Received: 3 December 2018 | Revised: 5 July 2019 | Accepted: 14 August 2019

DOI: 10.4218/etrij.2018-0684

ORIGINAL ARTICLE

ETRI Journal WILEY

Combining replication and checkpointing redundancies for reducing resiliency overhead

Hassan Motallebi

Department of Electrical and Computer Engineering, Graduate University of Advanced Technology, Kerman, Iran

Correspondence
Hassan Motallebi, Department of Electrical and Computer Engineering, Graduate University of Advanced Technology (GUAT), Kerman, Iran.
Email: h.motallebi@kgut.ac.ir

Abstract

We herein propose a heuristic redundancy selection algorithm that combines resubmission, replication, and checkpointing redundancies to reduce the resiliency overhead in fault-tolerant workflow scheduling. The appropriate combination of these redundancies for workflow tasks is obtained in two consecutive phases. First, to compute the replication vector (number of task replicas), we apportion the set of provisioned resources among concurrently executing tasks according to their needs. Subsequently, we obtain the optimal checkpointing interval for each task as a function of the number of replicas and characteristics of tasks and computational environment. We formulate the problem of obtaining the optimal checkpointing interval for replicated tasks in situations where checkpoint files can be exchanged among computational resources. The results of our simulation experiments, on both randomly generated workflow graphs and real-world applications, demonstrated that both the proposed replication vector computation algorithm and the proposed checkpointing scheme reduced the resiliency overhead.

KEYWORDS

concurrency graph, extended upward rank, fault-tolerant scheduling, hybrid redundancy, resiliency overhead

1 | INTRODUCTION

Fault tolerance has become an indispensable requirement in the execution of *workflow* applications on large-scale distributed platforms. In some certain situations, the mean time between failures of computational resources may not exceed a few hours [1]. For a resilient application execution, fault-tolerance techniques have been widely used: *resubmission*, *replication*, and *checkpointing* [2]. Using these basic redundancies may result in an unacceptable increase in the execution time and/or cost. Resubmission redundancy may result in unacceptable time overhead; especially for long-running tasks that may encounter multiple successive failures during execution, a significant amount of time may be consumed. Checkpointing redundancy is an improved version of the resubmission technique, in which the execution state is saved to persistent storage at different points, for example, periodically and when a failure occurs, the execution is resumed from the most recently saved state and not from the beginning [3].

Meanwhile, in replication redundancy, if sufficient resources are available for concurrent execution of task replicas and one replica succeeds at the least, the task succeeds with almost no time overhead. However, concurrent execution of task replicas is not always possible because (i) few resources may be provisioned (for cost reduction) and (ii) a large number of tasks may be executing concurrently in some duration.

This is an Open Access article distributed under the term of Korea Open Government License (KOGL) Type 4: Source Indication + Commercial Use Prohibition + Change Prohibition (http://www.kogl.or.kr/info/licenseTypeEn.do).
1225-6463/$ © 2020 ETRI

Additionally, replication redundancy may result in an unacceptable overprovisioning cost.

As each of these basic redundancies has its own drawbacks, recent studies have combined these redundancies to reduce the overhead [4–6]. In [4,5], a *hybrid* of replication and resubmission redundancies is proposed. In [6], a hybrid of replication and checkpointing is proposed assuming that checkpoint files can be exchanged among resources. In such a situation, once the execution of a replica fails, the most recent checkpoint file is requested from other resources. If even one of these replicas succeeds in a checkpointing interval, other replicas can have its result and proceed further.

We herein propose a hybrid redundancy selection algorithm based on combining resubmission, replication, and checkpointing redundancies. The problem of obtaining the appropriate combination of redundancies is divided into two subproblems: (i) first, we decide the set of tasks that are replicated and the number of replicas of each task; and (ii) subsequently, we determine the appropriate checkpointing scheme for tasks that require checkpointing. For small tasks whose execution time is less than the optimum checkpointing interval, we use the resubmission and replication techniques. That is, once a replica of such a task fails, we restart the execution on the same resource; when one of the replicas of the task succeeds, we cancel the execution of the other ones.

In using the checkpointing redundancy, we determine the *optimal checkpointing interval* [6–9]. If checkpoint savings are extremely frequent, the *failure-free overhead* (overhead in the absence of failures) will be extremely high; if checkpoints are extremely infrequent, a large amount of time will be wasted owing to re-computations of failed tasks.

We encounter a form of this problem, where multiple, say k, replicas of a task are being executed concurrently and checkpoint files are exchanged among the resources upon request. If q denotes the probability of failure in one interval, the probability of failure for all replicas is q^k. As k increases, the optimal interval length increases. The effect of varying the checkpointing interval on the average execution time of a task when k replicas of the task are being executed is shown in Figure 1, for $k = 1, 2, 3$. The results show that the optimum value for checkpointing interval increases with k. Thus, when idle resources are available, we can increase the checkpointing interval (and reduce the failure-free overhead) by increasing the number of task replicas. Furthermore, according to Figure 1, as the number of replicas increases, the deviation from the optimal interval length has a less significant impact on the redundancy overhead.

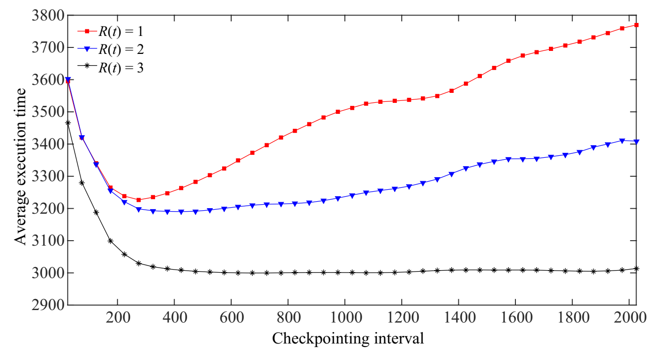

FIGURE 1 Effect of varying checkpointing interval on average execution time of a long-running task in case of replication redundancy with k replicas for $k = 1, 2, 3$

1.1 | Motivation and contribution

To reduce the overhead, we propose a hybrid redundancy selection approach based on combining resubmission, replication, and checkpointing redundancies. In the proposed approach, the redundancy selection is performed in two phases: (i) Computing an appropriate replication vector such that the processing capacity of all resources is used for task replication, and task replicas can be executed almost concurrently. (ii) Determining the optimal checkpointing interval of each task as a function of number of replicas, characteristics of tasks, and computational environment.

The basic idea of the proposed approach is as follows: First, idle durations of resources are found according to the graph structure and concurrency relations among tasks. Subsequently, these idle durations are used for increasing the number of replicas of the most competent tasks. Increasing the number of replicas of tasks decreases the resiliency overhead in two aspects: (i) It increases the probability of success in each interval, and because only idle durations of resources are used for task replication, the waste of resources is minimized. (ii) As the number of replicas of a task increases, the optimal checkpointing interval increases; thus, the failure-free overhead decreases.

