FINE: A Fully Informed aNd Efficient Communication-Induced Checkpointing Protocol for Distributed Systems

Yi Luo and D.Manivannan *

Department of Computer Science, University of Kentucky,
Lexington, KY 40506, USA
Technical Report # TR 492-08

Abstract

Communication-Induced Checkpointing (CIC) protocols are classified into two categories in the literature: *Index*-based and *Model*-based. In this paper, we discuss the intrinsic relation between the two categories and provide our classification of CIC protocols based on how they achieve the same goal of ensuring Z-Cycle Free (ZCF) property by tracking the checkpoint and communication pattern (CCP) that can lead to Z-cycles and preventing them. Then, based on our Transitive Dependency Enabled TimeStamp (TDE_TS) mechanism, we present our Fully Informed aNd Efficient **(FINE)** algorithm which can not only improve the performance of Fully Informed **(FI)** CIC protocol proposed by Helary et al. but also decrease the overhead of information piggybacked with application messages.

Key words: Distributed systems, communication-induced checkpointing protocols, consistent global checkpoints

1. Introduction

Fault-tolerance (or graceful degradation) is the property that enables a system to continue operating properly in the event of the failure of some of its components. Checkpointing and rollback recovery are recognized techniques for providing fault tolerance for distributed computations. Depending on saved states of processes in the stable storage during execution, such techniques allow processes to make progress in spite of failures.

In case of a system failure, it is desirable to restart an application from an intermediate state instead of from the beginning since it reduces the amount of recomputation and saves valuable computing time, especially for long-running, time-critical applications. However, taking checkpoints independently in each process can not guarantee that the whole system can rollback to a consistent global checkpoint which minimizes the amount of recomputation when failure happens. Due to the interprocess communication, rollback propagation can occur when the rollback of a message sender leads to the rollback of the corresponding receiver. Moreover, it is likely to result in unbounded, cascading rollback propagation known as *domino effect*. Therefore, in case of failures, how to minimize the lost amount of computation due to roll back is a fundamental problem in distributed computations.

To address this problem, several checkpointing protocols have been presented in the literature. They can be classified into three categories: *uncoordinated* checkpointing, *coordinated* checkpointing, and *communication-induced* checkpointing[10]. Uncoordinated Checkpointing Protocols allow each process to take checkpoints at each process's own convenience. Coordinated Checkpointing Protocols require processes to synchronize their checkpointing procedures so that each checkpoint in each process is guaranteed to be useful, and the synchronization among processes is usually achieved by some kind of two-phase commit algorithm. Communication-Induced Checkpointing (CIC) protocols allow processes to take checkpoints independently and ensure no useless checkpoints in the system by forcing processes to take additional checkpoints when needed. With regard to the possibly unbounded rollback propagation of *uncoordinated* checkpointing protocols and the possibly large latency involved in some kind of two-phase commit algorithms to form a consistent global checkpoint of *coordinated* checkpointing

* Corresponding author.
Email addresses: yiluo@cs.uky.edu (Yi Luo), mani@cs.uky.edu (D.Manivannan).
URL: http://www.cs.uky.edu/~manivann (D.Manivannan).

protocols, *Communication-Induced Checkpointing (CIC)* protocols[1,6–8,13,16,18,19,21,24,30,31] have received more and more attention due to their attractive features: CIC protocols can not only bound rollback propagation but also allow each process to take checkpoints independently while at the same time guaranteeing that each checkpoint is useful. This goal is achieved by taking some forced checkpoints to ensure no useless checkpoint in the system. Moreover, CIC protocols provide distributed algorithms so that they don't need any centralized strategy to collect information from each process then make offline analysis and decision during recovery.

CIC protocols are classified into two categories: *Model*-based and *Index*-based CIC protocols in the literature. However, in this paper we argue that this classification of CIC protocols is not clear enough. The basic idea behind a classification of protocols is to facilitate and hence to promote understanding of their features, advantages and disadvantages. Thus a clear classification of CIC protocols is essential to better understand the characteristics of CIC protocols, which in turn can help us design more efficient CIC protocols. This leads to the first contribution of this paper, namely, our classification of CIC protocols from the point of view of data structures used in these protocols; moreover we show that *Model*-based and *Index*-based protocols are intuitively derived from the same source by revealing the intrinsic relation between them. Our second contribution is the Basic *Fully Informed aNd Efficient* CIC protocol, which creates the stronger checkpoint-inducing condition than the Fully Informed CIC protocol presented in [16] without any extra overhead of piggybacked information. It is followed by the proof of its correctness. The last but not least contribution of this paper is the Advanced *Fully Informed aNd Efficient* CIC protocol. Based on the above intrinsic relation analysis, we apply the *Transitive Dependency Enabled TimeStamp* (TDE_TS) mechanism to the *Basic FINE* protocol to obtain our *Advanced FINE* protocol which uses a precise checkpoint-inducing condition while at the same time decreasing the overhead of piggybacked information. The simulation results show the performance comparison of our proposed FINE protocol with five other protocols.

The rest of the paper is structured as follows. Section 2 introduces the model of the distributed computation and background required. In Section 3, we provide our finer classification of CIC protocols based on the characteristics and the data structures used in existing CIC protocols. Then, in Section 4, we present our Basic and Advanced FINE CIC protocols, followed by the proof of its correctness. We also compared performance of our protocol with five other CIC protocols through simulation in Section 5. And Section 6 summaries related work. Finally, we conclude in Section 7.

2. Background

In this section, we introduce the system computational model, assumptions, definitions, fundamental concepts and principal theorems which our research is based on.

2.1. *System Model*

A distributed system consists of a collection of autonomous computers linked by a computer network and equipped with distributed system softwares. We define a distributed computation as a finite set of n processes $P_0, P_1, \ldots, P_{n-1}$ running concurrently on a set of computers in the network. The only way for processes to communicate with each other is by passing messages through a reliable, asynchronous directed channel with uncertain but finite transmission delays. The failures of processes follow fail-stop model[26], namely, if a process fails, then it simply stops.

We can view a process as the occurrence of certain events within the computation environment and these events can be among *internal*, *sending* and *receiving* events. An internal event doesn't include communication. However a sending event and a receiving event correspond to the execution of the statements "*send(m)*" and "*receive(m)*" respectively. Let H be the set of all the events produced by processes in a distributed computation, then the well-known Lamport's *happened-before* relation[20] on events, $\xrightarrow{hb}$, is used to model the distributed computation as a partially ordered set $\hat{H} = (H, \xrightarrow{hb})$.

Definition 1 *A checkpoint and communication pattern (CCP) is a 2-tuple $(\hat{H}, \mathscr{C}_{\hat{H}})$, where $\hat{H}$ is a partially ordered set modeling a distributed computation and $\mathscr{C}_{\hat{H}}$ is a set of local checkpoints involved in $\hat{H}$ [10,32].*

A local checkpoint of a process is considered as an event which records the state of the process at a given instance. Each checkpoint of a process is assigned a unique sequence number. $C_{i,x}$ denotes the x-th local checkpoint of process P_i and $I_{i,x}$ is called a checkpoint interval which contains the sequence of events happening at P_i between $C_{i,x}$ and $C_{i,x+1}$, including $C_{i,x}$ but excluding $C_{i,x+1}$. A *global checkpoint* is a set of local checkpoints, one from each process.

Definition 2 *A global checkpoint is consistent if it does not contain messages received but not sent (*orphan messages*) with respect to any pair of local checkpoints in this global checkpoint.*

A consistent global checkpoint (also called a recovery line) is where the whole system can rollback to in case of a failure. The latest consistent global checkpoint is always desirable for reducing the amount of recomputation and saving valuable computing time in rollback recovery.

2.2. *Z-Dependency*

As we know, two local checkpoints can be causally related or unrelated according to *happened-before* relation. Two causally unrelated checkpoints may belong to the same consistent global checkpoint, but only causally unrelated condition is not enough. The truth that a type of "hidden" dependency between two local checkpoints (even if they are causally unrelated) can prevent them from being part of the same consistent global checkpoint was first discovered by Netzer and Xu[23] based on the notion of Z-path (or Zigzag path), described below.

Definition 3 *A Z-path from $C_{i,x}$ to $C_{j,y}$ is said to exist if there is a sequence of messages $[m_1, m_2, \ldots, m_q]$ $(q \geq 1)$ such that:*
(i) m_1 is sent by P_i after taking its checkpoint $C_{i,x}$, and
(ii) for each m_i, $1 \leq i < q$: $receive(m_i) \in I_{k,s} \wedge send(m_{i+1}) \in I_{k,t} \wedge s \leq t$, and
(iii) m_q is received by P_j before taking its checkpoint $C_{j,y}$.

A Z-path is *causal* if each receiving event in the sequence happens before the sending event of the next message in the sequence[23]. A Z-path is *noncausal* if it is not causal. A causal Z-path is also called a *causal path* (or C-path for short). A Z-path containing only one message is always causal. In this paper, the first (resp. last) message of a Z-path ζ is represented by $\zeta.first$ (resp. $\zeta.last$).

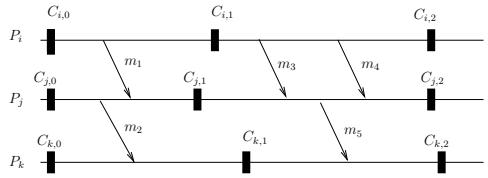


Fig. 1. A checkpoint and communication pattern (CCP)

For example, in the checkpoint and communication pattern shown in Fig.1, the message sequence $[m_1, m_2]$ forms a noncausal Z-paths from $C_{i,0}$ to $C_{k,1}$. The message sequence $[m_3, m_5]$ forms a causal Z-path from $C_{i,1}$ to $C_{k,2}$. However, the message sequence $[m_3, m_2]$ is not a Z-path from $C_{i,1}$ to $C_{k,1}$. To better understand why the "hidden" dependency exists among checkpoints and how it affects the usefulness of a checkpoint, the following definitions are presented in [16]:

Definition 4 *A Z-pattern consists of two consecutive messages m_α and $m_{\alpha+1}$ in a Z-path $[m_1, m_2, \ldots, m_q]$ such that $send(m_{\alpha+1}) \xrightarrow{hb} receive(m_\alpha)$.*

Two local checkpoints can have Z-dependency relation which is described as follows:

Definition 5 *A checkpoint $C_{j,y}$ Z-depends on a checkpoint $C_{i,x}$ (denoted $C_{i,x} \xrightarrow{Z} C_{j,y}$) if:*
(i) $(j = i) \wedge (y > x)$, or
(ii) a Z-path exists from $C_{i,x}$ to $C_{j,y}$.

