Optimal Checkpointing Strategy for Real-time Systems with Both Logical and Timing Correctness

LIN ZHANG*, Syracuse University, USA
ZIFAN WANG*, Syracuse University, USA
FANXIN KONG, Syracuse University, USA

Real-time systems are susceptible to adversarial factors such as faults and attacks, leading to severe consequences. This paper presents an optimal checkpoint scheme to bolster fault resilience in real-time systems, addressing both logical consistency and timing correctness. First, we partition message-passing processes into a directed acyclic graph (DAG) based on their dependencies, ensuring checkpoint logical consistency. Then, we identify the DAG's critical path, representing the longest sequential path, and analyze the optimal checkpoint strategy along this path to minimize overall execution time, including checkpointing overhead. Upon fault detection, the system rolls back to the nearest valid checkpoints for recovery. Our algorithm derives the optimal checkpoint count and intervals, and we evaluate its performance through extensive simulations and a case study. Results show a 99.97% and 67.86% reduction in execution time compared to checkpoint-free systems in simulations and the case study, respectively. Moreover, our proposed strategy outperforms prior work and baseline methods, increasing deadline achievement rates by 31.41% and 2.92% for small-scale tasks and 78.53% and 4.15% for large-scale tasks.

CCS Concepts: • **Computer systems organization** → *Real-time system specification*; **Reliability**; • **Security and privacy** → **Security services**.

Additional Key Words and Phrases: Real-time systems, fault resilience, checkpointing, logical consistency, timing correctness

1 INTRODUCTION

As real-time systems such as automobiles become more complex and open architectures, they are vulnerable to many adversarial factors such as faults and attacks [2, 12, 40, 46]. With these adversaries, the controller may make dangerous decisions and cause serious consequences such as vehicle crashes and loss of human lives [1, 8, 21, 39, 44]. Resilience to such adversarial factors is essential to the safety of such systems[5, 9, 16].

In this paper, we study the problem of tolerating transient faults for a controller executing in computational nodes. In general, there are two popular research threads for fault resilience: redundancy and checkpointing. One thread relies on redundant components (e.g., standby processors [13, 19, 31, 33] or task replica [20, 24, 32, 42]), where if some components are faulty, other components can still process forward to finish the job. The other thread occasionally checkpoints system states, and the system rolls back to a consistent state (checkpointed in history) when detecting faults [14, 15, 17, 28]. This work aligns with the second thread and studies checkpointing protocols for real-time parallel processes.

Existing checkpointing works can be divided into two groups. One group focuses on checkpointing computing tasks in general-purpose (non-real-time) systems. The goal is to guarantee the logical consistency of checkpoints

*Both authors contributed equally to the paper

Authors' addresses: Lin Zhang, lzhan120@syr.edu, Syracuse University, Syracuse, New York, USA, 13244; Zifan Wang, zwang345@syr.edu, Syracuse University, Syracuse, New York, USA, 13244; Fanxin Kong, fkong03@syr.edu, Syracuse University, Syracuse, New York, USA, 13244.

1539-9087/2023/6-ART $15.00
https://doi.org/10.1145/3603172

(value correctness), which represents the cause-effect relation defined by messages sent and received among tasks [14, 17, 22]. The other targets real-time systems and carries out checkpointing under timing constraints (deadlines) [29, 34, 37, 47, 48]. There are three main protocols setting checkpoints: uncoordinated checkpointing, coordinated checkpointing, and communication-induced checkpointing(CIC). Uncoordinated checkpointing enables tasks to set checkpoints when convenient for a better schedule[18, 38]. Coordinated checkpointing forces all the tasks to synchronize checkpoints and make recoveries simpler [7, 23]. Communication-induced checkpointing, also known as message induced checkpointing, enables tasks to establish separate checkpoints while also creating compulsory additional checkpoints as needed [3, 6, 41]. This approach promotes independence in detecting errors during execution. However, these works are incapable of tackling checkpointing real-time parallel processes, where both logical and timing correctness need to be guaranteed.

To fill this gap, we propose a new three-step checkpointing protocol that considers both types of correctness. In the first step, we partition the real-time parallel processes into a directed acyclic graph (DAG) of tasks, where each edge represents the message communicated between tasks. We then place compulsive checkpoints to ensure logical correctness for each task. The second step involves identifying the critical path, which is the longest execution path in the DAG. Finally, we ensure timing correctness by minimizing the overall execution time, which includes both task execution time and checkpointing overhead. To accomplish this, we propose effective and efficient algorithms for each step. Finally, we evaluate our checkpointing protocol with extensive simulations and a case study of a real system using CRIU [10]. Note that the checkpoints in this work only consider cyber states and are not related to physical states [43–46].

The rest of this paper is organized as follows. Section 2 describes the overview of the optimal checkpointing strategy, gives the system model and threat model, and lists most notations. Section 3, Section 4, and Section 5 present task partition, finding the critical path, and placing optional checkpoints for timing correctness, respectively. Section 6 evaluates our method, Section 8 concludes the paper, and Section 7 discusses the limitation of this paper.

2 PRELIMINARIES AND DESIGN OVERVIEW

this section, we present the problem statement, an overview of the proposed checkpointing strategy, system and fault models, and notations used in this paper.

2.1 Problem Statement

We consider a multiprocessor real-time system whose processes perform repetitive tasks, where each process aims at a specific function such as sensor data collection and data processing. During process execution, each process may send messages to other processes, forming the processes' dependencies.We study a checkpointing problem in such a system. When a fault is detected, the system rollbacks to the nearest valid checkpoint, which saves states of the processes and avoids redoing all valuable work during recovery. The objective is to determine an optimal checkpointing strategy, which achieves (i) logical consistency of checkpoints considering process dependencies and (ii) the shortest execution time considering checkpointing overhead and recovery time.

2.2 Overview of the Checkpointing Strategy

We obtain the optimal checkpoint strategy in three steps, as shown in Figure 1: (i) process partition, (ii) critical path extraction, and (iii) optional checkpoint placement. The following briefly describes these steps, and we will present their detailed design in Sections 3, 4, and 5.

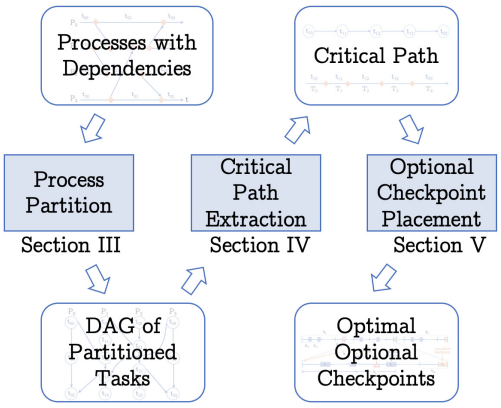


Fig. 1. The Overview of Optimal Checkpointing Strategy. The proposed optimal checkpointing strategy comprises three steps: process partitioning, critical path extraction, and optional checkpoint placement.

- *Process partition.* Execution of processes is dependent on message passing, and to ensure successful fault tolerance and recovery, checkpoints must be placed considering these dependencies. This involves partitioning dependent processes into a DAG graph and placing compulsive checkpoints that meet the requirements of logical dependency, while also avoiding the domino effect.
- *Critical path extraction.* In the DAG, most tasks can be executed in parallel with the multiprocessor. However, tasks in a dependent path must be executed in sequence. This step involves identifying the critical path, which is the longest dependent path in the DAG. The critical path determines the performance of the checkpointing and recovery process. While tasks in non-critical paths can also set up checkpoints, their execution time is typically less than that of tasks on the critical path. Therefore, in the proposed model, only the critical path should be considered.
- *Optional checkpoint placement.* If too few checkpoints are placed, a significant amount of progress will be lost on the critical path following a fault. Conversely, if too many checkpoints are placed, the checkpointing overhead will dominate. This step involves solving optimization problems to determine the optimal number and intervals of checkpoints, striking a balance between progress loss and checkpointing overhead.

2.3 Models and Preliminaries

Symbols and notations used in this paper are listed in Table 1.

2.3.1 System Model. We consider a multiprocessor real-time system that has multiple **processes**, where the messages exchanged between these processes form the process dependencies. We partition the processes into **tasks** (t_{BC} represents the C-th task on the B-th process) according to the dependencies. By partitioning the processes, we obtain a directed acyclic graph (DAG) where tasks are represented as nodes and process dependencies are represented as edges. We place *compulsory checkpoints* (see section 3) on this DAG to guarantee logical consistency and prevent the domino effect. Then, the critical path (see section 4) is identified as the longest dependent path, and the **tasks** in this path are renamed to T_D (i.e, the D-th task in the critical path). Finally, we place *optional checkpoints* (see section 5) according to our strategy to achieve a shorter execution time. The optional checkpoints split each task into some **segments** (S_{EF} is the F-th segment in T_E). Figure 2 is an example that illustrates the relationship among processes, tasks, and segments.