Our contribution is twofold: First, we propose a redundancy selection algorithm, in which the number of replicas of tasks is determined with respect to the number of available resources and the number of replicas of concurrently executing tasks. To obtain the set of tasks that are likely to be executed concurrently, we introduce the concept of *concurrency graph* and propose a graph-theory–based algorithm for computing this graph. The number of replicas of tasks is determined such that (i) all resources are used for executing additional replicas of tasks, (ii) more replicas are considered for more competent tasks, and (iii) the summation of the number of replicas of concurrently executing tasks is maintained less than or equal to the number of resources. This guarantees that task replicas can be executed almost concurrently. In the proposed *Extended Upward Rank Based* (*EURB*) algorithm, provisioned resources are

apportioned among concurrently executing tasks according to the *Extended Upward Rank* (*EUR*) metric, which indicates the extent to which each task benefits from having additional replicas. Next, we formulate and address the problem of obtaining the optimal checkpointing interval in situations where checkpoint files can be exchanged among resources. Although a similar problem is addressed in [6], it is assumed that once a replica fails, it cannot be resumed immediately but has to wait until the next checkpointing time to restart. This assumption can be unreasonable especially in situations involving long checkpointing intervals. Hence, we address the problem herein in a more general form.

Section 2 describes our model of the application and the execution environment. In Section 3, we review some related studies. In Section 4, we present the proposed fault-tolerant approach. In Section 5, a comparison of the proposed algorithm with some existing algorithms is presented. Finally, Section 6 provides concluding remarks.

2 | SYSTEM MODEL

A precedence-constrained application is defined as follows:

Definition 1 (Workflow)
A *workflow* is a weighted directed acyclic graph (DAG) $\mathcal{W}=(\mathcal{T},\mathcal{D})$, where $\mathcal{T}=\{t_1, \ldots, t_n\}$ is the set of *tasks* and $\mathcal{D}$ is the set of *task dependencies*. The task size is assigned to each task, and the data size that is transferred from t_i to t_j is assigned to each dependency.

Using an ad hoc java-based simulator, we simulated a cloud service provider that offers several virtualized resources $S=\{s_1, \ldots, s_m\}$ with different quality of service (QoS) parameters, such as CPU type and memory size, as well as different prices as in [14]. The environment settings is based on four parameters: the number $m = 5$, average processing capacity P_{avg}, average failure rate λ_{avg}, and average cost per billing period C_{avg} of computational resources. The processing capacities of the resources is uniformly selected from $[0.5P_{avg}, 2P_{avg}]$ MIPS, that is, the fastest resource is approximately four times faster (and approximately four times more expensive) than the slowest. The billing period is assumed to be half an hour and $C_{avg} = 0.5$ \$. The user is charged based on the number of time periods and is charged fully for the last time interval even if he uses only a small fraction of it. We consider only sequential tasks, especially long-running ones, whose execution time is comparable to the mean time-to-failure of computational resources. We used additional resources not for parallel execution of tasks but for fault tolerance. Considering *silent errors* yields a completely different situation; therefore, in this study, we consider only *fail-stops* [27]. As in [3], the time-to-failure of resources is assumed to follow an exponential distribution with rate λ, which follows a normal distribution $N(\lambda_{avg} = 7.5 \times 10^{-7}, \sigma = 2.0 \times 10^{-7})$.

3 | RELATED STUDIES

Most studies pertaining to this area assumed a *fault-free* environment. However, some studies have considered *fault-prone* situations. While a large number of studies used only the resubmission technique, some researchers, such as [5,10], used the replication technique. In [10], a replication-based extension of the HEFT algorithm [11] is proposed, where k replicas of each task are scheduled instead of one. Having the same number of replicas for all tasks may result in an unacceptable overhead. Thus, in [5,12], a method is proposed for determining the number of replicas of each task with respect to the characteristics of the task. Furthermore, in [13], a graph-based method is proposed for replicating tasks. Studies for guaranteeing or improving scheduling QoS have been performed [14–16]. To reduce the effect of resource performance fluctuations, the algorithm in [4] uses idle durations of resources for task replication.

As each of the basic redundancies has its own drawbacks and merits, some researchers have attempted to obtain the appropriate combination of those redundancies [5,12,17–19]. In [5], based on combining replication and resubmission redundancies, the *Resubmission Impact* (*RI*) heuristic is proposed. The number of replicas of each task in the *RI* heuristic is determined according to how much workflow termination is delayed if resubmission redundancy is considered for that task. Furthermore, [17] considered a hybrid redundancy based on combining resubmission and checkpointing redundancies.

A well-studied problem is that of obtaining the optimal checkpoint interval that minimizes the average execution time [3,20–24]. The approximation of the optimum checkpoint interval was initiated by Young [7]. Young established a simple mathematical equation for obtaining the optimal checkpoint interval. Furthermore, in [8], a more accurate approximation is proposed. None of the abovementioned studies considered replicated tasks. Rahman and others [6] proposed a mathematical model for minimizing the average execution time for multiple interdependent parallel processes in the Volpex environment [25], each possibly with multiple replicas. In that study, it is assumed that once a replica fails in an interval, the resource receives the checkpoint file from other resources processing other replicas of the task. Another assumption, which simplifies the problem significantly, is that when a replica fails, it cannot be resumed immediately but has to wait until the next checkpointing to restart.

In some studies, in addition to fail-stops, silent errors are considered [23,26–29]. In [27], disk checkpointing is combined with memory checkpointing. In [30], the problem of obtaining the optimal checkpointing scheme is examined for cases involving checkpoint files of different sizes. Finally, in [26], the same problem is studied for different failure behaviors.

4 | REDUNDANCY SELECTION ALGORITHM

In this section, we describe the proposed algorithm for determining the appropriate redundancies for tasks in a workflow. We separate the redundancy selection problem into two subproblems: First, the *replication vector*, $\mathcal{R}(.)$, is computed. $\mathcal{R}(t_i)$, the i_{th} element of $\mathcal{R}$, which denotes the number of replicas of t_i ($i = 1, \ldots, n$), is computed with respect to the number of provisioned resources, number of replicas of concurrently executing tasks, and the extent to which each task benefits from having additional replicas. Once the replication vector, $\mathcal{R}(.)$, is computed, in the next phase, the optimal checkpointing interval of each task t_i (if necessary) is computed as a function of $\mathcal{R}(t_i)$, failure rate, and checkpoint *saving* and *recovery* costs. In the remainder of this section, the proposed method for computing the replication vector will be explained, followed by the description of the technique for obtaining the checkpointing interval $\mathcal{I}(t_i)$ for each task t_i. Table 1 lists the notation used in our discussion.

