i_2) + O(v_{b,i_2}, i_2) + O(v_{a,i_2}, i_2) \rceil$. This allows each interval to be scheduled independently, and the each workload is modeled to run with the same release time and deadline on m cores. From this, we propose the first rule for priority assignment.

Rule 1. $\forall i_n,\ i_m \in I, n < m,\ \forall v_a \in V_n, v_b \in V_m \rightarrow p_a > p_b$

The next step is how to run the nodes within the interval. Since there is no guarantee that the workloads of the nodes can be executed in parallel, the situation indicated by the upward arrow in Figure 6 can cause problems with parallel execution. To avoid this situation, we introduce the following method. First, concatenate the WCETs of all nodes in a sequence, and then divide them based on interval size. Afterward, execute each piece(size of each piece is equal to the interval except last piece) by assigning it to a core. Some nodes may be sliced ($v_{b,i_2,1}$ and $v_{b,i_2,2}$ in Figure 6) and some may not(v_{a,i_2}, v_{d,i_2} in Figure 6). To prevent parallel execution, the later part of the sliced node should release after WCET of the earlier part, i.e. $v_{b,i_2,1}$ should released at $e_{b,i_2,2}$ after i_2 starts. Finally, we give them priority from the front. In Figure 6, $v_{b,i_2,2}$ and v_{a,i_2} have the highest priority, followed by v_{d,i_2}, and then $v_{b,i_2,1}$. This avoids running nodes in parallel because the size of the node's workload cannot exceed the size of interval (Occupancy will never be greater than 1). Scheduling like this allows us to schedule nodes without parallel executing them, even in situations where the sum of Occupancy equals the number of cores, i.e., a density of 1. Let $V_n{}'$ be the set of nodes after concatenating the nodes in V_n and slicing them to the size of i_n, The rule that sums up this process follows.

Rule 2. $\forall i_n \in I, \forall v_a, v_b \in V_n',$

$$\left\lfloor \sum_{k=1}^{a-1} e_k \right\rfloor \% \ size(i_n) < \left\lfloor \sum_{k=1}^{b-1} e_k \right\rfloor \% \ size(i_n) \rightarrow p_a > p_b$$

F. Merge with previous method

Interval-Occupancy Model is specialized for DAG structures including self-looping nodes and shows strong performance in most cases, but it can produce worse results than existing methods for certain malformed DAGs. For example, in the case of a DAG with a very large number of nodes with the same preceding and succeeding node conditions, the occupancy value of the corresponding interval increases and the number of required cores increases, resulting in a failure decision. A simple way to solve this is to try both the previous method and ours. First, the number of required cores obtained using the Interval-Occupancy model is compared with the number of available cores to determine whether an ideal budget can be provided, and then, if possible, provide an ideal budget to the self-looping node, and if not possible, use the previous method to see if it can provide a lower budget.

IV. Evaluation

This section evaluates the proposed algorithm compare to the previous method by synthetic task.

A. Compare to previous method in [4]

We conducted a comparative evaluation of our method against the previous method in [4] on randomly generated identical DAG with 4 identical homogeneous cores ($m = 4$). The DAGs are generated under the following conditions: the number of nodes *N* is randomly chosen from uniform(15, 25), with self-looping nodes excluded; WCET of non-self-looping nodes are randomly chosen from uniform(30, 50); the length of the critical path is randomly chosen from uniform(6, 10). Edges between nodes were established by setting random pairs of nodes with varying numbers of edges. On average, each node has 3 dependent nodes.

In these experiments, we varied the DAG's utilization *U* from 0 to 4 in increments of 0.2. The DAG's period (which is equivalent to the deadline) was determined based on the average execution time e_{avg} of the generated DAG (excluding self-looping nodes) and the utilization value, as follows:

$$T = D = \frac{e_{avg} \times N}{U}$$

For each utilization value, we conducted experiments with 100,000 DAGs. In the previous method in [4], a schedule was considered successful if the computed budget for self-looping nodes was greater than or equal to zero. In our method, a schedule was considered successful if the computed required number of cores was four or fewer. The results are shown in Figure 7. In terms of schedulability, our method exhibits a reasonable success rate even for high utilizations. This can be attributed to the fact that the previous method, which employs classic bound-based analysis with strong pessimism, often incorrectly concludes that self-looping nodes cannot be allocated a budget.

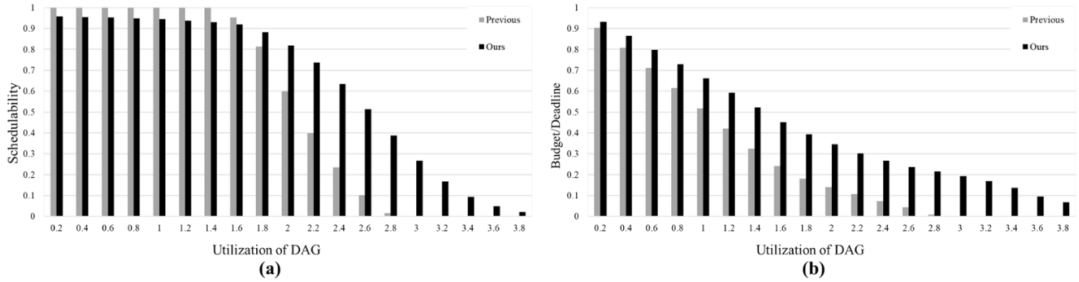

Figure 7. (a) Comparing schedulability based on utilization
(b) Comparing budget based on utilization