Table 1. Symbols and Notations Used in This Paper

Symbol	Description
P_i	i-th process in the system
T_i	i-th task in the critical path
S_{ij}	j-th segment in the Task i
s	the recovery overhead from the initial state
r	the recovery overhead from checkpoints
t_c	the overhead of checkpointing
q_i	the invalid rate of optional checkpoints in the Task i
p_i	the valid rate of optional checkpoints in the Task i
n	the number of tasks in the critical path
m_i	the number of optional checkpoints in the Task i
m_{i*}	the optimal number of optional checkpoints in the Task i
λ_i	the failure rate of the Task i
w_{ij}	the total execution time before S_{ij} in the Task i
h_{ij}	the total execution time of S_{ij} in the Task i
W_{ij}	the expectation of w_{ij}
d_{ij}	the completion time of S_{ij} before a fault
D_{ij}	the expectation of d_{ij}
F_{ij}	the probability that the fault occurs in S_{ij}
I_i	the fault-free computation time (excluding t_c) of Task i
τ_{ij}	the fault-free execution time (including t_c) of Segment S_{ij}

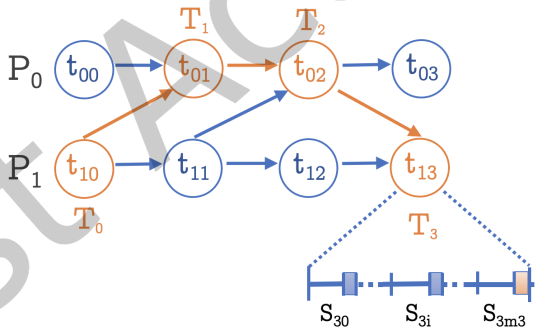


Fig. 2. Relationship of Process, Task, And Segment. Each of the two processes (P_0 and P_1) has four tasks. After inserting compulsory checkpoints, the critical path (i.e., longest dependent path) is determined (marked in orange). It includes four tasks (t_{10}, t_{01}, t_{02}, and t_{12}) that are renamed as T_0, T_1, T_2, and T_3. The task T_3 is divided by optional checkpoints into $m_3 + 1$ segments from S_{30} to S_{3m_3}, where m_3 and the intervals between checkpoints are derived by our approach.

We assume that the system stores all compulsory checkpoints in the current critical path and only stores the latest optimal checkpoint because of limited resources.

2.3.2 Fault Model. Fault occurrence is generally regarded as random and independent, so we assume that the arrival of faults is a Poisson process with a failure rate of λ. However, our proposed method is not limited to

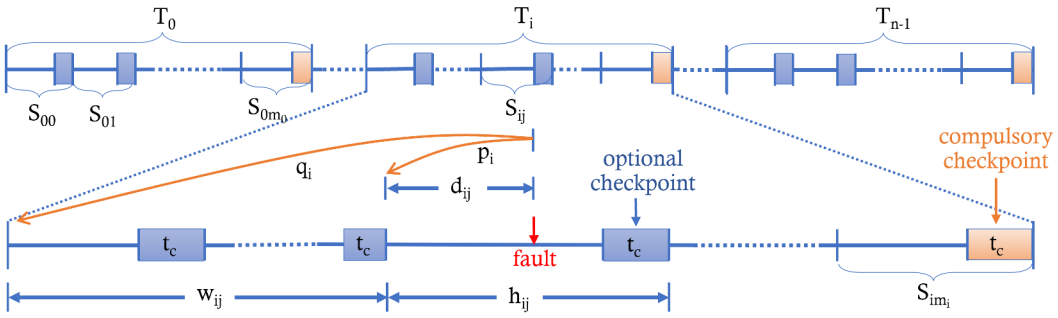


Fig. 3. Notations of the Model in Critical Path. Critical tasks $\{T_0, ..., T_{n-1}\}$ have $\{m0 + 1, ..., m_{n-1} + 1\}$ segments that are divided by $\{m0, ..., m_{n-1}\}$ optional checkpoints. w_{ij} and h_{ij} represent the total execution time before segment j and the execution time of segment j in Segments T_i, respectively. q_i and p_i are the probability of recovering from the beginning and last checkpoint, respectively.

a specific distribution and can be applied to various other distributions. The reason why our paper adopts the Poisson process is for easy presentation and its wide use in many existing works such as [11, 25, 26, 35, 36]. For generality, we set a different failure rate for each task on the critical path, and λ_i denotes the failure rate of the i-th task. Then, the time interval F between two faults is subject to an exponential distribution with a constant failure rate λ_i, and its probability density function (PDF) is $f_F(t) = \lambda_i e^{-\lambda_i t}, t \geq 0, \lambda \geq 0$.
When a fault arrives, there is a p_i chance for task T_i rollback to the latest checkpoint. However, there is also a $q_i = 1 - p_i$ chance for a task to roll back to the latest compulsory checkpoint because of the optional checkpoint availability. If a task rollback to a checkpoint, there is a checkpoint recovery overhead r. However, if a task rollback to the initial state of the critical path, there is a restart recovery overhead s.

2.3.3 Notations in the Critical Path. The critical path is illustrated in Figure 3. For the i-th task T_i on this path, the original fault-free computation time is I_i. We place optional checkpoints to divide the task into m_i task segments, i.e., $S_{i0}, S_{i1}, \ldots, S_{im_i}$. τ_{ij} denotes the fault-free execution time of S_{ij} that includes task computation time and checkpoint overhead t_c.
The probability that the fault occurs in Segment S_{ij} is

$$F_{ij}(\tau_{ij}) = P(T \leq \tau_{ij}) = 1 - P(T \geq \tau_{ij}) = 1 - e^{-\lambda \tau_{ij}}, \quad for \quad 1 \leq i < n, 0 \leq j \leq m_i \tag{1}$$

Besides, d_{ij} denotes the time interval from the beginning of S_{ij} to the time the fault occurs in S_{ij}, that is, the task performed before the fault during S_{ij}. The PDF of d_{ij} is

$$f_{d_{ij}}(t) = \frac{f_F(t)}{F_{ij}(\tau_{ij})} = \frac{\lambda_i e^{-\lambda_i t}}{1 - e^{-\lambda_i \tau_{ij}}}, \quad for \quad 0 \leq t \leq \tau_{ij}$$

The expectation of d_{ij} is

$$D_{ij} = \int_0^{\tau_{ij}} t f_{d_{ij}}(t) dt = \frac{1}{\lambda_i} - \frac{\tau_{ij} e^{-\lambda_i \tau_{ij}}}{1 - e^{-\lambda_i \tau_{ij}}} \tag{2}$$

Considering rollbacks, w_{ij} denotes the execution time from the beginning of Task i to the first beginning of Segment S_{ij}. W_{ij} is the expectation of w_{ij}. h_{ij} denotes the execution time of the segment S_{ij}.

To make the derivation brief, we also define some abbreviations as follows:

$$a_i = \frac{1}{\lambda_i} + s \tag{3a}$$

$$b_i = \frac{1}{\lambda_i} + p_i r + q_i s \tag{3b}$$

$$c_i = \frac{1}{\lambda_i} + r \tag{3c}$$

$$u_{ij} = e^{\lambda_i \tau_{ij}} - 1 \tag{3d}$$

$$v_{ij} = q_i e^{\lambda_i \tau_{ij}} + p_i \tag{3e}$$

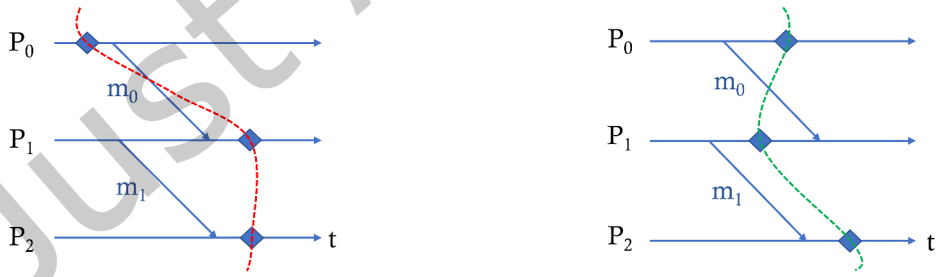
3 PROCESS PARTITION WITH LOGICAL CONSISTENCY

Processes in real-time systems perform some recurrent tasks, between which the sending and receiving messages form the dependencies. When a transient fault occurs, the system needs to be recovered back to normal. We back up some checkpoints so that the systems can re-execute from these states to tolerate the fault.

