Advanced Science and Technology Letters
Vol.76 (CA 2014), pp.45-49
http://dx.doi.org/10.14257/astl.2014.76.12

Dynamic Monitoring Tool based on Vector Clocks for Multithread Programs

Hyun-Ji Kim[1], Byoung-Kwi Lee[2], Ok-Kyoon Ha[3], and Yong-Kee Jun[1]

[1] Department of Informatics, Gyeongsang National University, 501, Jinjudea-ro, Jinju, Republic of Korea

[2] POSCO ICT Company Ltd., 68, Hodong-ro, Nam-gu, Pohang, Republic of Korea

[3] Engineering Research Institute, Gyeongsang National University, 501, Jinjdea-ro, Jinju, Republic of Korea
hecho3927@gmail.com, bk.lee@poscoict.com, {jassmin, jun}@gnu.ac.kr

Abstract. Monitoring program execution to collect the information of program executions, such as the occurrence of concurrent threads, memory accesses, and thread synchronization, are important to locate data races for debugging multithread programs. This paper presents an efficient and practical monitoring tool, called VcTrace that analyzes partial ordering of concurrent threads during an execution of the program based on vector clock system, and VcTrace is applied to detect data races for debugging real applications using multithreads. Empirical results on C/C++ benchmarks using multithreads show that VcTrace is a sound and practical tool for on-the-fly data race detection as well as analyzing multithread programs.

Keywords: Multithreads, data races, debugging, dynamic monitoring.

1 Introduction

It is well known that data races [1,2] are the hardest defect to handle in multithread programs because of their non-deterministic interleaving of concurrent threads. The analysis techniques for debugging data races generally use one of static and dynamic techniques [3,4]. Static techniques analyze the program information to report data races from source codes without any execution, while dynamic techniques locate data races from execution information of the program. Static techniques are sound, but imprecise since they report too many false positives. Dynamic techniques are precise and provide higher reliability of analysis than static techniques, but these techniques are unsound and inefficient because they require additional runtime overhead to monitor the information of program executions, such as the occurrence of concurrent threads, the accesses to shared memory locations, and thread synchronization. Dynamic techniques employ trace based post-mortem methods or on-the-fly methods, which report data races occurred in an execution of a programs. Post-mortem methods analyze the traced information or re-execute the program after an execution. On-the-

ISSN: 2287-1233 ASTL

fly methods are based on three different analysis methods: lockset analysis, happens-before analysis, and hybrid analysis.

2 Background

For detecting data races during an execution of multithread programs, it is important to identify the partial order relation of two accesses (threads) by analyzing the Happens-before Relation [5]. The partial order relation for two accesses, e_t and e_u on thread t and u respectively, can be identified as following:

1. If e_t must happen at an earlier time than e_u, e_t happens before e_u, denoted by $e_t \rightarrow e_u$.
2. If $e_t \rightarrow e_u$ or $e_u \rightarrow e_t$ is not satisfied, we say that e_t is concurrent with e_u, denoted by $e_t \parallel e_u$.

Vector clock (VC) [6] is a representation of the Happens-before relation to identify the partial order relation of concurrent threads considering thread operations and synchronization operations, such as fork-join operation. A VC is a set of clock value c for each thread while the program is executing. Thus, a VC C_t for thread t represents $<c_1, \ldots, c_n>$, where n designates the maximum number of simultaneously active threads during an execution.

Using the VCs of each thread, we simply analyze the partial order relation between any two threads. If the clock value for a thread t, denoted by $C_t[t]$, is less or equal to $C_t[u]$ which is the corresponding clock value of another thread u, ($\equiv \underline{iff}\ C_t[t] \leq C_t[u]$), we can analyze $t \rightarrow u$. Otherwise, $C_t[t] > C_u[t]$ or $C_t[u] < C_u[u]$, $t \parallel u$.

