Asian Journal of Information Technology 15 (4): 723-729, 2016
ISSN: 1682-3915

An Effective Standing Exposure in Gridlock (Deadlock)

[1]A. Mohan and [2]P. Senthil Kumar
[1]Saveetha Engineering College, Anna University, 602105 Chennai, India
[2]SKR Engineering College, 600123 Chennai, India

Abstract: Gridlock (Deadlock) independence is the most important dispute in developing multithreading programs. To avoid the potential risk of blocking a program, prior monitoring of threads can be used during the execution process. The proper monitoring scheme can able to monitor the threads that might enter to a deadlock stage, it maintains a backup to store the threads so after the execution of one thread the injection of the other thread can be made from backup into the processing stage. Today's parallel programs are difficult with deadlock problem further problem by the shift to multicore processors. By using this process, the deadlock can be avoided in the multithreading environment. Moreover, fixing other concurrency problems like races often involves introducing new synchronization which also cause the new set of deadlock. In the proposed system, we apply the different necessary condition for a deadlock, we implement the algorithm and report upon our experience applying it to a suite of multithreaded java program. It helps the threads to recover from deadlock situation and lets the threads complete their execution.

Key words: Deadlock, synchronization, multithreading, gridlock, thread maps, scrider

INTRODUCTION

Deadlock is a situation in which two computer programs sharing the same resource are effectively preventing each other from accessing the resource, resulting in both programs ceasing to function. The earliest computer operating systems ran only one program at a time. Deadlock-freedom is a major challenge in developing multi-threaded programs as a deadlock cannot be resolved until one restarts the program (mostly by using manual intervention). To avoid the potential risk of blocking, a program may use try lock operations rather than lock operations. In this case, if a thread fails to acquire a lock, it can take appropriate action such as releasing existing locks to avoid a deadlock (Mohan and Kumar, 2014). In the existing system, the usage of mapping is not implemented and in another mechanism, there is an approach that specifically clears the circular mutex wait for deadlocks and lock graphs but this model is not suited for all environments (Mohan and Kumar, 2014). The existing dynamic method has less efficiency compared to the static deadlock analysis method so the proposed system provides an efficient mapping technique for avoiding deadlocks depending upon priority.

The main aim of the project is to avoid the deadlocks occurred in the threads during execution by providing a map that stores the thread objects and locks acquired and requested by the thread. In this case, if a thread fails to acquire a lock, it can take appropriate action such as releasing existing locks to avoid a deadlock.

In order to avoid deadlocks in threads during the execution process in the proposed system, the monitor that identifies the threads running in the program, i.e., the thread objects are identified. After this process, there will be a map generated that store the thread objects and the locks acquired and requested by them. Whenever, a thread tries to acquire a lock if the access is denied then it waits for the certain period of time. After the time period is over, the thread again tries to access the lock, since due to some reasons if accessing the locks still denies then thread traverses the map to identify what are all the threads have requested or hold the same locks requested by it. So, if it finds any such threads then it detects that deadlock condition occurred, after this, the deadlocked thread wait for each other for an infinite time. When, it finds deadlock condition prevails, the thread now releases all the locks acquired by it, so that it might allow the deadlocked threads to complete their required operation. In another case if more deadlocks are detected then according to the priority the threads given to process their execution, the execution of the priority based threads are executed in a random manner during this process the threads involving in execution are been backed up for a while during other threads are in execution. According to the priority, the threads execution states are changed, this helps he threads to recover from deadlock situation and let the other threads complete their execution.

A problem that no existing analysis can directly solve effectively in terms of different problems that can be solved effectively.

Corresponding Author: A. Mohan, Saveetha Engineering College, Anna University, 602105 Chennai, India

Available: In a multithreaded program thread A (ta1) acquires the lock LA1 and while holding LA1, it proceeds to reach LA2, simultaneously holding both LA1 and 2. Similarly, thread B (tb1) acquires a lock LB1 while holding LB1, it proceeds to reach LB2 such that it holds both LB1 and 2.

Signaling: During the thread B holds LB1, LB2 then lock acquired by thread A, i.e., LA1 will it execution of a multithreaded program if thread A holds LA1, LA2 and is same as the lock acquired by thread B, i.e., LB2. Similarly, the lock acquired by thread A at LA2 will it be same as the lock acquired by thread B, i.e., LB1.

Avoidance: In a multithreaded program for example, thread A acquires LA1, then thread B, thread C and so on Can also acquire and access the lock at LA1. Similarly, for each lock acquired at LA2, LB1, LB2 can it be accessible from more than one threads like thread A, thread B and thread C.

Comparable: During the execution of program, if suppose thread A acquires LA2, then will it be possible for thread B. To acquire LB2 parallel, i.e., can different threads abstracted by TA and TB simultaneously reach LA2 and LB2, respectively.