In this step, our checkpoints should meet the logical consistency requirements (Definition 1) to ensure that the recovery process runs smoothly [14]. For example, the states in Figure 4a are inconsistent because P_1 (process 1) indicates the m_0 reception while P_0 (process 0) does not reflect the sending of m_0; the states in Figure 4b satisfy Definition 1 because P_2 (process 2) indicates the m_1 reception and P_1 reflects the sending of m_1, although P_1 does not reflect the m_0 reception particularly. We can conclude that all states satisfy Definition 1 if all processes checkpoint after sending a message.

Definition 1 (Logical Consistency). *Logical consistency is defined as an attribute of the system state that ensures that sender processes reflect the sending of messages once the corresponding receiver processes indicate the message reception.*

Another notable phenomenon is the domino effect, which occurs when an invalid message reception leads to an invalid message sending. This can result in a series of rollbacks, ultimately returning the system to its initial state. The domino effect leads to a significant loss of useful work and makes real-time performance uncontrollable.



(a) Logical inconsistency. After recovering, system states roll back to the dotted line. Process 1 reflects the reception of message 0 while the corresponding process 0 does not reflect the message sending. Thus, system states marked by the red dotted line are not logically consistent.

(b) Logical consistency. After recovering, process 1 and process 2 reflect the sending and reception of message 1 at the same time. Although process 1 does not reflect the reception of message 0, this does not violate the definition of logical consistency.

Fig. 4. Examples of Logical Inconsistency and Logical Consistency. P and m are processes and messages. Blue diamonds are checkpoints, and dot lines denote the system state after recovering.

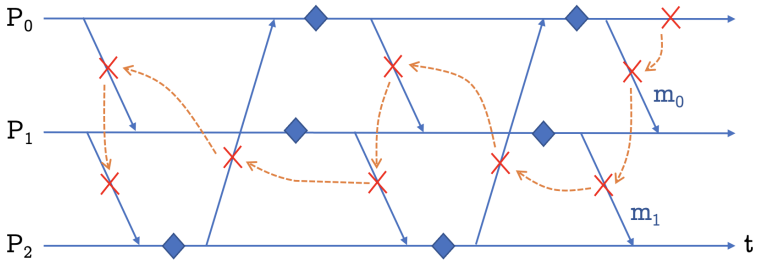
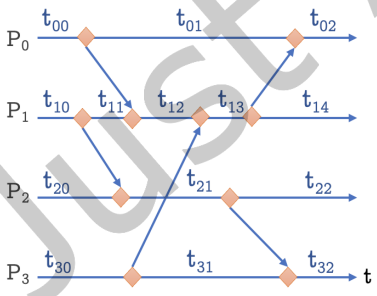


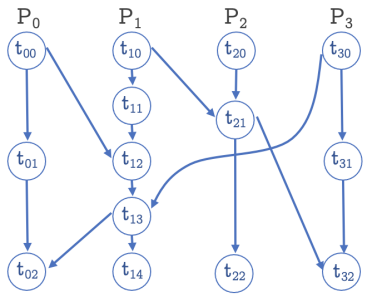
Fig. 5. **Domino Effect.** Incorrect checkpointing may lead to the domino effect, resulting in a series of rollbacks that ultimately return the system to its initial state.

For instance, Figure 5 shows a rollback scenario, in which a fault occurs on the process P_0. The fault forces P_0 to roll back to its latest checkpoint, and message m_0 becomes invalid, which forces process P_1 to roll back to its latest checkpoint as well. In turn, the message m_1 becomes invalid, and this backward propagation stops until the initial state. If all processes checkpoint before receiving a message, the domino effect can be avoided.

We partition processes into tasks based on their message sending and receiving behavior. We add edges between neighboring tasks within the same process and from a task sending a message to a task receiving the message, creating a partition graph. The weight of each vertex in this graph represents the fault-free execution time of the corresponding task, while the edges represent the dependencies between tasks. We also account for checkpoint overhead in the weight of each vertex. When a process sends a message, a compulsory checkpoint is placed immediately following it to reflect the message sent and ensure logical consistency. When a process receives a message, a compulsory checkpoint is placed to backup all work before the message, avoiding the domino effect. This can typically be accomplished by piggybacking a command to set up a compulsory checkpoint with the actual processing command after receiving a message. Now, we get a task graph in which all states satisfy Definition 1. This graph is a DAG because no task can send a message back to a previous time to form a



(a) Dependencies of processes. Compulsory checkpoints (orange diamonds) are placed immediately after message sendings and before message receptions to ensure logical consistency and prevent the domino effect.



(b) DAG of tasks. Edges are the dependencies between tasks, while vertices represent partitioned tasks. The weight of each vertex is the sum of the fault-free execution time and the overheads of the compulsory checkpoints.

Fig. 6. An Example of Process Partition. The processes are partitioned into fourteen tasks based on their message sending and receiving behavior.

(a) Extract critical path from DAG (b) Critical path includes compulsory checkpoints

Fig. 7. An Examples of The Critical Path

circle. For instance, Figure 6a shows a basic scenario with 4 processes - P_0, P_1, P_2 and P_3. After partitioning, we obtain a DAG graph, i.e., Figure 6b. In this graph, each vertex is a task partitioning from processes. The weights of vertices in the partition graph are equal to the fault-free execution time of the corresponding task plus the overheads of the compulsory checkpoints placed on it.

Remark 3.1. *Compulsory checkpoints satisfy logical consistency and avoid the domino effect.*

When a failure occurs, and the system rolls back to the compulsory checkpoints, the sender processes show that they have sent the messages, while the corresponding receiver processes indicate that they have not yet received them. This demonstrates that the messages have been sent but not yet received [14], which is in line with the definition of logical consistency. On the other hand, because the sender processes set compulsory checkpoints after sending the messages, the system will not invalidate the messages, thereby preventing further rollbacks, i.e., the domino effect.

4 CRITICAL PATH EXTRACTION

In a system with multiple processors, most tasks can be executed in parallel. However, tasks connected by edges in the DAG must be performed in sequence because of the dependencies. Identifying the critical path (Definition 2) in the DAG gives us the maximum length of tasks that must be executed in sequence to ensure logical consistency and avoid the domino effect. This critical path determines the total execution time of these processes. It's important to note that the critical path is extracted after inserting the compulsory checkpoints.

Definition 2 (Critical Path). *In a DAG of tasks, the critical path is a path in which the total sum of the weight of the vertices is no less than that of any other path.*

Algorithm 1 shows an algorithm to find a critical path based on the topological sort and dynamic programming. First, we topologically sort the DAG and get an ordered vertex set V with a complexity of $O(|V|)$. Then, we use dynamic programming to calculate the maximum total weight ending with vertex v, i.e., $tw(v)$. A transition function is defined as

$$tw(v) = max\{tw(u) + w(v)\} \tag{4}$$

where (u, v) is an edge in E and $w(v)$ is the weight of the vertex v. The computation is performed in the topological order of v. Given that the maximum number of computing $tw(\cdot)$ is equal to the number of edges, the algorithmic complexity is $O(|E|)$. Finally, we select the maximum value of $tw(v)$ as the highest total weight of l and reconstruct the critical path P using each optimal choice $v.choice$ we record.

The critical path is the worst-case for the following analysis. Figure 7a shows a critical path partitioned from processes, in which we already placed compulsory checkpoints (shown in Figure 7b) for logical consistency. However, it's challenging to meet real-time requirements using only compulsory checkpoints with long intervals. When a fault occurs, the system needs to roll back to a consistent state containing only compulsory checkpoints. This rollback can violate timing correctness (Definition 3), making it difficult to meet real-time requirements.

Algorithm 1 The Critical Path Extraction

Input: DAG $G = (V, E)$
Output: Critical path P, largest total weight l

```
Initialization: Topologically sort G
for each vertex v ∈ V in topological order do
    tw(v) ← 0
    for each edge (u, v) ∈ E do
        if tw(u) + w(v) > tw(v) then
            tw(v) ← tw(u) + w(v)
            v.choice ← u
        end if
    end for
    if tw(v) > l then
        l ← tw(v), x ← v
    end if
end for
for x.choice ≠ NULL do
    P.add(x)
    x ← x.choice
end for
return P, l
```

Definition 3 (Timing Correctness). *Timing Correctness means that all tasks in each process catch up with the deadline of this process in a real-time system.*

5 CHECKPOINT PLACEMENT FOR TIMING CORRECTNESS

In this section, we focus on placing optional checkpoints on each task of the critical path to minimize the total execution time, denoted as W_n. If no optional checkpoints are placed, a long rollback may occur, leading to a waste of useful work. However, placing too many checkpoints results in significant checkpoint overhead. To find a balance, we formulate an optimization problem for each task to determine the appropriate number and length of checkpoint segments.

As shown in Figure 7b, the first task T_0 in the critical path starts from the initial state, while other tasks $T_1, \ldots, T_n - 1$ start from a compulsory checkpoint. Thus, we apply the proposed optimization method to the two conditions separately.

