2016 IEEE International Conference on Cluster Computing

A Lightweight Causal Message Logging Protocol to Lower Fault Tolerance Overhead

Jin-Min Yang
Department of Computer Science and Electronic Engineering, Hunan University
Changsha, China
E-mail: rj_jmyang@hnu.edu.cn

***Abstract*—Rollback recovery is a trustworthy and key approach to fault tolerance in high performance computing and to parallel program debugging. In various rollback recovery protocols, causal message logging shows some desirable characteristics, but its high piggybacking overhead obstructs its applications, especially in large-scale distributed systems. Its high overhead arises from its conservation in the assumption on program execution model. This paper identifies the influence of non-deterministic message delivery on the correct outcome of a process, and then gives a scheme to relax the constraints from the piecewise deterministic execution model. Subsequently, a lightweight implementation of causal message logging is proposed to decrease the overhead of piggybacking and rolling forward. The experimental results of 3 NAS NPB2.3 benchmarks show that the proposed scheme achieves a significant improvement in the overhead reduction.**

Keywords—High performance computing; fault tolerance; rollback recovery; execution model; message logging

I. INTRODUCTION

Driven by modern scientific and engineering applications, the node scale of a computing system is constantly expanded to achieve higher computing performance. For example, the number of nodes of IBM's Blue Gene/P high performance computer has reached to 163,840 [1]. The expansion of the node scale has improved computing performance. However, it also leads to serious dependability problems, since faults increase dramatically with the node scale [2]. At present, for many high performance computing systems, their continuously normal running time is usually less than 40 hours [3]. But many of scientific and engineering computations may require a processing time of tens of hours even several days to obtain a final outcome. So fault tolerance is indispensable for high performance computing (HPC). Rollback recovery [1], [4] is a key approach to fault tolerance in HPC due to its simplicity, as it is based on time redundancy strategy to achieve fault tolerance, no node redundancy introduced [2].

In HPC, the execution of an application consists of many processes, each of which runs in a node of a system. These processes coordinate to complete computing task by message passing. In rollback recovery, every process takes a checkpoint periodically and logs the passing messages in its normal execution time. Upon a fault, the failed process stops executing and its state data in memory is lost. For the fault recovery, an incarnation process is created in a normal node to displace the failed process. The lost process state is recovered by first resetting to its last checkpoint and then replaying the logged messages. This course of replaying is called as rolling forward. So the fault loss is limited to the computation done during the period from its last checkpoint to the fault occurrence, rather than the whole from the startup to the failure. The overhead of rollback recovery refers to checkpointing and logging, as well as restarting and rolling forward. In various rollback recovery protocols, causal message logging [4] can achieve independent fault recovery to any one failed process as compared to others. Therefore, it is widely applied to fault tolerance of high performance computing [1-2] and parallel program debugging. However, its high overhead of message logging is hardly acceptable, especially in large-scale systems [5] and in application scenarios with frequent message passing.

For existing causal message logging protocols [6-9], one source of their high message logging overhead is associated with their assumption on program execution model. The piecewise deterministic execution model [4], called as PWD model, is adopted in almost all existing causal message logging protocols. The model is based on the consideration that, as to a process, the transition of its process state is driven by events of message reception and delivery. Since the arrival of any message is non-deterministic in a process due to no assumption on communication and response latency, the process state is also non-deterministic. The protocols address that, in order to guarantee a process to behave in its recovery completely same as it did before the fault occurrence, it must replay all logged messages since its last checkpoint serially in the same order as it delivered in the past. This treatment is of conservation. It can be inferred that PWD is a sufficient condition to obtain a correct recovery, but not a necessary one. This conservation leads to a high message logging overhead, because the meta of a delivered message must record its delivery order number in its receiver, and is piggybacked to all other processes depending on it directly or transitively. Furthermore, it causes a high rolling forward overhead due to waiting for some specified message. For example, process *P* delivered message m_1 and then message m_2 in its normal execution time, as m_1 arrived in *P* before m_2. Later on, assume *P* fails. In the period of its recovery, it is possible that the logged m_2 arrives in *P* first before the logged m_1, because of no assumption on communication and response latency. According to the PWD model, *P* must wait for m_1 before delivering m_2. Actually,

2168-9253/16 $31.00 © 2016 IEEE
DOI 10.1109/CLUSTER.2016.64

IEEE computer society

perhaps delivering m_2 first and then m_1 doesn't impact the correctness of its recovery at all.

In existing causal logging protocols, another reason for having to employ the PWD model is to achieve the independence of the rollback recovery module [10] on the application module in a process. Consequently, the rollback recovery module is general and can adapt to any kind of applications. The cost of the achievement is that the application module of a process has to be considered as a black box. Thus every message delivery also has to be considered as a non-deterministic event to ensure the correctness of rollback recovery. In this frame, the PWD model is a natural choice. However, the choice is of conservatism, resulting in a high overhead [5], [10]. This paper identifies the influence of non-deterministic message delivery on the correct outcome of a process, and then gives a scheme to relax the PWD constraint. Subsequently, a lightweight causal message logging protocol is proposed to decrease the overhead of message logging and of process's rolling forward.

This paper is organized as follows. Section 2 presents the system model. Section 3 describes our message logging scheme in normal execution time, and fault recovery scheme when a process fails. In section 4, the effectiveness of the proposed scheme is evaluated by experiments. Section 5 introduces the related work. The last section is the conclusion.

II. THE PWD MODEL AND THE RELAXATION TO IT

A. The PWD model

A high-performance computing task is delivered to n processes, each one in a node of a computing system, executing in parallel. The n processes, denoted by P_1, P_2, ..., P_i, ..., P_n, cooperate with each other by message passing to finish the computing task. Message passing makes n processes interrelated with and depended on each other, thereby leading to the consistency issue of the processes' states in fault recovery. In fault tolerance based on rollback recovery, every process does checkpointing periodically in its normal execution time, and does message logging as well. Upon a fault, the failed process stops executing and its state data in memory is lost. To recover from the fault, an incarnation process is created in a normal node. The lost process state is recovered by resetting the process state to its last checkpoint and then replaying the logged messages. This course is called as rolling forward.

The transition of the process state is driven by message delivery. Suppose that the process P_i, with the initial state P_i^0, receives and then deliveries messages m_i^1, m_i^2, ... , m_i^x in sequence from other processes (also including itself) in its lifetime. In P_i, the delivery of m_i^1 drives its state to transit from P_i^0 to P_i^1. During the period from P_i^s to P_i^{s+1}, called as process state interval I_i^{s+1}, it is possible for P_i to send one or more messages to other processes. Assume that the last checkpoint of P_i is taken at P_i^s, $1 \le s \le x$, and then a fault makes P_i failure after P_i has delivered the messages m_i^{s+1}, m_i^{s+2}, ..., and m_i^t in sequence, $s < t \le x$. The procedure of its fault recovery is as follows. The incarnation process is created in a normal node, with P_i's last checkpoint as its initial state first, and then rolls forward by redelivering the logged messages m_i^{s+1}, m_i^{s+2}, ... , and m_i^t in sequence. In order to ensure that any process can be recovered from a fault, all delivered messages must be logged in normal execution time.