Fig. 1 represents a multithread execution using a direct acyclic graph (DAG), and the VCs are allocated for each thread by the above partial order relation. In Fig. 1, we can easily analyze that a read r_1 on a thread t_1 happens before a read r_3 on t_0, $r_1 \rightarrow r_3$, because their VCs are satisfied ($C_{t1}[t_0] = 0$) $\leq$ ($C_{t0}[t_0] = 2$) by a synchronization between t_1 and t_0. On the other hand, we know that r_4 on t_1 is concurrent with r_5 on t_2, $r_4 \parallel r_5$, due to the fact that their VCs are satisfied ($C_{t1}[t_1] = 3$) > ($C_{t2}[t_1] = 2$).

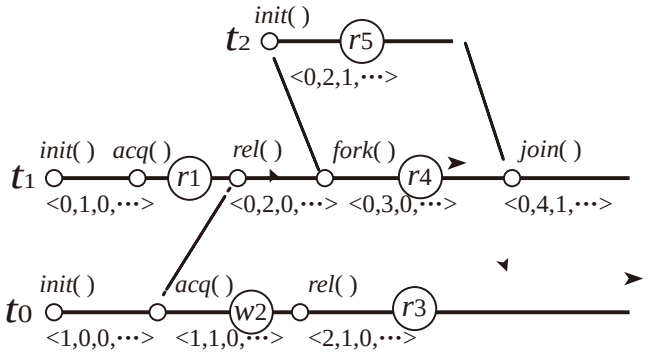

Fig. 1. An example of DAG for an execution of multiple threads

3 Design of Dynamic Monitoring Tool

To locate data races during an execution of multithread programs, we must monitor important operations, such as thread operations and synchronization operations, and provide thread identification, such as VCs, which maintains logical concurrency information of multiple threads for the partial order relation. This paper presents a dynamic monitoring tool, called *VcTrace* that analyzes partial ordering of concurrent threads and their accesses to the shared memory locations during an execution of the program based on vector clock system, and applies it to detect data races for debugging applications.

The tool considers following thread operations: *init*( ) for a thread initialization, *fork*( ) for the creation of a child thread, and *join*( ) which terminates forked child threads and spawns a single thread. It also collects the information of thread synchronization considering *acq_lock*( ) and *rel_lock*( ) used to provide the mutual exclusion of concurrent threads, *wait*( ) and *signal*( ) for the internal interleaving of concurrent threads, and *barrier*( ) for the waiting of multiple threads until the threads satisfy a waiting condition. Additionally, the tool monitors the thread accesses (*read/write*) to the shared memory locations.

We design VcTrace to monitor these thread and synchronization operations during programs executions based on a dynamic binary instrumentation framework, called PIN [7]. The Tool requires no source code annotation to monitor an execution of the programs, because it uses a just-in-time (JIT) compiler to recompile target program binaries for dynamic instrumentation. Fig. 2 depicts the overall architecture of VcTrace.

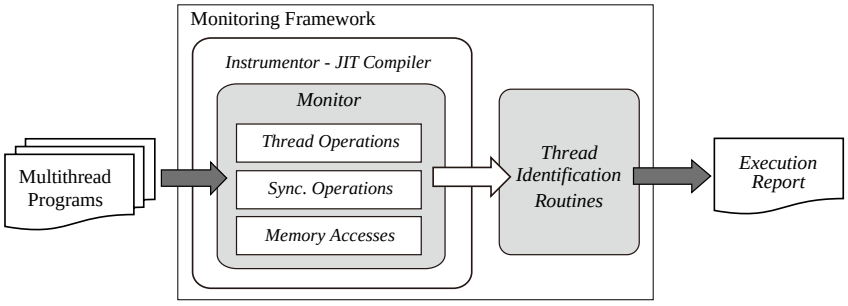

Fig. 2. The overall architecture of VcTrace

The tool consists of three monitoring modules which monitor thread operations, synchronization operations, and the memory accesses of threads, and the thread identification routines which create and maintain VCs to analyze the partial order relation of concurrent threads. To offer the correct identification of concurrent threads, VcTrace uses System Thread Id, Pthread Id, and PIN Thread Id for each thread. The System Thread Id is the thread id allocated by the operating system, and the Pthread Id is a identifier allocated by the pthread functions such as *pthread_create*( ). The PIN Thread Id is the logical identifier created whenever the Monitor catches a thread start operation, such as *init*( ) and *fork*( ). Finally, VcTrace provides comprehensive

information which is possible to analyze dynamically data races in an execution only using the binary codes of multithread programs.