Non safe: In some execution of a multithreaded program, is it possible for a thread abstracted by TA that does not hold any lock to acquire a lock at LA and while holding this lock, proceed to lock LA2 which is not already held by it..similarly can thread TB acquire lock LB1 which is not already held and while holding this can it proceed to acquire lock LB2 which is not already held.

Thus, if a thread which holds a lock acquires the same lock then the lock caused at the second time cannot create a deadlock because the locks can be reentered again as reentrant is possible in java.

Unmindful: In a multi threaded program, if two threads say TA and TB hold a common lock say lock "G" then it is called as guarding lock or simply gate lock. Thus, in case if there isn't any common lock, i.e., gate lock between different threads abstracted by TA and TB then is that possible for the threads to reach or acquire the locks LA1 and LB1.

Riterature review: The static and dynamic techniques used for exposing deadlock potentials (Agarwal *et al*., 2010). It has three extensions to the basic algorithm (logic graph) to eliminate, label as low severity and false warning of possible deadlocks. The extensions of lock graph algorithm to detect the deadlock in static and dynamic techniques.

They propose a new technique is practical static race detection for java parallel loops and the use of these constructs and libraries improves accuracy and scalability (Radoi and Dig, 2013). The new tool called ITE Race has been introduced which includes, a set of techniques that are specialized to use the intrinsic thread, safety and dataflow structure of collections. The ITE Race is fast and perfect enough to be practical. It scales to programs of hundreds of thousands of lines of code and it reports few race warnings, thus avoiding a common consequence of static analyzes. The tool implementing this method is fast, does not delay the program with many warnings and it finds latest bugs that were confirmed and fixed by the developers.

The detecting atomicity violations using dynamic analysis technique is present (Flanagana and Freund, 2008). A more fundamental non interference property is atomicity. A method execution is not affected by concurrently-executing threads means that method is called as the atomic method. It contains both formal and informal correctness arguments. Detecting atomicity violations combine an idea of both Lipton's theory of reduction and early dynamic race detectors. It is effective error detecting to unintended interactions between threads. It will be more effective than standard race detectors.

Flanagana and Freund (2006) proposes the type inference algorithm for RCC java. The performance of the algorithm is applied on programs of up to 30,000 lines of code. The resulting annotations and race-free guarantee provided by our type inference system. Type inference algorithm applied to the concurrent program to manipulate the shared variable without synchronization. This algorithm has some lock variables. Extending this inference algorithm to larger benchmark has some issue. It produces reliable error reporting.

Hasanzade and Babamir (2012) describe an approach for online deadlock detection for multithreaded programs using the prediction of future behavior of threads. About 74% of deadlock were predicted using the proposed method. Some specific behaviors of threads are extracted at run time and converted into the predictable format using time series method. The proposed method has several advantages compared to the existing static methods. A powerful technique used for predicting complex deadlock s.

Bodden and Havelund (2010) implement an efficient algorithm to sense concurrent programming errors online. In that system programmers to monitor program events

where locks are granted or handed back and where values are accessed that may be shared among multiple Java threads. The proposed RACER algorithm uses ERACER for memory model of java and AspectBench compiler for implementation. In that project, they proposed a language extension to the aspect-oriented programming language AspectJ. The proposed AspectJ have implemented the following three points. There are Lock(), Unlock(), Maybeshared().

Chen *et al.* (2011) examines the performance scaling of various processor cores and application threads. It analyzes the performance and scalability by correlating low-level hardware data to JVM threads and system components. It uses the JVM tuning techniques to solve the problems regarding lock conditions and memory access latencies. The study of performance, scalability of multithreaded java application on multicore systems is done. The proposed method reduces the bottlenecks using JVM tuning techniques. Inappropriate use of synchronization leads to a large number of stall cycles.

Joshi *et al.* (2009) present a novel dynamic analysis method to find real dead-locks in multi-threaded programs. Deadlock-fuzzer is the new technique used to find the deadlocks in two phases. In the first phase, find potential deadlocks in a multithreaded program using dynamic analysis technique by execution of the program. In the second phase, to control the deadlock creation using threads scheduler. Deadlock-fuzzer is implemented to find the all previously known deadlocks in large benchmarks and but it does not discover previously unknown deadlocks in an efficient manner. This technique needs both static and dynamic techniques.

