Efficient Data Race Detection for Structured Fork-join Parallelism*

Ok-Kyoon Ha and Yong-Kee Jun**

Department of Informatics,
Gyeongsang National University,
Jinju 660-701, Repblic of Korea
{jassmin,jun}@gnu.ac.kr

Abstract. Data races in parallel programs represent the most notorious class of concurrency bugs. It is still difficult and cumbersome to locate when a program runs into data races, because they may lead to unpredictable results of the program. To detect data races occurred during an execution of parallel programs, previous work provides large runtime and space overhead or focuses on reducing false positives. Thus, the previous detectors are still imprecise and inefficient, when applied to large scale parallel programs which use a structured fork-join parallelism with a large number of threads. This paper presents an efficient data race detection algorithm which analyze conflicting accesses to every shared memory location. The detection algorithm precisely reports data races because it guarantees to locate at least one data race for each shared memory location, if there exists any. Moreover, the technique provides a significant improvement of efficiency as $O(1)$ space and time overheads for each access history.

Key words: data races, data race detection, detection algorithm, structured fork-join parallelism, efficiency, preciseness

1 Introduction

The structured fork-join parallelism [6, 8] such as OpenMP is a model of parallel thread. This model requires at least one set of fork operations which create a set of concurrent threads, and a corresponding join operation which terminates the forked threads and spawns a single thread. The structured fork-join parallelism makes it easier to identify accesses to shared memory location on parallel threads and to produce a parallel program that is more robust and efficient. This can contain nested parallelism or series parallelism with other fork-join constructs,

* "This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology(2011-0026340) "

** Corresponding author: In Gyeongsang National University, he is now the director of GNU Embedded Software Center for Avionics (GESCA) which is a national IT Research Center (ITRC) of Republic of Korea.

where nested parallelism is a nestable fork-join model of parallel execution in which a parallel thread can generate a team of other parallel threads.

Data races [1] in parallel programs is the most notorious class of concurrency bugs that cause non-atomic execution of critical section or harmful conflicting accesses to a shared memory location with at least one of them being a write. A parallel program may not exhibit the same execution instance with the same input due to the fact that the execution order of threads is timing-dependent. It is still difficult and cumbersome to locate when a program runs into data races, because they may lead to unpredictable results of the program.

On-the-fly methods dynamically detect data races with still less overhead in storage space than other dynamic techniques due to the fact that unnecessary information are removed as the detection advances. On-the-fly methods generally use either the *happens-before analysis* [2, 6] which is inefficient due to the additional runtime overhead, or the *lockset analysis* [7, 9] which is imprecise due to the amount of reported false positives. Happens-before analysis which based on uses a logical time stamp[3, 5] and a protocol for race detection. This technique reports data races between current access and maintained previous accesses by comparing their happens-before relation. Lockset analysis reports races of monitored program by checking violations of a locking discipline. This technique is simple and can be implemented with low overhead. However, lockset analysis technique may lead to many false positives.

There is the trade off between the efficiency and the preciseness in both on-the-fly techniques, happens-before and lockset analysis. *Hybrid analysis* [4, 10] combines the happens-before and the lockset analysis to obtain both improved preciseness and performance. Most hybrid detectors improved accuracy on reporting data races by partially applying the happens-before analysis on lockset analysis, but still report false positives. We believe that detecting data races for debugging parallel programs requires a precise and efficient technique which locates at least one data race, if there exists any. This work addresses to offer practically precise on-the-fly data race detection.

2 Efficient Data Race Detection

The idea of our data race detection algorithm is originated by the Dinning and Schonberg's protocol [2]. This protocol defines an access history considering thread locking mechanisms and maintains it to report data races on every conflicting access to a shared memory location. In the protocol, an access history consists of a *Read set* ($\mathbb{R}$), a *Write set* ($\mathbb{W}$), a *CS-read set* ($\mathbb{CR}$), and a *CS-write set* ($\mathbb{CW}$), where CS stands for critical section. The $\mathbb{R}$ and $\mathbb{W}$ are for events outside critical sections, and the $\mathbb{CR}$ and $\mathbb{CW}$ are for events inside critical sections. The detection protocol precisely reports data races by guaranteeing the detection of at least one apparent data race for each shared memory location, if there exists any. However, the protocol requires large space and time overheads which are proportional to the maximum parallelism T for each entry set of the access histories in the worst case, because it maintains all earlier events in $\mathbb{R}$, $\mathbb{CR}$, and

$\mathbb{CW}$ until a new write event is recorded in $\mathbb{W}$. Thus, we efficiently improve the protocol to solve its overhead problem.

Our efficient detection protocol (EDP) reports data races in constant amount of space and time even in the worst case by maintaining only two concurrent events in $\mathbb{R}$, $\mathbb{CR}$, and $\mathbb{CW}$ using the *left-of-relation*. The original left-of-relation [6] is a relative notion about two parallel thread segments that always maintain two concurrent read events to detect at least one data race with a write event. For our EDP, we additionally apply the left-of-relation to CS-read and CS-write events.