Since $\xrightarrow{Z}$ is only a partial order defined on $\mathscr{C}_{\hat{H}}$, it is possible that a checkpoint Z-depends on itself and becomes useless because it cannot belong to any consistent global checkpoint.

Definition 6 *A Z-cycle is formed when a checkpoint $C_{i,x}$ Z-depends on itself: $C_{i,x} \xrightarrow{Z} C_{i,x}$[16].*

The main goal of CIC protocols is to allow processes to take checkpoints whenever they want and also ensure each checkpoint is useful. To achieve this goal, processes may be forced to take additional checkpoints to avoid Z-cycles. Based on the following two corollaries[16], no useless checkpoints in the system is equivalent to *Z-Cycle Free* property.

Corollary 1 *A global checkpoint is consistent iff no Z-dependency exists among the local checkpoints in the global checkpoint.*

Corollary 2 *A local checkpoint $C_{i,x}$ cannot belong to any consistent global checkpoint (we call $C_{i,x}$ useless checkpoint) iff $C_{i,x} \xrightarrow{Z} C_{i,x}$.*

Hence, if none of the checkpoints is useless, we call such a system *Z-Cycle Free*. Theorem 1 provides a necessary condition to guarantee *Z-Cycle Free* property based on a timestamp function.

Theorem 1 *Let each checkpoint C be associated with a timestamp $C.ts$. If for every pair of checkpoints $C_{i,x}$ and $C_{j,y}$ with $C_{j,y}$ Z-depending on $C_{i,x}$ ($C_{i,x} \xrightarrow{Z} C_{j,y}$), we have $C_{i,x}.ts < C_{j,y}.ts$, then there is no Z-cycle[17,21].*

Next, we present our classification of CIC protocols.

3. Classification of CIC Protocols

First, we bring to light the intrinsic relation between *Model*-based and *Index*-based CIC protocols, and try to make a finer classification of CIC protocols.

3.1. *Existing Classification of CIC Protocols*

CIC protocols are classified into two categories: *Model*-based and *Index*-based CIC protocols[10]. However, checkpointing algorithms under both these categories try to achieve the same goal of ensuring no useless checkpoints in the system by making processes take forced checkpoints, in

addition to independently taken basic checkpoints. Based on Corollary 2, a local checkpoint is useless iff it is involved in a Z-cycle. Consequently, both types of CIC protocols basically try to prevent Z-cycles from being formed in order to ensure that each checkpoint is useful. With the same motivation, some features which make this classification of CIC protocols ambiguous are:

- Both types of protocols associate a sequence number with each local checkpoint. The numbers can be treated as *Indices* which represent timestamps or checkpoint-interval numbers, and need to be piggybacked with application messages.
- While applying the idea of associating a timestamp of each local checkpoint to each checkpoint interval, ensuring ZCF property is equivalent to satisfying *Virtual Precedence* (VP) property[17], it shows that timestamping is not the prerogative of *Index*-based CIC protocols. Actually *Model*-based CIC protocols which prevent Z-cycles need to track the transitive dependency among checkpoint intervals, so that the VP property allows timestamping to be applied to *Model*-based CIC protocols as well.
- *Model*-based CIC protocols track some communication patterns which may lead to the formation of Z-cycles in the future. On the other hand, *Index*-based CIC protocols associate local checkpoints with timestamps in such a way that any two checkpoints along a non-causal Z-path (also called Z-pattern[16]) are put in order. In other words, *Index*-based CIC protocols also try to track some communication patterns (Z-pattern) indirectly and ensure that two checkpoints involved in such patterns are indexed in ascending order.

3.2. *Our Classification of CIC Protocols*

The classification of CIC protocols as *Model*-based and *Index*-based CIC protocols appears to be vague since it is losing the common intuitive meaning of *MODEL* and *INDEX* as the CIC protocols have been more and more studied in the literature. In this paper, we provide a classification of CIC protocols based on how they achieve the same goal of ZCF property from the point of view of the data structures used in these protocols.

Two types of *Index Numbers* are used among CIC protocols: *Checkpoint Interval (CI)* and *TimeStamp (TS)*. First, let us distinguish the purpose of these two types of indexing strategies from the point of view of their management and usefulness. They both can be scalar integers or n-size vectors (note that n denotes the number of processes involved in the computation unless otherwise specified). Among CIC protocols, *CI* is often used as a vector to track the dependency information among checkpoints of processes, whereas *TS* is the Lamport's scalar clock[20] or the Fidge-Mattern's vector time[11,22] and it is used for tracking causal dependency among events in processes. Later in this section, we discuss how these two Index Numbers in two formats, integer or vector, can be used in detail. While being used as vectors, their corresponding vectors are called Checkpoint Interval vector (CI-vector) and TimeStamp vector (TS-vector) respectively. The former records the information of checkpoint interval sequence numbers and is used to track the transitive dependency among checkpoint intervals of processes. The latter is used to timestamp the local checkpoints in each process. $CI_i[i]$ is the checkpoint interval number associated with the checkpoint taken by P_i and $CI_i[k]$ is the latest checkpoint interval of P_k to P_i's knowledge. $TS_i[i]$ is the timestamp of P_i and $TS_i[k]$ is the highest timestamp of P_k known to P_i. The management of these two vectors follows different indexing strategies, which are called the usual way[11] for CI-vector and the classical way[20] for TS-vector. The corresponding indexing strategies are illustrated in Table 1.

As a result, these two Index Numbers *CI* and *TS* have different meanings and play different roles in enforcing ZCF property. We provide a classification of CIC protocols solely depending on which *Index Number* is used to guarantee the usefulness of all checkpoints in the system. Hence we classify the CIC protocols into two categories, namely, *CI*-CIC and *TS*-CIC as follows:

(i) **CheckpointInterval-based CIC Protocols (*CI*-CIC):** Protocols in this family[2,3,6,13–15,24,31,33] try to track the direct dependency or transitive dependency among checkpoints (or checkpoint intervals) of processes. Based on its meaning and management in Table 1, the data structure *CI* in each process takes the successive integer values. So *CI* is such an index number that can be used to directly deduce which checkpoint interval an event belongs to or depends on from its *CI* value. Based on its format, *CI* index number has different dependency-tracking abilities:

- *Direct Dependency Tracking: CI* takes an integer value. To track direct dependencies, each application message is required to piggyback one integer as *CI* value. A central observer process is needed to collect the direct dependency information of local checkpoints in each process so that it can generate a consistent global checkpoint that includes a given set of local checkpoints[3].
- *K-dependency Tracking: CI* takes the format of a k-size vector ($1 \leq k \leq n$). k-size vector is called k-dependency vector, which piggybacks a constant number k of integers on each application message[5]. There is a tradeoff between the size of *CI* vector and the dependency-tracking ability. If $k < n$, only a subset of the causal dependencies can be tracked on-the-fly. So a good strategy for selecting k is needed to fit the specified application requirements.
- *Transitive Dependency Tracking: CI* is an n-size

Table 1
Comparison between the two indexing strategies and their maintenance. (where $\forall j, j \neq i$)

Management of CI-vector and TS-vector of each process P_i		
	Usual way to maintain vector CI_i[11]	Classical way to maintain vector TS_i[20]
Initialization	$CI_i[i] = 1,\ CI_i[j] = 0$	$TS_i[i] = 1,\ TS_i[j] = 0$
Taking a checkpoint	$CI_i[i] = CI_i[i] + 1$	$TS_i[i] = TS_i[i] + 1$
Sending a message m	P_i assigns m with its current CI_i	P_i assigns m with its current TS_i
Upon receiving a message m from P_k	$CI_i[i] = \max(CI_i[i], m.CI[i])$, $CI_i[j] = \max(CI_i[j], m.CI[j])$	$TS_i[i] = \max(TS_i[i], m.TS[k])$, $TS_i[j] = \max(TS_i[j], m.TS[j])$

vector. Baldoni et al.[4] have proved that it is impossible to ensure the *Rollback-Dependency Trackability* property (RDT, introduced by Wang[31]) with scalar clock. It is also known that the only way to detect the complete causality relationship among events in n processes is by using a vector with n entries[9]. The transitive dependency information can be used not only to detect some communication patterns which may lead to the formation of Z-cycles in the future[6,13,24,31], but also to ensure a stronger property, namely RDT property[2,14,15,25].

(ii) **TimeStamp-based CIC Protocols (*TS*-CIC):** Protocols in this family[1,7,8,18,19,21,30] assign timestamps to local checkpoints such that Theorem 1 is satisfied and hence ensure the ZCF property. The timestamps of successive checkpoints in each process, working as index numbers, can skip some integer values according to the classical way of management in Table 1. So timestamp is such an index number that can be used to track the causal relationship among checkpoints or events in distributed systems, but it cannot be used to directly deduce, to which checkpoint interval an event belongs. The checkpoint-inducing conditions of protocols in this family can be abstractly expressed as $(m.ts > ts_i) \wedge \mathscr{P}$, where $\mathscr{P}$ is a predicate that depends on each protocol[16].

3.3. *Intrinsic Relation between* Model-*based and* Index-*based CIC protocols*

In this section, we show that *Model*-based and *Index*-based protocols are intuitively derived from the same source. We do this by clarifying the assertion "For *Model*-based CIC protocols there always exists an equivalent time-stamping function that would cause the same forced checkpoints", made in [17].

First we show how to use the vector timestamp to track the transitive dependency among checkpoints (or checkpoint intervals). As illustrated in Fig.2, a causal path $[m_1, m_2]$ makes $C_{k,z}$ in process P_k depend on $C_{i,x}$ in process P_i. Assume $TS_i[i]$ of $C_{i,x}$ is t_1 and the timestamp of its successive checkpoint $C_{i,x+1}$ is t_2 ($t_2 > t_1$). Upon receiving m_2, $TS_k[i]$ of P_k will be updated according to the classical way in Table 1 to maintain the vector of timestamps. Then we get $t_1 \leq TS_k[i] < t_2$. The information about P_i known to P_k is from $TS_k[i]$ and the only way we can track the dependency relation is from the piggybacked vector of timestamps $m_1.TS$ and $m_2.TS$. So the only way we can obtain the dependency information (i.e., P_k depends on which checkpoint or checkpoint interval of P_i) is by means of knowing $TS_k[i]$ and the timestamp of latest checkpoint of process P_i. Since the timestamps of consecutive checkpoints in each process can have gaps, we can not get the checkpoint interval numbers directly from the timestamps. However, we can still obtain the dependency relation by choosing the checkpoint in P_i, say $C_{i,w}$, with the maximum timestamp $TS_i[i]$ among all checkpoints in P_i whose timestamps are less than or equal to $TS_k[i]$. And $C_{k,z}$ transitively depends on $C_{i,w}$ (e.g., $w = x$ in Fig.2).