TABLE 1 Notation used in explaining the proposed algorithm

Notation	Description
$\mathcal{W} = (\mathcal{T}, \mathcal{D})$	Workflow
$\mathcal{T} = \{t_1, \ldots, t_n\}$	Set of workflow tasks
$\mathcal{T}_i$	Execution time of t_i
$\mathcal{R}(t_i)$	Number of replicas of t_i
λ	Average failure rate
$\mathcal{I}(t_i)$	Optimal checkpointing interval for t_i

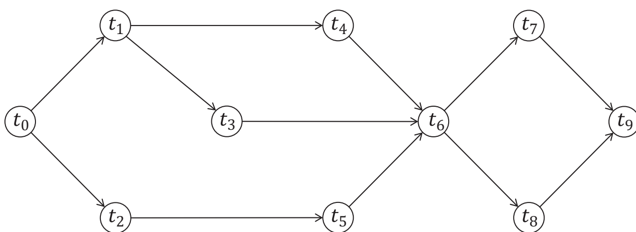

FIGURE 2 Example of workflow application

4.1 | Computing the redundancy scheme

The main idea of the proposed algorithm for computing the replication vector is that one replica for each task is considered initially. Subsequently, while some resources are idle, we increase the number of replicas of the most competent task among the set of tasks whose number of replicas can be increased. To maximize the gain, in each round, we increase the number of replicas of the task that benefits the most from an additional replica. The number of replicas of t_i can be increased if one idle resource at the least exists during the execution of t_i. Assume m resources are provisioned and apportioned among the set of concurrently executing tasks. To verify whether idle resource(s) exist during the execution of t_i, we should verify if the summation of the number of replicas of concurrently executing tasks is less than m. Hence, we introduce the concept of concurrency graph according to which we obtain the concurrency relations among tasks.

4.1.1 | Concurrency graph

Consider a workflow $\mathcal{W} = (\mathcal{T}, \mathcal{D})$ with the set $\mathcal{T}$ of tasks and the relation $\mathcal{D}$ denoting the dependencies among them. Assume that the *transitive closure* of $\mathcal{D}$ is denoted by $\mathcal{D}^*$. For a pair of tasks t_i and t_j, if we have either $(t_i, t_j) \in \mathcal{D}^*$ or $(t_j, t_i) \in \mathcal{D}^*$, then t_i and t_j may not be executed concurrently. Thus, t_i and t_j are likely to be executed concurrently if and only if

$$(t_i, t_j) \in C = \overline{\mathcal{D}^* \cup \mathcal{D}^{*T}} \tag{1}$$

where R^T and $\overline{R}$ are the *transpose* and *complement* of R, respectively. We use the terms *concurrency relation* and *concurrency graph* to refer to the relation C and the graph $\mathcal{P} = (\mathcal{T}, C)$, respectively. As an example, the concurrency graph of the workflow in Figure 2 is shown in Figure 3.

4.1.2 | Algorithm

The concurrency graph, once computed, is decomposed into the set $CC = \{c_1, \ldots, c_k\}$ of its *connected components*. For example, consider the workflow graph of Figure 2, whose concurrency graph (Figure 3) is composed of five connected components. As tasks in a component cannot be executed concurrently with tasks in other components, the problem of selecting the appropriate redundancies for tasks in a workflow is separated in a divide-and-conquer manner into the same problem for each of its connected components.

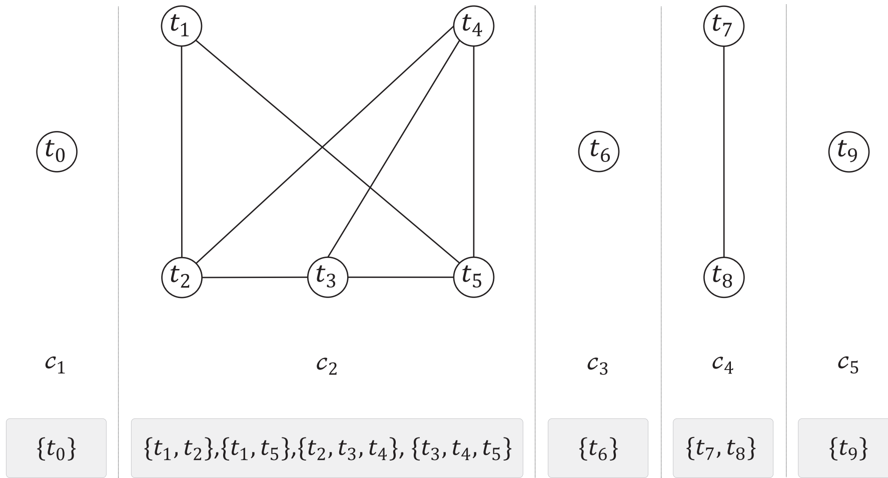

FIGURE 3 Concurrency graph of workflow in Figure 2

Thus, the appropriate redundancies for each connected component can be selected separately and independently. Next, we explain the proposed algorithm for computing the replication vector for a connected component. In this algorithm, the available resources are apportioned among concurrent tasks, and more replicas and resources are assigned to more competent tasks. This means that the summation of the number of replicas of concurrently executing tasks should be maintained less than or equal to the number of resources. As we need to obtain the maximal set of tasks that may be executed concurrently, we obtain the set of *maximal cliques* [32] in each connected component. For example, consider the connected component $c_2 = \{t_1, \ldots, t_5\}$ in Figure 3, which has the following set of maximal cliques:

$$Q = \{q_1 = \{t_1, t_2\}, q_2 = \{t_1, t_5\}, q_3 = \{t_2, t_3, t_4\}, \\ q_4 = \{t_3, t_4, t_5\}, q_5 = \{t_4, t_5\}\}. \quad (2)$$

Now, we describe the algorithm for the general case. Assume c is a connected component whose set of maximal cliques is denoted by Q. To obtain the replication vector for c, we first consider one replica for each task and subsequently compute the set $\mathcal{T}_{\text{possible}}$ of tasks, whose number of replicas can be increased as follows:

$$\mathcal{T}_{\text{possible}} = \left\{ t_i \in c \mid \forall q \in Q \bullet \left(t_i \in q \Rightarrow \sum_{t_j \in q} \mathcal{R}(t_j) < m \right) \right\}. \quad (3)$$

Now, while the set $\mathcal{T}_{\text{possible}}$ is nonempty, we obtain the most competent task t_{best} (Section 4.1.4) and increase $\mathcal{R}(t_{\text{best}})$. Subsequently, the set $\mathcal{T}_{\text{possible}}$ is recomputed and the procedure is repeated until $\mathcal{T}_{\text{possible}}$ becomes empty. Table 2 shows the steps of the proposed algorithm, applied to the connected component c_2 of the concurrency graph in Figure 3. The proposed replication vector computation procedure is given in Algorithm 4. In the first two lines, the concurrency graph of $\mathcal{W}$ is computed and decomposed into its connected