In causal message logging protocols, message logging consists of metadata logging and raw-data logging. The metadata of a message is distributed adaptively to those processes depending on it by the way of piggybacking, while the raw-data of a message is buffered in memory by its sender. The metadata of a message is referred to its unique identifier, including the sender identifier *sender_id* and the sending order number *send_index*, as well as the receiver identifier *receiver_id* and the delivery order number *deliver_index*.

B. A perspective to high overhead in existing causal message logging protocols under the PWD model

In causal message logging protocols, for a message m, its raw data may be logged in memory of its sender. When its receiver fails, its sender resends it to the incarnation of the receiver for recovery. If both its sender and its receiver fail, the raw data of m is lost. But fault recovery is not impacted, because m will be regenerated and resent by the incarnation of m's sender in the period of rolling forward. However, this logging strategy cannot be applied to the metadata of m. The reason is that the delivery order number of m becomes unknown when both its sender and its receiver fail. For two messages m_v and m_u, it is possible that the process P_i delivers m_v first and then m_u in its normal execution time, but first m_u and then m_v in its rolling forward, because their arrival time is random. If this case happens, P_i may not be recovered strictly consistently, leading to the problem of the occurrence of orphan message [4] under the PWD model.

In existing causal message logging protocols, the metadata of a message is logged in those processes depending on it directly or transitively to avoid the occurrence of orphan message. When a process sends a message m, it piggybacks on m the metadata of all messages it depends on. When a process delivers a message m, it merges the piggybacked metadata on m into its own set of dependency to update its current dependency set. This scheme leads to a very high overhead. The reasons are as follows:

1) When a process sends a message, all metadata it depends on directly or transitively has to be piggybacked. For example, in Fig.1, the process P_1 has to piggyback the metadata of m_0, m_1 and m_2 on m_3 to the process P_2, when it sends m_3 to P_2. For those applications with a lot of message passing, the high piggybacking overhead is hardly acceptable.

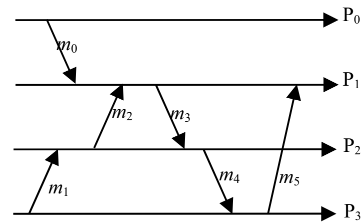

Fig. 1. The dependency among messages

2) There exist a lot of unnecessary piggybacks. For

example, in Fig.1, when P_1 sends m_3 to P_2, the piggyback of the metadata of m_1 is unnecessary. The reason is that P_2 already has held it. But P_1 does not know this situation. So it has to piggyback all metadata. In addition, assume at most f simultaneous failed processes. For any fault recovery, even though it is sufficient to let only f+1 processes log the metadata of a message, there is no way to let a process know how many processes have logged the metadata of the message. Therefore, every process has to conservatively piggyback all metadata onto every message it sends, leading to a high piggybacking overhead.

3) The protocols enforce too serious control in message delivery order. For example, the process P_1 delivered the message m_0 first and then message m_2 in its normal execution time, as m_0 arrived in P_1 before m_2. Later on, in a P_1's fault recovery, it is completely possible that the logged m_2 arrives in P_1 before m_0, because of no assumption on communication and response latency. According to the PWD constraint, P_1 must wait for m_0 before delivering m_2. Actually, perhaps delivering m_2 first and then m_0 doesn't impact the correctness of recovery at all. So the PWD constraint leads to a high recovery overhead.

C. *An observation to non-determinism in parallel computing*

In high performance computing, the sequence of messages a process delivers in the course of its execution is decided by its program logic. Non-determinism of message delivery should be obedient to the program logic. That is to say, non-deterministic message delivery is allowed to the degree that does not impact the correct outcome of program execution. For example, in a system with n processes, suppose every process sends its result to the process P_0 to calculate their sum. For those n messages, any delivery order in P_0 does not impact its correct outcome. So the program of P_0 may be designed to deliver any message it receives, no care about the arrival order of those messages. In this situation, although non-deterministic message delivery exists, the outcome of program execution is same. In contrast, for two messages m_i and m_j from other two different processes, if their delivery order impacts the correct outcome in P_0, the programmer of P_0 will indicate their delivery order. So the deliveries of m_i and m_j are deterministic. This observation has been confirmed to be true in all applications we investigated.

Almost all parallel programming interfaces, like MPI [10-11], support the expressions of both non-deterministic and deterministic message delivery. For example, an API function in MPI, *MPI_Recv*, is defined as follow: *int MPI_Recv(void *buf, int count, MPI_Datatype datatype，int source, int tag, MPI_Comm comm, MPI_Status *status)*, where *buf* is the memory buffer to store the received message, *source* is the sender identifier, and *tag* is the specified identifier for the desired message, such as the type of message. In an program, when *source* is set to *MPI_ANY_SOURCE,* non-deterministic message delivery is introduced, or else deterministic message delivery is indicated.

Based on the above observation, we can find that it is unnecessary to track the delivery order of every message in every process for rollback recovery. Hence the PWD constraint can be relaxed to reduce the overhead. After the PWD constraint is relaxed, causal dependency among messages becomes the sole necessary constraint in rollback recovery. For example, in Fig. 1, assume that the process P_1 delivers m_0 and m_2 in a non-deterministic mode. In a P_1's fault recovery, it is completely possible that message m_5 arrives in P_1 earlier than m_2, since they come from two independent log components in P_2 and P_3, respectively. If there is no causal dependency constraint, P_1 would deliver m_5 before m_2 due to the use of non-deterministic mode. It is obvious that the recovery is wrong, because m_5 depends on m_2 causally. Therefore, a scheme should be developed to track the causal dependency among messages in normal execution time and to follow it in the course of rollback recovery of any process.

III. A LIGHTWEIGHT CAUSAL MESSAGE LOGGING PROTOCOL

A. *Basic ideas*

From the above analyses, we can find that the PWD model assumes that any non-deterministic message delivery would impact the correct outcome of a recovery process. Hence the track of causal dependency among messages is intensified to every message, which leads to a high tracking overhead in normal execution time and a high rolling forward overhead in fault recovery time. In fact, non-deterministic message delivery is allowed in those places that the correct outcome is not impacted. Therefore, the track of causal dependency among messages can be relaxed to the level of process state interval. This kind of relaxation would reduce significantly both the tracking overhead and the rolling forward overhead.