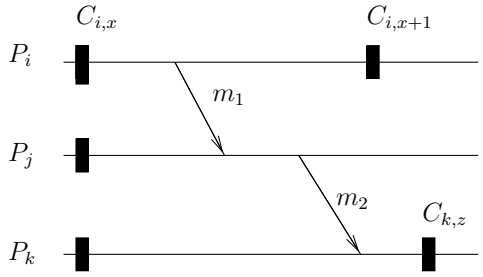


Fig. 2. The causal path $[m_1, m_2]$ brings dependency information

Second, we develop a mechanism by which we can not only timestamp each event but also get the transitive dependency information upon receiving a message (note that the timestamp of latest checkpoint in each process must be known by the receiver upon receiving a message if no centralized mechanism is assumed). In order to satisfy these two requirements, we propose a mechanism called TDE-timestamp (Transitive Dependency Enabled) mechanism, which contains 1) a timestamp function, $TDE(A)$ and 2) a TDE data structure, an n-size vector $TDE\_TS[i]$, $(0 \leq i \leq n-1)$ and its management. Each event or checkpoint, say A, is associated with a timestamp and its value is determined by the function $TDE(A)$. For example, if A is a checkpoint, say $C_{j,x}$, then $TDE(C_{j,x})$ is the TDE-timestamp of the checkpoint $C_{j,x}$, denoted by $C_{j,x}.TDE$ for simplicity; A can also be an event, e.g., send(m_1), then $TDE(send(m_1))$ is the TDE-timestamp of the sending event for m_1, denoted by $m_1.TDE$. The TDE-timestamp is made up of two com-

ponents: $\boldsymbol{TS}$ (timestamp of the last checkpoint) and $\boldsymbol{\Delta TS}$ (the incremental timestamp since the last checkpoint). Its value is calculated as follows:

$$TDE(A) = TS(A) + \Delta TS(A) \quad (1)$$

Since this function is used to calculate the TDE-timestamp of each process known to one process, TDE must be an n-size vector. Here comes the question: "Can we just piggyback the vector of TDE-timestamp on the application messages to track the transitive dependency?" The answer is no, because we also need to know the timestamp of latest checkpoint in each process from the above discussion. So we design the data structure of TDE-timestamp mechanism to be the following one:

$$TDE\_TS_i[k] = CB \times TS_i[k] + \Delta TS_i[k]\ ^1 \quad (2)$$

Where CB is an arbitrary large constant to combine two factors $TS_i[k]$ and $\Delta TS_i[k]$, on the other hand it also helps in easily separating the two factors when we need. It is preferable to take CB to be 2^t and t is large enough so that $TS_i[k] + \Delta TS_i[k] < 2^{t-2}$. Each entry of TDE_TS_i is not the TDE-timestamp but the combination of two factors of TDE-timestamp as in Equation (2). In our simulation runs, we set $CB = 2^{10}$. Because in binary computation, multiplying (or dividing) by 2 to the power t is the same as shifting t places left (or right), we can easily combine (or separate) the two factors ($TS_i[k]$ and $\Delta TS_i[k]$) into (or from) one for each entry of TDE_TS_i according to Equation (2) each time when P_i sends (or receives) a message.

Third, we discuss the management of the vector TDE_TS_i to maintain its meaning (note that we only consider the management of two factors respectively and the corresponding entry of TDE_TS_i can be easily obtained from Equation (2)). Therefore, we get the following TDE-indexing strategy, which is different from that in Table 1:

- Each entry of TS_i is initialized to be zero, and $TS_i[i]$ to be 1. $\forall j,\ \Delta TS_i[j] := 0$.
- When P_i sends a message m to P_k, it computes TDE_TS_i according to Equation (2), and appends the current value of TDE_TS_i to m.
- When P_i takes a checkpoint, $TS_i[i] := TS_i[i] + \Delta TS_i[i] + 1$, $\Delta TS_i[i] := 0$.
- When P_i receives a message m from P_j, P_i first separates two factors TS_i and ΔTS_i from each entry of TDE_TS_i, then updates them as follows (note that for each entry of $m.TDE\_TS_i$, $m.TS_i$ and $m.\Delta TS_i$ can be extracted easily):

  · $\forall k$, do case
  $\boldsymbol{m.TS[k] < TS_i[k]} \rightarrow skip$
  $\boldsymbol{m.TS[k] > TS_i[k]} \rightarrow TS_i[k] := m.TS[k],\ \Delta TS_i[k] := m.\Delta TS[k]$
  $\boldsymbol{m.TS[k] = TS_i[k]} \rightarrow \Delta TS_i[k] := \max(\Delta TS_i[k],\ m.\Delta TS[k])$
  end docase
  · for i, do case
  $\boldsymbol{m.TS[j] + m.\Delta TS[j] \leq TS_i[i] + \Delta TS_i[i]} \rightarrow skip$
  $\boldsymbol{m.TS[j] + m.\Delta TS[j] > TS_i[i] + \Delta TS_i[i]} \rightarrow \Delta TS_i[i] := m.TS[j] + m.\Delta TS[j] - TS_i[i]$
  end docase

The advantage of our designed data structure TDE_TS is that a process can get extra information about other processes upon receiving a message without increasing message overhead. Upon receiving a message m, a process can get the following information from $m.TDE\_TS$:

- The timestamp of last checkpoint in each process known to the sender, say P_j, while sending m: it can be obtained from each entry of $m.TS$ and it is necessary for the receiver to track the transitive dependency.
- The TDE-timestamp value in Equation (1): it works as the Lamport's logical clock in distributed systems.

So the TDE_TS mechanism helps to timestamp each event and also get the transitive dependency information upon receiving a message. Moreover, the first two contributions of this paper are based on the analysis in Section 3.3. First, the intrinsic relation between *Model*-based and *Index*-based CIC protocols is the primitive motivation to get our finer classification of CIC protocols in Section 3.2. Second, the TDE_TS mechanism can be applied to some existing CIC protocols to decrease the overhead of piggybacked information without degrading the checkpoint-inducing decision. In the next section, we apply our TDE_TS mechanism to Fully Informed CIC protocol[16] and develop an enhanced CIC protocol with better performance.

4. Fully Informed aNd Efficient (FINE) CIC Protocol

The CIC protocol introduced by Helary et al.[16] is considered to be one of the best CIC protocols in the literature since it exploits all possible information available from the causal past to prevent useless checkpoints while at the same time minimizing forced checkpoints. So it has been called *Fully Informed (FI)* protocol in [27]. However, we observe that the FI protocol[16] does not fully exploit the collected information from the causal past to make a precise checkpointing decision. Hence in this section, first we briefly introduce FI protocol to make our paper self-contained. Then we propose our *Basic FINE* protocol whose checkpoint-inducing condition is stronger than that of FI protocol without increasing the overhead of piggybacked information, followed by the proof of its correctness. Finally, we apply our TDE_TS mechanism to Basic FINE protocol and obtain our *Advanced FINE* protocol. And the overhead comparison demonstrates that *Advanced FINE* not only has better performance than *FI* protocol but also decreases its message overhead.

[1] We assume that $\Delta TS_i[k]$ is much smaller than CB (as in our simulation, $CB = 2^{10}$) and this assumption is reasonable and safe, because $\Delta TS_i[k]$ is a small integer denoting the incremental timestamp since the last checkpoint and it will be reset to 0 after taking a checkpoint.

4.1. *The Fully Informed (FI) Protocol*

This section introduces the design principles and checkpoint-inducing condition of FI protocol[16]. To better understand its checkpoint-inducing condition, the meaning and maintenance of data structures used in FI protocol[16] to are briefly explained below:

- ***Local logical clock (timestamp).*** The timestamps of a checkpoint $C_{i,x}$ and a message m are denoted by $C_{i,x}.t$ and $m.t$ respectively. Each process P_i manages a vector of timestamps cl_i in the classical way shown in Table 1. $cl_i[i]$ is the current timestamp of P_i and $cl_i[k]$ denotes the timestamp of P_k known to P_i.
- ***Boolean Array sent_to$_i$.*** Each process P_i manages a boolean array $sent\_to_i[1..n]$. $sent\_to_i[k]$ has the value *true* iff P_i has sent a message to P_k since its last checkpoint.
- ***Array min_to$_i$.*** Each process P_i manages an array of integers $min\_to_i[1..n]$. $min\_to_i[k]$ keeps the timestamp of the first message sent by P_i to P_k since P_i took its last checkpoint.
- ***Array ckpt$_i$.*** This array is a vector that counts how many checkpoints have been taken by each process. $ckpt_i[k]$ is the number of checkpoints taken by P_k to P_i's knowledge. This vector is managed in the usual way shown in Table 1. In particular, $m.ckpt$ denotes the value of this vector appended to m by its sender.
- ***Boolean Array taken$_i$.*** It is used to track a causal path with a checkpoint inside. The entry $taken_i[k]$ is *true* iff there is a causal Z-path from the last checkpoint of P_k known to P_i to the next checkpoint of P_i, and this C-path includes a checkpoint. The array $taken_i$ of P_i is managed in the following way to maintain its meaning[16]:
  - When P_i takes a checkpoint, for $\forall k \neq i$, $taken_i[k]$ is set to *true* and $taken_i[i]$ always remains $false$.
  - When P_i sends a message m to P_k, it merely appends to m the current value of $taken_i$, denoted by $m.taken$.
  - When P_i receives a message m from P_j, it updates $taken_i$ as follows:

$\forall k \neq i$, **do case**
$\boldsymbol{m.ckpt[k] < ckpt_i[k]} \rightarrow skip$
$\boldsymbol{m.ckpt[k] > ckpt_i[k]} \rightarrow taken_i[k] := m.taken[k]$
$\boldsymbol{m.ckpt[k] = ckpt_i[k]} \rightarrow taken_i[k] := taken_i[k] \vee m.taken[k]$
end docase

In FI protocol[16], the forced checkpoints are triggered in such a way that Theorem 1 is satisfied to ensure the ZCF property.

