Intention Preservation in Deterministic Multithreading

Wenbing Zhao
Department of Electrical and Computer Engineering
Cleveland State University, Cleveland, OH 44115
Email: wenbing@ieee.org

Abstract

In this paper, we present an analogy between the race condition resolution in a multithreaded application, and conflict resolution for concurrent updates in collaborative editing applications. We introduce the intention preservation problem in deterministic multithreading for programs with race conditions. We point out that current state-of-the-art deterministic multithreading approaches do not have any mechanism to preserve the intention of a program when it is instrumented for deterministic execution. Furthermore, we propose a solution for intention preservation in deterministic multithreading provided that the unprotected shared variables are primitive types.

Keywords

Deterministic multithreading, race condition, collaborative editing, intention preservation, conflict resolution

I. Introduction

Multithreaded applications, in particular Web based applications, are pervasively used to offer higher system throughput and lower end-to-end response time than single-threaded applications. Multithreading is also the way to go to exploit the availability of modern multi-core processors. However, multithreading also introduces a number of challenges:

- Multithreaded applications are significantly more difficult to develop compared with single-threaded applications due to the need for proper thread synchronization. Hence, multithreaded applications are more error-prone. Perhaps the most insidious issue with multithreaded applications is the race condition, where a shared variable is accessed by multiple threads concurrently without mutex protection.
- It is incredibly difficult to debug multithreaded applications because of the nondeterministic execution of such programs. In general it is impossible to enumerate all possible thread interleavings during testing. The nondeterminism intrinsic to multithreaded applications not only makes it hard to identify bugs during testing, it is hard to reproduce bugs reported.
- It is difficult to ensure strongly consistent replication of multithreaded applications for fault tolerance. State-machine replication requires that replicas be deterministic or rendered deterministic. Most existing proposed approaches to rendering multithreaded applications deterministic rely on capturing the thread interleavings on shared variables on one replica, and replay such interleaving on other replicas [1], [2], [3]. Such approaches are in general not practical because of the high runtime and communication overhead.

In the past several years, deterministic multithreading, which aims to enable reproducible deterministic execution of multithread applications, has attracted intense research [4], [5], [6] because it offers a promising new way to alleviate most of the above problems with multithreading:

- Because the thread interleavings on access of shared variables are made deterministic, debugging of multithreaded applications becomes much easier.
- Some deterministic multithreading approaches that offer strong determinism could even render race conditions deterministic. This would help debug race conditions.
- Strongly consistent replication may become possible for multithreaded applications without the need of recording, communicating, and replaying thread interleavings.

In this paper, we highlight the issue on how race conditions should be handled for deterministic threading. We argue that the current approaches used to handle race conditions are less desirable and could be enhanced such that the application developer's intention is preserved.

II. Race Condition and Collaborative Editing

We find that the problem of reconciling concurrent updates by different threads to the same shared variable is rather similar to the problem of reconciling concurrent updates to a shared document in collaborative editing applications [7], [8]. A common approach to achieving determinism in the presence of race conditions is to isolate different threads and updates to a shared variable are first applied to a private copy [5], [6]. Similarly, many collaborative editing applications allow users to update a private copy of the shared document without seeking prior approval from other users. Both scenarios face the challenge of reconciling updates made by different threads/users (initially to private copies) to the shared entity (a variable or a document) properly.

In deterministic multithreading, a common mechanism for deterministic conflict reconciliation is to impose a logical order on updates to a shared variable, and to propagate the update from one thread to all other threads. Furthermore, a later