5.1 Optimization for the First Task

The first task, T_0, is exceptional. If the task rollback to an optional checkpoint, the rollback overhead is r. However, if the task rollback to a checkpoint unsuccessfully, the task restarts with a higher overhead s, because there is no checkpoint at the beginning of the task T_0. In Segment S_{00}, the task is restarted if a fault is detected. Hence, the total execution time is updated because of it:

$$w_{01} = \begin{cases} \tau_{00} & P = 1 - F_{00}(\tau_{00}) \\ d_{00} + s + w_{01} & P = F_{00}(\tau_{00}) \end{cases} \tag{5}$$

From Eq.(1), (2), (5), we can derive the expectation of w_{01}:

$$W_{01} = \tau_{00} + \frac{F_{00}(\tau_{00})}{1 - F_{00}(\tau_{00})}(D_{00} + s) = (\frac{1}{\lambda_1} + s)(e^{\lambda_0 \tau_{00}} - 1) = a_0 u_{00} \tag{6}$$

In other segments, S_{0j}, there is no fault with $1 - F_{0j}(\tau_{0j})$ chance. The task rollbacks to the latest checkpoint with probability p_0 or restarts with probability q_0 if a fault is detected. Hence, the total execution time is

$$w_{0(j+1)} = w_{0j} + h_{0j} \tag{7}$$

where

$$h_{0j} = \begin{cases} \tau_{0j} & P = 1 - F_{0j}(\tau_{0j}) \\ d_{0j} + r + h_{0j} & P = F_{0j}(\tau_{0j}) p_0 \\ d_{0j} + s + w_{0(j+1)} & P = F_{0j}(\tau_{0j}) q_0 \end{cases}$$

From Eq.(1), (2), (7), we can get the expectation of $w_{0(j+1)}$:

$$\begin{aligned} W_{0(j+1)} &= \frac{1 - p_0 F_{0j}(\tau_{0j})}{1 - F_{0j}(\tau_{0j})} W_{0j} + \tau_{0j} + \frac{F_{0j}(\tau_{0j})}{1 - F_{0j}(\tau_{0j})}(D_{0j} + p_0 r + q_0 s) \\ &= (q e^{\lambda_1 \tau_{1j}} + p_1) W_{1j} + (e^{\lambda_1 \tau_{1j}} - 1)(\frac{1}{\lambda} + p_1 r + q_1 s) = v_{0j} W_{0j} + u_{0j} b_0 \end{aligned} \tag{8}$$

Using Eq.(6) and Eq.(8) recursively, we derive the expectation of the total execution time of the first task:

$$\begin{aligned} W_0 &= W_{0(m_0+1)} \\ &= a_0 u_{00} \prod_{i=1}^{m_0} v_{0i} + b_0 u_{01} \prod_{i=2}^{m_0} v_{0i} + b_0 u_{02} \prod_{i=3}^{m_0} v_{0i} + \ldots + b_0 u_{0(m_0-1)} v_{0m_0} + b u_{0m_0} \end{aligned} \tag{9}$$

The optimization problem is expressed as:

$$\begin{aligned} \arg\min_{m_0, \tau_{0j}} \quad & W_0 \\ \text{s.t.} \quad & \sum_{j=0}^{m_0} \tau_{0j} = I_1 + (m_0 + 1) t_c \end{aligned} \tag{10}$$

Because there are two variables in the problem, we first assume that m_0 is given and try to find the relations between τ_{0j}. Then, we try to compute the optimal m_0.
We introduce a Lagrange multiplier θ and get the Lagrange function.

$$\mathcal{L}(\tau_{0j}, \theta) = W_0(\tau_{0j}) - \theta g(\tau_{0j}), \text{ where } g(\tau_{0j}) = I_0 + (m_0 + 1) t_c - \sum_{j=0}^{m_0} \tau_{0j} \tag{11}$$

The optimal solution satisfies

$$\nabla \mathcal{L}(\tau_{0j}, \theta) = 0 \tag{12}$$

From Eq.(12), we get the equation:

$$\frac{\partial W_0}{\partial \tau_{00}} = \frac{\partial W_0}{\partial \tau_{01}} = \ldots = \frac{\partial W_0}{\partial \tau_{0m_0}} \tag{13}$$

From Equation (13), we can obtain the relations between the fault-free execution time of segments $\tau_{00}, \tau_{01}, \ldots, \tau_{0m_0}$:

$$\begin{aligned} \tau_{01} &= \tau_{02} = \cdots = \tau_{0m_0} = \tau_{0*} \\ \tau_{00} &= \tau_{0*} + \tau_d \end{aligned} \tag{14}$$

where $\tau_d = \frac{1}{\lambda_0} ln(\frac{1+\lambda_0 r}{1+\lambda_0 s})$. Hence, W_0 becomes

$$W_0 = a_0 u_{00} v_{0*}^m + b_0 u_{0*} \sum_{i=0}^{m_0-1} v_{0*}^i = (a_0 u_{00} + b_0 q_0^{-1}) v_{0*}^m - b_0 q_0^{-1} \tag{15}$$

where

$$\begin{aligned} \tau_{0*} &= \frac{I_0 - \tau_d}{m_0 + 1} + t_c \\ u_{0*} &= e^{\lambda_0 \tau_{0*}} - 1 \\ v_{0*} &= q_0 e^{\lambda_0 \tau_{0*}} + p_0 \end{aligned}$$

We get the optimal number of checkpoints in the first task m_0* by solving $\frac{\partial W_0}{\partial m_0} = 0$, and choose the best value of two nearest integers as the optimal solution. According to the optimal m_0* and Equation (14), we can determine where to checkpoint in Task 0.

5.2 Optimization for Other Tasks

For tasks after T_0, the task rolls back to a compulsory or optional checkpoint with overhead r, if a fault is detected. There is no restart overhead because there is a compulsory checkpoint at the beginning of the task T_i. Therefore, the total execution time for S_{ij} is

$$w_{i(j+1)} = \begin{cases} h_{i0} & j = 0 \\ w_{ij} + h_{ij} & 1 \le j \le m_i \end{cases} \tag{16}$$

where

$$h_{ij} = \begin{cases} \tau_{ij} & P = 1 - F_{ij}(\tau_{ij}) \\ d_{ij} + r + h_{ij} & P = F_{ij}(\tau_{ij}) p_i \\ d_{ij} + r + w_{i(j+1)} & P = F_{ij}(\tau_{ij}) q_i \end{cases} \tag{17}$$

From Eq.(1), (2), (16), (17), we can derive the expectation of $w_{i(j+1)}$:

$$\begin{aligned} W_{i(j+1)} &= \frac{1 - p_i F_{ij}(\tau_{ij})}{1 - F_{ij}(\tau_{ij})} W_{ij} + \tau_{ij} + \frac{F_{ij}(\tau_{ij})}{1 - F_{ij}(\tau_{ij})} (D_{ij} + r) \\ &= (q_i e^{\lambda \tau_{ij}} + p_i) W_{ij} + (e^{\lambda \tau_{ij}} - 1)(\frac{1}{\lambda_i} + r) = v_{ij} W_{ij} + u_{ij} c_i \end{aligned} \tag{18}$$

We add Eq.(18) recursively, and we can derive the expectation of the total execution time of the task T_i:

$$\begin{aligned} W_i &= W_{i(m_i+1)} \\ &= c_i u_{i0} \prod_{j=1}^{m_i} v_{ij} + c_i u_{i1} \prod_{j=2}^{m_i} v_{ij} + c_i u_{i2} \prod_{j=3}^{m_i} v_{ij} + ... + c_i u_{i(m_i-1)} v_{im_i} + c_i u_{im_i} \end{aligned} \tag{19}$$

The optimization problem is expressed as:

$$\begin{aligned} \arg\min_{m_i, \tau_{ij}} \quad & W_i \\ \text{s.t.} \quad & \sum_{j=0}^{m_i} \tau_{ij} = I_i + (m_i + 1) t_c \end{aligned} \tag{20}$$

The Lagrange function is

$$\mathcal{L}(\tau_{ij}, \theta) = W_i(\tau_{ij}) - \theta g(\tau_{ij}), \text{ where } g(\tau_{ij}) = I_i + (m_i + 1)t_c - \sum_{j=0}^{m_i} \tau_{ij} \tag{21}$$

The optimal solution satisfies

$$\nabla \mathcal{L}(\tau_{ij}, \theta) = 0 \tag{22}$$

From Eq.(22), we can get the following equation:

$$\frac{\partial W_i}{\partial \tau_{i0}} = \frac{\partial W_i}{\partial \tau_{i1}} = \ldots = \frac{\partial W_i}{\partial \tau_{im_i}} \tag{23}$$

From Equation (23), we can get the relations between the fault-free execution time of Segments $\tau_{i0}, \tau_{i1}, \ldots, \tau_{im_i}$:

$$\tau_{i0} = \tau_{i1} = \cdots = \tau_{im_i} = \tau_{i*} \tag{24}$$

Hence, W_i becomes

$$W_i = c_i u_{i*} \sum_{j=0}^{m_i} v_{i*}^j = -c_i q_i^{-1}(1 - v_{i*}^{m_i+1}) \tag{25}$$

where

$$\tau_{i*} = \frac{I_i}{m_i + 1} + t_c$$
$$u_{i*} = e^{\lambda_i \tau_{i*}} - 1$$
$$v_{i*} = q_i e^{\lambda_i \tau_{i*}} + p_i$$

By solving $\frac{\partial W_i}{\partial m_i} = 0$, and choosing the best value of two nearest integers, we get the optimal m_{i*}. Then, we can checkpoint at equidistance in Task i according to Equation (24).

