Distrib. Comput. (2011) 23:359–372
DOI 10.1007/s00446-010-0124-2

Efficient online algorithm for identifying useless states in distributed systems

Lung-Pin Chen · Der-Johng Sun · William Chu

Received: 28 April 2008 / Accepted: 25 August 2010 / Published online: 1 January 2011
© Springer-Verlag 2010

Abstract In a distributed system, detecting whether a given logical predicate is true on the global states is fundamental for testing and debugging the program. Detecting predicates by examining all global states is intractable due to the combinatorial nature of the problem. This work designs an efficient online algorithm that identifies the consistent and useless states each time a new state is reported. This paper formulates the optimality of detecting algorithms in terms of pseudo states, which are employed to represent unknown states to the monitor process. Based on this technique, memory space of the debugger can be minimized by removing the useless states without affecting the debugging results. While minimizing memory space, the proposed algorithm requires only $O(p^2M)$ time in total, where p is the number of processes, and M is the number of reported states.

Keywords Checkpoint · Distributed debugging · Global predicate detection

L.-P. Chen (✉) · W. Chu
Department of Computer Science and Information Engineering, TungHai University,
Taichung, Taiwan
e-mail: lbchen2007@gmail.com; lbchen@thu.edu.tw

W. Chu
e-mail: cchu@thu.edu.tw

D.-J. Sun
Department of Computer Science and Information Engineering, National Chiao-Tung University,
Hsinchu, Taiwan
e-mail: derjohng@gmail.com

1 Introduction

Detecting whether a logical predicate is true on some execution states is essential for testing and debugging a sequential program. In the distributed system, a *global predicate* Φ is comprised of variables from distinct processes. The *global predicate detection problem* finds a consistent global state of the distributed program execution on which Φ is true [4,10,13]. Usually Φ is used to formalize the *undesired* situation of the distributed system. For example, for a 2-process system, consider the common *conjunctive global predicate* [5,8,20] with form $\Phi = LP_1 \wedge LP_2$, where LP_1 (or LP_2) is a local variable of P_1 (or P_2) indicating whether the process P_1 (or P_2) is in the critical section. If there is some consistent global states on which Φ is true, then both processes are in the critical section, an error may occur.

A *checker process* is a distinguished process which collects the execution states and performs global predicate detection. The program execution generates a large number of execution states in a short period of time. However, instead of examining every state, the checker process only needs to examine the *checkpointed states*. For example, for a global predicate Φ, if the state is not related to the variables involved in Φ then the state can be disregarded when detecting Φ. On the other hand, if a variable is involved in Φ and its value is altered in a state, this state needs to be checkpointed. Note that each application process takes the checkpoints locally without communication.

This paper simply refers to the checkpointed state as a *checkpoint* hereafter. The checker process finds a consistent global checkpoint on which Φ is true, given a set of collected checkpoints H. Checkpoints in set H having no chance to become members of any consistent global checkpoint are *useless* or *removable* for the detection algorithm. Among all the checkpoints, the removable checkpoints may degenerate

the program debugger or monitor, since they are stored but useless in the debugging process. A smart way to accelerate the global predicate detection algorithm is to identify and remove useless checkpoints before invoking the detection procedure [2,3,18,7,21].

In this paper, the *online checkpoint identifying problem* identifies the consistent and removable checkpoints each time a new checkpoint is reported. This work formulates the *optimality* of the online checkpoint identifying problem in terms of pseudo states which are employed to represent unknown states to the checker process. An online identifying algorithm is *optimal* if it correctly identifies useless states as *early* as possible.

This paper also contributes an efficient optimal algorithm to identify useful/useless checkpoints online. This new algorithm is based on the *Z-path* [19,14,15,17], employed to determine precisely whether two checkpoints belong to the same consistent global checkpoint. Many researches have proposed checkpoint identifying algorithms using Z-paths [19,14]. However, these investigations assume all checkpoints are given ahead of the processing time. Instead, this paper investigates Z-paths for a sequence of growing checkpoint sets.

This study makes a distinction between *conjunctive global predicate detection* and *useless state detection* (to be solved in this paper) for the distributed system, although both concern consistent global states. The algorithms [6,8] of the first problem detect the consistent global checkpoint, rather than trying to find useless checkpoints. In [3], the algorithm incrementally removes some checkpoints to save memory space. However, the above research only explores partial useless checkpoints by their algorithm and does not mention optimality. The researcher in [2] discusses a similar problem for the problem model, differing from that used in this paper, called the *definitely-true* model. Assume that there are p processes and M checkpoints, i.e. M stages. This work shows that although there are M stages, the data structure associated with a checkpoint is updated at most p times. Based on this property, the time complexity of our algorithm is only $O(p^2M)$. This result represents an improvement over the previous $O(M^2)$-time algorithm in [3]. Note that $p \ll M$ since p is usually constant, while M usually grows as the program runs.

The rest of this paper is organized as follows. Section 2 describes the background of this paper and gives the basic definition of useless checkpoints. Section 3 describes the model for the optimal useless checkpoint identifying problem. Section 4 presents the algorithm for constructing pseudo states and proves their optimality. Section 5 presents the fundamental properties of checkpoints and outlines the basic algorithm. Section 6 discusses a new incremental checkpoint identifying algorithm. Finally, concluding remarks are made in Sect. 7.

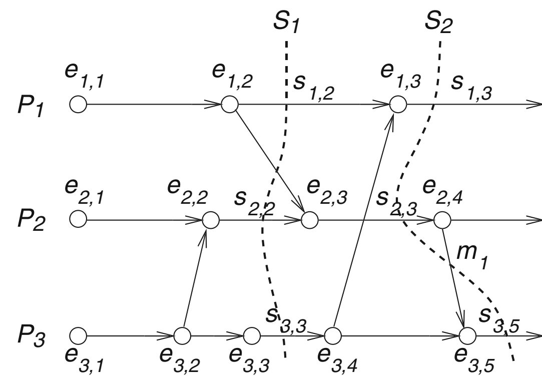

Fig. 1 Events and states of a 3-process distributed program

2 Preliminaries

2.1 Distributed computation

A distributed program consists of *application processes* (or simply, *processes*) that communicate via a network. For simplicity, this work assumes that the distributed program consists of p processes. These processes share no memory and no global clock. Each pair of processes must communicate by exchanging *messages* via a network *channel*. This work assumes that every message is sent and received correctly.

The *local states* (or, *states*) of a process change only when *events* (atomic actions) are executed. Let $e_{i,x}$ denote the xth event in process P_i. The state $s_{i,x}$ refers to the *local state* of P_i after P_i executes event $e_{i,x}$, but before $e_{i,x+1}$ is executed. Execution of a distributed program can be modeled as a *communication graph* $G = (U, E)$, where each vertex in U refers to an event, and each edge (u, v) in E refers to the precedence relation from event u to v [9,11,12]. In the communication graph, each edge $(e_{i,x}, e_{i,x+1})$ in E refers to the state $s_{i,x}$. Furthermore, each edge $(e_{j,y}, e_{i,x})$ refers to the *message* sent by event $e_{j,y}$ and received by event $e_{i,x}$, $j \neq i$.

Figure 1 illustrates the events and states of a distributed program. State $s_{j,y}$ *happens before* state $s_{i,x}$, denoted by $s_{j,y} \rightarrow s_{i,x}$, if and only if there is a path from $e_{j,y+1}$ to $e_{i,x}$ in the communication graph. For example, in Fig. 1, state $s_{3,3} \rightarrow s_{1,3}$ since there is a path from $e_{3,4}$ to $e_{1,3}$ in the communication graph.

In the distributed system (with p processes), a *global state* is a collection of p states, one from each process. A *consistent global state* is a global state in which no happen-before relationship occurs between the members. Consistent global states represent the execution states of a distributed program [12]. For example, in Fig. 1, the global state $S_1 = \{s_{1,2}, s_{2,2}, s_{3,3}\}$ is consistent, representing the state immediately after executing events $e_{1,2}$, $e_{2,2}$ and $e_{3,3}$. Note that an *inconsistent global state* can not represent any

execution state. For example, in Fig. 1, the system never enters the state corresponding to the inconsistent global state $S_2 = \{s_{1,3}, s_{2,3}, s_{3,5}\}$, since P_3 cannot receive message m_1 before P_2 sends the message out.

2.2 Checkpointed state

For a process in the distributed system, the important execution states are *checkpointed* and stored in a stable storage. These checkpointed states are called *checkpoints* hereafter. This work assumes that the checkpoints are generated by the process *locally* without communication. Let $c_{i,x}$ denote the xth checkpoint generated in process P_i. The value x is called the *index* of checkpoint $c_{i,x}$. A *global checkpoint* is a set of p checkpoints, one from each process. Moreover, a *consistent global checkpoint* is a global checkpoint in which no happen-before relation exists between the members.

As mentioned in Sect. 1, a distributed program can be tested and debugged by examining the consistent global checkpoints. However, not every checkpoint belongs to some consistent global checkpoint. For example, in Fig. 2, checkpoint $c_{1,2}$ and $c_{1,3}$ belong to no consistent global checkpoint, i.e. they are useless/removable.