In FI protocol[16], the checkpoint-inducing condition $\mathscr{C}2$ evaluated by a process P_i upon receiving a message is as follows:

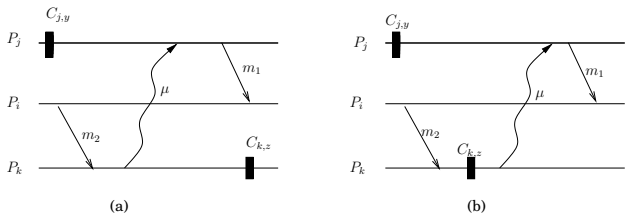


Fig. 3. Two possible cases to consider if the timestamp restriction is violated

$$\mathscr{C}2 \equiv (\exists k : sent\_to_i[k] \wedge (m.t > min\_to_i[k]) \wedge (m.t > cl_i(k) \vee \mathscr{C}')) \quad \text{(from [16])}$$

$$\mathscr{C}2 \equiv \mathscr{C}2\_1 \vee \mathscr{C}2\_2 \quad \text{(from the distributive law)}$$

where

$$\mathscr{C}2\_1 \equiv (\exists k : sent\_to_i[k] \wedge (m.t > min\_to_i[k]) \wedge (m.t > cl_i(k)))$$

$$\mathscr{C}2\_2 \equiv (\exists k : sent\_to_i[k] \wedge (m.t > min\_to_i[k]) \wedge \mathscr{C}')$$

$$\mathscr{C}' \equiv (m.ckpt[i] = ckpt_i[i]) \wedge m.taken[i] \quad \text{(from [16])}$$

$$\mathscr{C}2\_2 \equiv (m.ckpt[i] = ckpt_i[i]) \wedge m.taken[i] \quad \text{(from theorem7.2 in [16])}$$

Conditions $\mathscr{C}2\_1$ and $\mathscr{C}2\_2$ examine any violation under two cases: the value $cl_i(k)$ has been brought to P_i by a causal Z-path that started from P_k 1) before $C_{k,z}$ as in Fig.3(a), denoted as **case $\mathfrak{A}$** or 2) after $C_{k,z}$ as in Fig.3(b), denoted as **case $\mathfrak{B}$**. The meaning and maintenance of each data structure in above conditions are given in detail in FI protocol[16]. We make the following Observation 1 from the strategy to maintain the meaning of $taken_i[k]$:

Observation 1 *Upon receiving a message m, if $m.ckpt[k] \geq ckpt_i[k]$, P_i will set $taken_i[k]$ to false only if $m.taken[k]$ is false.*

4.2. *Basic FINE protocol*

We observe that upon receiving a message, although the receiver has been fully informed about the information from the causal past, the checkpoint-inducing condition of FI protocol does not fully exploit all the information. Let us consider what happens in FI protocol under the situation shown in Fig.4(a) and (b): the condition $\mathscr{C}2\_1$ of FI protocol is satisfied under both cases, so before delivering m_1, P_i must take a forced checkpoint, however it is not necessary to break any Z-cycle. Based on Definition 6 and Corollary 2, a cycle which doesn't contain any checkpoint is not a menace to the usefulness of any checkpoint. Hence it is desirable to identify these harmless cycles and reduce some unnecessary forced checkpoints without incurring any extra overhead of piggybacked information.

In this section, we present our Basic FINE protocol which recognizes those harmless cycles under each case in Fig.4 and makes precise checkpointing decision. The most attractive feature of Basic FINE protocol is that it uses exactly the same data structures as FI protocol without incurring any extra overhead of piggybacked information, however it

does obtain a stronger checkpoint-inducing condition than that of FI protocol while ensuring the ZCF property.

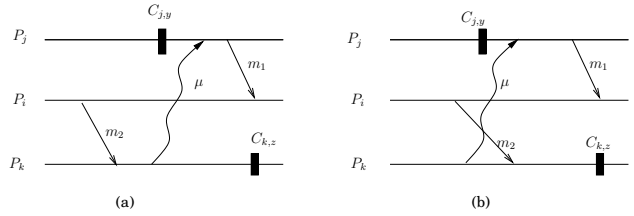


Fig. 4. No forced checkpoint should be taken upon receiving m_1

Our updated checkpoint-inducing condition $\mathscr{C}_{FINE}$ which is stronger than condition $\mathscr{C}2$ in FI protocol[16] is defined as follows:

$$\begin{aligned}
&\mathscr{C}_{FINE} \equiv \mathscr{C}_{FINE}_1 \vee \mathscr{C}_{FINE}_2 \\
&\quad \text{where} \\
&\mathscr{C}_{FINE}_1 \equiv (\exists k : sent_to_i[k] \wedge (m.t > min_to_i[k]) \\
&\qquad \wedge (m.t > cl_i(k)) \wedge m.taken[k]) \\
&\mathscr{C}_{FINE}_2 \equiv \mathscr{C}2_2
\end{aligned}$$

Upon receiving a message m, P_i is forced to take a checkpoint only if $\mathscr{C}_{FINE}$ is satisfied. Theorem 2 proves that our Basic FINE protocol guarantees that all checkpoints are useful.

Theorem 2 *The condition $\mathscr{C}_{FINE}$ of Basic FINE protocol is a Z-cycle free checkpoint-inducing condition.*

Proof: First, FI protocol[16] is a Z-cycle free checkpointing protocol and its design is based on Theorem 1. While considering if each Z-pattern [m_1,m_2] is consistent with the assumption of Theorem 1, we need to check if the following property holds:

$$\begin{aligned}
&(m_1.t \leqslant m_2.t) \vee \mathscr{P}, \\
&\text{where } \mathscr{P} \equiv (C_{j,y}.t \leqslant m_1.t \leqslant cl_i(k) < C_{k,z}.t)
\end{aligned}$$

Second, FI protocol guarantees ZCF by forcing the process to take a checkpoint whenever the above property is violated. In order to track such property violation, two possible cases illustrated in Fig.3(a) and (b) arise[16].

- **Case $\mathfrak{A}$.** The value $cl_i(k)$ has been brought to P_i by a causal Z-path that started from P_k *before* $C_{k,z}$. Any property violation under this case is captured by $\mathscr{C}2\_1$.
- **Case $\mathfrak{B}$.** The value $cl_i(k)$ has been brought to P_i by a causal Z-path that started from P_k *after* $C_{k,z}$. Any property violation under this case is captured by $\mathscr{C}2\_2$.

Since $\mathscr{C}_{FINE}\_2 \equiv \mathscr{C}2\_2$, we only need to prove that $\mathscr{C}_{FINE}\_1$ can track all the property violation under **case $\mathfrak{A}$**. The difference between $\mathscr{C}2\_1$ and $\mathscr{C}_{FINE}\_1$ is that the latter also needs to check whether $m.taken[k]$ is true or false upon receiving m. If it is true, then under **case $\mathfrak{A}$**, a forced checkpoint is taken only when $\mathscr{C}2\_1$ is also satisfied; otherwise no forced checkpoint. Under **case $\mathfrak{A}$**, two subcases can arise: $cl_i(k)$ is known to P_i through $[\mu, m_1]$ 1) in Fig.5(a) and/or 2) in Fig.5(b).

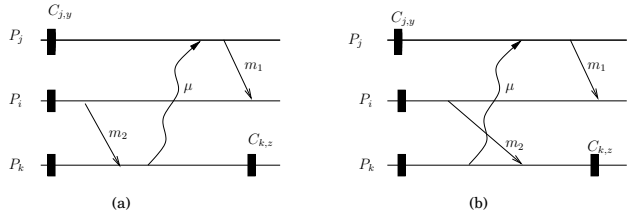


Fig. 5. Two possible cases to consider when $cl_i(k)$ is a lower bound of $C_{k,z}.t$

Next, we prove under **case $\mathfrak{A}$**, no Z-cycle can be formed along Z-path $[m_1, m_2, \mu, m_1]$ if $m_1.taken[k]$ is false. In other words, either there is no cycle along Z-path $[m_1, m_2, \mu, m_1]$ or any possible cycle formed along Z-path $[m_1, m_2, \mu, m_1]$ doesn't contain any checkpoint if $m_1.taken[k]$ is false. The following four cases arise under this condition.

- **Case I. Z-path $[m_1, m_2]$ doesn't contain any checkpoint.**
  Because Z-path $[m_1, m_2]$ forms a Z-pattern[1] and no checkpoint inside this Z-path trivially holds.
- **Case II. Z-path $[m_2, \mu.first]$ doesn't contain any checkpoint.**
  No checkpoint can exist between m_2 and μ since 1) under the case in Fig.5(a), no checkpoint can exist after $receive(m_2)$ and before $send(\mu.first)$ (Recall that $C_{k,z}$ is the first checkpoint taken by P_k after the delivery of m_2), otherwise it falls into **case $\mathfrak{B}$**. 2) under the case in Fig.5(b), if a checkpoint exists after $send(\mu.first)$ and before $receive(m_2)$, $[m_2, \mu]$ is no longer a zigzag path.
- **Case III. Z-path $[\mu.last, m_1]$ doesn't contain any checkpoint.**
  If $m_1.taken[k]$ is false, it means $taken_j[k]$ is false when P_j sends m_1 (Note that P_j is the sender, so $m_1.taken[k] = taken_j[k]$). Hence no checkpoint can exist between receive($\mu.last$) and send(m_1), otherwise $m_1.taken[k]$ must be true.
- **Case IV. Causal path μ doesn't contain any checkpoint.**
  Let us denote $\mu.last$ as m'. We first prove that upon receiving m', we can only have $m'.ckpt[k] \geqslant ckpt_j[k]$, otherwise if $m'.ckpt[k] < ckpt_j[k]$ it will fall into **case $\mathfrak{B}$**. We prove it by contradiction. Assume $m'.ckpt[k] < ckpt_j[k]$ upon receiving m', and there must exist a causal path ν from P_k to P_j which brings the dependency information about $I_{k,x}$ of P_k to P_j where $x \geqslant z$ and $\nu.last \xrightarrow{hb} \mu.last$. It is illustrated in Fig.6. Then the causal path $[\nu, m_1]$ forms a causal path from P_k to P_i after $C_{k,z}$, which falls into **case $\mathfrak{B}$**. So upon receiving m', P_j will update $taken_j[k]$ according to the condition $m'.ckpt[k] \geqslant ckpt_j[k]$. If $taken_j[k]$ is false after being up-

dated, $m'.taken[k]$ must be false according to Observation 1. Since μ is a causal path from P_k to P_j and $m' = \mu.last$ and $m'.taken[k]$ is false, no checkpoint can be inside μ in order to maintain $m'.taken[k]$ being false.

