Computer Life (CPL) ISSN: 1819-4818
DELIVERING QUALITY SCIENCE TO THE WORLD

Racing Message in Concurrent Programs

Yi Zeng

School of Computer and Electronic Information /School of Artificial Intelligence, Nanjing Normal University, Nanjing, China

Abstract: 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 have 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: Debugging, message-passing, checkpointing, message logging.

1. INTRODUCTION

Nowadays, with the improvement of user's demand of efficient software, concurrent technology is used widely. But, according to the added complexity communication and unintended interferences, the synchronization is not well taken care of by the developers the programs often exhibit non-deterministic behaviors [1]. Messages races are major sources of non-determinism in concurrent programs execution, which occurs when messages order of arrival at a process is not guaranteed by unpredictable scheduling decisions and variations in message latencies. Furthermore, sometimes programmers intentionally choose to allow a data race for better performance. Even though the controlled non-determinism might be important for matching the problems which are concurrent by nature or efficient concurrent programming, unintended non-determinism usually causes the program to behave in unexpected ways These bugs are caused by violation to programmer's order intentions, which may not be easily expressed via traditional debugging method. Even worse, the reason is that, a message race may indicate a concurrency bug, but it can also be a benign race in many cases. Different bug patterns usually demand different detection and diagnosis approaches. The programmers generally put their intentions on atomic regions and execution orders, but it is not easy to enforce all these intentions correctly and completely in implementation. It is also common for programmers to assume an order between two operations from different processes, but programmers may forget to enforce such an order. As a result, one of the two operations may be extracted faster or lower than the programmers' assumption, and it makes the order bug manifest.

2. MESSAGE RACING

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.

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.

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.

3. RECORD RACING MESSAGES

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 i^{th} slot of it (Timestamp[i] of p_j) equals the current event counter C_i of the process pi that happened before the most recent event in process p_j. 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:

After each checkpoint $C_{i,j}$ (the current process is process p_i)
IDV[i] = j
IDV[l] = null when $l \neq i$
Before each send operation
Attach IDV onto the sending message
After each receive operation(the current process is process p_j)
1. IDVMsg = IDV of the receiving message
2. if IDV[j] > IDVMsg[j] (the sender dependes on some event in an earlier interval of p_j)
then log the message's content
else IDV[j] = RdvMsg[j]

Fig 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. 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, p280-298.
[2] Fu X, Chen Z, Yu H, et al. Symbolic execution of MPI programs. IEEE, International Symposium on High Assurance Systems Engineering, IEEE, 2015, p809-810.
[3] Tianqi Bao, William B. Gardner, Log Visualization Tool for Message-Passing Programming in Pilot. Parallel and Distributed Processing Symposium Workshops, IEEE, 2017,p331-338.
[4] Volk M, Junges S, Katoen J P. Fast Dynamic Fault Tree Analysis by Model Checking Techniques. IEEE Transactions on Industrial Informatics, 2020, Vol. 14(18), p.370-379.
[5] E. Leu, A. Schiper, A. Zramdini, Execution Replay on Distributed Memory Architectures. IEEE Proceedings Second Symposium on Parallel and Distributed Processing, 2017, p106-112.
[6] Liu D.,Shen J.,Yang H. et al. Recognition and localization of actinidic arguta based on image recognition. Image Video Proc,2019, p19-201.
[7] Zerdoumi S.， Sabri A.Q.M.， Kamsin A. et al. Image pattern recognition in big data: taxonomy and open challenges: survey . Multimed Tools Appl,2018, p10091-10121.
[8] Kaur S.， Pandey S. Goel S. Plants Disease Identification and Classification Through Leaf Images: A Survey . Arch Computat Methods Eng,2019, p507-530.