For example, in Fig.1, with #m denoting the metadata of message m, the causal dependency set of message m_5 is the set S (#m_0, #m_1, #m_2, #m_3, #m_4) in the PWD model. In the existing causal dependency tracking protocols, it is necessary to let the message m_5 piggyback S to the process P_1. After the PWD constraint is relaxed, the causal dependency set of m_5 can be simplified into a vector comprised of the process state interval index of every process, i.e. V (0, 2, 2, 1). Since the size of the metadata of a message is 4, we can see that the size of the causal dependency set of m_5 is reduced from 20 to 4. Furthermore, with this simplification, non-determinism of message delivery can still keep valid in rolling forward of a process. This kind of benefit can be seen in the following example. Suppose that the process P_1 uses the non-deterministic mode to deliver messages, and fails after delivering the message m_5. As to m_0 and m_2, the process state interval indices of P_1 they depend on are both 0 in the logs, which means that no message was delivered before m_0 and m_2. Therefore, P_1 can deliver any one of them in its rolling forward for fault recovery as soon as it arrives in P_1. But this situation is not true for the message m_5, because the process state interval index of P_1 that m_5 depends on is 2 in the log. This means that P_1 cannot deliver m_5 until it has delivered other 2 messages. Therefore, with the relaxation to the PWD constraint, the correctness of recovery is still achieved.

B. *Data structure for the lightweight dependency tracking*

In causal message logging protocols, the dependency among messages must be followed in rollback recovery. For

the process P in rollback recovery, when it receives a logged message m from other processes, whether P can deliver it or not? This is a key question to address. Based on the above observations and ideas, the dependency among massages means that message m cannot be delivered until all messages it depends on have been delivered by P. It is unnecessary to care about the delivery order of those depended messages in P. As long as this rule is applied to every message in rollback recovery, the correctness of rollback recovery can be achieved. Therefore, the following data structure is built up for rollback recovery.

Every process maintains a vector *depend_interval*[n] to track the index of the process state interval of all processes a message depends on, where n is the number of processes in the system. For the process P_i, its vector element *depend_interval*[i] records P_i's current process state interval index, with an initial value 0. Whenever P_i delivers a message, its *depend_interval*[i] is incremented by 1, tracking the number of messages that it has delivered. Later on, suppose that P_i sends a message m_i to the process P_j, it is obvious that m_i depends on *depend_interval*[i] messages that P_i has delivered. Therefore, P_i's *depend_interval*[i] is piggybacked on m_i. Subsequently, when P_j sends back a message m_j to P_i, P_j logs m_j along with P_i's *depend_interval*[i] to record the number of messages in P_i that m_j depends on. With this tracking scheme, when P_i fails and rolls back, it knows what time to redeliver m_j from the log of P_j. That is to say, P_i can not deliver m_j until it has delivered *depend_interval*[i] messages.

Additionally, it is known that the dependency among messages can be transitive. For example, in Fig.1, the message m_5 depends on the messages m_0 and m_2 transitively. So other elements in the vector *depend_interval* are used to track the index of the process state interval of all other processes in the system a process depends on at the present time. When a process sends a message, its vector *depend_interval* should be piggybacked on it, as the sent message depends on it. Correspondingly, when a process delivers a message, the piggybacked vector *depend_interval* should be merged into the vector *depend_interval* of its own to report the new dependency of its current process state interval. For example, in Fig.1, before P_1 delivers the message m_5, its vector *depend_interval* is (0, 2, 1, 0). The piggybacked vector on m_5 is (0, 2, 2, 1). When P_1 delivers m_5, its vector *depend_interval* is updated to (0, 2, 2, 1) by merging the piggybacked one.

When a process fails, in order to understand which messages are lost due to the failure, every process also maintains two counting vectors *last_send_index*[n] and *last_deliver_index*[n] to record the number of the sent messages and of the delivered ones respectively, where n is the number of processes. For the process P_i, when it delivers a message m from process P_j, its vector element *last_deliver_index*[j] is incremented by 1 to record the total number of the messages delivered from P_j. When P_i sends a message m to P_j, its vector element *last_send_index*[j] is incremented by 1 to record the total number of the messages sent to P_j. When P_i takes a checkpoint, its vectors *last_send_index* and *last_deliver_index* are also saved as a part of the checkpoint. Later on, suppose P_i fails. Its incarnation is created in a normal node and its initial process state is reset to its last checkpoint. In the situation, all messages that P_i has received since its last checkpoint are lost. In order to let other processes understand which messages are lost, P_i's incarnation broadcasts its vector *last_deliver_index* in the system. Therefore, other processes can understand to send which logged messages to P_i for its recovery.

Algorithm 1 Rollback Recovery protocol

Local Variables:
1: n, P_i {the number of processes, the ID of the process i}
2: $Log_i \leftarrow \varnothing$ {the set of logged messages in P_i}
3: $depend_interval_i \leftarrow (0, ..., 0)$ {the vector of the current dependency in P_i }
4: $last_send_index_i \leftarrow (0, ... ,0)$ {the vector of the last send index in P_i }
5: $last_deliver_index_i \leftarrow (0, ..., 0)$ {the vector of the last delivery index in P_i }
6: $last_ckpt_deliver_index_i \leftarrow (0, ..., 0)$ {the vector of the delivery index of the last checkpoint in P_i }
7: $rollback_last_receive_index_i \leftarrow (0, ..., 0)$ {the vector of the last receiving index from other processes in the case of rollback }
$rollback_last_send_index_i[j] \leftarrow (0, ..., 0)$ {the vector of the last receiving index from the failed process P_i in the case of rollback}

8: **Upon sending message m to P_j**
9: $last_send_index_i[j] = last_send_index_i[j]+1$;
10: if $last_send_index_i[j] > rollback_last_send_index_i[j]$ then
11: Send ($MESSAGE$, $depend_interval_i$, $send_index_i[j]$, m) to P_j;
12: $Log_i \leftarrow Log_i \cup log_item(j, last_send_index_i[j], depend_interval_i, m)$

13: **Upon receiving (*MESSAGE*, *depend_interval*, *send_index*, *m*) from P_j**
14: put the message m (j, *depend_interval*, *send_index*, m) into the *receiving queue*

15: **Upon delivering message**
16: suppose m is the first message in the *receiving queue*
17: If $depend_interval_i[i]$ >= $m.depend_interval[i]$ then
18: $j = m.sender_id$
19: if $m.send_index = last_deliver_index_i[j]+1$ then
20: $depend_interval_i[i] = depend_interval_i[i]+1$
21: $last_deliver_index_i[j] = last_deliver_index_i[j]+1$
22: for $k = 1$ to n
23: if $k \neq i$ and $m.depend_interval[k] > depend_interval_i[k]$ then
24: $depend_interval_i[k] = m.depend_interval[k]$
25: *get_deliverable_message* = *TRUE*
26: takeout m from the *receiving queue* and deliver it to the application
27: else
28: Discard m from *receiving queue*
29: If *get_deliverable_message* ≠ *TRUE* then
30: let m is the next message in the *receiving queue*
31: if $m \neq$ null then goto 17th row else wait;

