Incremental Record and Replay Mechanism for Message Passing Concurrent Programs

Yi Zeng

School of Computer Science and Technology, Nanjing Normal University, Nanjing 210023, China.

05277@njnu.edu.cn

Abstract

Replaying the execution of a concurrent program from a runtime trace is a standard approach to performance analysis and debugging of a program with non-deterministic behavior. In this paper, we present a scheme that integrates checkpointing and message logging technique into record and replay mechanism in the debugging circumstances. Our incremental record and replay mechanism has shortened the debugging time considerably. There are two phases in it: one is record phase, which records the minimal information of the programs original execution to introducing minimal overhead; the other is replay phase, which uses the recorded information to force the replay behavior as the same with the original execution and provides the instant replay of the programs with re-execution from intermediate instead of from the beginning.

Keywords

Non-determinism, Message-passing, Record and Replay, Checkpointing, Message logging.

1. Introduction

Nowadays, with the improvement of user's demand for efficient software, concurrent technology is used widely. But numerous concurrency bugs exist in concurrency programs, such as event race bugs, data race bugs and message race bugs. These bugs have made a lot of challenges for developers and testers to test and fix them due to their inherent non-determinism. This is especially true for concurrent programs communicating via message-passing. Even with the same input, the concurrency programs might produce different executions, because of variation in message latencies and process scheduling. Recent work focuses on detecting races and classifying them as harmful or harmless by static analysis and dynamic analysis [1-5]. Although many concurrency bugs have been detected, they often cannot be replayed. Obviously, replay these detected bugs requires adequate information about how its processed communicated during the original execution.

So, guaranteeing reproducibility is a major issue in the concurrent program debugging. One of the main reproducibility techniques is record and replay [6-13]. It faithfully records the execution interleaving and deterministically relays the same interleaving to reproduce concurrency bugs. Moreover, the message-passing programs are often running hours or days, so it is not possible to replay the whole execution from the beginning. For this, we introduce independent strategy from fault tolerant systems which can save the intermediate states of each process independently without any coordination from each other. With it, we can re-execute programs from any interested checkpoint instead of the beginning.

For these reasons started above, we have researched and developed an adaptive algorithm named incremental record and replay, which integrates the checkpointing and message logging technique into

the record and replay mechanism. In the algorithm, we must face two difficult problems. On the one hand, when we replay some intervals in the program, beginning from the interest checkpoint of the process, and stopping until the next checkpoint, all the messages it received must be reproduced the same as the original. On the other hand, only the recorded message content is not enough, the order of message delivery must also be recorded and preserved during the replay, to grant deterministic re-execution.

This paper proposes an adaptive algorithm to resolve the two difficult problems. The benefits of our approach algorithm include the follow aspects: The record strategy optimizes the amount of ordering information saved by dynamically performing a transitive reduction on the racing messages. It is shown to reduce trace overheads by saving one order in each race to eliminate the non-determination. The message logging mechanism only logs the domino messages and the other messages are re-computed through re-execution of the corresponding intervals so that we can break domino effect and limit logging overhead while bounding the time required satisfying an instant replay request.

The rest of this paper is organized as follows: Section 2 briefly introduces the system model. Section 3 and 4 explain the record and message logging strategy. Section 5 describes the replay dependence sets and the replay algorithm, and section 6 makes the conclusion of the study.

2. System Model

In this paper, we treat the concurrent programs which only use explicit synchronous/asynchronous message passing primitives. Each process is dependently executed and communicates with other processes only by exchanging messages, and the messages are assumed to be delivered reliably.

Definition 1 A message-passing program P P=<p1, p2, p3, …, pn >, where pi is a process (message-passing program consisting of n processes).

Definition 2 A checkpointed program execution G, G=<E, hb, S, db>, where E is a finite set of events, and hb is the executed dependence relation defined over E; S is also a finite set of checkpointed intervals and db is the interval dependence relation defined over S.

Definition 3 $e_{i,j}$ represents the jth event in the process pi, include the send and receive operation.

Definition 4 $C_{i,j}$ represents the jth checkpoint in the process pi.

Definition 5 A checkpointed interval $I_{i,j}$, it represents all computation performed in process pi between the checkpoint $C_{i,j}$ and $C_{i,j+1}$(and includes $C_{i,j}$ but not $C_{i,j+1}$).