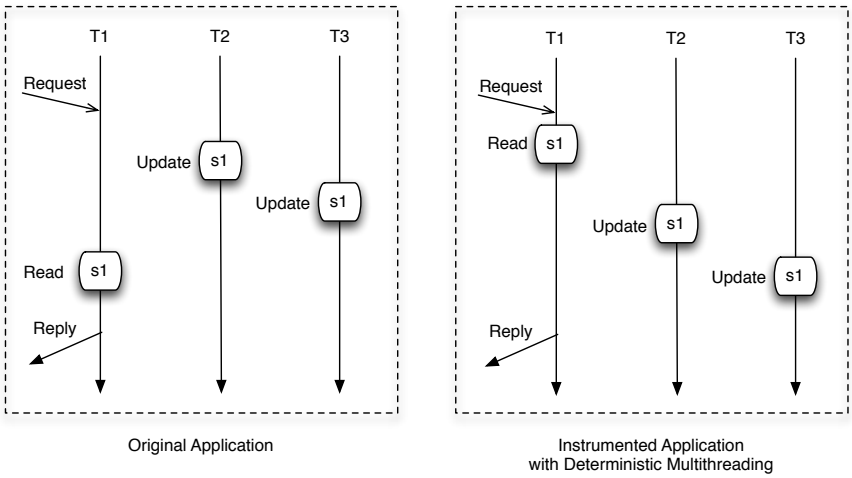

Figure 1. An example of intention violation due to deterministic multithreading. The left shows a possible run of the original multithreaded application where $T1$ would read $s1$ *after* $T2$ and $T3$ updated $s1$. The right shows what happens if the application is instrumented with a deterministic multithreading library where $T1$ would be forced to read $s1$ effectively prior to the updates on $s1$ from $T2$ and $T3$, which violates the program's intention.

update to the same shared variable would overwrite an earlier update (*i.e.,* the-last-write-wins strategy). Even though the same mechanism can be used to reconcile concurrent updates in collaborative editing applications as well, it is apparently problematic because it is against the objective of collaborative applications, which is to facilitate concurrent contribution to a shared document from multiple users. If one user's update is overwritten by another user's update, the former's contribution would be completely lost.

Hence, more sophisticated mechanisms have been used to reconcile concurrent updates to a shared document in collaborative editing applications. Such mechanisms are often based on operational transformation [9]. An important correctness criteria for any operational transformation mechanism is intention preservation [10], *i.e.,* the users intentions for their updates to the shared document must be preserved.

In light of this observation, we argue that the current mechanism to ensure deterministic execution of programs with race conditions, although serving the purpose of ensuring determinism, is insufficient to preserve the intention of the program. Of course, one may counter our claim in that race conditions are programming errors and instead of preserving the intention of the original developers in this case, it is more desirable to identify such errors and correct them. Indeed, some deterministic multithreading implementation [6] can be used as a fail fast tool for developers because it would help them realize that their intention is violated[1].

However, developing multithreaded application is a difficult task and even experienced software developers might make mistakes by forgetting to protect shared variables with mutexes, or fail to realize that data structures provided by some third party is not thread-safe. Furthermore, fixing race conditions may not always be straightforward, especially in the presence of thread-unsafe data structures in legacy or third party libraries.

Even though it is useful for a deterministic multithreading library to help reveal race condition bugs, wouldn't it be more desirable for a deterministic multithreading library to not only render the execution of a program with race conditions deterministic, but also obey the intention of the original developers? In effect, such a library would possess the capability of automatically fixing any race condition in the program during runtime without the intervention of software developers, which would reduce the cost of software development and also increase the robustness of multithreaded applications.

III. Intention Violation in Deterministic Multithreading

The mechanisms used to render race conditions deterministic in deterministic multithreading [5], [6], which we outlined in the previous section, completely ignore read access to shared variables. Because read and write to the same variable conflict, this could cause serious causality problems because once a thread reads the shared entity, it may subsequently update other local or shared variables based on what it has read.

To understand the intention violation issue better, consider the example shown in Figure 1. Note that this example only shows one of many potential ways of intention violation. We assume that a server application has 3 threads $T1$, $T2$, and $T3$, and a shared variable $s1$. Unfortunately, $s1$ is not protected by a mutex, hence, all accesses including read and write to it are not synchronized (*i.e.,* we have a race condition).

[1]Private communication with Dr. Xu Zhou, one of the authors in [6].

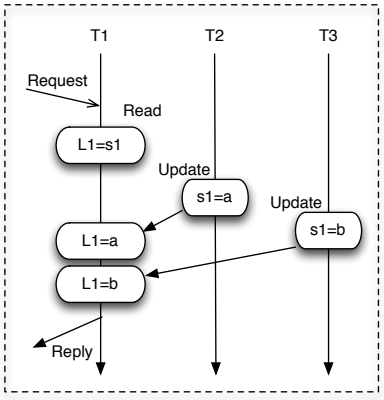

Figure 2. An example scenario for intention preservation.

