Development of Technique for Healing Data Races based on Software Transactional Memory

Eu-Teum Choi¹, Kun Su Yoon², Ok-Kyoon Ha³, Yong-Kee Jun¹

¹ Department of Informatics, Gyeongsang National University, Jinju, 52828, Republic of Korea
² Aviation Campus of Korea Polytechnic, Sacheon, 52549, Republic of Korea
³ Department of Aeronautics & Software Engineering, Kyungwoon University, Gumi, 39160, Republic of Korea
etchoi@gnu.ac.kr, unikyoon@kopo.ac.kr, okha@ikw.ac.kr, jun@gnu.ac.kr

Abstract. Data races in multi-threaded programs may occur when multiple accesses on different threads access a shared location without proper synchronization, and one of them is a store. It is difficult to develop data race free programs and to manually fix existing data races in the programs, because they may lead to unpredictable results to the programmer. This paper presents a technique that heals data races using software transactional memory. This technique consists of Transactional Region Specifier, which localizes the incorrect code blocks including shared variables, and Instrumentor, which inserts transactional memory codes to heal data races. We evaluate the accuracy of our healing technique using a set of synthetic programs.

Keywords: data races, software transactional memory, multi-threaded programs

1 Introduction

Data races [1, 2] in multi-threaded programs is a kind of concurrency bugs that may occur when two concurrent threads access a shared location without proper synchronization, and at least one of these accesses is a store. It is difficult to develop data race free programs due to a complex interaction among threads, and to manually fix existing data races in the programs. Data race detection techniques [3-6], which use either static analysis or dynamic analysis, have some limitations. The static analysis reports many false alarms, and the dynamic analysis requires heavy additional overheads for detecting data races.

This paper presents a technique that heals data races using software transactional memory (STM) [13, 14] during an execution of a target program which is implemented using multi-threaded APIs, such as Pthread. This technique consists of two modules, Transactional Region Specifier (TRS) and Instrumentor (INS). TRS module localizes the incorrect code blocks considering shared variables, such as iteration statement and conditional statement. INS module inserts transactional memory codes to heal data races occurred during an execution of the target program.
To evaluate the accuracy of our technique, we developed a set of synthetic programs that considers iteration statements and conditional statements, and shared variables.

2 Background

2.1 Techniques for Healing Data Races

In multi-threaded programs, data races is one of the common and the dangerous concurrency bugs due to the non-deterministic results of the program executions. Previous work has presented several approaches to develop and manually fix data races. However, these detecting approaches are inefficient because static analysis reports many false alarms, and dynamic analysis needs additional overheads for detecting data races. Thus, recent works have been introduced the techniques of healing data races rather than detecting the bugs.

Existing techniques that heal data races [7-11] can be categorized into three approaches: always-on, failure-recovery, and post-mortem. Always-on approach constrains program execution all the time to prevent potential manifestations of some concurrency bugs [10]. Failure-recovery approach rolls back program execution to a recent checkpoint when failures or errors occur, and relies on re-execution for automated recovery. Post-mortem approach aims to prevent future manifestations of a concurrency bug, after triaging earlier manifestations of the bug. Our healing technique that based on STM belongs to the failure-recovery approach.

2.2 Software Transactional Memory

The basic idea of transactional memory is simple. The properties of transactions provide a convenient abstraction for coordinating concurrency reads and writes of shared variables in the multi-threaded programs. Generally, a simple transactional memory interface is comprised operation for managing transactions and performing memory accesses.

STM is a concurrency synchronization mechanism only by software approach without hardware support. Unlike lock mechanism, STM is non-blocking characteristic that can perform by simultaneous approach of several threads. Therefore, STM is free from concurrency bugs, such as data races, deadlocks, and priority inversion.

STM provides a set of operations for managing transactions, such as StartTX, CommitTX, and AbortTX. StartTX operation begins a new transaction in the current thread. CommitTX attempts to commit the current transaction. If there is no conflict, the operation returns a true value and continuously executes the program, whereas it returns a false value and aborts the current transaction of the program. AbortTX is used to trigger an abort or to re-execute failed transactions.