For a global checkpoint G, let $G[i]$ denote the ith member from process P_i. Let $\mathcal{L}(c_{i,x})$ be the set of checkpoints occur before the checkpoint $c_{i,x}$. Specifically, $\mathcal{L}(c_{i,x}) = \{c_{i,x'} \mid c_{i,x'}$ is a checkpoint and $x' \le x)\}$. Some notations are defined as follows:

- $c_1 \prec c_2$ (or $c_1 \preceq c_2$): For checkpoints c_1 and c_2, $\mathcal{L}(c_1) \subset \mathcal{L}(c_2)$ (or $\mathcal{L}(c_1) \subseteq \mathcal{L}(c_2)$). For example, in Fig. 2, $c_{1,1} \prec c_{1,2}$.
- $c \prec S$ (or $c \preceq S$): For checkpoint $c = c_{i,x}$ and global checkpoint S, $c \prec S[i]$ (or $c \preceq S[i]$). For example, in Fig. 2, $c_{2,1} \prec G_2$ and $c_{2,2} \preceq G_2$.
- $S \prec c$ (or $S \preceq c$): For global checkpoint S and checkpoint $c = c_{i,x}$, $S[i] \prec c$ (or $S[i] \preceq c$). For example, in Fig. 2, $G_2 \prec c_{1,3}$.
- $S_1 \prec S_2$ (or $S_1 \preceq S_2$): For global checkpoints S_1 and S_2, the condition $c_1 \prec c_2$ (or $c_1 \preceq c_2$) holds, where $c_1 = S_1[i]$ and $c_2 = S_2[i]$, $1 \le i \le p$. Note that c_1 and c_2 are single checkpoints. For example, in Fig. 2, $G_1 \prec G_2$ and $G_2 \preceq G_3$. Obviously, if $S_1 \preceq S_2$ and $S_2 \preceq S_1$ then $S_1 = S_2$.
- $S_0 = \min(S_1, S_2)$: For global checkpoints S_0, S_1 and S_2, for each checkpoint $c_{i,x} \in S_0$, the following properties hold: (1) $c_{i,x} \in S_1$ or $c_{i,x} \in S_2$, (2) $c_{i,x} \preceq S_1$ and $c_{i,x} \preceq S_2$. For example, in Fig. 2, $G_2 = \min(G_3, G_4)$. Moreover, $G_2 \preceq G_3$ and $G_2 \preceq G_4$.
- $T = prev(S)$: For two global checkpoints S and T, $T[i] = c_{i,x}$ and $S[i] = c_{i,x+1}$ for all i. For example, in Fig. 2, $G_1 = prev(G_2)$. The notation extends to the checkpoints. For example, $c_{2,2} = prev(c_{2,3})$ in Fig. 2.
- $T = next(S)$: This holds when $S = prev(T)$.

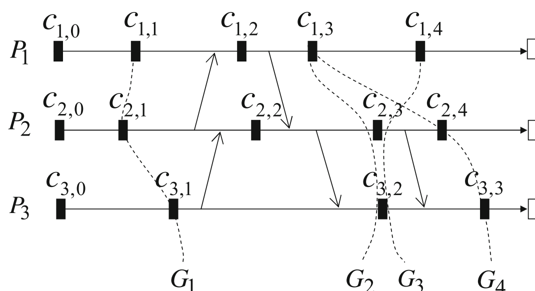

Fig. 2 Global checkpoints

2.3 Vector clock

The happen-before relation between a pair of checkpoints can be determined using the vector clocks [12], maintained by every computation process. In the vector clock mechanism, each process P_i maintains a variable V_i which is a vector of p integers. Initially, $V_i = [0, 0, \ldots, 0]$. The vector clocks are updated as follows (see Fig. 3):

1. When P_i sends a message, associate the message with its vector clock V_i.
2. When P_i receives a message which is associated with a vector clock V, let $V_i[k] = max(V_i[k],\ V[k])$ for each k.
3. When P_i generates a checkpoint $c_{i,x}$, proceeds $V_i[i]$ by one, i.e. let $V_i[i] = V_i[i] + 1$. Then, let vector clock of $c_{i,x}$ by $V(c_{i,x}) = V_i$.

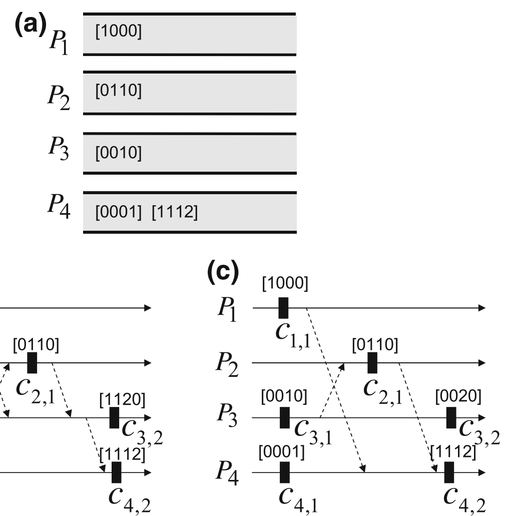

Fig. 3 **a** C-pattern H_5 with 5 checkpoints. **b** Possible H_6. **c** Another possible H_6. Note that the directed edges represent the precedence relations instead of real messages

From above, each checkpoint $c_{i,x}$ is associated with a vector clock denoted by $V(c_{i,x})$. Figure 3 illustrates a set of checkpoints and their vector clocks.

For a pair of p-tuple vector clocks V and V', define that $V < V'$ if and only if $V[i] \leq V'[i]$ for all i, but $V[j] < V'[j]$ for some j. Some important properties can be easily observed from Fig. 3 [1,11,12,16]:

P1 $V(c_{i,x})[i] = x$ indicates the index of checkpoint $c_{i,x}$.

P2 $c_{j,y} \rightarrow c_{i,x}$ iff $V(c_{j,y}) < V(c_{i,x})$
If $c_{j,y}$ happens before $c_{i,x}$ then $V(c_{j,y})[i] < V(c_{i,x})[i]$, thus, $V(c_{j,y})$ and $V(c_{i,x})$ can not be fully equal.

P3 $V(c_{j,y}) < V(c_{i,x})$ iff $V(c_{j,y})[j] < V(c_{i,x})[j]$.
Consider a checkpoint $c_{i,x}$ in process P_i with $V(c_{i,x})[j] = y'$. From the above vector clock protocol, the checkpoints in process P_j $(j \neq i)$ can be partitioned into two parts, checkpoints with indices less than or equal to y', which all happen before $c_{i,x}$, and checkpoints with indices larger then y', which do not happen before $c_{i,x}$. In other words, the happen-before relation between $c_{i,x}$ and any checkpoint $c_{j,y}$ in P_j can be determined by merely examining whether $c_{j,y}$'s index (i.e. value y) is less than or equal to y'. Therefore, Property P3 holds. Consider the example depicted in Fig. 3, $V(c_{3,1})[3] < V(c_{4,2})[3]$ and thus $c_{3,1} \rightarrow c_{4,2}$.

From Property P2 and P3, the happen-before relationship between a pair of checkpoints $c_{i,x}$ and $c_{j,y}$ can be determined from their vector clocks within $O(1)$ time.

3 Optimal online checkpoint identifying problem

3.1 C-pattern and possible future C-pattern

In the current problem model, each application process determines its checkpoints locally without communication. When a checkpoint is taken, it is reported to the *checker process*. The checkerprocess is a distinguished process which collects the timestamps of checkpoints and performs the checkpoint identifying algorithm in an online manner. This work assumes that when reporting a checkpoint c, only the local information (including the time-stamp and local variables) of checkpoint c is reported, and no other checkpoint information. We also assume that each application process reports its checkpoints in an FIFO order, i.e. checkpoint $c_{i,x}$ is received before $c_{i,x+1}$ to the checker process. The well-known TCP protocol supports such communication.

In a time instance, the set of checkpoints (with the associated timestamps) collected by the checker process is called a *C-pattern*, defined in Definition 1.

Definition 1 The set of checkpoints collected by the checker process is called a *C-pattern*. A C-pattern with t checkpoints is denoted by H_t. Thus, $H_t = H_{t-1} \cup \{c\}$ where c is the tth collected checkpoint. Each C-pattern H_t is FIFO, that is, the condition $c_{i,x} \in H_t$ implies that $c_{i,x'} \in H_t$ for all $x' < x$ (note that both $c_{i,x}$ and $c_{i,x'}$ are from the same process P_i).

For the checker process, let *stage t* refer to the time period of processing H_t. A stage $t' > t$ is called a *future stage* of stage t, and, $H_{t'}$ is called a *future C-pattern* of H_t. In H_t, each checkpoint c is associated with the vector clock $V(c)$. The vector clock $V(c)$ remains unchanged in $H_{t'}$ for all $t' \geq t$.

For the online problem model, in a stage t (before stage $t+1$ begins), any checkpoint set that has a possibility to be the C-pattern in some future stage of t is called a *possible future C-pattern* of H_t. The formal definition is in Definition 2.

Definition 2 For the C-pattern H_t in stage t, a checkpoint set H is called a *possible future C-pattern* of H_t if the following properties hold:

– There is a distributed program whose execution generates the vector clocks recorded in H_t and H (both are FIFO), and
– $H_t \subset H$.

Figure 3 illustrates a C-pattern H_5 (subfigure (a)) and its two possible future C-patterns (subfigures (b) and (c)). Each of them is a candidate for real H_6 in the next stage 6.

3.2 Optimal online checkpoint identifying problem

Given a single C-pattern H_t, if a checkpoint c belongs to no consistent global checkpoint in H_t then we say that c is *removable/useless* in H_t (or in stage t). Otherwise, c is useful in H_t (or in stage t). The *online checkpoint identifying problem* identifies the consistent and removable checkpoints over a sequence of C-patterns in the stages. Formally, in *each* stage $t, t = 1, 2, \ldots, M$, the checker process classifies all checkpoints into one of the following three statuses:

– *t-consistent*: A checkpoint c is t-consistent if and only if c is useful in all stages t' satisfying $t' \geq t$. Restated, in stage t, no matter how the newly received checkpoints are added, c belongs to at least one consistent global checkpoint.
– *t-removable*: A checkpoint c is t-removable if and only if c is useless in all stages t' satisfying $t' > t$. Restated, in stage t, regardless of how checkpointed are added based on the FIFO model, no consistent global checkpoints containing c can be found.
– *t-potential*: A checkpoint c in H_t is t-potential if c is neither t-consistent nor t-removable in this stage. Restated, c belongs to no consistent global checkpoint in H_t in stage t, but may belong to a consistent global state in $H_{t'}$ for some $t' > t$.

For example, in Fig. 3a, checkpoint $c_{2,1}$ with $V(c_{2,1}) = [0, 1, 1, 0]$ is potential in stage 5 since it might be useless (see Fig. 3b) or useful (see Fig. 3c) in the next stage 6.