32: **Upon checkpointing**
33: Save ($Image_i$, Log_i, $last_deliver_index_i$, $last_send_index_i$, $depend_interval_i$) of P_i on stable storage
34: for $k = 1$ to n
35: if $last_deliver_index_i[k] > last_ckpt_deliver_index_i[k]$ then
36: send($CHECKPOINT_ADVANCE$, $last_deliver_index_i[k]$) to P_k
37: $last_ckpt_deliver_index_i[k] = last_deliver_index_i[k]$

38: **Upon Receiving (*CHECKPOINT_ADVANCE*, *last_deliver_index*) from P_j**
39: Release the log items with *receiver_id* = j and *send_index* <= *CHECKPOINT_ADVANCE.last_deliver_index* in Log_i

40: **Upon recovering from a failure**
41: P_i's process image $\leftarrow P_i$'s last *checkpoint.Image*
42: $depend_interval_i \leftarrow P_i$'s last $checkpoint.depend_interval_i$
43: $last_send_index_i \leftarrow P_i$'s last $checkpoint.depend_interval_i$

44: *last_deliver_index*$_i$←P_i's last *checkpoint.depend_interval*$_i$
45: *last_ckpt_deliver_index*$_i$←*depend_interval*$_i$
46: Broadcast (*ROLLBACK*, *last_deliver_index*$_i$) to all other processes

47: **Upon receiving (*ROLLBACK*, *last_deliver_index*) from P_j**
48: Send (*RESPONSE*, *last_deliver_index*$_i$[*j*]) to P_j;
49: while (∃ log item *m* in *log*$_i$ and *m.receriver_id* = *j* and *m.send_index* = *ROLLBACK.last_deliver_index* [*i*] + 1 then
50: send (*MESSAGE, m.depend_interval*, *m.send_index*, *m.msg*) to P_j
51: *ROLLBACK.last_deliver_index*[*i*] = *ROLLBACK.last_deliver_index* [*i* + 1]

52: **Upon receiving (*RESPONSE*, *last_receive_index*) from P_j**
53: *rollback_last_send_index*$_i$[*j*] = *RESPONSE.last_receive_index*

C. Rollback recovery protocol

The rollback recovery protocol consists of three parts: logging and checkpointing, recovering from a fault, and identifying repetitive messages as shown in Algorithm 1.

1) Logging and checkpointing

The logging protocol is shown from the 8th line to the 31th line in Algorithm 1, in which sender-based logging strategy is employed due to its desirable characteristics, such as asynchronous memory buffering. When the process P_i sends a message *m* to the process P_j, it increments its vector element *last_send_index*[*j*] by 1 to count the total number of messages sent to P_j, and piggybacks it and the vector *depend_interval* onto *m* to notify P_j of the dependency of *m*. In the meantime, P_i builds up a log item with the destination *j*, the sending index *last_send_index*[*j*], the vector *depend_interval*, and *m* itself, and puts it in the memory buffer as *m*'s log. When P_i delivers a message *m* from P_j, P_i increments its vector element *depend_interval*[*i*], and *last_deliver_index*[*j*] by 1, respectively, and merges the piggybacked vector *depend_interval* on *m* into its own vector *depend_interval* to update its current dependency, and then delivers *m*.

The checkpointing protocol is shown in Algorithm 1 (from the 32th line to the 39th line). Every process may take independently a checkpoint before it is going to deliver a message. Besides the application state, the vectors *depend_interval*, *last_deliver_index* and *last_send_index*, and the logged messages should also be the content of the checkpoint. After a process takes a checkpoint, the messages the process has delivered are not rolled back any more due to a fault. Therefore, their corresponding logs may be discarded. The vector *ckpt_last_deliver_index* serves this purpose.

2) Recovering from a fault

As to fault recovery, suppose that the process P_i fails. An incarnation is created in a spare normal node to take over and act on P_i's behalf. Thus the incarnation becomes P_i and it executes the following recovery operations, shown in Algorithm 1 (from the 40th line to the 53th line):

(1) Reads P_i's last checkpoint, including the vectors *depend_interval*, *last_deliver_index* and *last_send_index*, and then sets its process state to the last checkpoint.

(2) Broadcasts a *rollback* notification containing the vector *last_deliver_index* to other processes. The purpose of *last_deliver_index* is to let other processes know which messages are lost due to the fault of P_i.

For any process getting the *rollback* notification from P_i, supposing P_j, it sends to P_i a *response* with its vector element *last_deliver_index*[*i*], letting P_i avoid sending some repetitive messages to P_j during its rolling forward. In the meantime, P_j also resends to P_i those logged messages with the destination equal to *i* and the sending index larger than *last_deliver_index*[*j*] contained in the notification. Note that every resent message should be piggybacked with the logged vector *depend_interval* as in normal execution mode to let P_i understand what time to deliver it, as well as rebuild P_i's vector *depend_interval*.

Let's shift to the side of P_i. There is a *receiving queue* in P_i. The received messages are first put in the *receiving queue*. Whenever P_i is going to deliver a message, the dependency among messages must be followed. That is to say, for any message in the *receiving queue*, only when P_i's vector element *depend_interval*[*i*] is larger than or equal to the piggybacked *depend_interval*[*i*] on it, the message can be delivered. Or else the message has to remain staying in the *receiving queue*. This control is shown from the 16th line to the 31th line in Algorithm 1.

3) Identifying repetitive messages

During the period of rolling forward in a fault recovery, when the process P_i sends a message *m* to another process, supposing P_j, perhaps *m* is a repetitive message as to P_j. For a repetitive message, it is unnecessary for P_i to send it. If the *response* from P_j has been gotten, and P_i's vector element *last_send_index*[*j*] is less than or equal to the piggybacked *last_deliver_index*[*i*] on the *response*, P_i knows that *m* is repetitive, and then omits sending it. Or else P_i sends *m*.

Before P_i gets the response from P_j, it cannot judge whether *m* is repetitive or not. So P_i has to send it conservatively. Therefore, it is possible for a process to receive a repetitive message from P_i. Before a process delivers a message *m* from P_i, it first checks if the piggybacked *send_index* on *m* is less than or equal to the vector element *last_deliver_index*[*i*] of its own. If so, *m* is repetitive, and P_j discards it.

The identification of repetitive messages is shown from the 52th line to the 53th line and from the 10th line to the 11th line in Algorithm 1.