Algorithm 1 EURB Redundancy Selection Algorithm

Input: $\mathcal{W} = (\mathcal{T}, \mathcal{D})$: Workflow graph, m: Number of provisioned resources
Output: $\mathcal{R}$: Replication vector, $\mathcal{I}$: optimal checkpointing interval
$\mathcal{P} \leftarrow$ the concurrency graph $(\mathcal{T}, \mathcal{C})$ of $\mathcal{W}$
$\mathcal{CC} \leftarrow$ the set $\{c_1, \ldots, c_k\}$ of connected components of $\mathcal{P}$
for e **do**ach connected component $c_r \in \mathcal{C}$ do
  $Q \leftarrow$ the set of maximal cliques in c_r
  for do$t_i \in c_r$ do
    $\mathcal{R}(t_i) \leftarrow 1$
  end for
  $\mathcal{T}_{\text{possible}} \leftarrow \left\{ t_i \middle| \forall q \in Q \bullet (t_i \in q \Rightarrow \sum_{t_j \in q} \mathcal{R}(t_j) \leq m) \right\}$
  while do$\mathcal{T}_{t\text{possible}} \neq \emptyset$
    $t_{\text{best}} \leftarrow$ the most competent task in $\mathcal{T}_{\text{possible}}$ (according to Section 4.1.4)
    $\mathcal{R}(t_{\text{best}}) \leftarrow \mathcal{R}(t_{\text{best}}) + 1$
    $\mathcal{T}_{\text{possible}} \leftarrow \left\{ t_i \middle| \forall q \in Q \bullet (t_i \in q \Rightarrow \sum_{t_j \in q} \mathcal{R}(t_j) \leq m) \right\}$
  end while
end for
for e **do**ach task $t_i \in \mathcal{T}$ do
  $\mathcal{I}(t_i) \leftarrow$ the optimal checkpointing interval for t_i (according to Section 4.2)
end for
return $\mathcal{R}(.), \mathcal{I}(.)$

components. In the loop iteration of lines 3–14, each connected component c_r is processed independently. In line 4, we use the Bron-Kerbosch algorithm [32] to obtain the set of maximal cliques in c_r. In the loop in lines 5–7, the number of replicas of all tasks is set to 1. In line 8, the set $\mathcal{T}_{\text{possible}}$ is computed. Subsequently, in the loop in lines 9–13, while the set $\mathcal{T}_{\text{possible}}$ is nonempty in lines 10 through 12, the number of replicas of the most competent task (according to Section 4.1.4) t_{best}, $\mathcal{R}(t_{\text{best}})$ is incremented, and the set $\mathcal{T}_{\text{possible}}$ is recomputed. Finally, in lines 15–17, the optimal checkpointing

TABLE 2 Steps of replication vector computation algorithm for connected component c_2 of the concurrency graph in Figure 3

$\mathcal{R}(t_i)$						Summation of number of replicas of tasks in cliques			
t_1	t_2	t_3	t_4	t_5	$\mathcal{T}_{\text{possible}}$	$\{t_2, t_3, t_4\}$	$\{t_3, t_4, t_5\}$	$\{t_1, t_2\}$	$\{t_1, t_5\}$
1	1	1	1	1	$\{t_1,t_2,t_3,t_4,t_5\}$	3	3	2	2
1	1	2	1	1	$\{t_1,t_2,t_3,t_4,t_5\}$	4	4	2	2
2	1	2	1	1	$\{t_1,t_2,t_3,t_4,t_5\}$	4	4	3	3
3	1	2	1	1	$\{t_1,t_2,t_3,t_4,t_5\}$	4	4	4	4
3	2	2	1	1	$\{t_1,t_5\}$	5	4	4	4
3	2	2	1	2	$\{\}$	4	5	4	5

interval $\mathcal{I}(t_i)$ for each t_i is determined according to Section 4.2.

4.1.3 | Extended upward rank metric

To increase the number of replicas of the most competent task, a metric is required to determine which task benefits the most from an additional replica. In the following, we first introduce the concept of *extended upward rank*, $\text{rank}_{\text{eu}}(.)$; subsequently, we use this metric for determining the most competent task. The definition of the extended upward rank ($\text{rank}_{\text{eu}}(.)$) metric is based on the concept of upward rank [11], which is recursively defined as follows:

$$\text{rank}_u(t_i) = \begin{cases} T_i, & \text{if } t_i \text{ is an exit node,} \\ T_i + \max_{t_j \in \text{Succ}(t_i)} (c_{ij} + \text{rank}_u(t_j)) & \text{else,} \end{cases} \tag{4}$$

where c_{ij} is the average communication cost of edge (t_i, t_j) and T_i is the average execution time of t_i on a fault-free *reference resource* [11]. The reference resource is a virtual resource whose processing capacity is equal to the average processing capacity of all provisioned resources. In fact, $\text{rank}_u(t_i)$ is our estimation of the execution time of the longest path from t_i to the existing task, including the completion time of t_i. We define the extended upward rank of t_i, $\text{rank}_{\text{eu}}(t_i)$ as a function of T_i, $\mathcal{R}(t_i)$ and the average failure rate.

Definition 2 [Extended Upward Rank (EUR)]
The extended upward rank, $\text{rank}_{\text{eu}}(t_i,R(t_i))$ of a task t_i is

$$\begin{cases} \mathcal{E}(t_i,\mathcal{R}(t_i)), & \text{if } t_i \text{ is an exit node,} \\ \mathcal{E}(t_i,\mathcal{R}(t_i)) + \max_{t_j \in \text{Succ}(t_i)} (c_{ij} + \text{rank}_{\text{eu}}(t_j)) & \text{else,} \end{cases} \tag{5}$$

where $\mathcal{E}(t_i, \mathcal{R}(t_i))$, the average execution time of t_i on a fault-prone reference resource is computed as follows.

The probability of failure in each execution trial is $1 - e^{-\lambda T_i}$. Thus, when $\mathcal{R}(t_i)$ replicas of the task are being processed concurrently, the probability of failure in all these replicas is $(1 - e^{-\lambda T_i})^{\mathcal{R}(t_i)}$. Because the number of execution trials until success follows a geometric distribution, the time required for the successful execution of a task is.

$$\mathcal{E}(t_i,\mathcal{R}(t_i)) = T_i + \left(\frac{1}{(1 - e^{-\lambda T_i})^{\mathcal{R}(t_i)}} - 1 \right) \times E\left[\max\left(T_{f,1}, \ldots, T_{f,\mathcal{R}(t_i)}\right)\right] \tag{6}$$

where identically distributed random variables $T_{f,1}$ through $T_{f,\mathcal{R}(t_i)}$ denote the time-to-failure of the resources executing the 1_{st} to $\mathcal{R}(t_i)_{\text{th}}$ replica of t_i, respectively.