Figure 4 illustrates the notion of potential checkpoints of the above online problem. From the point of view of stage t, there are many possibilities for future stages. Hence, the status of certain checkpoints may not be determined in the current stage t unless all the possibilities are examined. This notion is formalized as the *correctness and minimum property* as follows:

Correctness property The *correctness* property is satisfied if in each stage t, all the removable and useful checkpoint are correctly identified, i.e. all the checkpoints marked as removable in stage t are indeed t-removable.

A surprisingly naive strategy for satisfying the correctness property is simply to treat all checkpoints as non-removable, and thereby, no checkpoint will be wrongly predetermined as removable in the earlier stage. To prevent the trivial solution, an *optimal* online checkpoint identifying algorithm identifies all removable states as early as possible for each stage, under the correctness prerequisite. The following minimum property formalizes this notion:

Minimum property The *minimum* property is to find the minimum stage t for which a checkpoint is t-removable. Specifically, the *minimum* property is satisfied if for each stage t, if there is no future C-pattern (of H_t) in which a real checkpoint c is useful, then this checkpoint c is marked as t-removable.

4 Volatile checkpoints

The volatile checkpoints are imaginary checkpoints representing unknown checkpoints in the current stage of the online algorithm. Conceptually, they establish a mapping between the current and future C-patterns. In this section, Sect. 4.1 describes the algorithm for constructing volatile checkpoints. Sections 4.2 and 4.3 explain how volatile checkpoints are helpful for developing an optimal checkpoint identifying algorithm.

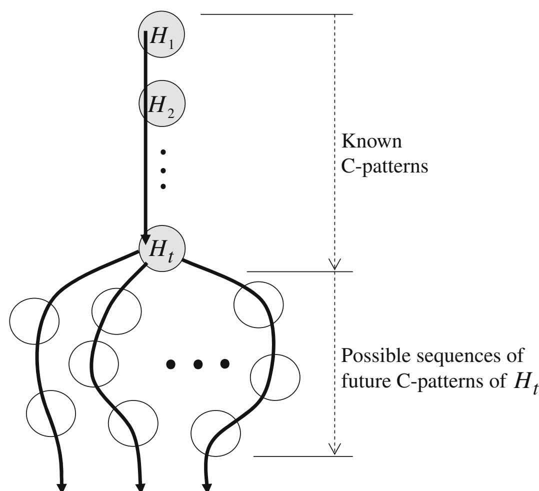

Fig. 4 Illustration of notion of possible future C-patterns

Algorithm 1 Construct_VC(H_t)

1: let $\widehat{H}_t = H_t$
2: **for** each process P_i, $1 \le i \le p$ **do**
3: let l_i be the index of the last real checkpoint from P_i in H_t;
4: let $L_i = \max\{V(c)[i] \mid c \in H_t, 1 \le i \le p\}$;
5:
6: */*create volatile checkpoints for the unknown checkpoints whose indices are between $l_i + 1$ and L_i */*
7: **for** $x = l_i + 1$ to L_i **do**
8: Add volatile checkpoints $d_{i,x}$ to $\widehat{H}_t$;
9: Set $V(d_{i,x}) = V(c_{i,l_i})$ but $V(d_{i,x})[i] = x$. (If such c_{i,l_i} does not exist, simply assume $V(c_{i,l_i}) = [0, 0, \ldots, 0]$.)
10: **end for**
11:
12: */*create last volatile checkpoint for the unknown checkpoints whose indices are larger than L_i */*
13: Add volatile checkpoint $d_{i,\infty}$ to $\widehat{H}_t$;
14: Let $V(d_{i,\infty}) = V(c_{i,l_i})$ but $V(d_{i,\infty})[i] = \infty$. (If such c_{i,l_i} does not exist, simply assume $V(c_{i,l_i}) = [0, 0, \ldots, 0]$.)
15: **end for**

4.1 Constructing volatile checkpoints

In the C-pattern H_t, let l_i be the maximum checkpoint index (of process P_i) reported by process P_i itself, and L_i be the maximum checkpoint index (of process P_i) reported by all processes. Formally, $l_i = \max\{V(c)[i] \mid c \in H_t$ and c is reported by $P_i\}$ and $L_i = \max\{V(c)[i] \mid c \in H_t\}$. Clearly, $l_i \le L_i$. In a C-pattern H_t, for each process P_i, those checkpoints occurring after c_{i,l_i} are *unknown* and represented using the *volatile checkpoints*, constructed as shown in Algorithm 1. The C-pattern H_t with the added volatile checkpoints is called the *extended C-pattern*, denoted by $\widehat{H}_t$. The original checkpoints in H_t are referred to as *real* or *non-volatile* checkpoints.

Figure 5 demonstrates the volatile checkpoints in $\widehat{H}_t$ (Fig. 5b) which establish the mapping between current C-pattern H_t (Fig. 5c) and future C-pattern $H_{t'}$ (Fig. 5a). Let c_{l_i} be the *last* real checkpoint in each process P_i in $\widehat{H}_t$. Some properties of volatile checkpoints are described as follows:

P4 For any checkpoint a and volatile checkpoint b in $\widehat{H}_t$, if b is in process P_i, $(a \to c_{l_i}) \Rightarrow (a \to b)$.
Clearly, all volatile checkpoints in P_i occur after c_{l_i}. This property follows from the fact that $(a \to c_{l_i})$ and $(c_{l_i} \to b)$.

P5 For any checkpoint a and volatile checkpoint b in $\widehat{H}_t$, if b is in process P_i, $(a \to b) \Rightarrow (a \to c_{l_i})$.

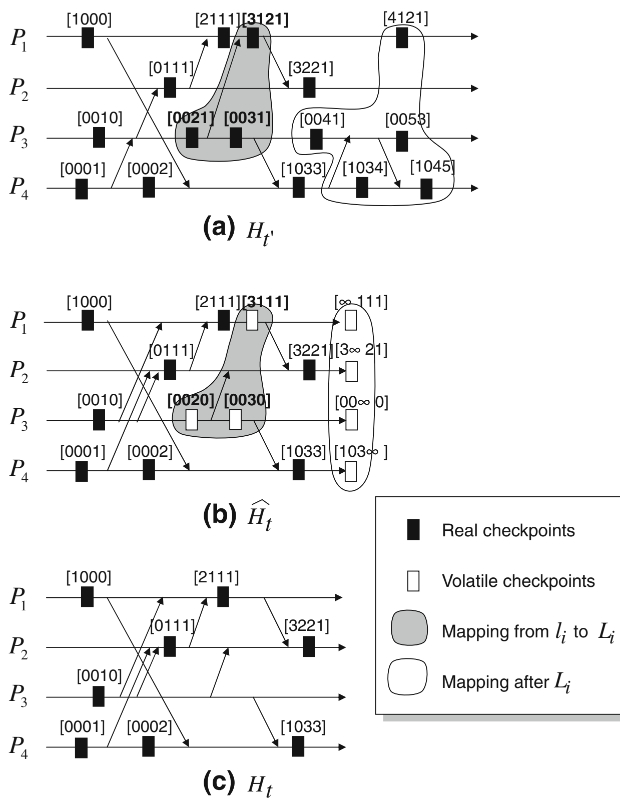

Fig. 5 An example of C-pattern H_t (in **c**) and its future C-pattern $H_{t'}$ (in **a**) with $t = 8$ and $t' = 16$. The extended C-pattern $\widehat{H}_t$ (in **b**) contains some volatile checkpoints to represent checkpoints in $H_{t'} \setminus H_t$

If b is volatile, $V(b)$ is set to equal to $V(c_{l_i})$ except index $V(b)[i]$ (See Step 9 in Algorithm 1). According to Property P2 and P3, b can not incur happen-before relation $a \rightarrow b$ unless the relation $a \rightarrow c_{l_i}$ (and $c_{l_i} \rightarrow b$) is confirmed by the real checkpoint c_{l_i}.

4.2 Correctness of volatile checkpoints

Suppose that H_t is a C-pattern and $H_{t'}$ is a future C-pattern of H_t. Recall that $\widehat{H}_t$ is the same as H_t but incorporating volatile checkpoints (see Sect. 4.1). From $H_{t'}$ to $\widehat{H}_t$, there is a one-to-one mapping for checkpoints without ∞ indices; and there is a many-to-one mapping for checkpoints with ∞ indices. The mapping is denoted and defined as follows:

$$map_{t',t}(c_{i,x}) = \begin{cases} c_{i,x} \text{ in } \widehat{H}_t & \text{if } x \leq l_i \\ d_{i,x} \text{ in } \widehat{H}_t & \text{if } l_i < x \leq L_i \\ d_{i,\infty} \text{ in } \widehat{H}_t & \text{if } L_i < x \end{cases} \quad (1)$$

where $c_{i,x} \in H_{t'}$ is a real checkpoint in $H_{t'}$, l_i is the index of the last real checkpoint reported from P_i in H_t, and $L_i = \max\{V(c)[i] \mid c \in H_t\}$.

From Fig. 5b, a volatile checkpoint does not incur any new precedence relation to the existing checkpoints, unless the relation is confirmed by the existing non-volatile checkpoints. For example, in Fig. 5b, the volatile checkpoint [3, 1, 1, 1] has no precedence relation except those relations confirmed by real checkpoints [2, 1, 1, 1] and [3, 2, 2, 1]. Based on this observation, Lemma 1 shows the useful property of the volatile checkpoints.

Lemma 1 *Assume that $H_{t'}$ is a future C-pattern of H_t with $t < t'$ and $H_t \subset H_{t'}$. For a checkpoint $c_{i,x}$ in $H_{t'}$ and its corresponding checkpoint $map_{t',t}(c_{i,x})$ in $\widehat{H}_t$, the following properties hold:*

- *The index of volatile checkpoint is equal to that of the corresponding real checkpoint. That is,*

  $$V(map_{t',t}(c_{i,x}))[i] = V(c_{i,x})[i] = x.$$

- *The vector clock of the volatile checkpoint is less than or equal to that of the corresponding real checkpoint. That is, $V(map_{t',t}(c_{i,x}))[j] \leq V(c_{i,x})[j]$, $j \neq i$.*

Proof The first property directly holds from the above construction procedure for the volatile checkpoints.