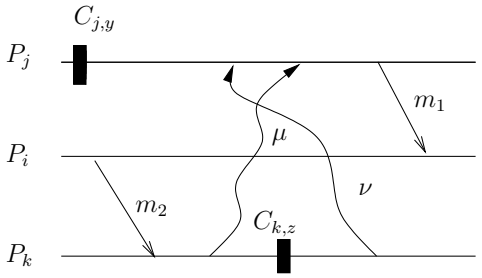


Fig. 6. A causal path ν from P_k to P_j after $C_{k,z}$ makes CCP fall into case $\mathfrak{B}$

After verifying all the possible situations, we conclude: under **case $\mathfrak{A}$**, no Z-cycle can be formed along Z-path $[m_1, m_2, \mu, m_1]$ if $m_1.taken[k]$ is false. In other words, under **case $\mathfrak{A}$**, there is potential for Z-cycles to be formed only if $m_1.taken[k]$ is true. So if $\mathscr{C}2\_1$ can guarantee ZCF property under **case $\mathfrak{A}$**, the stronger condition $\mathscr{C}_{FINE}\_1 \equiv \mathscr{C}2\_1 \wedge m_1.taken[k]$ can also guarantee ZCF property under **case $\mathfrak{A}$**. Moreover since $\mathscr{C}_{FINE}\_2 \equiv \mathscr{C}2\_2$ under **case $\mathfrak{B}$**, $\mathscr{C}_{FINE}\_2$ can guarantee ZCF property under **case $\mathfrak{B}$**. Therefore $\mathscr{C}_{FINE}$ can capture all the property violation under both **case $\mathfrak{A}$** and **$\mathfrak{B}$**, hence the checkpoint-inducing condition $\mathscr{C}_{FINE}$ can guarantee ZCF property.

$\square_{Theorem\ 2}$

4.3. *Advanced FINE protocol*

In order to get a precise checkpoint-inducing condition, each process needs to collect enough information about others which can only come from information piggybacked with messages. Therefore, in an effort towards obtaining efficient CIC protocols, it is desirable to decrease message overhead without weakening the checkpoint-inducing decision. In this section, we introduce our Advanced FINE protocol using the TDE_TS mechanism introduced in Section 3.3. TDE_TS mechanism provides a good balance between message overhead and computation overhead in each process. When receiving (resp. sending) a message, the receiver (resp. sender), say P_i, needs to do a small amount of computation (e.g., several shift operations in binary computation) and gets (resp. transfers) both the transitive dependency information and timestamps from the piggybacked n-size vector TDE_TS_i.

We now describe the Advanced FINE protocol briefly. The line numbers refer to the detailed protocol description in Pseudocode 1.

As noted earlier, upon receiving a message, each process can easily separate the values of TS and ΔTS from each

```
1: procedure take_checkpoint at P_i
2: for k = 0, ..., n − 1 do
3:    sent_to_i[k] := false
4: end for
5: for ∀k ≠ i do
6:    taken_i[k] := true
7: end for
8: TS_i[i] := TS_i[i] + ΔTS_i[i] + 1; ΔTS_i[i] := 0
9: Save state to stable storage

10: initialization of P_i
11: for k = 0, ..., n − 1 do
12:    TS_i[k] := 0; ΔTS_i[k] := 0
13: end for
14: taken_i[i] := false
15: take_checkpoint

16: when P_i sends a message m to P_k
17: if ¬sent_to_i[k] then
18:    sent_to_i[k] := true
19: end if
20: for l = 0, ..., n − 1 do
21:    TDE_TS_i[l] := 2^10 × TS_i[l] + ΔTS_i[l]
22: end for
23: send (m, TDE_TS_i, taken_i) to P_k

24: when P_i receives (m, TDE_TS, taken) from P_j
25: % P_i separates two factors TS_j and ΔTS_j from
    TDE_TS_j %
26: if ((∃k : sent_to_i[k] ∧ (m.TS[j] + m.ΔTS[j]) >
    (m.TS[k] + m.ΔTS[k]) ∧ (m.TS[j] + m.ΔTS[j]) >
    (TS_i[i] + ΔTS_i[i]) ∧ m.taken[k]) ∨ (m.TS[i] = TS_i[i] ∧
    m.taken[i])) then
27:    take_checkpoint          % forced checkpoint %
28: end if
29: ∀k, do case
30: m.TS[k] < TS_i[k] → skip
31: m.TS[k] > TS_i[k] → TS_i[k] := m.TS[k];
    ΔTS_i[k] := m.ΔTS[k]; taken_i[k] := m.taken[k]
32: m.TS[k] = TS_i[k] → ΔTS_i[k] :=
    max(ΔTS_i[k], m.ΔTS[k]);
    taken_i[k] := taken_i[k] ∨ m.taken[k]
33: end docase
34: for i, do case
35: m.TS[j] + m.ΔTS[j] ≤ TS_i[i] + ΔTS_i[i] → skip
36: m.TS[j] + m.ΔTS[j] > TS_i[i] + ΔTS_i[i] →
    ΔTS_i[i] := m.TS[j] + m.ΔTS[j] − TS_i[i]
37: end docase
38: deliver message m
```

Pseudocode 1: Advanced FINE protocol

Table 2
Overhead Comparison

	FI-1	FI-2	FINE
Each process maintains	$3n$ integers	$n+1$ integers	n integers
	$2n$ booleans	$3n$ booleans	$2n$ booleans
Each message piggybacks	$2n$ integers	$n+1$ integers	n integers
	n booleans	$2n$ booleans	n booleans

entry of $m.TDE\_TS$ with *div* and *mod* operations (line 25 of Pseudocode 1) according to Equation (2). Followed by a decision whether a forced checkpoint is taken based on the checkpoint-inducing condition in line 26. Then vectors TS_i, ΔTS_i and $taken_i$ are updated (line 29–line 37) to maintain their meaning. When sending a message, the process combines the corresponding entry of TS_i and ΔTS_i into TDE_TS_i (line 21) and appends the current vector TDE_TS_i to the message (line 23). The meaning of $TS_i[k]$ is the timestamp of last checkpoint of P_k, known to P_i, so that it records the information about the last checkpoint (or interval) of each process known to P_i. On the other hand, the value of $(TS_i[k] + \Delta TS_i[k])$ is the latest timestamp of P_k, known to P_i.

In the Advanced FINE protocol, each process P_i maintains three n-dimensional vectors: a vector of TDE-timestamps (TDE_TS_i) and two boolean vectors ($sent\_to_i$ and $taken_i$). Among these three vectors, only TDE_TS_i and $taken_i$ need to be piggybacked on each message sent by P_i. Table 2 compares our Advanced FINE protocol with FI-1 and FI-2 protocols in terms of data structures maintained in each process and piggybacked information (Note: FI-1 is the original protocol described in Fig.7 of [16] and FI-2 is FI protocol with reduced data structures, given in Fig.8 of [16]).

5. Performance Evaluation

In this section we present the performance evaluation of both Basic and Advanced FINE protocols compared with some existing CIC protocols: BCS, FI, VP-accordant, RDTPartner and RDTMinimal. We choose these protocols mainly because (1)BCS[7,8,19] is the first algorithm proposed to ensure ZCF property even before the concept of ZCF was known, and its checkpoint-inducing condition satisfies the abstract expression of checkpoint-inducing conditions for *TS*-CIC protocols ($(m.ts > ts_i) \wedge \mathscr{P}$, shown in Section 3.2); (2)the comparison between our protocol and FI[16] is to illustrate the improvement of our FINE protocol over FI protocol based on simulation results; (3)VP-accordant[6] was chosen to represent the protocol in the class of *CI*-CIC protocols; (4)both RDTPartner[15] and RDTMinimal[14] ensure RDT property and the comparison with them reflects the tradeoff between checkpointing overhead and property of CCP. Our performance evaluation reveals that FINE protocol performs better than the FI protocol both in terms of the number of forced checkpoints and message overhead.

5.1. *Simulation Model*

The behavior of our FINE protocol has been carefully analyzed and compared with other protocols through simulation using the Distributed Checkpointing Simulator, ChkSim[28]. ChkSim has the property of reproducibility which is ensured by a completely deterministic simulation model[29]. Only with this property, the comparison among different CIC algorithms in the simulation is ensured to be correct and persuasive.

Simulation Scenarios We conducted the simulation under five simulation scenarios (described below), namely, SP, SI, AP, AI and AD conditions which represent major Checkpointing and Communication Patterns and mimic closely the bahavior of real distributed applications[29]. The topology of the distributed system under each scenario of our simulation is set to be a complete graph [1] , in which each pair of processes is directly connected by a bidirectional communication channel. Other common features shared by all scenarios include that the channels do not lose, corrupt or change the order of the messages sent. Hence the message latency can be ignored, compared to the checkpoint interval length[29]. The difference among these five scenarios is described as follows: (note under five scenarios, "symmetric" stands for the assumption that the communication events are homogeneously distributed among all processes involved in the distributed computation; and "asymmetric" stands for the assumption that the number of communication events in one of the processes is different from others and among other processes the communication events are homogeneously distributed.)

SP (Symmetric Processes) Under this scenario, we assume that the communication events are homogeneously distributed among all processes involved in the distributed computation. For our simulation we assume that each process has an average of 50 communication events in each basic checkpoint interval. The simulated distributed computation consists of processes ranging from 10 to 100.

SI (Symmetric Intervals) Under this scenario, we assume that the communication events are homogeneously distributed among all processes involved in the distributed computation. For our simulation we assume that the systems is composed of 20 processes. Each process has the number of communication events ranging from 10 to 200 in each basic checkpoint interval.

AP (Asymmetric Processes) Under this scenario, we assume that the number of communication events in one process is different from others and among other processes the communication events are homogeneously distributed. For our simulation we assume that the average number of communication events in each basic checkpoint interval for one particular process is 20, while

[1] Each directed communication channel can be defined individually in a network definition file, which generates flexible topologies on demand.

the average number of communication events for each of other processes is 50. The simulated distributed computation consists of processes ranging from 10 to 100.

AI (Asymmetric Intervals) Under this scenario, we assume that the communication events in one process are different from others and among other processes the communication events are homogeneously distributed. For our simulation we assume that the system is composed of 20 processes. The average number of communication events in each basic checkpoint interval for one particular process is 30 less than the average number for each of other processes. This particular process has the number of communication events ranging from 10 to 200 in each basic checkpoint interval and others have the number of communication events ranging from 40 to 230 in each basic checkpoint interval accordingly.