Remark 5.1. *Optional checkpoint placement in the critical path, which is computed by solving the above optimization problem, achieves the shortest total execution time.*

The optimization objectives for both the first and other tasks are the total execution time including overheads. Thus the number of optional checkpoints and their intervals that are computed by solving the optimization problem lead to the shortest total execution time.

6 EVALUATION

In this section, we evaluate the effectiveness of our proposed optional checkpointing strategy through extensive simulations and a case study. For the simulations, we randomly generate a system with dependent processes and evaluate our approach based on four aspects: prevention of the domino effect, optimization of optional checkpoint intervals, optimization of checkpoint numbers, and performance under different scales. In the case study, we demonstrate how our approach works in a real system.

6.1 Simulation

6.1.1 Simulation Setting. The generated processes with dependencies are shown in Figure 8a. According to Section 3, we partition these processes into a DAG of tasks, shown in Figure 8b. The fault-free execution times of each task are marked beside the vertices. According to Section 4, We extracted the critical path from the DAG and marked it in orange. The critical path consists of four tasks, each with time lengths of 400, 300, 200, and 200. The overall deadline for completing all tasks is 3300, which is equivalent to three times the fault-free computation time of the tasks on the critical path.

Considering the simplicity of setting up the experiment and the directness of illustrating the advantage of our model, the parameters mentioned in the Section 2.3.3 are as follow. Some parameters are based on real-world experiences, such as the checkpointing overhead for placement and recovery, while others can be set arbitrarily and do not affect the fairness among different strategies. We choose the same fault rate for each task, i.e., $\lambda_0 = \lambda_1 = \lambda_2 = \lambda_3 = 0.01$, which means one fault is expected to occur every 100 units of time. The checkpoint placement overhead t_c is 4. When a fault arrives, the system rollback to an optional checkpoint with a probability of $p = 0.8$, and to a compulsory checkpoint (the initial state for J_1) with a probability of $q = 0.2$. The recovery overhead from a checkpoint is $r = 12$, and the recovery overhead from the initial state is $s = 20$.

To simulate the time interval between two faults, we use the equation in [27] for the exponential distribution:

$$nextInterval = \frac{-lnU}{\lambda}$$

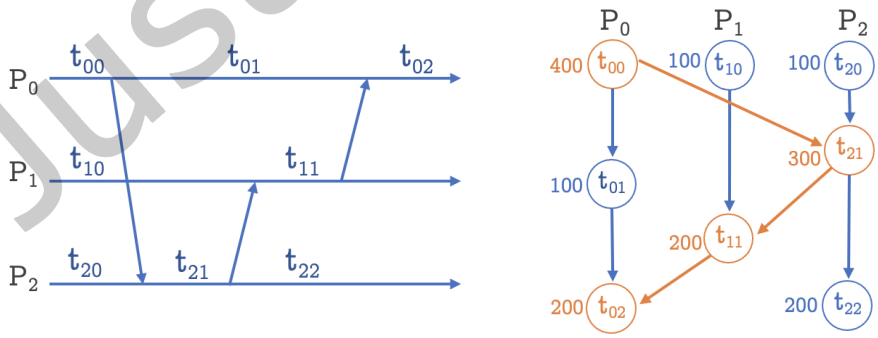
where U is a random value between 0 and 1.

According to the Section 5, we can obtain the optimal number of checkpoints in each task of the critical path, which are 13, 9, 6, 6 checkpoints. According to Eq.(14) and (24), we can also get corresponding intervals between every two checkpoints.

6.1.2 Simulation Result. We perform four simulations to evaluate our strategy of setting checkpoints. The first simulation aims to determine whether our model can prevent the domino effect and thus reduce the execution time. The second and the third simulation aims to prove our model optimizes the interval of checkpoints and the number of checkpoints, respectively. The fourth simulation shows that our model performs well in a wide range of scales.

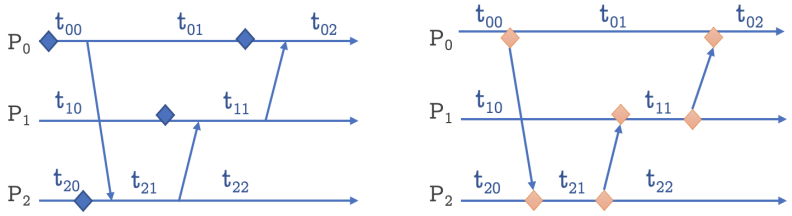
Simulation 1: Domino Effect Prevention. The processes with dependencies in Fig.9a suffer from the domino effect if we set checkpoints randomly, for example, the blue checkpoints. Instead, setting checkpoints based on our strategy prevents the system from rolling back to the initial state whenever faults happen. Thus, the system's execution time will be shortened. We simulate the critical path 100000 times with four strategies: (a) no checkpoints, place no checkpoints in the system; (b) only compulsory checkpoints, place no optional checkpoints; (c) only optional checkpoints, place no compulsory checkpoints; (d) optimal checkpoints, place both compulsory and optional checkpoints (the proposed strategy). Among them, (a) and (c) are affected by the domino effect.

The result shows that the domino effect leads to a large execution time. There are two observations: (i) the average execution time of strategy (a) is about 750 times longer than that of strategy (b), (ii) the average execution



(a) Simulation processes (b) DAG of tasks

Fig. 8. Simulation Setting



(a) With domino effect (b) Without domino effect

Fig. 9. Simulation 1 - Domino Effect Prevention

Table 2. The Result of Simulation 1 - Domino Effect Prevention. Avg Exec: Average Execution Time, Min Exec: Minimum Execution Time, Max Exec: Maximum Execution Time, %Deadline: the Percentage of Simulations that Meet the Deadline. Strategies: NC: (a) No Checkpoints, CO: (b) Only compulsory Checkpoints, OO: (c) Only Optional Checkpoints, OP: (d) Optimal Checkpoints (the Proposed Strategy).

Strategy	Avg Exec	Min Exec	Max Exec	%Deadline
NC	7581500.85	8378.91	45725624.63	0.00
CO	10067.38	1116.00	70697.98	6.22
OO	10340.65	1287.56	111719.27	20.67
OP	2616.95	1264.34	10702.18	81.82

time of (b) is about 4 times longer than that of strategy (d). The reason behind the noteworthy difference is from the compulsory checkpoints, which make the system free from the domino effect. For (a), the system rollback to the initial state when a fault occurs, but for (b), the system only rollback to the nearest compulsory checkpoints. Thus, a large amount of useful work can be saved. Although the system can rollback to an optional checkpoint when a fault occurs for strategy (c), but there is also a possibility q that the optional checkpoint is not valid. Under this condition, the system has to rollback to the initial state, which leads to a larger execution time in strategy (c). The domino effect is also avoided in strategy (d). This analysis highlights the importance of compulsory checkpoints that prevent the domino effect, reduce the execution time, and increase the finishing process percentage on time.

Simulation 2: Performance Regarding Checkpoint Interval. The second simulation compares our checkpoint placement strategy with four other strategies on the critical path with respect to checkpoint intervals. We consider five types of different checkpoint placement strategies, the same in the number of checkpoints but different in the checkpoint interval. They are: (a) optimal placement strategy obtained from Section 5; (b) two-state strategy [36]: a strong prior work that has two stages of setting checkpoints, and the first stage delays the checkpoints as much as possible avoiding checkpointing overheads. (c) uniformI (I stands for intervals): place checkpoints based on uniform distribution; (d) Gauss distribution placement strategy: place checkpoints based on Gauss distribution with $I/2$ as the mean and $I/4$ as the standard deviation; (e) narrowing placement strategy: gradually narrow the interval between two checkpoints; (f) widening placement strategy: gradually widen the interval between two checkpoints. The placement strategy (d) is based on the algorithm: the $i+1$-th checkpoint in a task is placed at the first third of the interval between the i-th checkpoint and the end. The placement strategy (e) is the reverse process of strategy (d). We simulate the critical path process 100000 times and list the result in Table 3.