4.1.4 | Obtaining the most competent task

Next, we describe the method for obtaining the most competent task among the set $\mathcal{T}_{\text{possible}} = \{t_1, \ldots, t_p\}$ of tasks. First, the amount of decrease in the average execution of the remainder of the workflow that can be obtained by increasing the number of replicas of each task should be considered. When the number of replicas of a tasks t_i is increased from $\mathcal{R}(t_i)$ to $\mathcal{R}(t_i)+1$, $g(t_i)$, the amount of decrease in the EUR of t_i is computed as follows:

$$g(t_i) = \text{rank}_{\text{eu}}\left(t_i, \mathcal{R}(t_i)\right) - \text{rank}_{\text{eu}}\left(t_i, \mathcal{R}(t_i)+1\right). \tag{7}$$

Furthermore, the amount of decrease in the critical path length of the remainder of the application, denoted by $g^*(t_i)$, is computed as follows:

$$g^*(t_i) = \max\left(\max_{t_j \in \mathcal{T}_{\text{possible}}} \text{rank}_{\text{eu}}(t_j, \mathcal{R}(t_j)) \right) - \max\left(\max_{t_j \in \mathcal{T}_{\text{possible}} \setminus t_i} \text{rank}_{\text{eu}}(t_j, \mathcal{R}(t_j)), \text{rank}_{\text{eu}}(t_i, \mathcal{R}(t_i)+1) \right). \tag{8}$$

To compute $\text{gain}(t_i)$, that is, the gain for increasing the number of replicas of t_i, we combine $g(t_i)$ and $g^*(t_i)$ as $\text{gain}(t_i) = \beta g(t_i) + (1-\beta) g^*(t_i)$, where $0 \le \beta \le 1$. In this study, we have $\beta = 0.5$. To obtain the most competent task, we select the task t_{best} with the maximum $\text{gain}(t_i)$.

4.2 | Optimal checkpointing interval

Next, we formulate the problem of obtaining the optimal checkpointing interval for a task t_i. Assume that k replicas of t_i are processed on k resources and when a resource saves a checkpoint file, all other resources are informed regarding the amount of progress such that they can request for the file when required. That is, once a replica fails, it can compare its progress against the others' and obtain the latest checkpoint file if required. For simplicity and without loss of generality, we assumed that the resources were identical in terms of both failure behavior and processing capacity.

Herein, the execution time and lag of t_i are denoted by T_j and L_i, respectively. Consider the problem of obtaining the optimal checkpoint interval, $\mathcal{I}(t_i)$ that minimizes the redundancy overhead owing to rework and checkpoint saving and recovery. Herein, the checkpoint saving and recovery costs of t_i are denoted by r and s, respectively. Assume that at time 0, the execution of all replicas is started and the checkpointing interval is c. The probability of encountering failure in [0, c] for each replica is $q=\int_0^c \lambda e^{-\lambda z}dz=1-p$, where $p=e^{-\lambda c}$. Therefore, the number of successful replicas among k replicas is a binomially distributed random variable with parameters k and p. Thus, the probability of j successful replicas succeeding is

$$P(j)=\begin{pmatrix} m+n \\ m \end{pmatrix} p^j q^{k-j}. \tag{9}$$

Three cases are possible for j, the number of replicas that succeed: (i) $j=k$, the probability that all replicas successfully reach the checkpoint saving state at time c is $P(j=k)=p^k$, in which case the time required for all replicas to save the checkpoint file is $\mathcal{E}_{j=k}(c)=c+s$. (ii) j = 0, the probability of all replicas encountering failure before finishing the interval is $P(j=0)=q^k$, in which case the time wasted owing to failure is approximated by c, that is, $\mathcal{E}_{j=0}(c)=E[\max(T_{f,1},\dots,T_{f,k})]\approx c$. (iii) $0 < j < k$, the probability that some replicas succeed and some fail is $P(0 < j < k) = 1 - p^k - q^k$. In this case, we have $k-j$ failed and j successful replicas, each of which has to send the checkpoint file to $(k-j)/j$ failed resources, as shown in Figure 4. Thus, the time required is as follows:

$$\mathcal{E}_{0<j<k}(c)=c+s+\sum_{j=1}^{k-1}\begin{pmatrix} k \\ j \end{pmatrix} p^j q^{k-j}\frac{\left((k-i)r+i\times\sum_{w=1}^{\frac{k-j}{j}}(wu+l)\right)}{k}, \tag{10}$$

where u and l are the time required for uploading the checkpoint file and the average latency among resources, respectively. We have

$$\mathcal{E}_{0<j<k}(c)=c+s+\left(r+l-\frac{u}{2}\right)\left(1-p^k-q^k\right)+\sum_{j=1}^{k-1}\begin{pmatrix} k \\ j \end{pmatrix} p^j q^{k-j}\left(\frac{uk}{2j}-\frac{j(r+l)}{k}\right). \tag{11}$$

According to the discussion above, the average time required for all resources to either save or receive the checkpoint file is obtained as follows:

$$\mathrm{ET}(t_i,c)=P(j=0)(\mathcal{E}_{j=0}(c)+r+\mathrm{ET}(t_i,c))+P(0<j<k)\mathcal{E}_{0<j<k}(c)+P(j=k)\mathcal{E}_{j=k}(c). \tag{12}$$

We have

$$\mathrm{ET}(t_i,c)=q^k(E[\max(T_{f,1},\dots,T_{f,k})]+r+\mathrm{ET}(t_i,c))+(1-p^k-q^k)\mathcal{E}_{1\le j\le k-1}(c)+p^k(c+s), \tag{13}$$

where $\mathrm{ET}(t_i,c)$ denotes the time required for processing the replicas until the checkpoint file is saved, assuming that the checkpointing interval is c. The time required for executing the task t_i is obtained as $\mathrm{AET}(t_i,c)=T/c\times\mathrm{ET}(t_i,c)$. For brevity, we omit the extended formula of $\mathrm{AET}(t_i,c)$. To obtain the optimal value $\mathcal{I}(t_i)$ of c for which $\mathrm{AET}(t_i,c)$ is minimized, we obtain the first partial derivative of $\mathrm{AET}(t_i,c)$ with respect to c and obtain the roots using the bisection method. The result is a function of $\mathcal{R}(t_i)$, checkpoint saving and recovery costs, failure rate, and network latency. Additionally, the first-order approximation of the optimal checkpointing interval of each task t_i can be obtained as a function of $\mathcal{R}(t_i)$ and other characteristics of the computational environment. For tasks with different number of replicas, different formulas are obtained. The derivation of these formulas is beyond the scope of this study.