4 Experimentation

We implemented VcTrace using C/C++ on top of PIN software framework and evaluated the efficiency of the tool for data race. The implementation and experimentation was carried on a system with Intel Xeon Quard-core 2 CPUs and 48GB main memory under Linux operating system (kernel 2.6). The FastTrack algorithm was connected to VcTrace to detect data races. We employed three benchmarks using multithreads, MySql (an open source DBMS), Cherokee (an open source Web Server application), and X.264 (an open library for encoding video streams), and these applications were executed five times to measure the runtime and memory overheads.

Table 1 shows measured results of average runtime and memory consumption for the three benchmarks under VcTrace. For the experiments, X.264 used 64 multiple threads during an execution, and 70 threads was used for MySql and Cherokee. In the table, “Origin” means the original execution without VcTrace, and “Monitor” means measured results on monitoring phase only by VcTrace. “Detect” column indicates the overheads on VcTrace connected with data race detection, and “Races” column indicates the number of located data races under the detection phase.

Table 1. Measured runtime and memory overhead results for three benchmarks

Benchmarks	Time (Sec)			Memory (MB)			Races
	Origin	Monitor	Detect	Origin	Monitor	Detect	
X.264 (64)	0.4	2	12.4	30	228	1,134	3
MySql (70)	60	60	60	337	686	1,244	13
Cherokee (70)	60	60	60	912	1,349	1,453	6

In case of X.264, Monitoring phase incurred an average runtime overhead of 5X and an average memory overhead of 7.6X, and the tool required average runtime overhead of 31X and memory overhead of 37.8X for detection phase. For MySql and Cherokee, VcTrace incurred average memory overheads of 2X and 1.4X, respectively, in monitoring phase, and incurred average memory overheads of 3.7X and 1.6X, respectively, in detection phase. Additionally, the tool precisely located data races for three benchmarks, because these data races are well known that they exist in these applications by a prior work [8].

5 Conclusion

It is important to monitor the aspect of thread execution and the accesses to the shared memory location for debugging data races. This paper presented a dynamic monitoring tool, called VcTrace that analyzes partial ordering of concurrent threads and their accesses to the shared memory locations during an execution of the program based on vector clock system. Empirical results on C/C++ benchmarks using multithreads show that VcTrace is a sound and practical for on-the-fly data race detection as well as analyzing multithread programs. In the future, VcTrace might be developed as a practical software testing tool for detecting concurrency bugs, such as data races and deadlocks.

Acknowledgments. This work was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2013R1A1A2011389) and also was supported by the Korea Evaluation Institute of Industrial Technology (KEIT) under "the Development of Verification System for Mid-sized IMA Project" (10043591) funded by the Ministry of Trade, Industry & Energy.

References

1. Netzer, R.H.B., Miller, B.P.: What are race conditions?: Some issues and formalizations. ACM Lett. Program. Lang. Syst. 1, 74-88 (1992)
2. 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, PADTAD 2006, pp. 69–78. ACM, New York (2006)
3. Flanagan, C., Freund, S.N.: FastTrack: Efficient and precise dynamic race detection. In: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, pp. 121–133. ACM, New York (2009)
4. Ha, O.-K., Kuh, I.-B., Tchamgoue, G. M., Jun, Y.-K.: On-the-fly detection of data races in OpenMP programs. In: Proceedings of the 2012 Workshop on Parallel and Distributed Systems: Testing and Debugging. PADTAD 2012, pp. 1-10. ACM, New York (2012)
5. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21, 558–565 (1978)
6. Baldoni, R., Raynal, M.: Fundamentals of distributed computing: A practical tour of vector clock systems. IEEE Distributed Systems Online 3 (February 2002)
7. Bach, M., Charney, M., Cohn, R., Demikhovsky, E., Devor, T., Hazelwood, K., Jaleel, A., Luk, C.K., Lyons, G., Patil, H., Tal, A.: Analyzing parallel programs with pin. Computer 43(3), 34-41 (2010)
8. Yu, J., Narayanasamy, S.: A case for an interleaving constrained shared-memory multi-processor. SIGARCH Comput. Archit. News 37(3), 325-336 (2009)