In contrast, our method mitigates pessimism to a significant extent. This is particularly noticeable in the utilization range of 2.6 to 3.0, where our method achieves a success rate of 30% ~ 50%, while the previous method shows less than 10%. However, in scenarios with low utilization, our method occasionally fails (approximately 5%). This typically occurs when randomly generated DAGs exhibit extremely high parallelism. In such cases, a slight reduction in the budget could allow scheduling within given cores, but our method keeps the budget fixed at its maximum value, leading to this issue. To address this, we propose a system that employs both methods. If our method succeeds in scheduling, it provides the ideal budget. If it fails, previous method is used to calculate budget. Our method consistently provides larger budgets than the previous method, ensuring that giving self-looping nodes maximum execution time budget to reach the target accuracy and thereby enhancing the overall system stability.

Figure 7(b) illustrates how much budget was allocated on average for successful budget calculations, based on utilization. In consideration of varying deadlines and workloads assigned to each DAG, the y-axis has been normalized as budget/deadline. In contrast to previous method, our method demonstrates a distinct advantage by providing an ideal budget allocation that remains stable even as utilization increases. This stands in contrast to the rapid reduction in budget size observed in previous methods. This substantial increase in budget allocation compared to the previous method ensures that self-looping nodes have sufficient execution time to reach the target accuracy, thereby stabilizing the entire system.

B. Comparison based on DAG generation conditions

The following experiment compares the budget calculation as the DAG's generation conditions change for two factors: Parallelism, Dependency.

First, we conducted an experiment to control the parallelism by setting the average length of critical paths to 6, 7, 8, 9, and 10 (the rest of the conditions are the same as the above experiment, and Utilization is set to 2). The result at Figure 8(a) shows that the budget calculated by our method decreases as the critical path becomes longer, i.e., the parallelism decreases. This is because the budget is calculated by subtracting the size of the critical path excluding self-looping nodes from the deadline. Meanwhile, schedulablity has a maximum value at a certain length(8 at this condition) and decreases when it is less or more. First, if the critical path is too short, the schedulability decreases because the parallelism of the nodes increases and the occupancy value increases, causing more cases of scheduling failure. On the other hand, if the critical path is too long, the budget of the self-looping node decreases, which affects the artificial release time and deadline. When the budget of the self-looping node is much larger than other nodes, the occupancy value of the nodes that can run in parallel with the self-looping node is calculated very low. But when the budget is reduced to be similar to other nodes, the occupancy value of the section that runs in parallel with the self-looping node is increased, and the sum of occupancy increases.

In the second set of experiments, we adjusted the dependencies by increasing the average number of other nodes

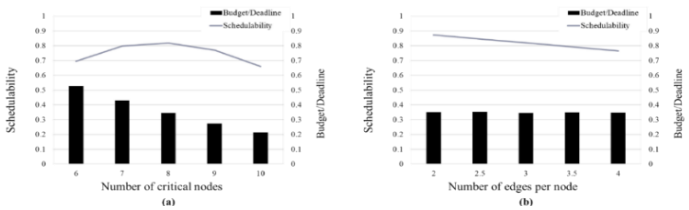

Figure 8. (a) Variations in schedulability and budget with changing levels of parallelism
(b) Variations in schedulability and budget with changing levels of dependencies

each node depends on from 2 to 4 in increments of 0.5, while keeping the other conditions the same as the previous experiments. The results at Figure 8(b) indicate that as the dependencies increase, the schedulability calculated using our method decreases. This is because considering inter-node dependencies leads to narrower intervals between artificial release times and deadlines, which, in turn, increases occupancy in each interval and subsequently results in a higher required maximum number of cores, leading to scheduling failures. Meanwhile, dependency is totally independent of the budget, as the budget is solely determined by the critical path.

V. CONCLUSION

In this paper, our paper present a new scheduling and analysis method, the Interval-Occupancy Model, which mitigating the pessimism present in previous papers. The model allocates an ideal budget to self-looping nodes and optimally schedules other nodes, minimizing the number of misjudgments due to physical error situations and improving the schedulability. Looking ahead, we plan to extend our study to a wider range of DAGs, i.e., structures with two or more self-loop nodes, or general structures that do not contain self-loop nodes.

REFERENCES

[1] 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..

[2] 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.I. S. Jacobs and C. P. Bean, “Fine particles, thin films and exchange anisotropy,” in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[3] 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.R. Nicole, “Title of paper with only first word capitalized,” J. Name Stand. Abbrev., in press.

[4] J. Han, S. Park, H. Jeon and C. -G. Lee, "Guaranteeing Safety Despite Physical Errors in Cyber-Physical Systems," 2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS), Milano, Italy, 2022, pp. 1-12, doi: 10.1109/RTAS54340.2022.00009.

[5] 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.

[6] J. Han, C. -G. Lee and S. Baruah, "Improved Results for Guaranteeing Safety Despite Physical Errors in CPS's," 2022 IEEE Real-Time Systems Symposium (RTSS), Houston, TX, USA, 2022, pp. 266-276, doi: 10.1109/RTSS55097.2022.0003