Now consider the second property. If $map_{t',t}(c_{i,x})$ is real in stage t (and t'), then $map_{t',t}(c_{i,x}) = c_{i,x}$, and the second property clearly holds. On the other hand, assume that $map_{t',t}(c_{i,x})$ is volatile. From the above construction procedure for the volatile checkpoints, we can derive that

$$V(map_{t',t}(c_{i,x}))[j] = V(c_{i,l_i})[j], \quad (2)$$

where $j \neq i$. Since $c_{i,x}$ occurs after c_{i,l_i} in $H_{t'}$, we can also derive that $c_{i,l_i} \rightarrow c_{i,x}$ and thus

$$V(c_{i,l_i})[j] \leq V(c_{i,x})[j] \quad (3)$$

For example, assume that $V(c_{i,x}) = [3, 1, 2, 1]$ in $H_{t'}$ in Fig. 5a. Since [3, 1, 2, 3] is $H_{t'}$ but not in H_t (Fig. 5c), there is a corresponding volatile checkpoint $V(map_{t',t}(c_{i,x})) =$ [3, 1, 1, 1] in $\widehat{H}_t$ (Fig. 5b). Furthermore, in Fig. 5b, the real checkpoint before [3, 1, 1, 1] is $V(c_{i,l_i}) = [2, 1, 1, 1]$. The above Eqs. 2 and 3 can be easily examined for this example.

Combining Eqs. 2 and 3 concludes that $V(map_{t',t}(c_{i,x}))[j] \leq V(c_{i,x})[j]$ and completes the proof. □

Lemma 2 shows the *correctness* property of the volatile checkpoints.

Lemma 2 *Consider a real checkpoint c in H_t. If c is removable in $\widehat{H}_t$ then c is removable in all future C-patterns $H_{t'}$ of H_t, $t' > t$.*

Proof The proof is by contradiction. Assume that c is non-removable in $H_{t'}$, $t' > t$, but c is removable in $\widehat{H}_t$. By this assumption, findings show a consistent global checkpoint $C^{t'}$ containing c in $H_{t'}$. Recall that $c_{j,y} \not\rightarrow c_{i,x}$ for each pair of checkpoints $c_{j,y}$ and $c_{i,x}$ in $C^{t'}$ (since $C^{t'}$ is a consistent global checkpoint).

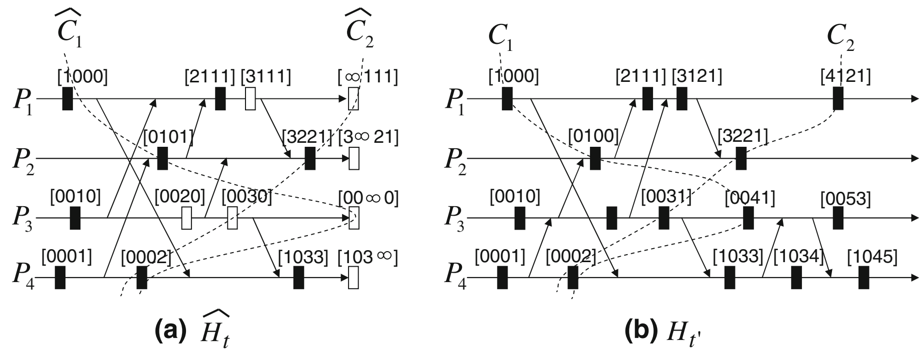

Fig. 6 Global checkpoint C_1 (or C_2) in $H_{t'}$ and its corresponding set $\widehat{C}_1$ (or $\widehat{C}_2$) in $\widehat{H}_t$

Now let $C^t = \{map_{t',t}(a) \mid a \in C^{t'}\}$ be a global checkpoint in $\widehat{H}_t$. Consider the checkpoints $map_{t',t}(c_{i,x})$ and $map_{t',t}(c_{j,y})$ in $\widehat{H}_t$ correspond to $c_{j,y}$ and $c_{i,x}$ in $H_{t'}$. From Lemma 1, Property P2 and Property P3, we can derive the following properties:

$$\begin{aligned}
&c_{j,y} \not\rightarrow c_{i,x} \text{ in } H_{t'}\\
\Rightarrow\ & V(c_{i,x})[j] < y\\
\Rightarrow\ & (V(map_{t',t}(c_{i,x}))[j] \le V(c_{i,x})[j]) < y\\
\Rightarrow\ & V(map_{t',t}(c_{i,x}))[j] < y\\
\Rightarrow\ & map_{t',t}(c_{j,y}) \not\rightarrow map_{t',t}(c_{i,x}) \text{ in } \widehat{H}_t
\end{aligned}$$

For example, in Fig. 6, both global checkpoints C_1 (or C_2) in $H_{t'}$ and $\widehat{C}_1 = \{map_{t',t}(a) \mid a \in C_1\}$ (or $\widehat{C}_2 = \{map_{t',t}(a) \mid a \in C_2\}$) in $\widehat{H}_t$ are consistent.

From above, C^t is a consistent global checkpoint in $\widehat{H}_t$ and contains the real checkpoint c. This contradicts that c is removable in $\widehat{H}_t$. Thus, the lemma follows. □

4.3 Minimum property of volatile checkpoints

Recall that the minimum property finds the minimum stage t such that the checkpoints are t-removable. On the other hand, if a checkpoint c is determined as non-removable in stage t, then c must be possibly useful in some future stage $t' > t$. To show this, we construct a possible future C-pattern H of H_t and show that all the non-removable checkpoints in H_t are also non-removable in H.

In our problem model, the checker process collects only the timestamps of checkpoints, instead of every execution state and message of the distributed program. Consequently, constructing such H is basically a re-engineering process which constructs a communication graph (used to model the execution of the distributed program) by given a set of timestamps of H_t.

Next, Lemma 3 proves the minimum property based on this principle.

Lemma 3 *If a real checkpoint c is non-removable (i.e. useful) in $\widehat{H}_t$, then it is possible that c is non-removable in some future stage t', $t' > t$.*

Proof For this lemma, it suffices to show the following: if a real checkpoint c is useful in $\widehat{H}_t$ then there must be a possible future C-pattern (of H_t) in which c is useful. In this proof, we shall show that $\widehat{H}_t$ is itself the possible future C-pattern.

Given the set of checkpoints in $\widehat{H}_t$ as well as their timestamps, a distributed computation $\mathcal{C}$ with p processes is constructed as follows. For each real checkpoint $c_{i,x}$ or volatile checkpoint $d_{i,x}$ in $\widehat{H}_t$, there is a corresponding checkpoint $a_{i,x}$ in computation $\mathcal{C}$. Let $a \mapsto b$ denote the message *directly* sent from checkpoint a to checkpoint b. Processes in computation $\mathcal{C}$ exchange messages based on the following rule:

M1 $a_{j,y} \mapsto a_{i,x}$ in computation $\mathcal{C}$ if and only if $c_{i,x} \in \widehat{H}_t$ and $V(c_{i,x})[j] = y$.

Based on Rule M1, the following property establishes the relations between $\mathcal{C}$ and $\widehat{H}_t$:

$$\begin{aligned}
&(a_{j,y} \rightarrow a_{i,x})\\
&\Leftrightarrow ((c_{j,y} \rightarrow c_{i,x}) \vee (d_{j,y} \rightarrow c_{i,x})\\
&\vee (c_{j,y} \rightarrow d_{i,x}) \vee (d_{j,y} \rightarrow d_{i,x}))
\end{aligned} \tag{4}$$

Equation 4 is explained as follows. First consider the case that real checkpoint $c_{i,x}$ is in $\widehat{H}_t$. If $(c_{j,y} \rightarrow c_{i,x}) \vee (d_{j,y} \rightarrow c_{i,x})$ then clearly $V(c_{i,x})[j] \ge y$ (since $c_{j,y}$ or $d_{j,y}$ may not be the last checkpoint in P_j that happens before $c_{i,x}$). Suppose that $V(c_{i,x})[j] = y'$ for some $y' \ge y$. Based on Rule M1, a message $a_{j,y'} \mapsto a_{i,x}$ must be involved in computation $\mathcal{C}$, which implies that $a_{j,y} \rightarrow a_{j,y'} \rightarrow a_{,x}$. Conversely, consider the case where $a_{j,y} \rightarrow a_{i,x}$ in computation $\mathcal{C}$. If this relation is induced by a direct message $a_{j,y} \mapsto a_{i,x}$, the condition $(c_{j,y} \rightarrow c_{i,x}) \vee (d_{j,y} \rightarrow c_{i,x})$ holds by Rule M1. Alternatively, assume that the precedence relation is induced by a sequence of direct messages $a_{j_{k-1},y_{k-1}} \mapsto a_{j_k,y_k}$, $2 \le k \le r$, in computation $\mathcal{C}$, where $a_{j_1,y_1} = a_{j,y}$ and $a_{j_r,y_r} = a_{i,x}$. From Rule M1, we have $c_{j_k,y_k} \in \widehat{H}_t$ and $V(c_{j_k,y_k})[j_{k-1}] = y_{k-1}$ for all $2 \le k \le r$. For $k = 2$, the condition $V(c_{j_2,y_2})[j_1] = y_1$ implies that either $c_{j_1,y_1} \rightarrow c_{j_2,y_2}$ or $d_{j_1,y_1} \rightarrow c_{j_2,y_2}$. Together with the properties that all $c_{j_2,y_2}, c_{j_3,y_3}, \ldots, c_{j_r,y_r}$ are real checkpoints and c_{j_2,y_2}

$\rightarrow c_{j_3,y_3} \rightarrow \cdots \rightarrow c_{j_r,y_r}$, we can derive that either $c_{j_1,y_1} \rightarrow c_{j_r,y_r}$ or $d_{j_1,y_1} \rightarrow c_{j_r,y_r}$ (i.e. either $c_{j,y} \rightarrow c_{i,x}$ or $d_{j,y} \rightarrow c_{i,x}$). From above, we have shown that $(a_{j,y} \rightarrow a_{i,x}) \Leftrightarrow ((c_{j,y} \rightarrow c_{i,x}) \vee (d_{j,y} \rightarrow c_{i,x}))$ if real checkpoint $c_{i,x}$ is in $\widehat{H}_t$. For another case that the volatile checkpoint $d_{i,x}$ is in $\widehat{H}_t$, since the precedence relation of volatile checkpoints depend on real checkpoints (see properties P4 and P5), Eq. 4 clearly holds for both real and volatile checkpoints.