AD (Asymmetric Difference) Under this scenario, we assume that the communication events in one process are different from others and among other processes the communication events are homogeneously distributed. For our simulation we assume that the system is composed of 20 processes. Except for one particular process, the average number of communication events in each basic checkpoint interval for all other processes is the fixed number 50. Only this particular process changes the number of communication events in its basic checkpoint intervals and the different number of communication events in each basic checkpoint interval with other processes ranges from 2 to 40.

A complete simulation run to compare our chosen algorithms is configured using a run definition file. This file sets up the different parameters to evaluate the performance of algorithms under different simulation environments. In our simulation, we compare the chosen algorithms under the following configurations in the run definition files: *Iterations*, denotes the number of times the simulation needs to be run for each data point and the collected data for each metric computed as the average of each iteration. We set $iteration = 10$ for each run. *Events*, indicates the overall duration of each execution in terms of the number of communication events per process. We set $events = 12000$ in most of our simulation runs, in order to display the stability of the algorithms' behavior in a long enough run. We also run our simulation with $events = 6000$ and 24000 to check if the overall duration has great effect on the behavior of those algorithms.

5.2. *Performance Metrics*

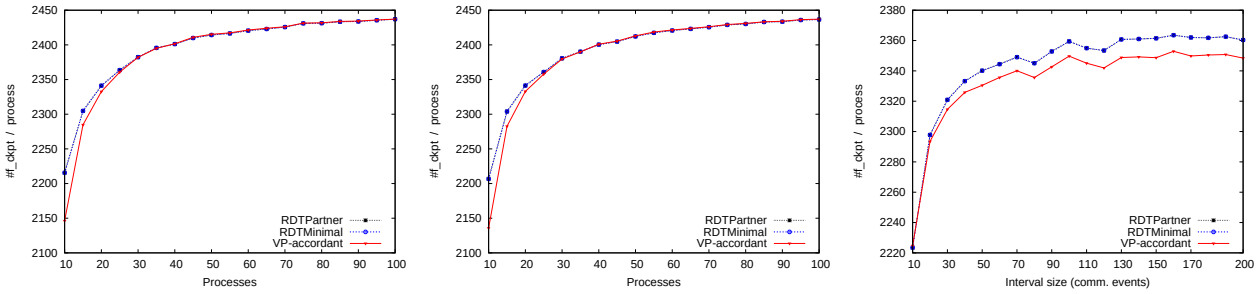
One of the metrics used in the evaluation of CIC protocols is the number of forced checkpoints induced by the protocols, since taking unnecessary checkpoints incurs runtime and resource overhead. Therefore, the checkpoint-inducing condition becomes very important due to its key role in inducing forced checkpoints. In order to get a precise checkpoint-inducing condition, each process needs to collect enough information about others which can only come from the information piggybacked with messages. Hence the number of forced checkpoints and the overhead of piggybacked information are two most important metrics for evaluating the performance of CIC protocols. Since we have compared the overhead of piggybacked information among FI-1, FI-2 and FINE in table 2, we only focus on the number of forced checkpoints in our simulation and we measure the two metrics: 1)the number of forced checkpoints per process ($\#f\_ckpt/process$); 2)the ratio of forced checkpoints with respect to the number of basic checkpoints ($\#f\_ckpt/\#b\_ckpt$).

5.3. *Simulation Results*

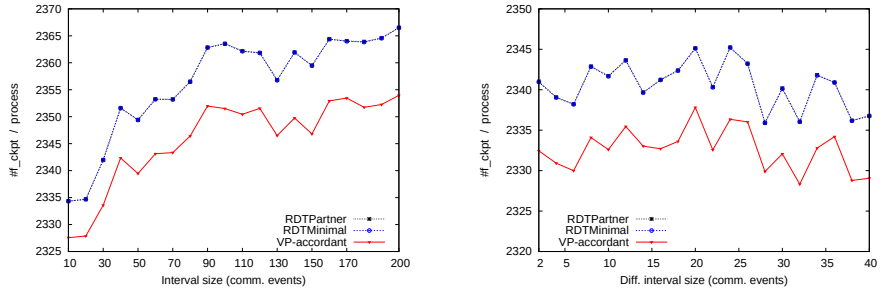
We observe that the six algorithms can be classified into two groups according to their behavior under five simulation scenarios proposed above. We place RDTPartner, RDTMinimal and VP-accordant algorithms in group I due to their similar behavior under all five scenarios and their simulation results are shown in Fig.7; while BCS, FI and AdvancedFINE belong to group II due to their similar behavior under five scenarios. As expected, under each scenario of our simulation, algorithms in group I always take more forced checkpoints than those in group II. Fig.8 shows the result under the AD scenario. This is due to the following two reasons. First, the algorithms ensuring RDT property have to satisfy two requirements: 1) Z-cycle free, and 2) the existence of a causal sibling for every non-causal Z-path. Hence the stronger property than ZCF, namely RDT, obtains its high efficiency in rollback recovery at the expense of taking more forced checkpoints in the system. Second, since VP-accordant algorithm belongs to CI-CIC protocols and it ensures ZCF property by taking forced checkpoints to break all suspect Z-cycles and our simulation results match the general hypothesis that TS-CIC protocols often have less forced checkpoints than CI-CIC protocols. Fig.7 also exhibits that the two protocols, RDTPartner and RDTMinimal, behave similar under all five scenarios. The message overhead and computation complexity of both protocols are $\mathcal{O}(n)$.

Since each data point in all figures is the result of the average over 10 executions for each simulation run, we also calculated the standard deviation of each run. Our results show that for all simulation scenarios the standard deviation (SD) for all the 10 executions are always less than 3.5% of the average. In this paper we only show the ratio of $SD/average$ under SP scenario in Fig.9 as an example, and the maximum value is shown to be less than 2.2% under SP.

The simulation results of algorithms in group II are depicted in Fig.10. Under all five scenarios, our AdvancedFINE algorithm always performs better than FI algorithm by inducing an average of $2\% - 5.5\%$ less forced checkpoints. As expected, the $\#f\_ckpt/process$ increases with increase in the number of processes for all three al-



(a) Sym, var. process # (SP) (b) Asym, var. process # (AP) (c) Sym, var. interval length (SI)



(d) Asym, var. interval length (AI) (e) Asym, var. diff interval length (AD)

Fig. 7. The behavior of the algorithms in group I under five scenarios

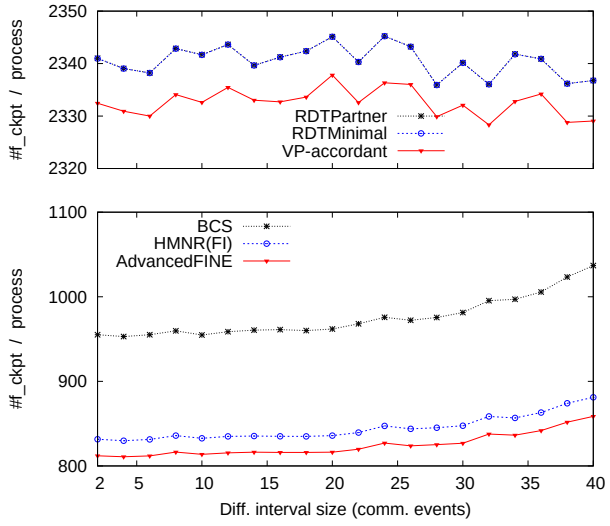


Fig. 8. Six Algorithms

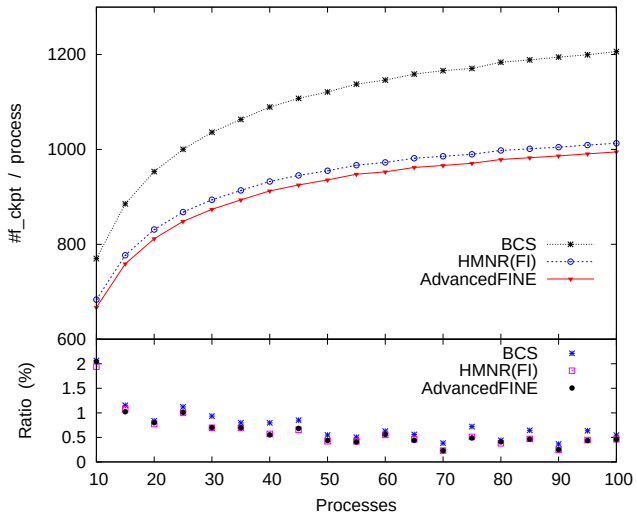
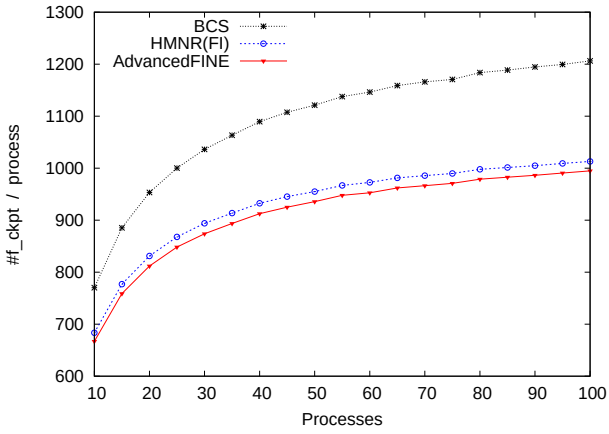
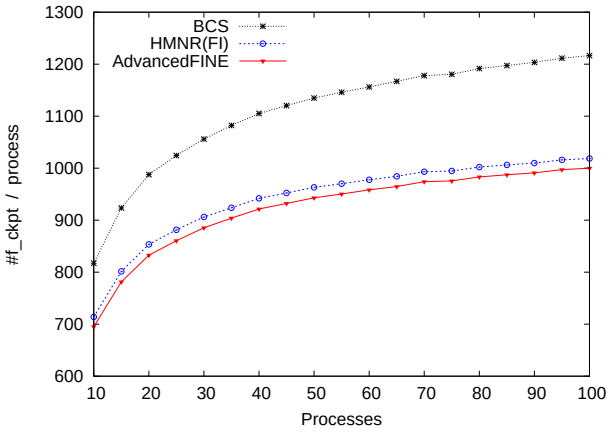