Table 3. The Result of Simulation 2 - Performance Regarding Different Checkpoint Interval. Strategies: (a) Optimal, (b) TwoState, (c)UniformI, (d) Gauss, (e) Narrowing, (f) Widening.

Strategy	Avg Exec	Min Exec	Max Exec	%Deadline
Optimal	2616.12	1252.00	11995.20	81.88
TwoState	2761.80	1398.50	8682.60	80.96
UniformI	2744.79	1236.00	11732.31	77.80
Gauss	2732.77	1254.44	10993.74	78.19
Narrowing	2946.51	1250.99	14325.78	70.97
Widening	2941.82	1265.67	13142.09	71.02

The result shows that our model optimizes the interval between checkpoints. We notice that our optimal strategy (a) has the shortest average execution time and the highest percentage of finishing processes on time. Strategy (b), (c), and (d) have shorter average execution times than strategy (d) and (e), but still longer than strategy (a). The prior work (b) yields a competitive rate of meeting deadlines with the proposed strategy but it behaves worse than the proposed strategy and baseline strategies (c) and (d) in terms of average and minimum execution time. This is because of its concentrated distribution of execution time. This strategy reduces the maximum execution time and increases the percentage of meeting deadlines, however, it also increases the minimum execution time and thus increases the average execution time. Strategy (c) places the checkpoints uniformly on each task, making it worse but close to our model's result. The performance of strategy (d) depends on the value of the mean and standard deviation, and in this case, it has a better performance than strategy (c). Note, the maximum execution time of strategy (d) being less than other strategies is due to randomness, i.e. fewer faults happen during some runs in strategy (d). Our model cannot guarantee to perform the best every time, but it promises a better average result when running time accumulates.

Simulation 3: Performance Regarding Checkpoint Number. The third simulation compares our checkpoint placement strategy with three other strategies on the critical path with respect to checkpoint numbers. We consider four types of checkpoint placement strategy in this simulation: (a) optimal placement strategy obtained from Section 5; (b) MelhemInt: The algorithm of determining the number of checkpoints that is used in many prior work [4, 30]; (c) uniformM (M stands for the number of checkpoints m), placing the same number of checkpoints in each task, and the total number of checkpoints is close to that in (a); (d) light-weight placement strategy, place fewer checkpoints than (a), but the number of checkpoints in each task is proportional to the computation time of each task; (e) heavy-weight placement strategy, place more checkpoints than (a), but the number of checkpoints in each task is proportional to the computation time of each task. All strategies share the same pattern of determining intervals, i.e., our model. We simulate the critical path process 100000 times, and the result is listed in Table 4.

The result shows that our model optimizes the number of checkpoints. The other three strategies slightly change the number of checkpoints for each task, and none of them performs better than our model. Our strategy reduces the average execution time and increases the percentage of processes completed on time. The prior work (b) and the lightweight placement baseline place inadequate checkpoints, wasting useful work and leading to longer execution time. On the other hand, the heavy-weight placement strategy places too many checkpoints, and their overheads contribute more execution time. Only our proposed strategy meets the trade-off between useful work waste and checkpointing overhead. Note that the light-weight placement strategy's minimum execution time is smaller than our model because it places fewer checkpoints. The scale, i.e. the size of the DAG and the length of the critical path, is small in this simulation, which leads to the slight improvement from other strategies to the proposed model. Improvement will be increased in simulation 4.

Table 4. The Result of Simulation 3 - Performance Regarding Different Checkpoint Numbers. CP No.: Number of Checkpoints in The Critical Path Tasks. %DDL: The Percentage of Simulations That Meet the Deadline. Strategies: (a) Optimal, (b) MelhemInt, (c) UniformM, (d) Light-weight, (e) Heavy-weight.

Strategy	CP No.	Avg Exec	Min Exec	Max Exec	%DDL
Optimal	13, 9, 6, 6	2617.39	1267.73	11791.90	81.90
MelhemInt	4, 3, 3, 3	3636.14	1167.65	35886.96	51.79
UniformM	9, 9, 9, 9	2653.03	1293.53	11901.07	80.67
Light-wt	10, 7, 5, 5	2628.57	1236.94	10821.43	81.47
Heavy-wt	16, 11, 7, 7	2637.40	1314.42	11609.88	81.27

Simulation 4: Performance Regarding Scalability. The fourth simulation shows the scalability of our checkpoint placement strategy. First, we gradually add the number of processes and their lengths. The execution time for each task is chosen randomly from 50 to 650 units of time. Second, we randomly generate dependencies between tasks: for each task, there is 0.4 chance that no message is sent out, 0.5 chance that 1 message is sent to another process, and 0.1 chance that 2 messages are sent to other processes. By selecting and adjusting the number and length of processes, the scale of the DAG can be controlled to an expected range. Then according to Section 3, we partition these processes into a DAG of tasks. Finally, according to Section 4, we extract the critical path of the DAG. The set of parameters, i.e. λ, t_c, p, q, r, s is the same as the above value. And the deadline for finishing all tasks is also 3 times the fault-free computation time of tasks on the critical path. The scale and critical path details are shown in Table 5 and Table 6 respectively.

Table 5. The Scales of Simulation 4 - Performance Regarding Scalability. Proc No.: Number of Processes, Avg Proc Len: Average Process length, Msg No.: Number of Messages, Critical Path Len: Length of Critical Path.

Proc No.	Avg Proc Len	Msg No.	Critical Path Len
3	40	91	48
4	80	206	93
5	120	444	142
6	160	680	191
7	200	978	238
8	240	1319	292

The optimal checkpoint numbers are calculated by the proposed model. We simulate the critical path process 100000 times and list the result in Table 7. We also simulate the two best baselines in the previous simulation for better comparison: the TwoState strategy in simulation 2 and the light-weight strategy in simulation 3. Besides, we simulate the strategy of using compulsory checkpoints only.

The result shows that our model stays strong in different aspects of scale and proves again that it performs better than other strategies. We notice that as the critical path becomes longer, the average execution time of all four strategies increases. The strategy to place only compulsory checkpoints cost the system over 10 times longer than the other three strategies to complete the tasks. Moreover, the percentage of meeting the deadline remains 0 on all scales. This unacceptable result is due to repeated work without proper checkpointing. The TwoState strategy that had a competitive performance as our approach in simulation 2 has unacceptable performance on execution time and deadline meeting rate at all scales. As the scale grows, its performance degrades significantly because its motivation to delay the first checkpoint leads to a long rollback for the first fault. Also, in contrast to

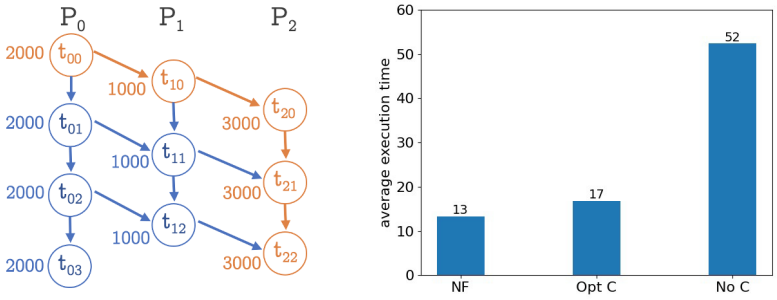
Table 6. The Detail of Critical Path of Simulation 4 - Performance Regarding Scalability. Path Len: Number of Tasks in the Critical Path. Opt CP No.: The Optimal Number of Checkpoints in the Critical Path, FF Exec: Fault-free Execution Time, Task Avg: Average Execution Time of All the Tasks, Task S.D.: Standard Deviation of All the Tasks.