From the vector clock mechanism discussed in Sect. 2.3, the timestamp of a checkpoint c records the index numbers of those checkpoints happened before c. Thus, Eq. 4 implies that $V(a_{i,x}) = V(c_{i,x})$ or $V(a_{i,x}) = V(d_{i,x})$ for each $a_{i,x}$. That is, the execution of distributed computation $\mathcal{C}$ generates the vector clocks the same as that in $\widehat{H}_t$. Therefore, $\widehat{H}_t$ is a possible future C-pattern of H_t, and the lemma is satisfied. □

4.4 Compressing volatile checkpoints

Suppose that H is the C-pattern constructed by Algorithm 1 with input data H_t. Let $H' = H \setminus \{d_{i,x} \mid x \neq \infty,\ d_{i,x} \in H\}$. Lemma 4 shows that H and H' are equivalent for identifying consistent global checkpoints.

Lemma 4 *Suppose that c is a real checkpoint in both H and H'. A consistent global checkpoint $\mathcal{C}$ contains c in H if and only if there is a consistent global checkpoint $\mathcal{C}'$ that contains c in H'.*

Proof ($\Rightarrow$) Since $H' \subseteq H$, we can define a many-to-one mapping $m(a)$ from H to H': if a is real then $m(a) = a$, otherwise, $m(a)$ is equal to the last volatile checkpoint (with ∞-index) in the same process for a. For each consistent global checkpoint $\mathcal{C}$ containing c in H, the corresponding $\mathcal{C}' = \{m(a) \mid a \in \mathcal{C}\}$ must also be consistent, explained as follows. Since $\mathcal{C}$ is consistent, for each pair of a and b in $\mathcal{C}$, the property $a \not\rightarrow b$ holds. This implies that $m(a) \not\rightarrow m(b)$ in $\mathcal{C}'$, which implies that $\mathcal{C}'$ is consistent, as discussed in the following cases:

- Both a and b are real: In this case, $m(a) = a$ and $m(b) = b$. Clearly, $a \not\rightarrow b$ implies that $m(a) \not\rightarrow m(b)$ in $\mathcal{C}'$.
- a is volatile and b is real: In this case, $m(a) = a_\infty$ and $m(b) = b$, where a_∞ is the last volatile checkpoint in the process, and cannot have precedence relation to any other checkpoint. Thus, $a_\infty \not\rightarrow b$ in $\mathcal{C}'$.
- a is real and b is volatile: In this case, $m(a) = a$ and $m(b) = b_\infty$, where b_∞ is the last volatile checkpoint in the process. Note that both b and b_∞ are in process P_i. Let b_{l_i} be the *last* real checkpoint in P_i. Based on Property P4 and P5, $a \rightarrow b \Leftrightarrow a \rightarrow b_{l_i}$. Therefore, condition $a \not\rightarrow b$ implies that $a \not\rightarrow b_{l_i}$ and furtherly $a \not\rightarrow b_\infty$.
- Both a and b are volatile: the proof is the same as the previous one.

($\Leftarrow$) Since all checkpoints in H' are presented in H, every global checkpoint in H' must be exist in H. Moreover, the timestamps of checkpoints are unaltered from H' to H. Thus, every consistent global checkpoint in H' must also be a consistent global checkpoint in H, and the proof follows immediately. □

From above, a real checkpoint c is useless in H if and only if c is useless in H'. Thus, the normal volatile checkpoints in H and the compressed volatile checkpoints in H' are equivalent for identifying useless/useful checkpoints. The remainder of this paper only uses compressed volatile checkpoints.

5 Basic algorithms for identifying removable checkpoints

This section discusses the properties of the removable checkpoints, and presents the basic algorithms to identify all the removables. Section 5.1 discusses the properties of checkpoints for a single C-pattern, and Sect. 5.2 extends the properties to a sequence of C-patterns. Section 5.3 describes an efficient data structure, called a front/rear global checkpoint to improve the algorithm.

5.1 Identify removables in a single C-pattern

Researchers in [15,17] proposed the notion of *Z-path* and *Z-cycle*, and showed how to determine the removable checkpoints.

Definition 3 A Z-path exists from checkpoint $c_{i,x}$ to $c_{j,y}$ in C-pattern H if and only if a sequence of pairs of checkpoints in H exists as follows: $(c_{i,x} \rightarrow c_{i_1,x_1})$, $(prev(c_{i_1,x_1}) \rightarrow c_{i_2,x_2})$, $(prev(c_{i_2,x_2}) \rightarrow c_{i_3,x_3}), \ldots,$ $(prev(c_{i_k,x_k}) \rightarrow c_{j,y})$. If $c_{i,x}$ has a Z-path to $c_{j,y}$, then it is denoted by $c_{i,x} \rightsquigarrow c_{j,y}$; otherwise, it is denoted by $c_{i,x} \not\rightsquigarrow c_{j,y}$.

A Z-path with the same start and end point is called a Z-cycle.

Figure 7 has a Z-path from checkpoint $c_{2,1}$ to $c_{1,1}$, and a Z-cycle from $c_{3,2}$ to $c_{3,2}$ itself. Theorem 1 shows the fundamental relationship among Z-paths, Z-cycles and global checkpoints.

Theorem 1 *In a C-pattern H, $c \rightsquigarrow c'$ if and only if both checkpoints c and c' do not belong to the same consistent global checkpoint in H. Furthermore, $c \rightsquigarrow c$ if and only if c is removable in H, i.e. c does not belong to any consistent global checkpoint in H.*

Proof See [15,17]. □

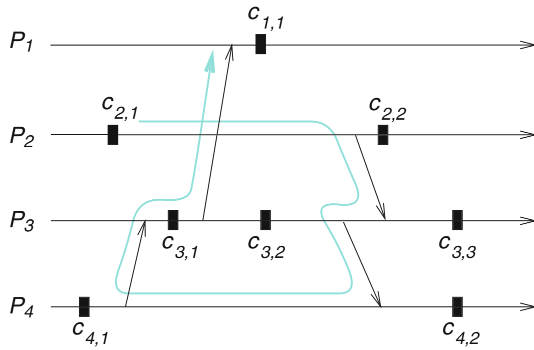

Fig. 7 Z-path $(c_{2,1} \rightsquigarrow c_{1,1}) = \{(c_{2,1} \rightarrow c_{3,3}), (prev(c_{3,3}) \rightarrow c_{4,2}), (prev(c_{4,2}) \rightarrow c_{1,1})\}$

5.2 Identify removables in a sequence of C-patterns

The previous subsection discusses the properties of Z-path to a single C-pattern. In this subsection. Lemmas 5 and 6 extend the properties to the incremental growing C-patterns $H_1, H_2, \ldots, H_M$, where M is the total number of checkpoints reported.

Lemma 5 *Assume that c and c' are two real checkpoints in $\widehat{H}_t$. If $c \rightsquigarrow c'$ in $\widehat{H}_t$, then $c \rightsquigarrow c'$ in $H_{t'}$, $1 \le t \le t'$.*

Proof This lemma shows that a Z-path in H_t will not be broken in the future as new checkpoints are reported. According to the FIFO assumption (Definition 1), the checker process appends each newly received checkpoint to the end of the process queue. This action does not alter the happen-before relationships between old non-volatile checkpoints. Thus, the lemma holds for the Z-paths with only non-volatile member checkpoints.

Next, assume that one volatile checkpoint d is involved in the Z-path. That is, $c \rightsquigarrow d$ and $prev(d) \rightsquigarrow c'$. From the construction algorithm of volatile checkpoints (Algorithm 1), $c \rightsquigarrow d$ in $\widehat{H}_t$ implies that $c \rightsquigarrow c''$ for some real checkpoint c'' which occurs before d. Thus, another Z-path $c \rightsquigarrow c''$ and $c'' \rightsquigarrow c'$, with only real member checkpoints, can be constructed in $\widehat{H}_t$. Thus, the lemma also holds for this case. □

Lemma 6 *The Z-path $c \rightsquigarrow c$ exists in H_t if and only if c is t-removable.*

Proof According to Lemma 5, if $c \rightsquigarrow c$ in H_t then $c \rightsquigarrow c$ in all $H_{t'}$, $t' > t$, which directly implies that c is removable in all $H_{t'}$ from Theorem 1. □

Based on Lemma 6, the z-cycles can be utilized to identify the removable checkpoints, as shown in the following straightforward Algorithm $\mathcal{A}$ which runs on the checker process. (Note that Algorithm $\mathcal{A}$ only serves as a high-level guideline for our concrete algorithms, no further implementation is provided here.)

Algorithm $\mathcal{A}$

1. When the checker process receives a new checkpoint c_t. Let H_t be the current C-pattern.
2. Derive all the Z-paths in H_t.
3. For each Z-path, if it is a Z-cycle of c then identify c as a t-removable.
4. Repeat the first step.

5.3 Using front/rear global checkpoints

In Algorithm $\mathcal{A}$, the number of Z-paths is obviously very large. This subsection describes a new algorithm based on an efficient data structure called *front/rear global checkpoint* as an alternative to using Z-paths directly. The front and rear global checkpoints are defined as follows.