Definition 6 The relation hb shows how events dependent on one and another, it is defined as follow:

(1) If $e_{i,j}$,$e_{i,k}$ are in the same checkpointed interval, and i<k ,then $e_{i,j}$ hb $e_{i,k}$

(2) If $e_{i,j}$ is a sending event and $e_{k,l}$ is the corresponding receiving event , then $e_{i,j}$ hb $e_{k,l}$

(3) If $e_{i,j}$ hb $e_{k,l}$ and $e_{k,l}$ hb $e_{s,t}$ then $e_{i,j}$ hb $e_{s,t}$.

Definition 7 The relation db is defined as follows:

(1) If $e_{i,j}$ is a sending event in the checkpointed interval $I_{i,j}$, and the $e_{k,l}$ is the corresponding receiving event in the $I_{k,l}$,then $I_{i,j}$ db $I_{k,l}$.

(2) If $I_{i,j}$ db $I_{k,l}$ and $I_{k,l}$ db $I_{s,t}$ then $I_{i,j}$ db $I_{s,t}$.

In our model, we must emphasize that a checkpointed interval is not dependent on the earlier ones in the same process, because we introduce the in-dependent checkpointing and when we replay the interval no events in earlier intervals in the same process need to be re-executed.

To illustrate the model, let us consider the execution in Fig1, where horizontal lines represent the execution of each process, blocks represent independent checkpoints, and the inter-process arrows represent message passing.

3. Record Strategy

3.1 Traditional Record Method

The record and replay method apply two phases: first it records minimal information about a particular message-passing program execution while introducing minimal overhead. Then this information is used to control the re-execution of program. The computation overhead should be limited as much as possible both in time and in space in order to avoid the probe effect.

Two executions of a process P are considered to be equivalent if the process P receives the same information from the other processes at the same instant. The instant of an event is defined by the interval in which only this event take place. This means that two executions of a concurrent program will be considered to be equivalent if the execution of each process is equivalent.

The equivalent re-execution can be enforced by using either a contents-based record or an ordering-based record. Contents based record methods record the contents of the messages received. The recorded data are then fed back during the replay phase. It is obvious that this will lead to immense record overhead. The second approach, ordering based record alleviates this drawback by recording the order of the communicating events. By imposing this ordering during replay, a faithful re-execution will occur.

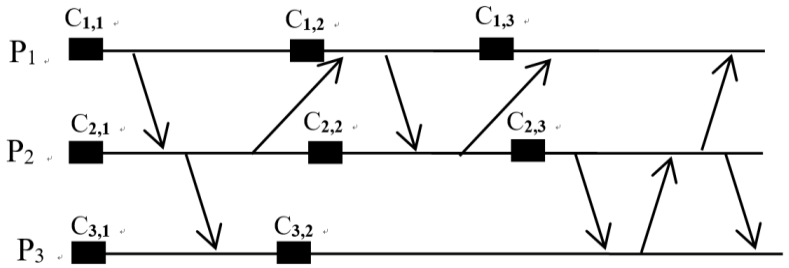

Fig 1. An execution with three processes

3.2 The Racing Messages

Our record algorithm is an improved ordering-based record scheme, by only recording the racing messages order. In message-passing program, not each message passing can cause the non-determination, so we don't need trace all the order of every message sends and receive operations. Only the message racing can result in the non-deterministic execution. The race happens between two messages which are simultaneously in transit and either could arrive first at some receive operation, due to variations in messages latencies or process scheduling.

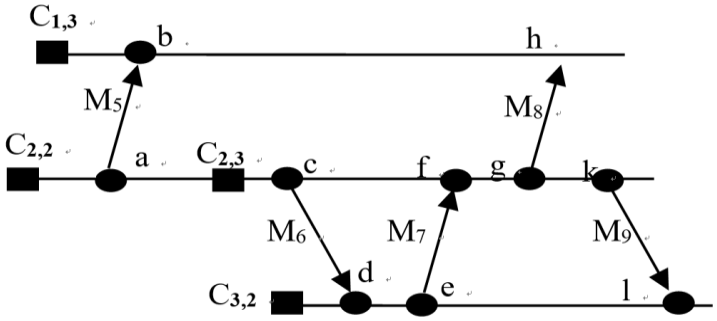

Fig 2. The racing messages

In this section we formally define the notion of a racing message [6]. Fig2 shows an example to describe the racing messages and non-racing ones. Checkpointed interval I1,3 has two receive events b and h, and the I3,2, also has two receive events d and l. Let's first look at the interval I1, 3: message M5 is received first by event b, but this phenomenon will be reversed when message M5 were delayed and message M8 could instead be received first by event b. In this case, we call the two messages M5

and M8 are racing. Second, let's analyze the checkpointed interval I3,2: could the message M9 be received before the M6? The answer is no. Because the receive event d always happens before the send event k. We illustrate this case by using the executed dependence relation .