The left-of-relation provides positive efficiency for on-the-fly data race detection, however, the efficiency of EDP can be firmly established by a filtering method which ignores repeated accesses to a shared memory location. With the filtering method, EDP considers only the first events of each event type in a thread segment, if the events are performed with redundant lockset. Given an event e_i, a later same type event e_j than e_i is filtered out by the following conditions:

$$\begin{aligned} \mathit{IsFiltered}(e_i, e_j) = \{(\mathfrak{T}(e_i) = \mathfrak{T}(e_j)) \wedge (T_i = T_j)\} \quad \wedge \\ (\mathit{lockset}(e_i) \supseteq \mathit{lockset}(e_j)) \end{aligned} \tag{1}$$

where $\mathit{lockset}(e_i)$ represents a set of locks living on a thread segment for event e_i, and $\mathfrak{T}(e_i)$ means the event type of e_i.

In order to report data races or update the access history, our EDP analyzes the happens-before relation between a current event and an earlier event kept in an access history, and checks violations of a locking principle between them. If we observe distinct events e_i and e_j, a data race is reported by the following conditions:

$$\begin{aligned} \mathit{IsRace}(e_i, e_j) = \{(\mathfrak{T}(e_i) = Write \vee \mathfrak{T}(e_j) = Write) \wedge (T_i \parallel T_j)\} \quad \wedge \\ (\mathit{lockset}(e_i) \cap \mathit{lockset}(e_j) = \phi) \end{aligned} \tag{2}$$

For reporting data races, EDP considers only $\mathbb{W}$ and $\mathbb{CW}$ for e_j which is an earlier event kept in an event history for a shared memory location, if $\mathfrak{T}(e_i)$ is Read. Otherwise, it considers all entries of the access history.

Our EDP is practically precise for data race detection in large scale parallel programs, because it guarantees to locate at least one data race for each shared memory location and offers epochally reduced time and space overhead by applying the left-of relation and the filtering method. Thus, our technique is naturally efficient and precise due to the fact that the EDP precisely reports data races without any false positives while providing a significant improvement of efficiency as $O(1)$ time and space overheads for each access history.

3 Conclusion

Data races in parallel programs can be occurred when two parallel threads access a shard memory location without proper inter-thread coordination, and at least

one of these accesses is a write. Race detection is important work for debugging of the parallel programs, but it is still difficult and cumbersome. The earlier data race detector is still imprecise with non-remarkable efficiency for large scale programs that use a structured fork-join parallelism with a large number of threads. In this paper, we presented an efficient detection algorithm to analyze conflicting accesses to every shared memory location. Our data race detection technique precisely reports data races without any false positives, while providing a significant improvement of efficiency as $O(1)$ space and time overheads for each access history.

References

1. Banerjee, U., Bliss, B., Ma, Z., Petersen, P.: A theory of data race detection. In: Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging. pp. 69–78. PADTAD '06, ACM, New York, USA (2006)
2. Dinning, A., Schonberg, E.: Detecting access anomalies in programs with critical sections. In: Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging. pp. 85–96. PADD '91, ACM, New York, NY, USA (1991)
3. Ha, O., Jun, Y.: Efficient thread labeling for on-the-fly race detection of programs with nested parallelism. In: Software Engineering, Business Continuity, and Education, Communications in Computer and Information Science, vol. 257, pp. 424–436. Springer Berlin Heidelberg (2011)
4. Jannesari, A., Kaibin, B., Pankratius, V., Tichy, W.F.: Helgrind+: An efficient dynamic race detector. In: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing. pp. 1–13. IPDPS '09, IEEE Computer Society, Washington, DC, USA (2009)
5. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21, 558–565 (July 1978)
6. Mellor-Crummey, J.: On-the-fly detection of data races for programs with nested fork-join parallelism. In: Proceedings of the 1991 ACM/IEEE conference on Supercomputing. pp. 24–33. Supercomputing '91, ACM, New York, USA (1991)
7. Nishiyama, H.: Detecting data races using dynamic escape analysis based on read barrier. In: Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium - Volume 3. pp. 10–10. USENIX Association, Berkeley, CA, USA (2004)
8. Raman, R., Zhao, J., Sarkar, V., Vechev, M., Yahav, E.: Efficient data race detection for async-finish parallelism. In: Proceedings of the First international conference on Runtime verification. pp. 368–383. RV'10, Springer-Verlag, Berlin, Heidelberg (2010)
9. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: a dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. 15, 391–411 (November 1997)
10. Yu, Y., Rodeheffer, T., Chen, W.: Racetrack: efficient detection of data race conditions via adaptive tracking. In: Proceedings of the twentieth ACM symposium on Operating systems principles. pp. 221–234. SOSP '05, ACM, New York, NY, USA (2005)