Definition 4 For a checkpoint c in the C-pattern H_t, its front/rear global checkpoint is defined as follows:

- The front global checkpoint of c, denoted by $F_t(c)$, represents the earliest checkpoint in each process which has no Z-path to c. Specifically, $F_t(c)[j] \not\rightsquigarrow c$, but $prev(F_t(c)[j]) \rightsquigarrow c$ for all $j = 1, 2, \ldots, p$.
- The rear global checkpoint of c, denoted by $R_t(c)$, represents the latest checkpoint in each process which has no Z-path from c. Specifically, $c \not\rightsquigarrow R_t(c)[j]$, but $c \rightsquigarrow next(R_t(c)[j])$ for all $j = 1, 2, \ldots, p$.

Note that both $F_t(c)$ and $R_t(c)$ are well-defined in H_t from the assumptions of existence of the initial and volatile checkpoints. Hereinafter, we use the compressed volatile checkpoint which is discussed in 4.4. Figure 8 illustrates an example of front/rear global checkpoints.

Next, some useful notations and properties are described as follows. For a checkpoint c, let $Z_t(c)$ be the set of checkpoints having Z-paths to c, and let $Z_t^{-1}(c)$ be the set of checkpoints having Z-paths from c. That is, $Z_t(c) = \{d \mid d \rightsquigarrow c\}$ and $Z_t^{-1}(c) = \{d \mid c \rightsquigarrow d\}$ in H_t. As shown in Fig. 8, $F_t(c)$

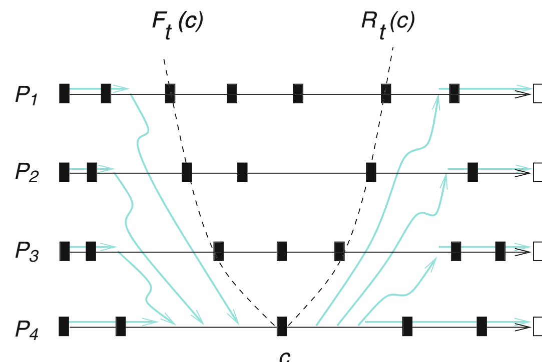

Fig. 8 Front and rear global checkpoints $F_t(c)$ and $R_t(c)$

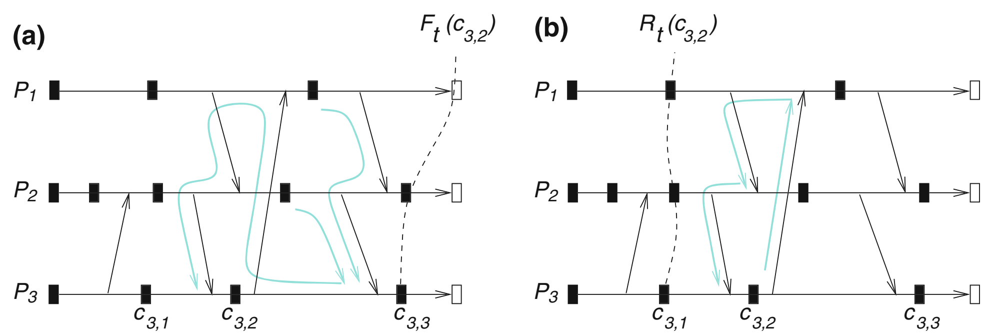

Fig. 9 Front and rear global checkpoints of checkpoint $c_{3,2}$

represents the right margin of $Z_t(c)$ while $R_t(c)$ represents the left margin of $Z_t^{-1}(c)$.

A0 As shown in Fig. 8, the following properties hold for checkpoints c and d:

- For some checkpoint $d, d \in Z_t(c)$ if and only if $d \prec F_t(c)$, and
- For some checkpoint $d, d \in Z_t^{-1}(c)$ if and only if $R_t(c) \prec d$.

A1 Assume that $c \rightsquigarrow c'$, then $F_t(c) \preceq F_t(c')$.
If $c \rightsquigarrow c'$ then $d \rightsquigarrow c \rightsquigarrow c'$ for any checkpoint d. That is, $Z_t(c) \subseteq Z_t(c')$. Hence, this property follows from the fact that $F_t(c)$ and $F_t(c')$ represent the right margins of $Z_t(c)$ and $Z_t(c')$, respectively.

A2 Assume that $c \rightsquigarrow c'$, then $R_t(c) \preceq R_t(c')$.
The proof is similar to that of A1 and is omitted.

A3 Assuming that $t < t^+$, the following properties hold for all checkpoints c in H_t.

- $F_t(c) \preceq F_{t^+}(c)$. Restated, as t increases, $F_t(c)$ moves *rightwards* in the time diagram as depicted in Fig. 8.
  From Lemma 5, a Z-path $d \rightsquigarrow c$ in H_t continues to exist in its descendant H_{t^+}. This implies that $Z_t(c) \subseteq Z_{t^+}(c)$. Hence, this property holds as $F_t(c)$ and $F_{t^+}(c)$ represent the right margins of $Z_t(c)$ and $Z_{t^+}(c)$, respectively.
- Similarly, $R_{t^+}(c) \preceq R_t(c)$. Restated, as t increases, $R_t(c)$ moves *leftwards* in the time diagram.

A4 In C-pattern H_t, a checkpoint c is in a Z-cycle if and only if $c \prec F_t(c)$ and $R_t(c) \prec c$.
This property holds by Definition 4. Consider the example in Fig. 9a, any checkpoint having Z-paths to $c_{3,2}$ is in the left side of line $F_t(c_{3,2})$. Thus, if $c_{3,2} \rightsquigarrow c_{3,2}$, then $c_{3,2}$ itself is in the left side of $F_t(c_{3,2})$. Similarly, $R_t(c_{3,2}) \prec c_{3,2}$ can be shown.

Based on Property A4, the new checkpointing algorithm employing the rear global checkpoints is described as follows.

Algorithm $\mathcal{B}$

1. When the checker process receives a new checkpoint c_t. Let H_t be the current C-pattern.
2. Derive $R_t(c)$ for all checkpoints c in H_t.
3. For each checkpoint c, if $c \notin R_t(c)$ then mark c as a t-removable.
4. Repeat the first step.

Algorithm $\mathcal{B}$ can use front global checkpoints instead of rear global checkpoints. However, this study adopted rear global checkpoints, since they were found to yield a better time complexity than front global checkpoints.

6 Improved checkpointing algorithm

Maintaining all the rear global checkpoints is advantageous in that many of them remain unchanged from one stage to the next. In this section, a C-pattern is partitioned into *changed area* and *unchanged area*. Only the data structures of the checkpoints in the changed area need to be maintained. A new efficient algorithm is developed in this section based on this observation.

6.1 Changed and unchanged areas

Let c_t be the tth reported checkpoint and $H_t = H_{t-1} \cup \{c_t\}$. Also, let $c_{t'} = prev(c_t)$. Note that $t' \leq t-1 < t$ and $H_{t'} \subseteq H_{t-1} \subset H_t$. As illustrated in Fig. 10, the set H_t excluding c_t can be partitioned into three disjoint areas by using $F_{t'}(c_{t'})$ and $F_t(c_t)$:

- First area: all the checkpoints c satisfying $c \prec F_{t'}(c_{t'}) \prec F_t(c_t)$.
- Second area: all the checkpoints c satisfying $F_{t'}(c_{t'}) \preceq c$ and $c \prec F_t(c_t)$.
- Third area: all the checkpoints c satisfying $F_t(c_t) \preceq c$.

In the three areas, only the checkpoints c in the second area can have the new relation $c \rightsquigarrow d$ for some d because:

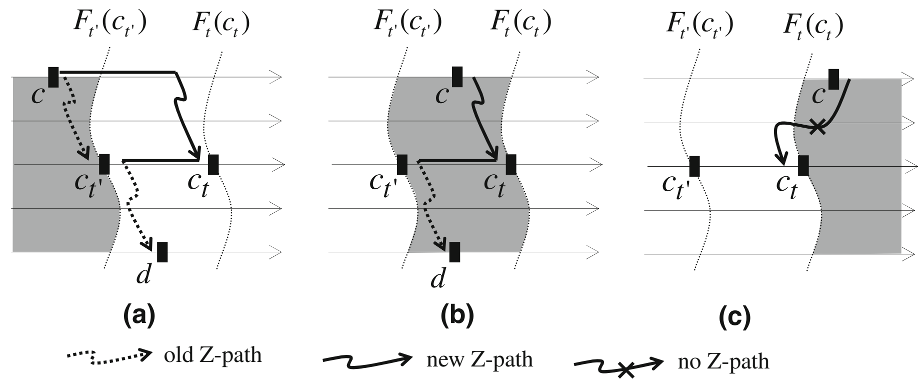

Fig. 10 Three areas in C-pattern H_t: **a** first area, **b** second area, and **c** third area

– If c is in the third area (Fig. 10c), then c satisfies $c \not\rightsquigarrow c_t$. Since any new Z-paths in H_t must be connected via the new checkpoint c_t (otherwise, all the checkpoints in the Z-path is old, implies that the Z-path is old), $c \not\rightsquigarrow c_t$ implies that c has no new Z-path to any other checkpoints.
– If c is in the first area (Fig. 10a), then a new Z-path $(c \rightsquigarrow c_t)$, $(c_{t'} \rightsquigarrow d)$ is generated in H_t. However, since $c \rightsquigarrow c_{t'}$, another old Z-path $(c \rightsquigarrow c_{t'})$, $(c_{t'} \rightsquigarrow d)$ can be constructed as illustrated by the dotted lines in the figure.

Lemma 7 gives the formal proof.

Lemma 7 *Let c_t be the tth reported checkpoint and $H_t = H_{t-1} \cup \{c_t\}$. For checkpoints c and d, $c \neq c_t$ and $d \neq c_t$, if $c \rightsquigarrow d$ in H_t but $c \not\rightsquigarrow d$ in H_{t-1} then c must be in the second area of H_t.*

Proof Initially, some additional notation is introduced. Let $c_{t'} = prev(c_t)$. Also, if a Z-path exists from c to d in H_t then it is denoted by $(c \rightsquigarrow d)^{(t)}$.