From Fig2, three relations can be found: d hb e, e hb f and f hb k. We can get the relation between the d and k: d hb k through the transitive attribute of the relation hb. So the message M6 always happens before the M9, and they are non-racing messages. But as to the first case, we cannot conclude the relation between the event b and c.

Definition 8: Two messages are racing messages iff

(1) a is a send operation and b is the corresponding receive operation of message Mi,

(2) c, d is the another sends and receive operations of message Mj;

(3) b and d are in the same checkpointed interval; but cannot conclude the executed dependence relation of the b and d.

3.3 Our Record Mechanism

Seen from above, we divide the messages into two parts: racing messages and non-racing ones. Non-racing messages cannot introduce non-determination and thus their deliveries need not be enforced during the replay phase. For each pair of racing messages, by forcing only one to be delivered to the appropriate receive, the other message will automatically be delivered to the correct operation.

Our record algorithm checks each message to determine if it races with the other, and traces only one of the racing messages. When each message received, the race check is performed by analyzing the executed dependence relation between the previous receive event and the message send event. We attach one event counter and one vector timestamp [13] per process. The event counter Ci is used to assign serial numbers to event in process pi, when a synchronization event (send and receive) takes place, the event counter Ci is incremental (+1). The vector timestamps are maintained so that at any point during execution, the ith slot of it (Timestamp[i] of pj) equals the current event counter Ci of the process pi that happened before the most recent event in process pj. To accomplish this, each process appends the current value of its timestamp onto sending messages and updates its timestamp with the receiving message's timestamp.

The record algorithm explained blow:

Before each send operation (the current process is process pi)

1. Ci =Ci +1
2. Timestamp[i] = Ci
3. attach <pi, Ci> and Timestamp onto the sending message

After each receive operation (the current process is process pj)

1. TimestampMsg = Timestamp of the receiving message
2. Cj =Cj +1
3. If Timestamp[j] > TimestampMsg[j] Then the message is racing message, trace the (<pi,Ci>,<pj,Cj>)
4. Timestamp[j]=max(Timestamp[j],TimestampMsg[j],Cj)Timestamp[i]=max(Timestamp[i], TimestampMsg[i])

Alga 1. Record algorithm

The on-line algorithm dynamically detects the racing message on the basis of the dependencies on past events of the execution that it introduces on the receiver process, Dependency information can be made dynamically available to processes by piggybacking vector timestamps into the sending messages.

4. Literature References

The message-passing concurrent programs are often long-running, so we introduce independent checkpointing strategy from fault tolerant systems, with which we can re-execute programs from any interested checkpoint instead of the beginning. The independent checkpointing divides each process into several independent checkpointed intervals, and we can select any of the checkpointed interval to replay. For example, when replaying the interval Ii,j, all the messages between the checkpoint Ci,j and Ci,j+1 must be regenerated again. To do this, one technique is to log all the received messages in the record phase, and then to read them from the trace files in the replay. Obviously, this technique is ineffective: logging all messages is impossible and costs a lot both in time and space. The other is re-executing the corresponded checkpointed intervals which send these messages in the original execution. Unfortunately, it will cause domino effect [12-13], and in the worst case, all the processes in the program may be forced to roll back to the beginning to re-compute the needed messages.

Our message logging protocol combines the advantages of the two techniques by only logging the special messages that could cause domino effect, and other messages being re-computed through re-execution of the corresponding checkpointed intervals. As the same with the detecting racing messages, we also on-line determinate whether a message needs to be logged, and this also based of the dependencies on the past events of the execution that introduces on the receiver process.

4.1 Domino Effect

Let us consider the execution in Fig3 and analyze how the domino effect happens. If we want to replay the checkpointed interval I1,3, two messages M5 and M8 must be reproduced. To re-compute the two messages, we could re-execute their sending interval. For message M5, we will re-execute the interval I2,2 that sends the message M5, and then, the checkpointed interval I2,2 receives the message M4, so we also should re-execute the interval I1,2, and so on. At last, we must re-execute the process p1 from its first checkpoint C1,1 y need to re-execute the checkpointed interval I3,2 and I2,3, but do not need to roll back the process p1 to the earlier interval.