D. The correctness of the protocol

Any rollback recovery protocol should ensure that the dependency among messages is not violated and there is no any lost message and no any repetitive message in the course of fault recovery. In our logging protocol shown in the above subsection, every sent message depends on the current process state interval of its sender, while the current process state interval of a process depends on both its last process state interval and the last delivered message. Our protocol tracks both the dependency of every message by the way of piggyback, and the dependency of the current process state interval by merging two dependency vectors. Therefore, for messages sent to a process, their dependency is tracked directly or transitively. This kind of track means that, when a message

is logged in normal execution time, its dependency can be logged along with it. Upon a fault with one failed process, for all messages needed for the fault recovery, the recovery process can obtain them from the logs in the surviving processes. The logged dependency is used to control the delivery order.

When multiple simultaneous process failures take place, the logged messages in failed processes are lost. For example, suppose that P_1, P_2 and P_3 fail at the same time as shown in Fig 2, thus the logged messages m_1, m_2, m_3, m_4, and m_5 are lost. Although P_1 possibly delivers m_0 first and then m_2 when it rolls forward in its recovery, not same as it did before the failure, the execution is correct. The reason is that m_0 and m_2 are independent on each other, thus their delivery order does not impact P_1's correct outcome. That is to say, m_6 does not become an orphan message. Nevertheless, as to m_7, the situation is completely different. Although P_1 possibly receives m_7 before m_2 in its recovery, but it cannot deliver m_7 before m_2. The reason is that the dependency of m_7 indicates that m_7 cannot be delivered until P_1 has already delivered other 2 messages. In addition, even though the logs of m_1, m_2, m_3, m_4, and m_5 are lost due to the failures of P_1, P_2 and P_3, these messages, along with their dependencies, are regenerated by P_1, P_2 and P_3 in the period of their rolling forward in recovery, respectively. Their logs are also rebuilt correctly. As a result, in the case of multiple simultaneous failed processes, the dependency among messages is still followed, and no any orphan message appears in rollback recovery.

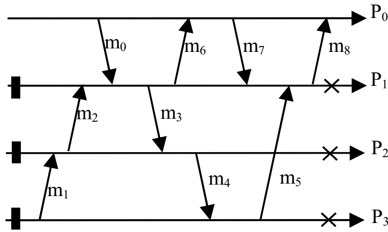

Fig. 2. Rollback recovery in the case of multiple simultaneous faults

Furthermore, our protocol identifies every message by its receiver and its sender via the vectors *last_send_index* and *last_deliver_index* respectively to guarantee no lost message and no repetitive message in rollback recovery. As to any process, its last checkpoint records the vector *last_deliver_index*, each element of which indicates the last message that is not lost due to a later fault. Once a process fails, its checkpointed vector *last_deliver_index* can let other processes know which messages are lost and should be resent by comparing the received *last_deliver_index* with their own *last_send_index*. Hence, all lost messages are identified and resent to let the failed process get recovery. For example, in Fig.2, As to P_1, the checkpointed vector element *last_deliver_index*[0] is equal to 0, indicating no any message delivery from P_0. After the incarnation of P_1 broadcasts a recovery notification with the vector *last_deliver_index* in the system, P_0 knows that P_1 lost the last 2 messages from P_0, as P_0's vector element *last_send_index*[1] is equal to 2.

In the meantime, our protocol also ensures that any process does not deliver any repetitive message. Although a process possibly sends a repetitive message in the period of its rolling forward, its receiver can identify it by comparing its *last_deliver_index* with the piggybacked *sending_index.* For example, in Fig.3, suppose that the process P_1 fails, it is possible for P_1 to send a repetitive message m_3 to P_3 in the period of its rolling forward in recovery, as P_1 does not get the *response* from P_3 yet before sending m_3. Nevertheless, the repetition can be identified by P_3, because the piggybacked *sending_index* on m_3 is 1, less than or equal to the vector element *last_deliver_index*[1] in P_3. Thus P_3 discards the repetitive message m_3.

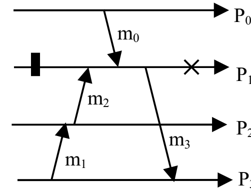

Fig. 3. The situation of repetitive messages

From the above analyses, it can be concluded that, in our protocol, the dependency among messages is followed, and no lost message and repetitive message appears. So it is correct.

E. The implementation of complete non-blocking

There are two other issues to address in the implementation of rollback recovery. The first one is that many communication components are implemented in a synchronization mode in message passing in parallel computing, such as MPICH [12]. That is to say, when a process sends a message by calling a communication API, its execution is blocked until the message has been received by its receiver. If the receiver fails, the sending will fails and the execution of the sender will be blocked until the receiver recovers. So the failure of a process hampers the execution of other normal processes. The second one is that, during the period of recovery of a process, those messages for rolling forward do not arrive in the same order as in normal execution time. The reason is that there is no any constraint in communication latency on sending a logged message to a recovery process. Hence there appears a possible situation that, even though a message arrives, it cannot be delivered in recovery until all depended messages arrive and are delivered. Therefore, it is vital for rollback recovery to have a scheme to manage sending, receiving and delivering messages effectively. This paper proposes a scheme based on buffering and multithreading as shown in Fig. 4.

In the proposed scheme, the rollback recovery component is embedded between the application component and the communication component, as shown in Fig.4. Moreover, two message buffering queues *A* and *B* are created in memory in the rollback recovery component. The queue *A* is for receiving messages, while the queue *B* is for sending messages. Corresponding to two buffer queues, two other threads are created for sending and receiving messages, respectively. When the application thread sends a message *m* to the process P_j, it attaches *j* as the receiver identifier and the vector element

last_send_index[*j*] as the sending index to *m*, puts *m* into the queue *A*, and then returns to continue its computation. Therefore, its execution is not blocked. The sending thread is responsible for sending the messages in the queue *A* based on the sending protocol shown in Algorithm 1. Note that after a message has been sent, it keeps in memory and is transformed into a log. On the other side, the receiving thread puts every received message in the queue *B*. When the application thread is going to deliver a message, the delivery manager chooses the desired one in the queue *B* according to the delivery protocol shown in Algorithm 1. If there is no desired message in the queue *B*, the application thread waits until the arrival of a desired message.

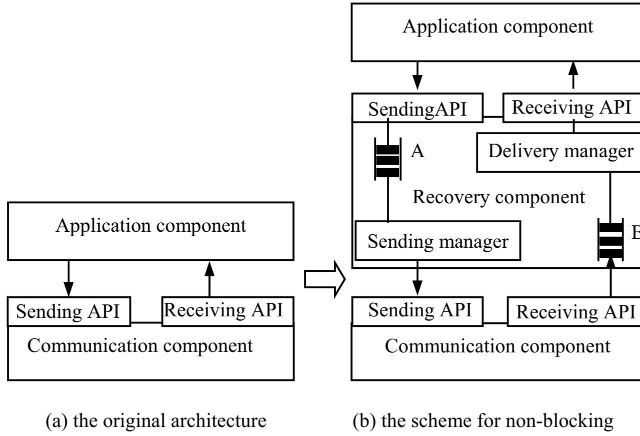

Fig. 4. The implementation scheme of complete non-blocking

It can be seen that this scheme completely eliminates blocking of computation rooted in other process's failure by the way of buffering and multithreading. Moreover, it enables computing, sending message and receiving message to execute concurrently. It contributes to a lower overhead of rollback recovery. Its effectiveness will be shown in the next section.