Also, STM provides a set of operations for data access, such as ReadTX and WriteTX. ReadTX takes the address of the value of a variable and returns the
transaction’s view of the data at that address. WriteTX takes an address of the value of a variable and a new value of the variable, writing the value to the transaction’s view of that address.

![Diagram of the Overall Architecture of healing data races based on STM.](image)

**Fig. 1.** The Overall Architecture of healing data races based on STM.

### 3 Our Technique for Healing Data Races

To heal data races using transactional memory, we need to insert proper transactional regions and the codes of STM on target programs considering code blocks and shared variables. Therefore, our technique for healing data races consists of two modules: Transactional Region Specifier (TRS), which collects the information of target program execution considering iteration statement, conditional statement, and shared variables, and Instrumentor (INS), which inserts transactional memory codes into specified transactional regions of target program by TRS module. Fig. 1 shows the overall architecture of our technique for healing data races.

During an execution of a multi-threaded program, TRS module localizes incorrect code blocks considering iteration statements, conditional statements, and shared variables. The module collects the information of shared variables in the target program. TRS classifies collected shared variables into an incorrect synchronization set if the shared variables are not protected a same lock. Otherwise, it classifies a correct synchronization set. The TRS also collects the information of control flow in the target program, such as iteration statements and conditional statements, considering the incorrect synchronization set.

INS module dynamically inserts the managing operations of STM, such as StartTX, CommitTX, and AbortTX, to specified transactional regions of the target program. If a shared variable of the incorrect synchronization set is located in the region of an
iteration statement, StartTX operation is inserted into immediately before the start point of the iteration statement, and CommitTX and AbortTX also are inserted into immediately after the end point of the region. If a shared variable is placed in a conditional block with a conditional statement, StartTX is inserted into before a statement which includes the shared variable, and both CommitTX and AbortTX are also inserted into after the statement, respectively. Finally, INS applies ReadTX and WriteTX considering the accesses, such as load or store, to the shared variable.

![Fig. 2. The Example of Synthetic Programs.](image)

### 4 Evaluation

To evaluate the accuracy of our technique, we used synthetic programs considering either involving data races or not. We developed the synthetic programs considering three criteria such as iteration statement, conditional statement, and none of them. Fig. 2 graphically shows the execution of synthetic programs.

We implemented our technique as a Pin-tool on top of the Pin binary instrumentation framework [12] which uses a just-in-time (JIT) compiler to recompile target program binaries for dynamic instrumentation. For STM, we introduced TinySTM, version 1.0.5.

Our implementation and experimentation were carried on a system with Intel Xeon Processor E5620 (4 core and 8 threads) and 48GB main memory under the kernel 2.6 of Linux operating system. The syntheses were compiled with GCC compiler 4.1.2.

The results of healing data races under the synthetic programs appear in Table 1. From the table, the original runs for R01 and R02 incurred data races related to control flow, such as iteration statements and conditional statements, whereas no data races were occurred during the execution of the syntheses, R01 and R02, with our technique using STM. As the results, our technique shows that data races effectively heal to the multi-threaded programs. So, it is useful for the programs that are needed data race free execution.
Table 1. The Result of Experimentation.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Original run of synthetic programs</th>
<th>Run of synthetic programs with our technique</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R01</td>
<td>R02</td>
</tr>
<tr>
<td>None</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Iteration</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Conditional</td>
<td>○</td>
<td>○</td>
</tr>
</tbody>
</table>

4 Conclusion

This paper presented a technique that consists of Transactional Region Specifier and Instrumentor to heal data races in multi-threaded programs using STM. We evaluated the accuracy of our healing technique using a set of synthetic programs that considers iteration statements and conditional statements, and shared variables. The technique shows that data races effectively heal to the multi-threaded programs. In the future works, we will verify the effectivity of our technique with the variety of data race patterns using real-world programs which are widely used in several open source projects.

Acknowledgments. This research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2014R1A1A2060082) and also was supported by the Agency for Defense Development, South Korea, under Grant (UD160014DD).

References