Path Len	Opt CP No.	FF Exec	Task Avg	Task S.D.
48	522	16945	353.02	166.84
93	1002	32656	351.14	173.85
142	1541	50097	352.79	174.42
191	2095	68033	356.19	171.24
238	2610	84779	356.21	174.76
292	3183	103359	353.97	175.49

Table 7. The Result of Simulation 4 - Performance Regarding Scalability. P Len: Length of Critical Path, Strategies: Opt: The Proposed Strategy, TwoState: Prior Work proposed in [36], L-wt: Light-weight Placement Strategy, CO: Only Compulsory Checkpoints

P Len	Strat	Avg Exec	Min Exec	Max Exec	%DDL
48	**Opt**	46658.36	29043.16	79035.80	79.52
	TwoState	54044.17	37780.91	79985.81	48.11
	L-wt	47159.98	29690.36	78796.84	76.60
	CO	549115.45	183697.60	1433248.68	0.00
93	**Opt**	90962.30	63480.37	131047.60	82.37
	TwoState	107625.32	82879.86	147974.87	26.12
	L-wt	92036.61	67042.33	132859.30	78.44
	CO	1174630.43	525642.00	2287054.97	0.00
142	**Opt**	142221.44	107544.19	189782.72	83.71
	TwoState	162806.04	129464.32	208093.57	14.80
	L-wt	143888.79	106922.95	191786.89	79.07
	CO	1893677.80	1048011.40	3309548.29	0.00
191	**Opt**	190392.98	148406.07	247362.28	88.54
	TwoState	231139.34	189596.35	285603.39	2.45
	L-wt	192664.35	148323.16	247667.30	84.14
	CO	2628444.25	1516961.52	4276293.98	0.00
238	**Opt**	238168.00	192514.41	309228.85	89.70
	TwoState	285246.53	237750.67	342616.35	13.18
	L-wt	240934.17	192965.68	304839.08	85.08
	CO	3223960.78	2017579.28	5132395.45	0.00
292	**Opt**	289887.27	240952.28	358688.84	92.30
	TwoState	324411.12	274018.52	387484.72	13.77
	L-wt	293287.77	239438.80	356610.61	88.15
	CO	3872775.91	2507589.88	5993822.13	0.00

the more concentrated distribution of execution times, the TwoState strategy has an overall longer execution time in average, minimum, and maximum, and the difference increases as the scale grows. The lightweight placing strategy has a competitive result, but still fails to surpass our model in terms of both average execution time and



(a) DAG of case study (b) Result of case study

Fig. 10. DAG and Result of Case Study. NF: No Faults happen, Opt C: Optimal Checkpoints (the Proposed Strategy), No C: No Checkpoints placed

percentage of meeting deadline. The reason is that given the relatively low fault rate, the overhead of setting checkpoints makes up the gap between performances. Another noteworthy observation shown in the result is that the uniform distribution strategy's percentage of meeting the deadline goes low as the scale becomes large, while our model and lightweight strategy have an opposite trend. This is because i) our model performs better as the scale grows and random data become stable, ii) the light-weight strategy's performance is relative to our model as it has a fixed ratio of fewer checkpoints, iii) the absolute more execution time of uniform strategy becomes large as the scale expands so the percentage of meeting the deadline drops.

6.2 Case Study

In this section, we apply and test our model on an environmental monitoring system that monitors, records, and analyses the campus environment, including atmospheric composition, tap water ingredients, environmental noise, and twelve more aspects of data. The program has three processes: (a) data processing process: iterate a database, filter, and retrieve the target records; (b) logic processing process: analyze and sort the records retrieved by process (a); (c) message sending process: send the sorted records to an external program and waiting for responses. Since there are trillions of records, all three processes should be partitioned into several tasks according to different record ID ranges to reduce the cost of faults. The inputs of the latter tasks depend on the results of the previous tasks. We choose a pair of boundaries of this program and plot the system in a DAG, Figure 10a. The number besides tasks is their estimated execution time: every task of the process (a) needs 2 seconds to run; every task of the process (b) needs 1 second to execute; every task of the process (c) consumes 3 seconds. The critical path is colored orange. We run the program 100 times and plot the results in Figure 10b.

We use CRIU [10], a library for process state management, to set and restore checkpoints. We build a separate C++ program to generate faults, save, and restore from checkpoints. The number of checkpoints and the interval between checkpoints is obtained from our proposed strategy, according to Section 5. The expected interval between every two faults is consistent with [27]. The fault rate $\lambda = 0.001$. Other parameters, i.e. t_c, p, q, r, s are the same as the simulation settings, and the deadline is also 3 times the fault-free critical path execution time.

The result in Figure 10b shows that if we do not set any checkpoints on the program, the average execution time (i.e., 52.37s) is about four times longer than the expected 13.32s execution time if no faults occur. However, if we set checkpoints based on the model described before, we can decrease the average execution time to 16.83s because the checkpoints prevent the program from falling back to the initial state and repeatedly doing the same

tasks. Thus, our model of setting checkpoints can reduce the average execution time in real-world programs if faults happen.

7 DISCUSSION

Compulsory and optional checkpoints do not impact the critical path. On the one hand, inserting compulsory checkpoints happens before extracting the critical path, which means we have considered the impact of compulsory checkpoints. On the other hand, the proposed strategy determines how to place optional checkpoints to achieve the minimum expected execution time on the critical path. However, the optional checkpoints are inserted coordinately in all processes; so they globally increase the execution time and equally affect all paths.

The proposed approach does not assume a deterministic number of faults. This work considers a probabilistic fault model that we cannot foresee the number of faults in advance. Considering the probabilistic fault model is more general because it encourages the proposed checkpointing strategy to be robust in a random environment. While, in contrast, addressing deterministic fault models (i.e., k-fault tolerance scenario) only guarantees performance in a limited number of faults. The k-fault model is not suitable in complex systems especially when the fault rate is high because a k-fault tolerant system drains out tolerance very soon and needs special treatment in such a high fault rate environment. The proposed strategy minimizes the total expected execution time, thus, leading to a higher probability of meeting deadlines than other strategies. All other strategies result in longer execution times and lower rates of meeting deadlines. The experimental results also show that the proposed strategy promises the shortest average execution time. In addition, we consider an overall deadline for all tasks instead of individual deadlines for different tasks.

The proposed strategy addresses transient faults. The proposed strategy addresses the transient fault under a certain distribution, and the permanent fault is out of the scope of this paper. To address permanent faults, the system must be designed with redundancy; thus, when permanent faults occur, it can migrate tasks from faulty processes to intact ones and establish new dependencies with new messages. The proposed approach can then be applied to ensure fault tolerance and minimize overall execution time. This migration of tasks can be done manually or automatically, depending on the system's configuration and the nature of the fault. Once the migration is complete, the proposed approach can be used to partition dependent processes into a DAG graph, identify the critical path, and optimize the number and intervals of checkpoints to minimize the impact of faults and ensure timely recovery.

8 CONCLUSION

The main contribution of the paper is the consideration of both logical consistency and timing correctness during checkpoint placement in real-time systems. We first partition processes with complex dependencies into a DAG, during which we place some compulsory checkpoints to guarantee logical consistency and avoid much useful work waste. Then we extract the longest critical path to analyze timing correctness. Finally, we build a model to illustrate how to minimize each task's execution time in the critical path to achieve minimum total execution time. Four simulations and a case study show the necessity to consider both logical and timing correctness, and our strategy performs the best among prior works and baselines.

ACKNOWLEDGMENTS

This work was supported in part by NSF CNS-2143256. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation (NSF).

REFERENCES

[1] Chuadhry Mujeeb Ahmed and Jianying Zhou. 2020. Challenges and Opportunities in Cyberphysical Systems Security: A Physics-Based Perspective. *IEEE Security & Privacy* 18, 6 (2020), 14–22. https://doi.org/10.1109/MSEC.2020.3002851

[2] Rasim Alguliyev, Yadigar Imamverdiyev, and Lyudmila Sukhostat. 2018. Cyber-physical systems and their security issues. *Computers in Industry* 100 (2018), 212–223. https://doi.org/10.1016/j.compind.2018.04.017

[3] Lorenzo Alvisi, Elmootazbellah Elnozahy, Sriram Rao, Syed Amir Husain, and Asanka De Mel. 1999. An analysis of communication induced checkpointing. In *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No. 99CB36352)*. IEEE, 242–249.

[4] Mohsen Ansari, Sepideh Safari, Heba Khdr, Pourya Gohari-Nazari, Jörg Henkel, Alireza Ejlali, and Shaahin Hessabi. 2022. Power-Aware Checkpointing for Multicore Embedded Systems. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (2022), 4410–4424.

[5] Reza Arghandeh, Alexandra von Meier, Laura Mehrmanesh, and Lamine Mili. 2016. On the Definition of Cyber-Physical Resilience in Power Systems. *Renewable and Sustainable Energy Reviews* 58 (2016), 1060–1069. https://doi.org/10.1016/j.rser.2015.12.193

[6] R. Baldoni, J. . Helary, A. Mostefaoui, and M. Raynal. 1997. A communication-induced checkpointing protocol that ensures rollback-dependency trackability. In *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*. 68–77. https://doi.org/10.1109/FTCS.1997.614079

[7] Guohong Cao and Mukesh Singhal. 1998. On coordinated checkpointing in distributed systems. *IEEE Transactions on Parallel and Distributed Systems* 9, 12 (1998), 1213–1225.

[8] Anirban Chakraborty, Manaar Alam, Vishal Dey, Anupam Chattopadhyay, and Debdeep Mukhopadhyay. 2018. Adversarial attacks and defences: A survey. *arXiv preprint arXiv:1810.00069* (2018).