IV. EXPERIMENTAL EVALUATION RESULTS

In this section, we compare the protocol presented in section 3 to the two existing representative protocols, one based on antecedence graph [7] and the other based on event logger [5], by several experiments to investigate the effect of our lightweight causal message logging protocol. Additionally, we evaluate the gain from the elimination of computation blocking rooted in other process's failure.

In research of rollback recovery, NAS NPB2.3 benchmark [13] is a widespread adoption to evaluate and compare the performance of various protocols. In our experiments, three NAS NPB2.3 benchmarks, LU, BT and SP, were used to compare the performance of 3 rollback recovery protocols. These benchmarks are MPI programs about molecular dynamics calculations written in Fortran language. They have different representative features: LU with high message frequency and relatively small checkpoint size, BT with large checkpoint size, large message data size and relatively low message frequency, and SP with moderate message frequency and checkpoint size, relative to LU and BT.

We have established an experimental testbed to execute parallel computing. The lightweight causal message logging protocols as well as the non-blocking communication scheme described in the section 3 were implemented in our rollback recovery support library WINDAR [14], while WINDAR was embedded in MPI support library MPICH [12] consisting of MPI API, ADI2 and NT_IPVISHM. The software hierarchy of the system is shown in Fig. 5. In our experiment configurations, every application consisted of 4, 8, 16, and 32 processes respectively, each of which runs on a PC with a AMD Athlon 2.3GHZ CPU, 896M RAM, a 640G hard disk and Windows XP operating system. All PCs are connected by 100M Ethernet. We set the checkpoint interval to 180 seconds effective computation. We examined logging overhead and recovery overhead in a checkpoint interval.

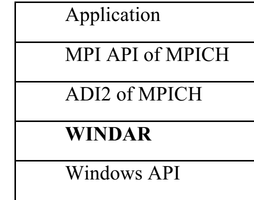

Fig. 5. Software stack in the rollback recovery experiment

A. *Comparisons of the piggybacking overhead in 3 protocols*

We first investigated the piggybacking overhead in three causal message logging protocols: TDI (Tracking based on dependent interval presented in section 3), TAG (tracking based on antecedence graph [7]) and TEL (tracking based on event logger [5]), respectively. The piggybacking overhead was measured in the two criteria: data amount and time.

The experimental results are shown in Fig. 6 and Fig. 7, respectively. We can observe that both time overhead and data amount overhead are remarkably smaller in our protocol than in other two protocols. Especially when applications have frequent message passing, such as LU, the effectiveness of our protocol is more prominent. It is also true for a large-scale system indicated by the number of processes. The reason is that, for our protocol to achieve correct recovery, it is sufficient only to track the current number of delivered messages in every process. In contrast, the other two protocols need to track the whole history of all events of message delivery in every process. Therefore, the data amount of piggyback is significantly reduced in our protocol, which also leads to a positively proportional reduction in time overhead. The large amount of piggyback is one source of high time overhead. Another source is the calculation of the increment of antecedence graph. The purpose of this calculation is to reduce of the amount of piggyback. Such a calculation does not exist in our protocol, while it refers to a traverse to an antecedence graph in the other two protocols. Consequently, our protocol has no calculation overhead.

The experimental results also show that our protocol has a better scalability in contrast to the other two protocols. The reason is that, for a process going to send a message, since there is no way for it to precisely know the antecedence graph that the receiver currently holds, it has to piggyback conservatively sufficient metadata to the receiver. Such a

conservative scheme leads to a lot of redundant piggyback of metadata. This redundancy is in the order of a vector in our protocol, while it is in the order of a two-dimensional graph of message meta in the other two protocols. Therefore, the effectiveness of our protocol is more significant in the scenarios of large system scale or frequent message passing.

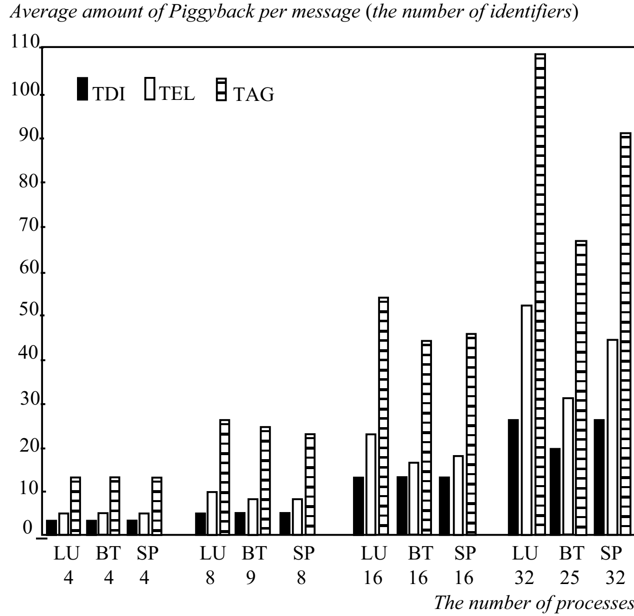

Fig. 6. The comparison of the average amount of piggyback in 3 protocols on 3 benchmarks LU,BT and SP

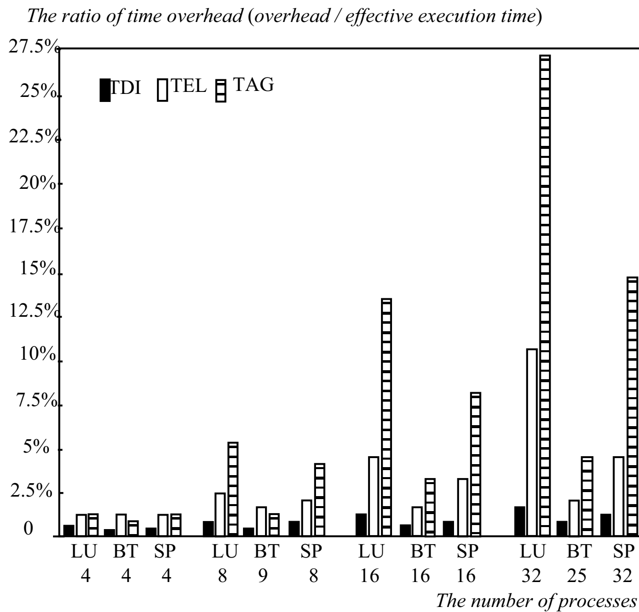

Fig. 7. The comparison of the time overhead of tracking in 3 protocols on 3 benchmarks LU,BT and SP

Additionally, we can observe that three applications have very different time overhead. The reason is that they have different message passing frequency and communication pattern. For example, LU has a frequent message passing, so its time overhead is much higher than that of BT and SP. TAG and TEL protocols need to maintain an antecedence graph for dependency tracking. The size of the antecedence graph is positively proportional to the frequency of message passing. So the piggyback amount and time overhead increase sharply with the frequency of message passing and the system scale in TAG and TEL protocols. In contrast, our protocol maintains only a vector with the size of node scale in the system, and has no increment calculation .Therefore, its piggyback overhead increase linearly with the system scale, while its time overhead is hardly relevant to the system scale.