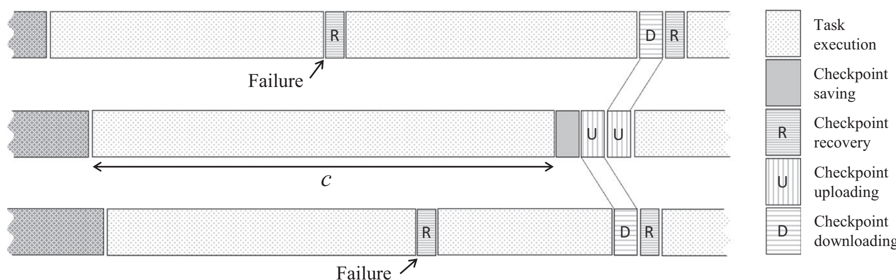

FIGURE 4 Exchanging checkpoint files among resources processing task replicas

5 | PERFORMANCE EVALUATION

In this section, we evaluate the proposed algorithm, and compare its performance against prior methods. The performance of the proposed algorithm is compared against the best combination of some recent studies: the method proposed by Rahman [6] for computing the optimal checkpointing interval combined with the (*RI*) heuristic [5] for obtaining the replication vector. To generate a synthetic workload, we develop a random task graph generation procedure similar to that in [33]. We use the following parameters to generate random task graphs: n, the number of tasks; $W_{mean} = 4.5$ and $W_{max} = 12$, the average and maximum number of tasks in each level of the graph, respectively; $J_{avg} = 2.5$ and $D_{avg} = 2.75$, the average jump and average input (output) degree of nodes, respectively. In addition to random workloads, we use some real-world applications namely Epigenomics, Montage, CyberShake, and SIPHT [31] to assess the performance of the proposed algorithm. The task sizes are uniformly selected such that the execution time of tasks is $[10^3, 10^5]$ s and the task input/output data sizes are uniformly selected from $[100, DS_{max} = 8000]$ MB. The default number of tasks and computational resources in these experiments are 5 and 200, respectively. In the following, we study the effects of different application and environment parameters on the relative performances of the algorithms.

5.1 | Effects of different parameters

To evaluate the relative performance of different redundancy selection algorithms in various situations, we assess the effects of different parameters, namely the number of resources, application size, failure rate, and graph width on the execution time and cost of these algorithms. Furthermore, we perform a comparative study of the performances of different checkpointing schemes.

5.1.1 | Number of resources

Figures 5 and 6 show the effects of varying the number of computational resources, m on the performances of different algorithms. The results show that increasing m to reduce the execution time increases the execution cost. However, increasing the number of provisioned resources is beneficial only if it results in a significant time improvement at the expense of an acceptable cost increase. Therefore, according to Figures 5 and 6, the maximum number of resources in our experiments is set to 7. The results show that especially for cases with fewer resources, the proposed algorithm achieves a significant performance improvement in terms of both time and cost.

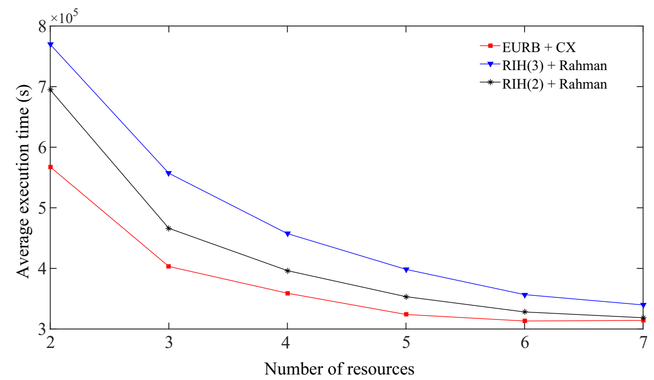

FIGURE 5 Effect of number of resources on average execution time

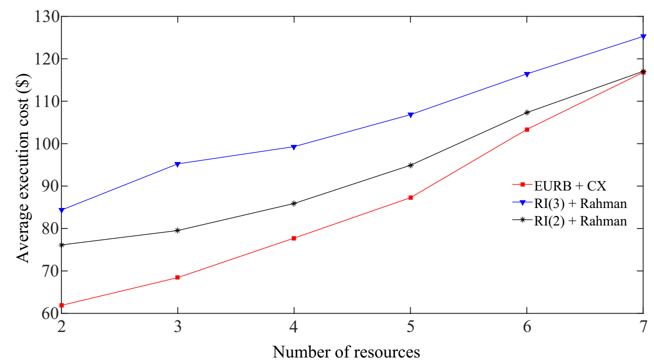

FIGURE 6 Effect of number of resources on average execution cost

5.1.2 | Application size

To assess the effect of varying the application size, these algorithms are applied on task graphs of different sizes. As shown in Figure 7, as application size increases, the relative performances of the algorithms remain the same, and the proposed algorithm achieves a significant improvement compared with prior algorithms.

5.1.3 | Failure rate

To assess the effect of the average failure rate on the performances of the algorithms, the algorithms are executed in environments with different failure rates. Figure 8 shows that the proposed algorithm achieves a significant improvement for all failure rates.

5.1.4 | Graph width

Figure 9 shows the effect of varying the average graph width on the performance of the algorithms. The results show that the proposed algorithm outperforms the previous algorithm

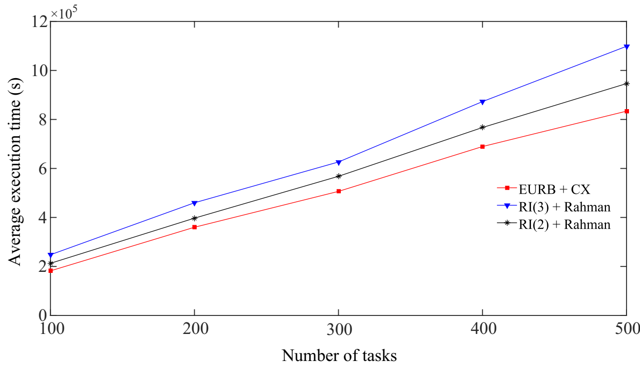

FIGURE 7 Effect of application size on average execution time

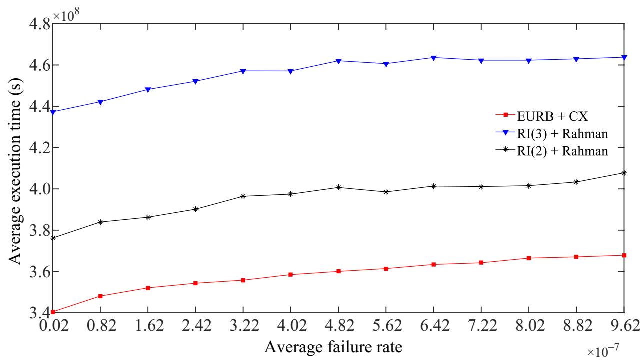

FIGURE 8 Effect of failure rate on average execution time

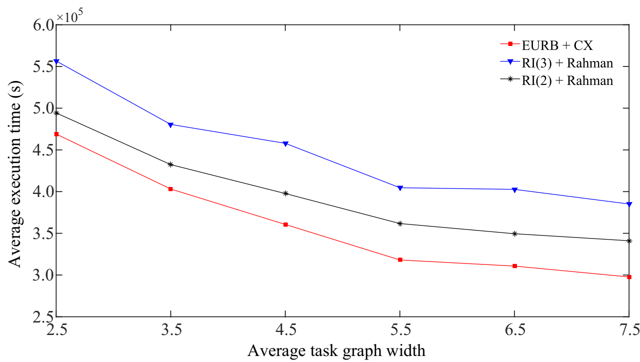

FIGURE 9 Effect of graph width on average execution time

for applications with different graph widths, especially for applications with wider graphs.