Definition 9: A domino dependence between the checkpointed intervals db is defined as:

Ii,x db Ij,y iff Ii,x db Ij,y and i=j , x < y

Definition 10: The domino messages are the messages that introduce the domino dependence by introducing execution dependence from the earlier checkpointed interval to the later interval in the same process.

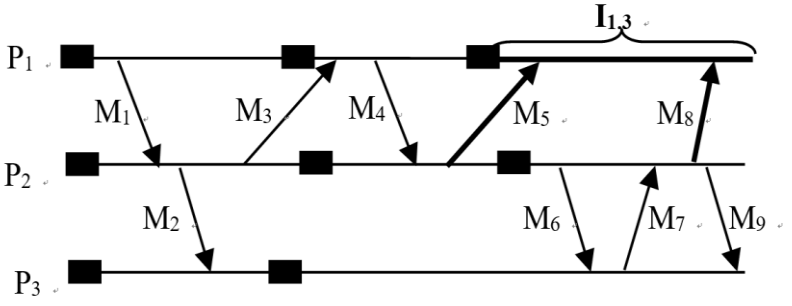

Fig 3. The domino messages

4.2 Detecting Domino Messages

Our algorithm detecting domino messages is similar to the algorithm of detecting racing message. It also dynamically traces the execution dependences among checkpointed intervals to determinate which messages cause a domino effect and only log their content. We must node that our algorithm is not optimal because that some special messages lie on more than one domino dependent path, and if these ones to be logged can break several domino dependences. However, the computation of the minimal number of domino messages is NP-hard.

Our algorithm to trace dependences, each process being kept count of its current state similar to a vector timestamp [14]. The checkpointed interval dependent vector (IDV) of the process pi contains the index of the earlier intervals in every other process on which process pi has interval dependence. Each process appends its IDV onto sending message, and upon receiving a message, updates its IDV. However, the most important is that after taking a checkpoint, a process reinitializes its IDV by nullifying all components except the one for itself, indicating that newly taken checkpoint is not executed dependent on any other event.

The domino algorithm is as follows:

After each checkpoint Ci, j (the current process is process pi)
IDV[i] = j
IDV[l] = null when l ≠i
Before each send operation
Attach IDV onto the sending message
After each receive operation (the current process is process pj)
1. IDVMsg = IDV of the receiving message
2. if IDV[j] > IDVMsg[j] (the sender dependes on some event in an earlier interval of pj) then log the message's content

Alga 2. Domino algorithm

5. The Replay Phase

We now outline how to replay an interest checkpointed interval. Compared to record algorithm, the replay phase is much more simply. Our adaptive trace algorithm guarantees that for any checkpointed interval, all messages on which it receives have either been logged or can be re-generated by re-executing the corresponding intervals. Therefore, the first problem needs to be tackled is how to determinate these corresponding checkpointed intervals in the replay phase. Our algorithm addressing this problem is to construct a dependent set for every checkpointed interval, and when replaying the special interval, we only need to re-execute the intervals of its dependent set.

The checkpointed interval relation db describes the dependent relation between the intervals. For instance, Ii,x db Ij,y indicates that there are events in Ij,y that are executed dependent on Ii,x.

Definition 11: for a given checkpointed interval Ii,j , the dependent set RS[Ii,j] = {Ix,y: Ix,y db Ii,j}

The interval dependences are introduced by the message deliveries in our system model. We can compute the interval dependent set after each receive operation takes place, the interval dependences set algorithm is as follows:

After each checkpoint Ci, j
RS [Ii,j] = { Ii, j}
Before each send operation
Attach the RS onto the sending message
After each receive operation (the Il, k is the current interval)
1. RSMsg = RS[Ii,j]
2. if message is not domino message then RS[Il.k] = RS[Il,k] ∪ RSMsg

Alga 3. Interval dependences set algorithm

IF the message is a domino message, it has been logged, and then it does not introduce any new dependency and does not require the update of the dependent set.

Once the dependent set RS has been maintained, we can replay any interested checkpointed interval. We can read the dependent set and have the corresponding intervals to re-execute concurrently. In the replay, eliminating the non-determination by the recorded information in the record phase is the most important. To effect replay faithful, we assign the event serial numbers (meaning the events' ordering). Our replay algorithm matches the racing messages with the correct receive during the replay, which have been original recorded in trace phase are modified to accept only the message whose serial number appears in the trace.