Without loss of generality, this proof assumes that a checkpoint can occur *at most once* in a Z-path. If a Z-path from c to d contains a multiple-occurrence checkpoint c', just employ the *simple* Z-path consists of the sub-Z-paths from c to the first occurrence of c', and, the sub-Z-path from the last occurrence of c' to d.

A new Z-path $\mathcal{P}$ in H_t must be connected via c_t, otherwise, $\mathcal{P}$ is old, since all its members are already presented in H_{t-1}. Thus, the Z-path $\mathcal{P}$ from c to d is new in H_t if and only if

$$(c \rightsquigarrow c_t)^{(t)} \text{ and } (c_{t'} \rightsquigarrow d)^{(t)} \tag{5}$$

Since the above Z-path is simple, c_t can occur only once, i.e. the sub-Z-path $(c_{t'} \rightsquigarrow d)^{(t)}$ contains no new checkpoint c_t. Thus, Eq. 5 implies that

$$(c \rightsquigarrow c_t)^{(t)} \text{ and } (c_{t'} \rightsquigarrow d)^{(t-1)} \tag{6}$$

Equation 5 immediately implies that $c \prec F_t(c_t)$, i.e. c is either in the first area or the second area. Next, the checkpoints in the first area are examined closely. In the first area, all the checkpoints c satisfy the following conditions based on Lemma 5:

$$\begin{aligned} c \prec F_{t'}(c_{t'}) &\Rightarrow (c \rightsquigarrow c_{t'})^{(t')} \\ &\Rightarrow (c \rightsquigarrow c_{t'})^{(t-1)} \end{aligned} \tag{7}$$

If c is in the first area, then an old Z-path from c to d can be constructed by combining Eqs. 6 and 7: $(c \rightsquigarrow c_{t'})^{(t-1)}$ and $(c_{t'} \rightsquigarrow d)^{(t-1)}$. Thus, the relation $c \rightsquigarrow d$ can not be new in H_t if c is in the first area. This completes the proof. □

Based on Lemma 7, Corollaries 1 and 2 show the derivation of the rear global checkpoints for all three areas.

Corollary 1 *Let c_t be the tth reported checkpoint and $H_t = H_{t-1} \cup \{c_t\}$. The following properties hold from H_{t-1} to H_t:*

– *For the checkpoints c in the first and third areas of H_t, their rear global checkpoints remain unchanged.*
– *For the checkpoints c in the second area of H_t, $R_t(c) = \min(R_{t-1}(c), R_{t-1}(c_{t'}))$, where $c_{t'} = prev(c_t)$.*

Proof By definition, deriving $R_t(c)$ requires evaluating all the Z-paths starting from c in H_t (see Fig. 8). From Lemma 7, checkpoints c in the first and third areas of H_t have no new Z-paths, which directly implies that $R_t(c)$ remains unchanged, i.e. $R_t(c) = R_{t-1}(c)$.

Next, consider the checkpoint c in the second area, where $c \prec F_t(c_t)$. From Eq. 5, a new Z-path $c \rightsquigarrow d$ in H_t must be connected via c_t and $prev(c_t)$. Thus,

$$\begin{aligned} &(c \rightsquigarrow d)^{(t)} \text{ is a new Z-path in } H_t \\ \Leftrightarrow\ &(c \rightsquigarrow c_t)^{(t)} \text{ and } (c_{t'} \rightsquigarrow d)^{(t)} \end{aligned} \tag{8}$$

Furthermore, by definition (see Definitions 3 and 4), the above equation can be rewritten as

$$\begin{aligned} &\Leftrightarrow (c \rightsquigarrow c_t)^{(t)} \text{ and } (c_{t'} \rightsquigarrow d)^{(t)} \\ &\Leftrightarrow c \prec F_t(c_t) \text{ and } R_t(c_{t'}) \prec d \end{aligned} \tag{9}$$

That is, every checkpoint c in the second area has new Z-paths to every checkpoint d, where d is the checkpoint having Z-paths from $c_{t'}$ (i.e. $R_t(c_{t'}) \prec d$) (see Fig. 10b). Thus, $R_t(c)$ can be obtained as follows:

$$\begin{aligned} R_t(c) &= \min(R_{t-1}(c), R_t(c_{t'})) \\ &= \min(R_{t-1}(c), R_{t-1}(c_{t'})) \end{aligned}$$

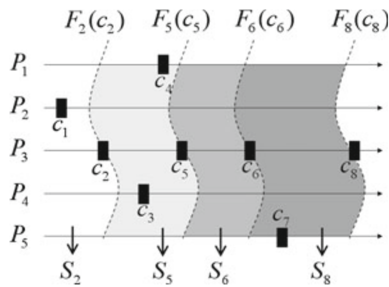

Fig. 11 Checkpoints c_2, c_5, c_6, c_8 that are sequentially reported by P_3. Disjoint areas S_2, S_5, S_6, S_8 refer to the second areas of stages 2, 5, 6, 8, respectively

The above condition $R_t(c_{t'}) = R_{t-1}(c_{t'})$ holds, since $(c_{t'} \rightsquigarrow d)^{(t)}$ implies $(c_{t'} \rightsquigarrow d)^{(t-1)}$ for any checkpoint d, as discussed in Eq. 6. □

Corollary 2 *For a checkpoint c, $R_t(c)$ is updated only p times among M stages (for $t = 1, 2, \ldots, M$), where p is the number of the processes and M is the number of checkpoints.*

Proof We shall prove that for a checkpoint c, $R_t(c)$ is updated only *once* for $t = t_1, t_2, \ldots, t_k$ if the checkpoints $c_{t_1}, c_{t_2}, \ldots, c_{t_k}$ are reported from the *same* process. Then, it can be shown that $R_t(c)$ is updated p times for total p processes.

Figure 11 illustrates the case of four checkpoints, c_2, c_5, c_6, c_8, that are reported from the same process. Their corresponding second areas are marked S_2, S_5, S_6, and S_8 in the figure. These areas are partitioned by $F_2(c_2)$, $F_5(c_5)$, $F_6(c_6)$ and $F_8(c_8)$. Since $F_2(c_2) \preceq F_5(c_5) \preceq F_6(c_6) \preceq F_8(c_8)$ (see Property A3), S_2, S_5, S_6, and S_8 are disjoint (see the definition of second area). In the other words, any checkpoint c can belong to only one of S_2, S_5, S_6, and S_8. From Corollary 1, $R_t(c)$ is updated at most once among stages 2, 5, 6, 8.

The formal proof is as follows. Assume that the checkpoints $c_{t_1}, c_{t_2}, \ldots, c_{t_k}$ are reported from the same process. Let S_{t_i} be the second area of stage t_i, $1 \le i \le k$. By definition, $S_{t_i} = \{d \mid F_{t_{i-1}}(c_{t_{i-1}}) \preceq d \preceq F_{t_i}(c_{t_i})\}$. Since $F_{t_1}(c_{t_1}) \preceq F_{t_2}(c_{t_2}) \preceq \cdots \preceq F_{t_k}(c_{t_k})$ (see Property A3), $S_{t_1}, S_{t_2}, \ldots, S_{t_k}$ are *disjoint*. In the other words, any checkpoint c can belong to only one of $S_{t_1}, S_{t_2}, \ldots, S_{t_k}$. From Corollary 1, $R_t(c)$ is updated at most once among stages $t_1, t_2, \ldots, t_k$. Directly implies that $R_t(c)$ is updated at most p times for all stages. □

6.2 New algorithm

This subsection presents the new algorithm that improves Algorithm $\mathcal{B}$. The main function of the new algorithm is shown in Sect. 6.2.1. Some subroutines are presented in Sects. 6.2.2 and 6.2.3.

6.2.1 Main function

According to Corollary 1, Algorithm $\mathcal{B}$ is improved as follows.

Algorithm $\mathcal{C}$

1. When the checker process receives a new checkpoint c_t. Let H_t be the current C-pattern.
2. Derive $F_t(c_t)$. Partition H_t into the three areas as illustrated in Fig. 10.
3. Update $R_t(c)$ for all c in the second area according to Lemma 1, where $c \neq c_t$.
4. Derive $R_t(c_t)$. (Note that $R_t(c_t)$ is excluded in Step 3.)
5. For each updated $R_t(c)$, if $c \notin R_t(c)$ then mark c as a t-removable.
6. Repeat the first step.

Time complexity of algorithm $\mathcal{C}$

Algorithm $\mathcal{C}$ is very efficient because it only updates the data structures of the checkpoints in partial areas of the C-patterns, rather than all. The time complexity of Algorithm $\mathcal{C}$ is analyzed as follows.

1. Since each checkpoint c updates its $R_t(c)$ at most p times (Corollary 2) and each update takes $O(p)$ time (for the minimum operation in Corollary 1), each checkpoint c takes $O(p^2)$ to update its $R_t(c)$. Therefore, the total time complexity to maintain $R_t(c)$ is $O(p^2M)$ for all M checkpoints.
2. In Step 5, identifying the removables takes no more time than updating all $R_t(c)$, because when updating each $R_t(c)$ the identification can be done simply by verifying whether $R_t(c)[i] = c$, where c is in process P_i.
3. In Step 2, $F_t(c_t)$ is derived to partition the C-pattern H_t. As proven in Sect. 6.2.3 (below), it takes $O(p^2M)$ total time to derive $F_t(c_t)$ for all t.
4. In Step 4, $R_t(c_t)$ is derived. As proven in Sect. 6.2.2 (below), it takes $O(p^2M)$ total time to derive $R_t(c_t)$ for all t.

The above discussion establishes that $O(p^2M)$ time is required in total by Algorithm $\mathcal{C}$.

6.2.2 Deriving $R_t(c_t)$ for all t

In Step 3 of Algorithm $\mathcal{C}$, all $R_t(c)$ for $c \neq c_t$ are obtained. This subsection discusses the derivation of $R_t(c_t)$ in Step 4.