5.1.5 | Real-world applications

Additionally, we conducted experiments for assessing the performances of the algorithms for real-world applications, namely Epigenomics, Montage, CyberShake, and SIPHT. As shown in Figure 10, we assessed the performances of the algorithms for these applications in

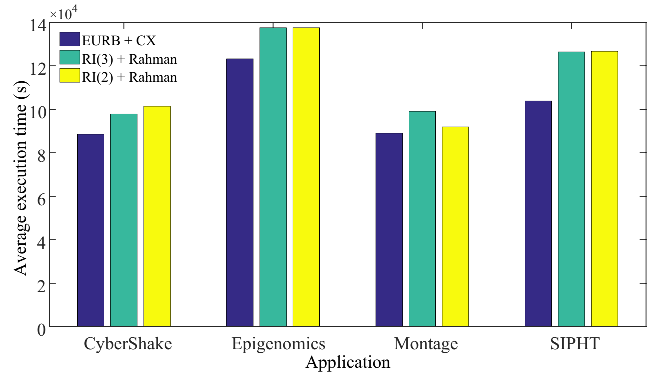

FIGURE 10 Average execution time of real-world benchmark applications

terms of the average execution time. The results show that the proposed algorithm performs well for real-world applications.

5.1.6 | Comparison of checkpointing schemes

The benefit of the proposed approach is attributed to two reasons: (i) the proposed replication method (EURB) and (ii) the proposed checkpointing scheme (CX). To assess the effects of each of these two aspects, we studied different combinations of the replication methods (EURB and RI) and checkpointing schemes, CX, the scheme proposed by Rahman et al. (Rahman), and a checkpointing scheme without exchanging checkpoint files (CNX). To assess the effect of the checkpointing scheme, the EURB replication method is combined with two checkpointing schemes, CX and Rahman. Figure 11 shows that the CX, which is obtained by relaxing the simplifying assumption in Rahman's work, achieves a better performance relative to the Rahman scheme for all failure rates. Furthermore, as shown in Figure 12, all combinations of the replication methods (EURB and RI) and checkpointing schemes CX and CNX are compared. The results show that the effect of the EURB replication method is more significant than that of CX.

5.1.7 | Other failure distributions

Furthermore, we assessed the relative performance of the redundancy selection algorithms on computing environments with more general time-to-failure distributions. We considered Weibull and Pareto distributions for the time-to-failure of resources. The parameters of these distributions were selected according to [34]. The results depicted in Figure 13 shows that the proposed approach performs efficiently for Weibull and Pareto time-to-failure distributions.

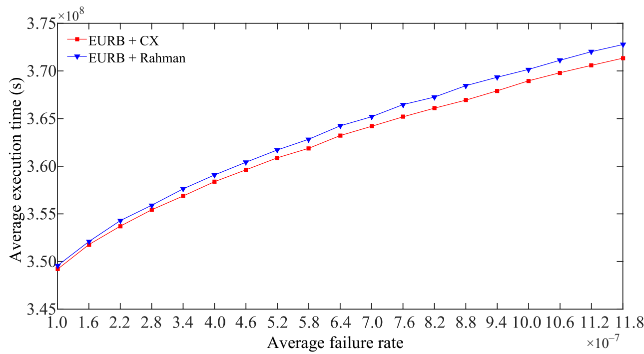

FIGURE 11 Effect of the proposed checkpointing method on average execution time

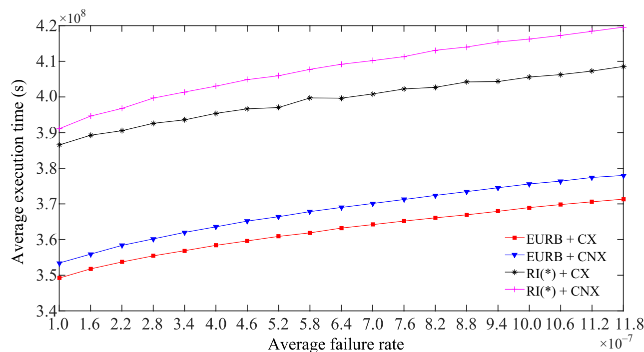

FIGURE 12 Effect of proposed replication and checkpointing method on relative performance of algorithms

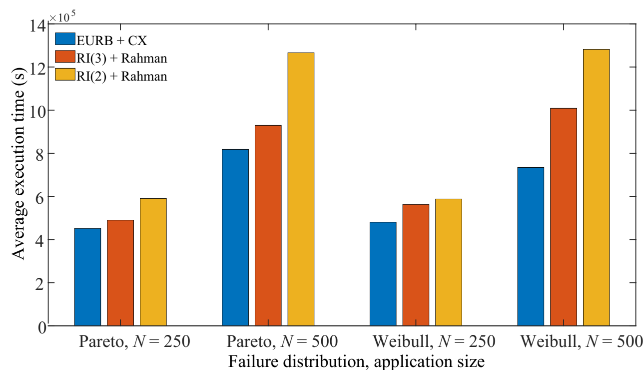

FIGURE 13 Relative performance of algorithms in environments with Pareto and Weibull time-to-failure distributions

6 | CONCLUSIONS

We focused on the problem of selecting the appropriate fault-tolerance strategies for scheduling workflow tasks in fault-prone environments. To reduce the redundancy overhead, we combined resubmission, replication, and checkpoint redundancies. The appropriate redundancies of tasks were determined in two phases: First, the appropriate replication vector was obtained; subsequently, the optimal checkpointing interval of each task was computed. As the proposed algorithm used the idle durations of resources for task replication and checkpoint files were exchanged among resources, fewer checkpoints files were required to be saved; thus, the failure-free overhead was reduced. Furthermore, as the proposed algorithm apportioned the provisioned resources among concurrently executing tasks, it achieved a significant performance improvement compared with prior studies, especially in situations with less resources.

Finally, because the simplifying assumption in a previous study was relaxed in our formulation of the optimal checkpointing problem, a better performance was achieved in terms of both time and cost. Our results demonstrated that both the proposed replication method and CX reduced the resiliency overhead.