B. 4.2 The gain from eliminating blocking of computation

Our second experiment was conducted to evaluate the gain from eliminating blocking of computation rooted in other process's failure presented in subsection 3.5. In this experiment, we constructed two communication modes shown in Fig.4 (a) and (b) respectively, and employed our protocol TDI for rollback recovery. After a process took a checkpoint, we let it run for 180 seconds, and then triggered a fault and then immediately carried out the rollback recovery. The gain is defined as the normalized difference of the accomplishment time of computation task in two communication modes: blocking and non-blocking.

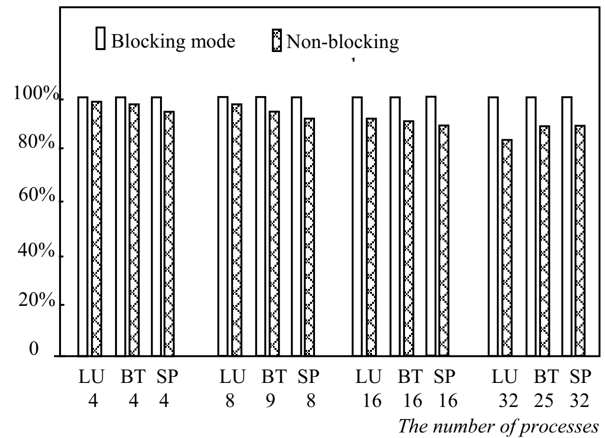

Fig. 8. The gain from non- blocking on 3 benchmarks LU,BT and SP

The gain is shown in Fig.8. It can be observed that it is explicit in 3 benchmarks, especially when the system scale becomes large. Additionally, we note that, although the gain in application LU is largest when the number of processes is 32, it is not positively proportional to the number of messages. The explanation is that the data amount of messages also plays an important role in the gain. Bigger is the size of a message, longer is its transmission time. Hence the blocking time of computation is also correspondingly longer when the blocking mode is employed. Moreover, the receiver of a message also plays an important role in blocking. When a process is sending a big message, if the receiver is executing a recovery or

application computation, it has to wait. The reason is that the buffering memory in communication subsystem is limit and not enough to cache the whole message. The blocking of the sending process does not end until the receiving process transits from recovering or computing to receiving a message. Therefore, for performance improvement, it is significant to eliminate blocking by the way of buffering and multithreading presented in Section 3.5.

Although the experimental results show some gain from the non-blocking of computation, it is not very significant. The reason is that all processes are close coupled with each other in three applications. Once a process fails, other processes cannot advance in their computation at all, as they need messages from the failed process. Therefore, they have to wait for the recovery of the failed process.

V. THE RELATED WORK

In various rollback recovery protocols, causal message logging [4] shows many good characteristics. For example, every process can take its checkpoints independently, and recover independently from a fault. Moreover, there is no synchronization constraint in message logging and delivering. Therefore, they are widely applied to fault tolerance of high performance computing and parallel program debugging. However, their piggybacking overhead is hardly acceptable, especially in large-scale systems [15-16] and in the scenarios of frequent message passing. Manetho [6] is the first to address the piggybacking overhead. It used an antecedence graph to describe the causal relationship between non-deterministic events of message delivery. In order to remove redundant piggybacking, when a process sends a message to another one, it does not piggyback the complete graph, but an incremental part. For those metadata that the sender knows that the receiver already holds, they do not need to be sent back to the receiver. LogOn [7] did an improvement to Manetho. It partially reorders events from the inheritance relationship in the antecedence graph by exploiting the semantics of the metadata of message. This treatment enables an adequate utilization of the semantics of the antecedence graph to easy and accurate the calculation of an incremental part.

The above two approaches do not have any assumption on the number of simultaneous faults in a system. Thus the condition ending the piggyback of the metadata of a message is that all processes hold it and know that all other processes already hold it. This situation inevitably gives rise to many unnecessary piggyback, thereby leading to an unacceptable piggybacking overhead. Alvisi [8] proposed 3 schemes to control the propagation of the metadata of a message under the assumption of at most f simultaneous faults in the system. With the assumption, it is sufficient for the metadata of a message to be piggybacked to only $f+1$ processes. To know how many processes already hold the metadata of a message, extra tracking information is introduced and also needs to be piggybacked as the metadata of messages does. The size of tracking information can be $O(n^2)$, $O(n)$ or $O(1)$, where n is the number of processes in the system. Of course, little tracking information generally corresponds to poor control accuracy. However, no matter how much extra tracking information is piggybacked, there is no way for a process to precisely know how many other processes already hold the metadata of a message.

To address the above problem, Event logger [5],[9] was proposed. It introduces a stable storage to save metadata, and keep asynchronous message logging. In this scheme, the piggyback of the metadata of a message ends as soon as it is saved on the stable storage. Although the piggyback overhead is remarkably reduced, besides the stable storage, extra notification messages are also introduced in the system.

As to the piggybacking overhead, the above-mentioned causal message logging protocols are poor in scalability. To address this problem, system partition schemes [15],[17-18] came out. After a big system is structured into some small units, conventional causal logging is conducted in a small scale. For those messages across the boundary, their dependency is dealt with via various measures, such as pessimistic logging [17], transformation [15], and replication [10].

All the above-mentioned researches are under the assumption of the PWD execution model. The track of the dependency refers to the delivery order number of every message. Although some improvements are achieved, they are not essential. In contrast with the PWD execution model, the send deterministic execution model [11], [19-20] was proposed to address the conservative characteristics of PWD, which argues that the sequence of messages sent by each process is the same in any correct execution of the application. This new model enables new rollback-recovery protocols to lower the message logging overhead [21].

Our protocol in Section 3 is similar to the send deterministic execution model. It is different from the existing causal message logging protocols in that it considers the correct recovery of a failed process can be achieved by following the dependency among messages, not necessarily by the strictly same delivery sequence of messages. Therefore, in the normal time, the track of dependency can be relaxed from logging the delivery order of every message to logging the number of delivered messages. Furthermore, this kind of logging only happens at the time of sending a message. This kind of relaxation leads to a significant reduction of the logging overhead from piggybacking a much smaller data amount of dependency and from eliminating the calculation of the incremental part of piggyback. In our protocol, the piggyback onto a message refers to only a vector of integer. In contrast, it involves a two-dimensional graph of message meta in existing causal logging protocols.