Fig. 9. Standard Deviation

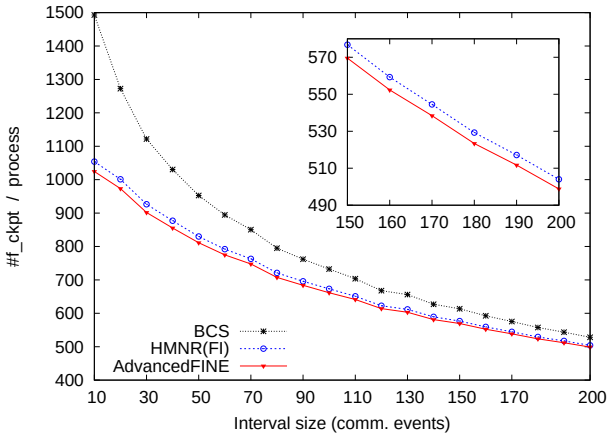
gorithms under SP and AP scenarios (Fig.10(a) and (b)). This is due to the fact that the increase in the number of processes results in the accumulation in dependency information which increases the chances of satisfying the checkpoint-inducing condition, which triggers more forced checkpoints to ensure ZCF property in each of the three algorithms. Fig.10(c) and (d) show that the number of forced checkpoints in each process decreases as the checkpoint interval increases both under SI and AI scenarios. An explanation for this is that longer checkpoint intervals mean lower basic checkpoints, which slows down the frequency at which timestamp in each process increases and this is the major factor in checkpoint-inducing conditions of these three algorithms. However, from the results in our further investigation, the ratio of $\#f\_ckpt$ to $\#b\_ckpt$ under both SI and AI scenarios increases as the checkpoint interval increases (Fig.11(c) and (d)). The asymmetry under AD scenario is obtained in such a way that it is caused by one process taking basic checkpoints faster than other processes and the different number of communication events in each of its basic checkpoint interval with other processes gets bigger (different range from 2 to 40 communication events) with the simulation run. As a result, the number of forced checkpoints increases (Fig.11(e)). It is due to the fact that the time for different processes to "catch up" with each other's timestamp decreases and it increases the frequency of satisfying the checkpoint-inducing condition in each of these three algorithms.



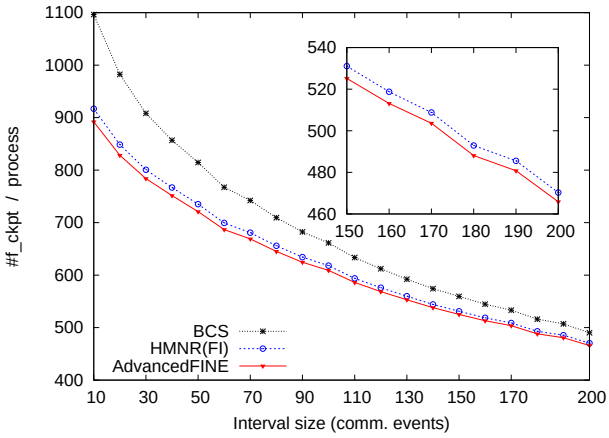
(a) Symmetric, var. process # (SP)



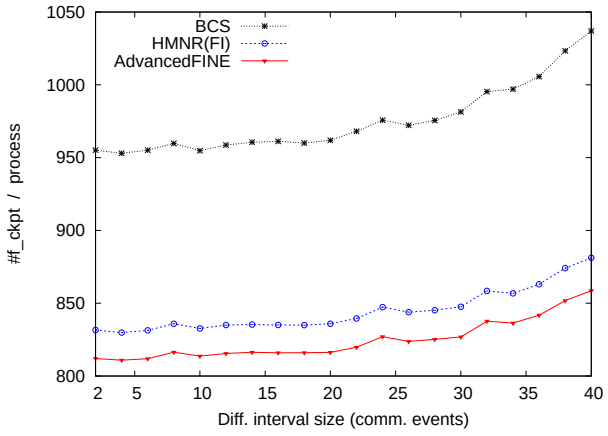
(b) Asymmetric, var. process # (AP)



(c) Symmetric, var. interval length (SI)



(d) Asymmetric, var. interval length (AI)



(e) Asymmetric, var. difference in interval length (AD)

Fig. 10. The behavior of the algorithms in group II under five scenarios

Let us consider the relationship between basic checkpoints and forced checkpoints. Basic checkpoints are taken to provide saved intermediate states of processes for rollback recovery in case of failures. However, allowing processes to take basic checkpoints independently without coordination can not guarantee the basic checkpoints are useful. Hence, forced checkpoints are taken to make sure that the whole system can rollback to a consistent global checkpoint which minimizes the amount of recomputation when failure occurs. Given that ChkSim has the feature of reproducibility and it can deterministically generate the same sequence of events using two statistic load generators with the same seed[29], it provides the deterministic model for evaluating the performance of algorithms with respect to the ratio $\#f\_ckpt/\#b\_ckpt$. Thus, we further investigate the cost of forced checkpoints based on the pattern of basic checkpoints under the five scenarios and show the results in Fig.11. As the number of processes increases, both for symmetric and asymmetric scenarios (Fig.11(a) and (b)), $\#f\_ckpt/\#b\_ckpt$ increases. It is quite reasonable because the more processes involved in the computation, the more will be the propagated dependency among processes. Our simulation reveals that performance of all algorithms gets worse as the size of the system increases. $\#f\_ckpt/\#b\_ckpt$ also increases as the number of communication events in a basic checkpoint interval increases (Fig.11(c),(d) and (e)). The results make it clear that even though it is possible to decrease the number of forced checkpoints by increasing the checkpointing interval (as shown in Fig.10(c) and (d)), we have to consider both absolute and relative measures ($\#f\_ckpt$ and $\#f\_ckpt/\#b\_ckpt$) while calculating the cost of checkpoints in the system. Taking either of these two measure in isolation gives us an incomplete picture. In particular, deciding the relative rankings of these two measurements is quite important. For example, under the case when the basic checkpoint intervals are predetermined by the system, only the absolute measurement is needed to evaluate the performance of a CIC protocol; on the other hand, while considering how to set the checkpointing interval, both measurements should be taken into consideration and it is one of the topics in our future work.

6. Related Work

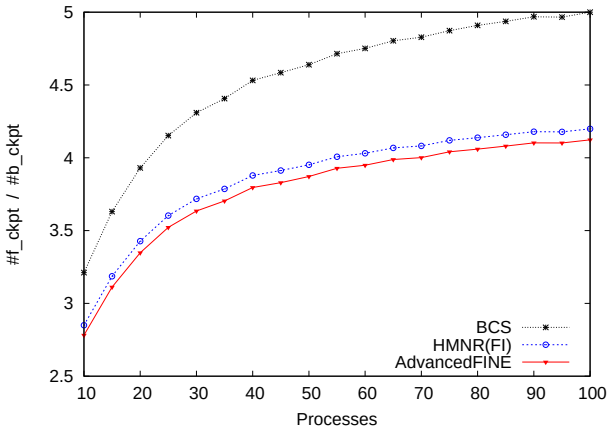
Communication-Induced Checkpointing (CIC) protocols work by evaluating a checkpoint-inducing condition upon message reception and a forced checkpoint must be taken if such condition is satisfied. So the checkpoint-inducing condition used for determing when to take forced checkpoints plays a very important role in the design of efficient CIC protocols.

Model-based CIC protocols[6,13,24,31] track some special communication models/patterns which may lead to the formation of Z-cycles in the future and prevent these potential Z-cycles from being formed. Baldoni et al.[6] present VP-accordant algorithm which ensures no useless checkpoint by preventing the formation of *suspect Z-cycles* based only on available information from the causal past. A "suspect Z-cycle" in [6] is a checkpoint and communication pattern satisfying several constraints. Although a *suspect* Z-cycle is not necessarily a part of real Z-cycle to be formed in the future, VP-accordant ensures that by preventing suspect Z-cycles, the system is guaranteed to be Z-Cycle Free. The PRL protocol presented in [13] decreases the complexity of the piggybacked information from $\mathcal{O}(n^2)$ of the algorithm in [6] to $\mathcal{O}(n)$ and it brings "*progressive view*"[12] of the restricted order for the sequence of consistent global checkpoints being observed. Moreover, it has been proved in [13] that "if PRL takes a force checkpoint, VP-accordant must also induce a forced checkpoint."

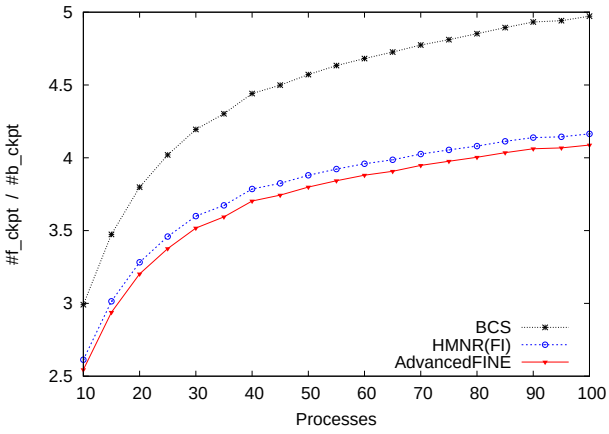
Index-based CIC protocols[1,7,8,16,18,19,21,30] ensure that checkpoints are taken in such a way that Theorem 1 is satisfied, thereby ensuring the ZCF property. These protocols make sure that the timestamps of checkpoints always increase along any Z-path and a forced checkpoint is taken if any violation to Theorem 1 happens. Essentially the checkpoint-inducing conditions of CIC protocols in this approach can be expressed as "$(m.ts > ts_i) \wedge \mathscr{P}$", where $m.ts$ is the timestamp of the message m upon receiving m, ts_i is the current timestamp of the receiver, and $\mathscr{P}$ is a predicate that depends on each protocol [16]. MS algorithm [21] checks if the condition "$m.ts > ts_i$" is satisfied upon receiving a message and it ensures the existence of a consistent global checkpoint containing the latest checkpoint of any process all the time. Helary et al.[16] present a CIC protocol which tries to capture all possibly available information from the causal past and use these information to generate a precise checkpoint-inducing condition which is beneficial to the system in terms of the number of forced checkpoints. It is considered to be one of the best CIC protocols in the literature. Two checkpoint-inducing conditions ($\mathscr{C}1$ and $\mathscr{C}2$) corresponding to two CIC algorithms are found in [16]. Condition $\mathscr{C}1$ is "$(send_i == TRUE) \wedge (m.ts > ts_i)$". And the much more restrictive condition $\mathscr{C}2$ exactly coresponds to *Fully Informed (FI)* protocol. The behavior of our proposed FINE protocol has been carefully analyzed (in Section 4) and compared with FI protocol[16] through simulation (in Section 5). Under all five simulation scenarios, FINE protocol performs better than the FI protocol both in terms of the number of forced checkpoints and message overhead.