ORCID

Hassan Motallebi https://orcid.org/0000-0001-5769-8532

REFERENCES

1. F. Cappello, *Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities*, Int. J. High Perform. Comput. Appl. **23** (2009), 212–226.
2. D. P. Chandrashekar, *Robust and Fault-Tolerant Scheduling for Scientific Workflows in Cloud Computing Environments*, Ph.D. dissertation, Dept. Computing and Inf. Syst., University of Melbourne, Melbourne, Australia, 2015.
3. G. Aupy et al., *Checkpointing strategies for scheduling computational workflows*, Int. J. Network. Comput. **6** (2016), 2–26.
4. R. N. Calheiros and R. Buyya, *Meeting deadlines of scientific workflows in public clouds with tasks replication*, IEEE Trans. Parallel Dist. Syst. **25** (2014), 1787–1796.
5. K. Plankensteiner and R. Prodan, *Meeting soft deadlines in scientific workflows using resubmission impact*, IEEE Trans. Parallel Dist. Syst. **23** (2012), 890–901.
6. M. T. Rahman et al., *Check pointing to minimize completion time for inter-dependent parallel processes on volunteer grids*, in Proc. IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (Cartagena, Colombia), May 16–19, 2016, pp. 331–335.
7. J. W. Young, *A first order approximation to the optimum check point interval*, Commun. ACM **17** (1974), 530–531.
8. J. T. Daly, *A higher order estimate of the optimum checkpoint interval for restart dumps*, Future Gener. Comp. Syst. **22** (2006), 303–312.
9. M.-S. Bouguerra et al., *A flexible checkpoint, restart model in distributed systems*, in Proc. Parallel Process. Appl. Math. (Wroclaw, Poland), Sept. 13–16 (2009), pp. 206–215.
10. A. Benoit, M. Hakem, and Y. Robert, *Fault tolerant scheduling of precedence task graphs on heterogeneous platforms*, in Proc. Int. Symp. Parallel Distrib. (Miami, FL, USA), Apr. 14–18, 2008, pp. 1–8.
11. H. Topcuoglu, S. Hariri, and M.-Y. Wu, *Performance-effective and low-complexity task scheduling for heterogeneous computing*, IEEE Trans. Parallel Dist. Syst. **13** (2002), 260–274.
12. S. K. Jayadivya, J. S. Nirmala, and M. S. S. Bhanu, *Fault tolerant workflow scheduling based on replication and resubmission of tasks in cloud computing*, Int. J. Comput. Sci. Eng. **4** (2012), 996–1006.
13. R. Sirvent, R. M. Badia, and J. Labarta, *Graph-based Task Replication for Workflow Applications*, in Proc. IEEE Int. Conf.

High Performance Comput. Commun. (Seoul, Rep. of Korea), June 25–27, 2009, pp. 20–28.
14. S. Abrishami, M. Naghibzadeh, and D. H. J. Epema, *Deadline constrained workflow scheduling algorithms for infrastructure as a service clouds*, Future Gener. Comp. Syst. **29** (2013), 158–169.
15. L. Zhao, Y. Ren, and K. Sakurai, *Reliable workflow scheduling with less resource redundancy*, Parallel Comput. **39** (2013), 567–585.
16. M. Wieczorek, R. Prodan, and A. Hoheisel, *Taxonomies of the multi-criteria grid workflow scheduling problem*, Institute on Resource Management and Scheduling, Innsbruck, Austria, CoreGRID Tech. Rep. TR-0106, Aug. 2007.
17. Y. Zhang et al., *Combined fault tolerance and scheduling techniques for workflow applications on computational grids*, in Proc. IEEE/ACM Int. Symp. Clust. Comput. Grid (Shanghai, China), May 18–21, 2009, pp. 244–251.
18. A. Benoit et al., *Combining checkpointing and replication for reliable execution of linear workflows, In Proc. IEEE Int. Paallel Distrib. Process. Symp. Workshops* (Vancouver, Canada), May 2018, pp. 793–802.
19. A. Benoit et al., *Optimal check- pointing period with replicated execution on heterogeneous platforms*, in Proc. Workshop FTXS@ HPDC (Washington, DC, USA), June 26–27, 2017, pp. 9–16.
20. M. Chtepen et al., *Adaptive task checkpointing and replication: toward efficient fault-tolerant grids*, IEEE Trans. Parallel Distrib. Syst. **20** (2009), 180–190.
21. J. Daly, *A model for predicting the optimum checkpoint interval for restart dumps*, in Proc. Int. Conf. Comput. Sci. (Melbourne, Australia), June 2–4, 2003, pp. 3–12.
22. S. Sadi and B. Yagoubi, *Communication-aware approaches for transparent checkpointing in cloud computing*, Scalable Comput.: Practice Experience **17** (2016), 251–270.
23. M. Bougeret et al., *Checkpointing strategies for parallel jobs*, in Proc. Int. conf. Hight Performance Comput. Netw. Storage Anal. (Seattle, WA, USA), Nov. 12–18, 2011, pp. 1–11.
24. G. Aupy and J. Herrmann, *Periodicity in optimal hierarchical checkpointing schemes for adjoint computations*, Optim Methods Softw. **32**, (2017), 594–624.
25. H. Nguyen et al., *An execution environment for robust parallel computing on volunteer PC Grids*, in Proc. Int. Conf. Parallel Process. (Pittsburgh, PA, USA), Sept. 10–13, 2012, pp. 158–167.
26. G. Aupyet al., *On the Combination of Silent Error Detection and Check-pointing*, in Proc. IEEE Pacific Rim Int. Symp. Dependable Comput. (Vancouver, Canada), Dec. 2–4, 2013, pp. 11–20.
27. A. Benoit et al., *Two-level check-pointing and verifications for linear task graphs*, in Proc. IEEE Int. Parallel Distrib. Process. Symp. (Chicago, IL, USA), May 23–27, 2016, pp. 1239–1248.
28. A. Benoit et al., *Multi-level check- pointing and silent error detection for linear workflows*, J. Comput. Sci. **28** (2018), 398–415.
29. L. Han et al., *A generic approach to scheduling and checkpointing workflows*, in Proc. Int. Conf. Parallel Process. (Eugene, OR, USA), July 29–Aug. 3, 2018, pp. 1–10.
30. S. Sadi and B. Yagoubi, *On the optimum checkpointing interval selection for variable size checkpoint dumps*, in Proc. Int. Conf. Comput. Sci. Applicat. (Saida, Algeria), May 20–21, 2015, pp. 599–610.
31. G. Juve et al., *Characterizing and profiling scientific workflows*, Future Gener. Comput. Syst. **29** (2013), 682–692.

AUTHOR BIOGRAPHY

Hassan Motallebi received his BS degree in software engineering from the School of Electrical and Computer Engineering, Isfahan University of Technology, Isfahan, Iran, in 2005, and his MS and PhD degrees in software engineering from the School of Computer Engineering, Iran University of Science and Technology, Tehran, Iran, in 2008 and 2013, respectively. Since 2013, he has been with the Department of Electrical and Computer Engineering, Graduate University of Advanced Technology, Kerman, Iran, where he is an Assistant Professor. His research interests include data mining, cloud and grid computing, performance evaluation of concurrent systems, and Petri-net–based formalisms.