Additionally, our protocol achieves a proactive perception of delivery order of messages. In existing causal logging protocols, as to a message, its metadata log and its raw data log are completely separated in different time and places. Specifically, its sender logs its raw data. Afterwards, its metadata is constructed when it is delivered, and then is logged in other processes depending on it. As a result, when a failed process rolls back for recovery, only after it receives both the metadata and the raw data of a message from different processes, it can determine what time to deliver this message. In our protocol, when a process sends a message, it knows the dependency of the message. Thus the dependency information

and the raw data of a message are logged together in its sender. Therefore, our protocol achieves a proactive perception of delivery order of messages. As to a logged message, it delivery order is determined as soon as it arrives in the recovering process. Consequently, the overhead of rolling forward is also reduced significantly.

VI. Conclusions

In various rollback recovery protocols, as causal message logging does not require any rollback of normal process in the case of fault recovery, and has no any synchronization constraint on message logging and delivering, it becomes a widespread adoption to fault tolerance in high performance computing and to parallel program debugging. However, existing causal message logging protocols have a hardly acceptable piggybacking overhead, especially when the system scale has been enlarging or there is frequent message passing. The high overhead arises from their conservation in the assumption of the piecewise deterministic execution model. As to correct rollback recovery, this model is sufficient, but not necessary. Correct rollback recovery can be achieved by following the dependency among messages, not necessarily by strictly same delivery sequence of messages. Therefore, in our causal logging protocol, the track of dependency is relaxed from logging the delivery order of every message to logging the number of delivered messages. This kind of relaxation leads to an effective reduction of the data amount of piggyback onto a message from a two-dimensional graph of message meta to a vector of integer. Consequently, the piggybacking overhead is significantly reduced, with a subsidiary benefit of the decrease of the rolling forward overhead in fault recovery. In addition, our protocol completely eliminates blocking of computation rooted in other process's failure by the way of buffering and multithreading, leading to a further decrease in the rollback recovery overhead. Our protocol achieves a good scalability, as its time overhead is hardly relevant to the node scale of the system.

Acknowledgment

This work was supported in part by National Natural Science Foundation of China under grant 61272401, Key Science and Technology Plan of Hunan province under grant 2013GK2003, and the Prospective Research Project on Future Networks of Jiangsu Future Networks Innovation Institute under grant 2013095-1-05.

References

[1] X. Yang, Z. Wang and J. Xue, "The reliability wall for exascale supercomputing," IEEE Transactions on Computers, 2012, Vol. 61, No. 6 , pp. 767–779.

[2] F. Cappello, A. Geis and W. Gropp et al,"Toward exascale resilience – 2014 update," Journal of Supercomputing Frontiers and Innovations, 2014,vol. 1, No.1, pp.5–27.

[3] R. Gupta1, H. Naik and P. Beckman,"Understanding checkpointing overheads on massive-scale systems: analysis of the IBM Blue Gene/P system," Int'l Journal of High Performance Computing Applications, 2011, Vol.25, No.2, pp.180–192.

[4] E. Elnozahy, L. Alvisi, and Y. Wang et al,"A survey of rollback recovery protocols in message passing systems," ACM Computing Surveys, 2002, Vol.33, No.3, pp.375–408.

[5] A. Bouteiller, B. Collin and T. Herault et al,"Impact of event logger on causal message logging protocols for fault tolerant MPI," 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05), 2005, pp. 97–106.

[6] E. Elnozahy and W. Zwaenepoel,"Replicated distributed processes in manetho," In 22nd International Symposium on Fault Tolerant Computing, 1992, pp.18–27.

[7] B. Lee, T. Park and H. Yeom et al,"An efficient algorithm for causal message logging," In 17th Symposium on Reliable Distributed Systems,1998, pp.19–25.

[8] K. Bhatia, K. Marzullo and L.Alvisi,"Tracking causality in causal message logging protocols," Distributed Computing, 2002, Vol. 15,No.1,pp. 1–15.

[9] T. Ropars, C. Morin,"Improving message logging protocols scalability through distributed event logging," Euro-Par 2010-Parallel Processing, 2010.

[10] E. Meneses, G. Bronevetsky and L. Kale, "Evaluation of simple causal message logging for large-scale fault tolerant HPC systems," IEEE International Symposium on Parallel and Distributed Processing , 2011, pp.1533–1540

[11] F. Cappello, A. Guermouche and M. Snir,"On communication determinism in parallel HPC applications," In 19th International Conference on Computer Communications and Networks (ICCCN), 2010, pp.1–8.

[12] MPICH | high-performance portable MPI. https://www.mpich.org/, 2014.

[13] NAS parallel benchmarks. NASA Ames Research Center. http://science.nas. nasa.gov/ Software/NPB/, 2012.

[14] J. Yang, D. Zhang and X. Yang,"WINDAR: a multithreaded rollback-recovery toolkit on Windows," In 10th IEEE Pacific Rim dependable Computing International Symposium, 2004.

[15] K. Bhatia and K. Marzullo and L. Alvisi, "Scalable causal messagelogging for wide-area environments," Concurrency and Computation: Practice and Experience, 2003, Vol.15, No.3, pp. 873–889.

[16] J. Lifflander, E. Meneses and H. Menon et al,"Scalable replay with partial-order dependencies for message-logging fault tolerance," IEEE International Conference onCluster Computing (CLUSTER), 2014, pp.19–28

[17] A. Bouteiller, T. Herault and G. Bosilca et al,"Correlated set coordination in fault tolerant message logging protocols for many-core clusters," Concurrency and Computation: Practice and Experience, 2013, Vol. 25, pp.572–585

[18] Y. Ci, Z. Zhang and D.Zuo et al, "Message fragment based causal message logging," Journal of Parallel and Distributed Computing. 2009, Vol.69, pp. 915–921.

[19] A. Guermouche, T. Ropars and M.Snir et al, "HydEE: Failure containment without event logging for large-scale send-deterministic MPI applications," IEEE 26th International Parallel and Distributed Processing Symposium, 2012, pp.1216–1227.

[20] A. Guermouche, T. Ropas and E. Brunet,"Uncoordinated checkpointing without domino effect for send-deterministic MPI applications," IEEE Int'l Parallel & Distributed Processing Symp.(IPDPS), 2011, pp. 989–1000.

[21] N. E. Sayed and B.Schroeder," Understanding Practical Tradeoffs in HPC Checkpoint-Scheduling Policies," IEEE Transactions on Dependable and Secure Computing, 2016, DOI 10.1109/TDSC. 2016.2548463