The replay algorithm is as follows:

For checkpointed interval Ii, l , read its RS and re-execute all the intervals of the RS[Ii, l]

Before each send operation (the current process is process pi)

1. Ci =Ci +1

2. if the record <<pi, ci>, <pj, cj>> is in the trace file

then set the message's type is an unique number

else set the message's type is 1

After each receive operation (the current process is process pj)

1. Cj =Cj +1

2. if the record <<pi, ci>, <pj, cj>> is in the trace file

then set the message's type is a unique number the same as the send operation

Alga 4. Replay algorithm

6. Conclusion

In this paper, we present a record and replay mechanism with checkpointing and message logging technique for message-passing programs. Our adaptive algorithm records the minimal amount of information that we have to trace in order to be able to provide an equivalent re-execution. We dynamically detect the racing messages, only record one order in each race to eliminate the non-determination. And at the same time, we determinate the domino messages, log their contents to break the domino effect; and construct the interval dependent set to compute which corresponding intervals should be re-executed for each checkpointed interval. In future work, more approaches can be adopted, such as predicating examination at the breakpoint and modeling checking of recorded event information, to further improve the efficiency.

References

[1] Marques E R B, Martins F, Ng N, et al. Protocol-based verification of message-passing parallel programs. ACM Sigplan International Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM, 2015, p. 280-298.

[2] Fu X, Chen Z, Yu H, et al. Symbolic execution of MPI programs. IEEE, International Symposium on High Assurance Systems Engineering, IEEE, 2015, p. 809-810.

[3] Tianyi Bao, William B. Gardner, Log Visualization Tool for Message-Passing Programming in Pilot. Parallel and Distributed Processing Symposium Workshops, IEEE, 2017 ,p. 331-338.

[4] S.Goldszmidt, Shaula Yemini, High-Level Language Debugging for Concurrent Programs. ACM Trans. on Computer Systems, Vol, 8(2015), p. 311-336.

[5] Volk M, Junges S, Katoen J P. Fast Dynamic Fault Tree Analysis by Model Checking Techniques. IEEE Transactions on Industrial Informatics, Vol. 14(2018), p.370-379.

[6] M.-Y. Park, S. J. Shim, Y.-K. Jun, and H.-R. Park, MPIRace-Check: Detection of message races in MPI programs. in Proc. of Advances in Grid and Pervasive Computing, Springer, 2007, p. 322–333.

[7] Irannejad M, Tchamgoue G M, Fischmeister S. A Reordering Framework for Testing Message-Passing Systems. IEEE, International Symposium on Real-Time Distributed Computing, IEEE, 2017, p. 109-116.
[8] E. Leu, A. Schiper, A. Zramdini, Execution Replay on Distributed Memory Architectures.IEEE Proceedings Second Symposium on Parallel and Distributed Processing, 2017, p. 106-112.
[9] F. Baiardi, N. De Francesco, G. Vagliani, Development of a Debugger for a Concurrent Language. IEEE Transaction on Software Engineering, Vol.12(2016), p. 547-553.
[10] C.Q. Adamsen, R. Karim, et al. Repairing event race errors by controlling nondeterminism. International Conference on Software Engineering , 2017, p. 289-299.
[11] D. Johnson, W. Zwaenepoel, Recovery in distributed systems using optimistic message logging and checkppointing, Journal of Algorithms, Vol. 11(2017), p. 462-491.
[12] Netzer R, Xu J. Adaptive message logging for incremental program replay. IEEE Parallel & Distributed Technology Systems & Applications, Vol. 1(1993), p. 32-39.
[13] Netzer R, Miller B. Optimal tracing and replay for debugging message-passing parallel programs. Kluwer Academic Publishers,1994.
[14]]Leslie Lamport. Time, Clock and the Ordering of Events in a Distributed System. Communications of the ACM, Vol21(1978), p. 558-565.
[15] Vetter J, Supinski B. Dynamic software testing of MPI applications with umpire.Supercomputing, ACM/IEEE 2000 Conference, IEEE, 2000, p. 51-51.
[16] K. Sen. Race directed random testing of concurrent programs. in Procc. of the Conf. on Programming Language Design and Implementation, ACM, 2008, p. 11–21.