[9] Silvia Colabianchi, Francesco Costantino, Giulio Di Gravio, Fabio Nonino, and Riccardo Patriarca. 2021. Discussing resilience in the context of cyber physical systems. *Computers & Industrial Engineering* 160 (2021), 107534. https://doi.org/10.1016/j.cie.2021.107534

[10] CRIU. 2022. Checkpoint/Restore In Userspace (CRIU). https://criu.org/Main_Page

[11] Sheng Di, Yves Robert, Frédéric Vivien, Derrick Kondo, Cho-Li Wang, and Franck Cappello. 2013. Optimization of Cloud Task Processing with Checkpoint-Restart Mechanism. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '13)*. Association for Computing Machinery, New York, NY, USA, Article 64, 12 pages. https://doi.org/10.1145/2503210.2503217

[12] Wenli Duo, MengChu Zhou, and Abdullah Abusorrah. 2022. A Survey of Cyber Attacks on Cyber Physical Systems: Recent Advances and Challenges. *IEEE/CAA Journal of Automatica Sinica* 9, 5 (2022), 784–800. https://doi.org/10.1109/JAS.2022.105548

[13] Alireza Ejlali, Bashir M Al-Hashimi, and Petru Eles. 2009. A standby-sparing technique with low energy-overhead for fault-tolerant hard real-time systems. In *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*.

[14] Elmootazbellah Nabil Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B Johnson. 2002. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)* (2002).

[15] Elmootazbellah N Elnozahy and James S Plank. 2004. Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery. *IEEE Transactions on Dependable and Secure Computing* 1, 2 (2004), 97–108.

[16] Francesco Flammini. 2019. Resilience of cyber-physical systems. *Springer* (2019).

[17] Erol Gelenbe and D Derochette. 1978. Performance of rollback recovery systems under intermittent failures. *Commun. ACM* 21, 6 (1978), 493–499.

[18] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello. 2011. Uncoordinated Checkpointing Without Domino Effect for Send-Deterministic MPI Applications. In *2011 IEEE International Parallel Distributed Processing Symposium*. 989–1000. https://doi.org/10.1109/IPDPS.2011.95

[19] Yifeng Guo, Dakai Zhu, and Hakan Aydin. 2013. Generalized standby-sparing techniques for energy-efficient fault tolerance in multiprocessor real-time systems. In *2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE, 62–71.

[20] Mohammad A Haque, Hakan Aydin, and Dakai Zhu. 2016. On reliability management of energy-aware real-time systems through task replication. *IEEE Transactions on Parallel and Distributed Systems* 28, 3 (2016), 813–825.

[21] Haibo He and Jun Yan. 2016. Cyber-physical attacks and defences in the smart grid: a survey. *IET Cyber-Physical Systems: Theory & Applications* (2016).

[22] Justin CY Ho, Cho-Li Wang, and Francis CM Lau. 2008. Scalable group-based checkpoint/restart for large-scale message-passing systems. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 1–12.

[23] Bran Selic Shiping Chen Ifeanyi P. Egwutuoha, David Levy. 2013. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing* (2013).

[24] Viacheslav Izosimov, Paul Pop, Petru Eles, and Zebo Peng. 2005. Design optimization of time-and cost-constrained fault-tolerant distributed embedded systems. In *Design, Automation and Test in Europe*. IEEE.

[25] Bentolhoda Jafary, Lance Fiondella, and Ping-Chen Chang. 2020. Optimal equidistant checkpointing of fault tolerant systems subject to correlated failure. *Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability* 234, 4 (2020), 636–648.
[26] N. Kaio, T. Dohi, and K. S. Trivedi. 2002. Availability Models with Age-Dependent Checkpointing. In *Reliable Distributed Systems, IEEE Symposium on*. IEEE Computer Society, Los Alamitos, CA, USA, 130. https://doi.org/10.1109/RELDIS.2002.1180181
[27] Donald E Knuth. 2014. In *Art of computer programming, volume 2: Seminumerical algorithms*. Addison-Wesley Professional.
[28] Fanxin Kong, Meng Xu, James Weimer, Oleg Sokolsky, and Insup Lee. 2018. Cyber-Physical System Checkpointing and Recovery. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*. 22–31. https://doi.org/10.1109/ICCPS.2018.00011
[29] Seong Woo Kwak, Byung Jae Choi, and Byung Kook Kim. 2001. An optimal checkpointing-strategy for real-time control systems under transient faults. *IEEE Transactions on reliability* 50, 3 (2001), 293–301.
[30] Rami Melhem, Daniel Mosse, and Elmootazbellah Elnozahy. 2004. The interplay of power management and fault recovery in real-time systems. *IEEE Trans. Comput.* 53, 2 (2004), 217–231.
[31] Matthias Pflanz and Heinrich Theodor Vierhaus. 1998. Generating reliable embedded processors. *IEEE Micro* (1998).
[32] Claudio Pinello, Luca P Carloni, and Alberto L Sangiovanni-Vincentelli. 2004. Fault-tolerant deployment of embedded software for cost-sensitive real-time feedback-control applications. In *Design, Automation and Test in Europe*. IEEE.
[33] Dhiraj K Pradhan. 1996. *Fault-tolerant computer system design*. Prentice-Hall, Inc.
[34] Sasikumar Punnekkat, Alan Burns, and Robert Davis. 2001. Analysis of checkpointing for real-time systems. *Real-Time Systems* 20, 1 (2001), 83–102.
[35] Siva Satyendra Sahoo, Bharadwaj Veeravalli, and Akash Kumar. 2020. Markov Chain-based Modeling and Analysis of Checkpointing with Rollback Recovery for Efficient DSE in Soft Real-time Systems. In *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE, 1–6.
[36] Mohammad Salehi, Mohammad Khavari Tavana, Semeen Rehman, Muhammad Shafique, Alireza Ejlali, and Jörg Henkel. 2016. Two-state checkpointing for energy-efficient fault tolerance in hard real-time systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24, 7 (2016), 2426–2437.
[37] Kang G Shin, Tein-Hsiang Lin, and Yann-Hang Lee. 1987. Optimal checkpointing of real-time tasks. *IEEE Transactions on computers* 100, 11 (1987), 1328–1341.
[38] Yi-Min Wang, Pi-Yu Chung, In-Jen Lin, and W. Kent Fuchs. 1995. Checkpoint space reclamation for uncoordinated checkpointing in message-passing systems. *IEEE Transactions on Parallel and Distributed Systems* 6, 5 (1995), 546–554.
[39] Marilyn Wolf and Dimitrios Serpanos. 2017. Safety and security in cyber-physical systems and internet-of-things systems. *Proc. IEEE* (2017).
[40] Jean-Paul A. Yaacoub, Ola Salman, Hassan N. Noura, Nesrine Kaaniche, Ali Chehab, and Mohamad Malli. 2020. Cyber-physical systems security: Limitations, issues and future trends. *Microprocessors and Microsystems* 77 (2020), 103201. https://doi.org/10.1016/j.micpro.2020.103201
[41] D. Manivannan Yi Luo. 2013. Theoretical and experimental evaluation of communication-induced checkpointing protocols in FE and FLazy-E families, Performance Evaluation. *Performance Evaluation* (2013).
[42] Luyuan Zeng, Pengcheng Huang, and Lothar Thiele. 2016. Towards the design of fault-tolerant mixed-criticality systems on multicores. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. 1–10.
[43] Lin Zhang, Xin Chen, Fanxin Kong, and Alvaro A. Cardenas. 2020. Real-Time Attack-Recovery for Cyber-Physical Systems Using Linear Approximations. In *2020 IEEE Real-Time Systems Symposium (RTSS)*. 205–217. https://doi.org/10.1109/RTSS49844.2020.00028
[44] Lin Zhang, Pengyuan Lu, Fanxin Kong, Xin Chen, Oleg Sokolsky, and Insup Lee. 2021. Real-Time Attack-Recovery for Cyber-Physical Systems Using Linear-Quadratic Regulator. *ACM Trans. Embed. Comput. Syst.* 20, 5s, Article 79 (sep 2021), 24 pages. https://doi.org/10.1145/3477010
[45] Lin Zhang, Kaustubh Sridhar, Mengyu Liu, Pengyuan Lu, Xin Chen, Fanxin Kong, Oleg Sokolsky, and Insup Lee. 2023. Real-Time Data-Predictive Attack-Recovery for Complex Cyber-Physical Systems. In *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*.
[46] Lin Zhang, Zifan Wang, Mengyu Liu, and Fanxin Kong. 2022. Adaptive Window-Based Sensor Attack Detection for Cyber-Physical Systems. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. Association for Computing Machinery, New York, NY, USA, 919–924. https://doi.org/10.1145/3489517.3530555
[47] Ying Zhang and Krishnendu Chakrabarty. 2004. Task feasibility analysis and dynamic voltage scaling in fault-tolerant real-time embedded systems. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Vol. 2. IEEE, 1170–1175.
[48] Dakai Zhu. 2006. Reliability-aware dynamic energy management in dependable embedded real-time systems. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium*.