Let $\gamma(c_t)$ be a set of checkpoints in H_t where $\gamma(c_t)[i]$ is the latest checkpoint in process P_i satisfying $c_t \not\rightarrow \gamma(c_t)[i]$ and $c_t \rightarrow next(\gamma(c_t)[i])$. Additionally, c_t is itself excluded from $\gamma(c_t)$. Figure 12a illustrates an example of $\gamma(c_t)$. Lemma 8 indicates that $R_t(c_t)$ can be derived from $R_t(c)$ for all $c \in \gamma(c_t)$.

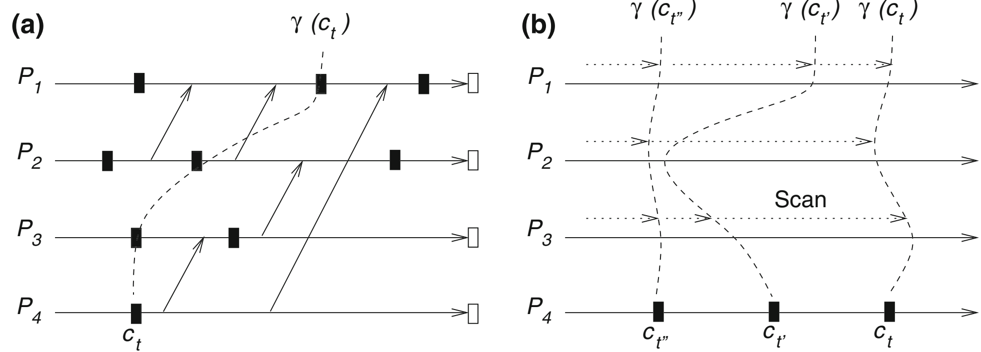

Fig. 12 **a** An example of $\gamma(c_t)$. **b** Illustration of scanning $\gamma(c_{t''})$, $\gamma(c_{t'})$, and $\gamma(c_t)$

Lemma 8 *In C-pattern H_t, $R_t(c_t)$ is $\min_{c,\forall c\in\gamma(c_t)} R_t(c)$.*

Proof Consider a checkpoint c in $\gamma(c_t)$. From Definition 3, each Z-path $c \rightsquigarrow c'$ implies another Z-path $c_t \rightsquigarrow c'$ due to $c_t \rightarrow next(c)$ and $c \rightsquigarrow c'$. Thus, $R_t(c_t) \preceq R_t(c)$. Furthermore, considering all $c \in \gamma(c_t)$, we have $R_t(c_t) \preceq \min_{c,\forall c\in\gamma(c_t)} R_t(c)$.

Consider a checkpoint $d \prec \min_{c,\forall c\in\gamma(c_t)} R_t(c)$. Since $d \prec R_t(c)$ for each $c \in \gamma(c_t)$, no Z-path $c \rightsquigarrow d$ exists. Hence, from Definition 3, this implies that no Z-path $c_t \rightsquigarrow d$ exists. Therefore, $R_t(c_t) = \min_{c,\forall c\in\gamma(c_t)} R_t(c)$. □

Figure 12 shows that each $\gamma(c_t)[i]$ can be obtained by scanning checkpoints in process P_i in the order $c_{i,0}$ to $c_{i,\infty}$. As this figure shows, scanning $\gamma(c_t)[i]$ starts at $\gamma(c_{t'})[i]$ (recall that $c_{t'} = prev(c_t)$) and ends when checkpoint $c_{i,x}$ (which satisfying $c_t \not\rightarrow c_{i,x}$ and $c_t \rightarrow next(c_{i,x})$) is found in process P_i.

The time complexity of deriving all $R_t(c_t)$ is discussed as follows:

- As depicted in Fig. 12b, $\gamma(c_{t''})$, $\gamma(c_{t'})$, and $\gamma(c_t)$ can be derived by scanning M checkpoints just once, where $c_{t''}$, $c_{t'}$, and c_t are in the same process and $t'' < t' < t$. This takes $O(M)$ time. Thus, $O(pM)$ time is required for p processes.
- In Lemma 8, each $R_t(c_t)$ can be derived by performing minimum operations among $R_t(c)$, $\forall c \in \gamma(c_t)$. This minimum operation takes $O(p^2)$ time. Thus, $O(p^2M)$ time is required for all c_t, $t = 1, 2, \ldots, M$.

From above, the time complexity of deriving $R_t(c_t)$ for all c_t is $O(p^2M)$.

6.2.3 Deriving $F_t(c_t)$ for all t

In Step 2 of Algorithm $\mathcal{C}$, the front global checkpoint $F_t(c_t)$ has to be obtained. In this subsection, we simply apply Garg and Waldecker's algorithm [5] to obtain the front global checkpoints. The time complexity of their algorithm is $O(pA)$, where A is the number of checkpoints. Now, assume that process P_i has k checkpoints $c_{t_1}, c_{t_2}, \ldots, c_{t_k}$. From Property A1 and A3, $F_{t_1}(c_{t_1}) \preceq F_{t_2}(c_{t_2}) \preceq \cdots \preceq F_{t_k}(c_{t_k})$ as illustrated in Fig. 11. Let these front global checkpoints partition all M checkpoints into k disjoint parts. By applying Garg and Waldecker's algorithm to each of these k parts, all the front global checkpoints $F_{t_j}(c_{t_j})$, $1 \le j \le k$, can be derived in time $O(pM)$. Thus, for p processes, the total time complexity to derive all $F_t(c_t)$ is $O(p^2M)$.

7 Conclusion

This work designs an efficient algorithm that can incrementally identify all consistent and removable checkpoints. Applications of this algorithm include testing and debugging of distributed programs.

The time complexity for the proposed new algorithm is only $O(p^2M)$, which is much lower than the previous $O(M^2)$-time algorithm, because M usually grows as the program runs, while p usually remains fixed. In our checkpointing algorithms, the memory space can be significantly reduced by disregarding all the removable checkpoints. This result also represents an improvement over previous works, which only delete a subset of removable checkpoints.

Acknowledgments The authors would like to thank the anonymous referees for their valuable comments, which greatly improved the presentation of this paper. The authors would also like to thank the National Science Council of the Republic of China (Taiwan) for financially supporting this research under contract No. NSC96-2221-E-029-027 and NSC97-2221-E-029 -022.

References

1. Chandy, K.M., Lamport, L.: Distributed snapshots: determining global states of distributed systems. ACM Trans. Comput. Syst. **3**(1), 63–75 (1985)
2. Chen, L.B., Wu, I.C.: An efficient distributed online algorithm to detect strong conjunctive predicates. IEEE Trans. Softw. Eng. **28**(11), 1077–1084 (2002)
3. Chiou, H.K., Korfhage, W.: Enhancing distributed event predicate detection algorithms. IEEE Trans. Parallel Distrib. Syst. **7**(7), 673–676 (1996)

4. Cooper, R., Marzullo, K.: Consistent detection of global predicates. Sigplan Notices, pp. 167–174 (1991)
5. Garg, V.K., Waldecker, B.: Detection of weak unstable predicates in distributed programs. IEEE Trans. Parallel Distrib. Syst. **5**(3), 299–307 (1994)
6. Garg, V.K., Waldecker, B.: Detection of strong unstable predicates in distributed programs. IEEE Trans. Parallel Distrib. Syst. **7**(12), 1323–1333 (1996)
7. Helary, J.M., Mostefaoui, A., Netzer, R.H.B., Raynal, M.: Communication-based prevention of useless checkpoints in distributed computations. Distrib. Comput. **13**(1), 29–43 (2000)
8. Hurfin, M., Mizuno, M., Raynal, M., Singhal, M.: Efficient distributed detection of conjunctions of local predicates. IEEE Trans. Softw. Eng. **24**(8), 664–677 (1998)
9. Kshemkalyani, A.D.: The power of logical clock abstractions. Distrib. Comput. **17**(2), 131–150 (2004)
10. Kshemkalyani, A.D.: A fine-grained modality classification for global predicates. IEEE Trans. Parallel Distrib. Syst. **14**(8), 807–816 (2003)
11. Lamport, L.: Time, clocks and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
12. Mattern, F.: Virtual time and global states of distributed systems. In: Parallel and Distributed Algorithms: Proceedings of the International Workshop on Parallel and Distributed Algorithms, pp. 215–226. Elsevier, New York (1989)
13. Mittal, N., Garg, V.K.: Techniques and applications of computation slicing. Distrib. Comput. **17**(3), 251–277 (2005)
14. Manivannan, D., Netzer, R.H.B., Singhal, M.: Finding consistent global checkpoints in a distributed computation. IEEE Trans. Parallel Distrib. Syst. **8**(6), 165–169 (1997)
15. Netzer, H.B., Xu, J.: Necessary and sufficient conditions for consistent global snapshots. IEEE Trans. Parallel Distrib. Syst. **6**(2), 165–169 (1995)
16. Schwarz, R., Mattern, F.: Detecting causal relationships in distributed computations: In search of the holy grail. Distrib. Comput. **7**(3), 149–174 (1994)
17. Wang, Y.M.: Consistent global checkpoints that contain a given set of local checkpoints. IEEE Trans. Comput. **46**(4), 456–468 (1997)
18. Wang, Y.M., Chung, P.Y., Lin, I.J., Fuchs, W.K.: Checkpoint space reclamation for uncoordinated checkpointing in message-passing systems. IEEE Trans. Parallel Distrib. Syst. **6**(5), 546–554 (1995)
19. Wang, Y.M., Lowry, A., Fuchs, W.K.: Consistent global checkpoints based on direct dependency tracking. Inf. Process. Lett. **50**(4), 223–230 (1994)
20. Wu, I.C., Chen, L.B.: On detection of bounded global predicates. Comput. J. **41**(4), 231–237 (1998)
21. Yen, L.H.: Precluding useless events for on-line global predicate detections. J. Parallel Distrib. Comput. **61**(8), 1077–1095 (2001)