7. Conclusion

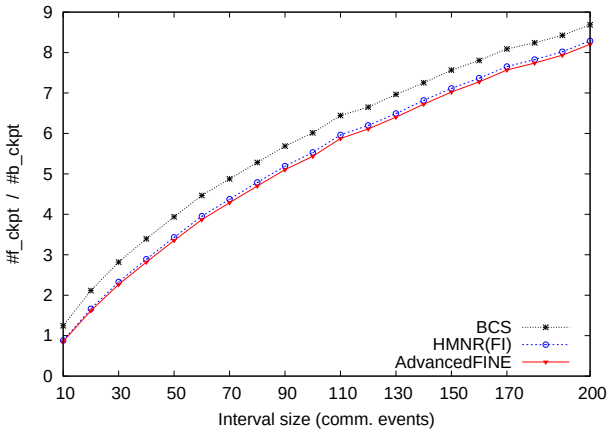
In this paper, first we propose a finer classification of CIC protocols based on the data structures used in these protocols to achieve the goal of ensuring ZCF property. Second, based on the analysis of the intrinsic relation between *Model*-based and *Index*-based CIC protocols, we design a data structure for use in piggybacked messages, namely, *TDE-TS vector*, which helps to get both timestamps and transitive dependency information upon receiving a mes-



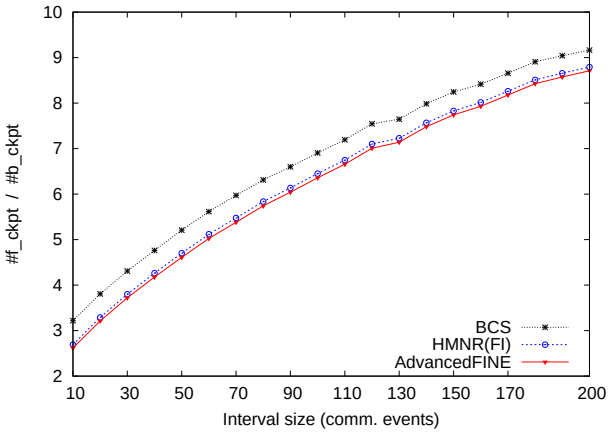
(a) Symmetric, var. process # (SP)



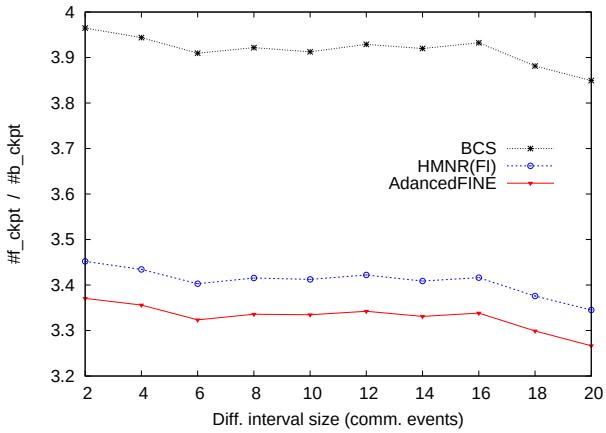
(b) Asymmetric, var. process # (AP)



(c) Symmetric, var. interval length (SI)



(d) Asymmetric, var. interval length (AI)



(e) Asymmetric, var. difference in interval length (AD)

Fig. 11. The ratio of $\#f\_ckpt$ to $\#b\_ckpt$ under five scenarios

sage. Hence it is possible to decrease the overhead of piggybacked information at the expense of a little computation overhead. Then, we propose our Basic and Advanced FINE CIC protocols which fully exploit the collected information from the causal past to make intelligent checkpoint-inducing decision and at the same time decrease the overhead of piggybacked information using TDE TimeStamping mechanism we designed. The simulation results show the performance comparison of our proposed FINE protocol with other five protocols. Last, a review of the related work shows us the major research directions and contributions related to our research area.

References

[1] L. Alvisi, E. N. Elnozahy, S. Rao, S. A. Husain, A. D. Mel, An analysis of communication induced checkpointing, in: Proceedings of the 1999 International Symposium on Fault-Tolerant Computing, 1999.

[2] R. Baldoni, A communication-induced checkpointing protocol that ensures rollback-dependency trackability, FTCS '97: 27th International Symposium on Fault-Tolerant Computing 00 (1997) 68.

[3] R. Baldoni, G. Cioffi, J.-M. Hélary, M. Raynal, Direct dependency-based determination of consistent global checkpoints, International Journal of Computer Systems Science and Engineering 16 (1).

[4] R. Baldoni, J.-M. Hélary, A. Mostéfaoui, M. Raynal, Impossibility of scalar clock-based communication-induced checkpointing protocols ensuring the rdt property, Inf. Process. Lett. 80 (2) (2001) 105–111.

[5] R. Baldoni, G. Melideo, k-dependency vectors: A scalable causality-tracking protocol., in: Proceedings of the 11th Euromicro Workshop on Parallel, Distributed and Network-Based Processing, IEEE Computer Society, 2003.

[6] R. Baldoni, F. Quaglia, B. Ciciani, A vp-accordant checkpointing protocol preventing useless checkpoints, in: SRDS '98: Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems, IEEE Computer Society, Washington, DC, USA, 1998.

[7] R. Baldoni, F. Quaglia, P. Fornara, An index-based checkpointing algorithm for autonomous distributed systems, IEEE Transactions on Parallel and Distributed Systems 10 (2) (1999) 181–192.

[8] D. Briatico, A. Ciuffoletti, L. Simoncini, A distributed domino-effect free recovery algorithm, in: Proc. of IEEE Symposium on Reliability in Distributed Software and Database Systems, Silver Spring (Maryland), 1984.

[9] B. Charron-Bost, Concerning the size of logical clocks in distributed systems, Inf. Process. Lett. 39 (1) (1991) 11–16.

[10] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, D. B. Johnson, A survey of rollback-recovery protocols in message-passing systems, ACM Comput. Surv. 34 (3) (2002) 375–408.

[11] C. Fidge, Logical time in distributed computing systems, Computer 24 (8) (1991) 28–33.

[12] I. Garcia, L. E. Buzato, Progressive construction of consistent global checkpoints, in: ICDCS '99: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, IEEE Computer Society, Washington, DC, USA, 1999.

[13] I. C. Garcia, L. E. Buzato, Checkpointing using local knowledge about recovery lines, Tech. Rep. IC-99-22, Univ. of Campinas, Brazil (Nov. 1999).

[14] I. C. Garcia, L. E. Buzato, An efficient checkpointing protocol for the minimal characterization of operational rollback-dependency trackability, SRDS '04: 23rd Symposium on Reliable Distributed Systems (2004) 126–135.

[15] I. C. Garcia, G. M. D. Vieira, L. E. Buzato, RDT–partner: An efficient checkpointing protocol that enforces rollback-dependency trackability, in: Proc. 19th Brazilian Symp. Computer Networks, 2001.

[16] J.-M. Hélary, A. Mostéfaoui, R. Netzer, M. Raynal, Communication-based prevention of useless checkpoints in distributed computations, Distributed Computing 13 (1) (2000) 29–43.

[17] J.-M. Hélary, A. Mostéfaoui, M. Raynal, Virtual precedence in asynchronous systems: Concept and applications, in: WDAG '97: Proceedings of the 11th International Workshop on Distributed Algorithms, Springer-Verlag, London, UK, 1997.

[18] J.-M. Hélary, A. Mostéfaoui, M. Raynal, Communication-induced determination of consistent snapshots, IEEE Transactions on Parallel and Distributed Systems 10 (9) (1999) 865–877.

[19] R. Koo, S. Toueg, Checkpointing and rollback-recovery for distributed systems, IEEE Trans. Softw. Eng. 13 (1) (1987) 23–31.

[20] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Commun. ACM 21 (7) (1978) 558–565.

[21] D. Manivannan, M. Singhal, A low-overhead recovery technique using quasi-synchronous checkpointing, in: Proc. 16th IEEE Int'l Conf. Distributed Computing Systems, 1996.

[22] F. Mattern, Virtual time and global states of distributed systems, in: Proceedings of the International Workshop on Parallel and Distributed Algorithms, 1989.

[23] R. H. B. Netzer, J. Xu, Necessary and sufficient conditions for consistent global snapshots, IEEE Transactions on Parallel and Distributed Systems 06 (2) (1995) 165–169.

[24] F. Quaglia, R. Baldoni, B. Ciciani, On the no-z-cycle property in distributed executions, Journal of Computer and System Sciences 61 (3) (2000) 400–427.

[25] T. C. Sakata, I. C. Garcia, Non-blocking synchronous checkpointing based on rollback-dependency trackability, in: SRDS '06: Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems (SRDS'06), IEEE Computer Society, Washington, DC, USA, 2006.

[26] R. D. Schlichting, F. B. Schneider, Fail-stop processors: an approach to designing fault-tolerant computing systems, ACM Trans. Comput. Syst. 1 (3) (1983) 222–238.

[27] J. Tsai, J.-W. Lin, On the fully-informed communication-induced checkpointing protocol, PRDC '05: 11th Pacific Rim International Symposium on Dependable Computing (2005) 151–158.

[28] G. M. D. Vieira, L. E. Buzato, Chksim: A distributed checkpointing simulator, Tech. Rep. IC-05-34, University of Campinas (Dec. 2005).

[29] G. M. D. Vieira, L. E. Buzato, Distributed checkpointing: Analysis and benchmarks, in: SBRC '06: Proceedings of the 24th Brazilian Symposium on Computer Networks, Curitiba, Paraná, Brazil, 2006.

[30] G. M. D. Vieira, I. C. Garcia, L. E. Buzato, Systematic analysis of index-based checkpointing algorithms using simulation, in: Proc. IX Brazilian Symp. Fault-Tolerant Computing, 2001.

[31] Y.-M. Wang, Consistent global checkpoints that contain a given set of local checkpoints, IEEE Trans. Comput. 46 (4) (1997) 456–468.

[32] Y.-M. Wang, A. Lowry, W. K. Fuchs, Consistent global checkpoints based on direct dependency tracking, Inf. Process. Lett. 50 (4) (1994) 223–230.

[33] J. Wu, Y. Luo, D. Manivannan, An enhanced model-based checkpointing protocol, in: Proc. 25th Parallel and Distributed Computing and Networks (PDCN 2007), Innsbruck, Austria, 2007.