Upon receiving a request from a client, all 3 threads are notified about the event. $T2$ and $T3$ would update $s1$ concurrently. $T1$ would first do something locally or write to some other shared variables before it would read $s1$. So, there is a physical time lag between updates to $s1$ by $T2$ and $T3$, and the read from $s1$ by $T1$, but without proper thread synchronization. Subsequently, $T1$ generates a reply to the client based on the value read from $s1$. The developer's intention is to let $T2$ and $T3$ perform some computation that involves with $s1$ in response to the request, and let $T1$ obtain the outcome of the computation by reading from $s1$.

If the application is instrumented with a deterministic multithreading library such as [5], [6], $T1$ would read an old value of $s1$ prior to updates from $T2$, or $T3$. This would violate the intention of the program.

IV. A Possible Solution for Intention Preservation

We propose a simple solution for intention preservation in deterministic multithreading provided that each unprotected shared variable is of primitive types such as integer. Much more sophisticated mechanisms are needed to preserve intention if the shared variable is of complex types such as string.

During execution, the runtime must track not only write operations to shared variables, but also read operations. When a thread performs a read operation on a shared variable in isolation, an entry is added to a log containing the shared variable, all other variables that are involved, and the operations on these variables. When an update to the shared variable is propagated from one thread to another, the log is examined to see if a read on the shared variable has been performed. If that is the case, the logged operation would be redone and all relevant variables are updated as a result. Note that if there exist further propagations, all such operations would have to be redone. Figure 2 shows a simple case for the proposed solution.

V. Conclusion

In this paper, we presented an interesting analogy between the race condition resolution in a multithreaded application, and conflict resolution for concurrent updates in collaborative editing applications. We introduced the intention preservation problem in deterministic multithreading for programs with race conditions. We pointed out that current state-of-the-art deterministic multithreading approaches do not have any mechanism to preserve the intention of a program when it is instrumented for deterministic execution. Furthermore, we proposed a simple solution for intention preservation in deterministic multithreading provided that the unprotected shared variables are primitive types. Further research is needed to deal with complex shared variables.

Acknowledgements

This work was supported in part by a Graduate Faculty Travel Award from Cleveland State University.

References

[1] W. Zhao, P. M. Melliar-Smith, and L. E. Moser, "Low latency fault tolerance system," *The Computer Journal*, vol. 56, no. 6, pp. 716–740, 2013.

[2] W. Zhao, "Byzantine fault tolerance for nondeterministic applications," in *Proceedings of the 3rd IEEE International Symposium on Dependable, Autonomic and Secure Computing*, Loyola College Graduate Center, Columbia, MD, USA, September 2007, pp. 108–115.

[3] H. Zhang, W. Zhao, P. M. Melliar-Smith, and L. E. Moser, "Design and implementation of a Byzantine fault tolerance framework for non-deterministic applications," *IET Software*, vol. 5, pp. 342–356, 2011.

[4] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble, "Deterministic process groups in dos." in *OSDI*, vol. 10, 2010, pp. 177–192.

[5] T. Liu, C. Curtsinger, and E. D. Berger, "Dthreads: efficient deterministic multithreading," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 327–336.

[6] K. Lu, X. Zhou, T. Bergan, and X. Wang, "Efficient deterministic multithreading without global barriers," in *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2014, pp. 287–300.

[7] W. Zhao, "Concurrency control in real-time e-collaboration systems," in *Encyclopedia of E-Collaboration*, N. Kock, Ed. Idea Group Publishing, 2008, pp. 95–101.

[8] W. Zhao and M. Babi, "Byzantine fault tolerant collaborative editing," in *Proceedings of the IET International Conference on Information and Communications Technologies*. IET, 2013, pp. 233–240.

[9] C. A. Ellis and S. J. Gibbs, "Concurrency control in groupware systems," in *ACM SIGMOD Record*, vol. 18, no. 2. ACM, 1989, pp. 399–407.

[10] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen, "Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 5, no. 1, pp. 63–108, 1998.