Wang *et al.* (2009) describes a new Java thread deadlock detection approach called as JDeadlockDetector. Existing system requires source code and built on non-official JVMs for Java thread deadlock detection solutions. Many numbers of Java programs cannot be evaluated with these solutions. The JDeadlockDetector is built on the official Java Virtual Machine (JVM), viz., OpenJDK's HotSpot. The JdeadlockDetector has three unique advantages compared to the existing system. There are application transparency, detection accuracy and minimized performance overhead. The JDeadlockDetector achieves no false negative and minimized false positive. JDeadlockDetector to detect Java thread deadlock based on holds the capability of monitoring the thread states and synchronization states on runtime. In this way, the technique achieves their advantages. To track the control flow and data flow of a Java program they want to extend the Hotspot introspection architecture. This will afford a capability to analyze the vulnerability of Java programs.

In a new two-phase deadlock detection scheme was introduced which provides efficient memory utilization and time constraints. The performance of the proposed system is much higher than the traditional approach to finding the potential deadlock in an application. The first phase reduces lock by filter out certain locks that cannot participate. The second phase creates smaller lock graph for potential deadlock detection. The proposed work can minimize the overall deadlock detection time and increases the performance. We focus on developing dynamic deadlock detection technique which reduces the deadlock occurrences.

MATERIALS AND METHODS

System architecture: The system architecture for the proposed system includes deadlock monitor and analyzing thread states and explains the efficient ways of detecting deadlock in the multithread program using thread map and priority assignment. The java thread has created based on the set of condition to occurring into a deadlock situation. Each thread has built based on a certain lock to access the resources. The deadlock monitor that identifies the threads running in the program, i.e., the thread objects is identified. After this process, there will be a map generated that store the thread objects and the locks acquired and requested by them (Fig. 1).

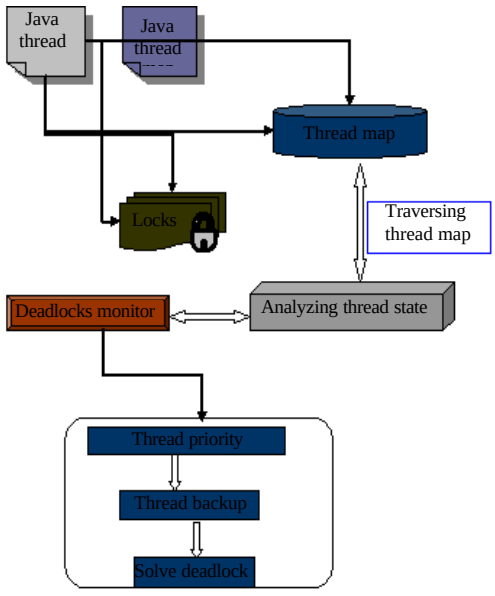

Fig. 1: System architecture

When, it finds deadlock condition prevails, the thread now releases all the locks acquired by it, so that it might allow the deadlocked threads to complete their required operation. In another case if more deadlocks are detected then according to the priority the threads given to process their execution, the execution of the priority based threads are executed in a random manner.

Module description

Estimation of available deadlock: For a tuple (ta, la1, la2, tb, lb1, lb2) to be a deadlock our aliasing condition must be satisfied: Can a lock acquired at la1 be the same as a lock acquired at lb2 (and similarly for la2, lb1)? Our algorithm uses the mayAlias property (Fig. 4) to approximate this condition:

availableDeadlock(ta, la1, la2, tb, lb1, lb2) if
mayAlias(la1, lb2)/\ mayAlias(la2, lb1)

For our running example, both tuples d1 and 2 satisfy availableDeadlock: predicates mayAlias (l1, l1) and available deadlock; additionally, mayAlias (l3, l2) holds because abstract object [h3] satisfies the two conjuncts in the definition of mayAlias and hence tuple d2 also satisfies availableDeadlock.

Estimation of signaling deadlock: The JDK contains many classes (e.g., java.util.vector) with synchronized methods. When such objects cannot be accessed by more than one thread, they cannot participate in a deadlock. Thus, for a tuple (ta, la1, la2, tb, lb1, lb2) to be a deadlock our escaping condition must be satisfied: can a lock acquired at la1 be accessible from more than one thread (and similarly for each of la2, lb1, lb2)?

We approximate this condition using a thread escape analysis. Our application of this analysis to static deadlock detection appears novel and we quantify the need for it in our experiments. The thread-escape problem is usually defined as follows:

"In some execution is some object allocated at a given site h accessible from more than one thread?" To increase precision, we refine the notion of thread-escape to track when an object escapes. This allows the escaping condition to eliminate some deadlock reports on objects that later escape to other threads. Formally, (c, v) must be in relation esc if argument v of abstract context c may be accessible from more than one thread. Our escaping condition is thus:

SignallingDeadlock (ta, la1, la2, tb, lb1, lb2) if
(la1, sync(la1))∈esc/\(la2, sync(la 2))∈esc/\
(lb1, sync(lb1))∈esc/\(lb2, sync(lb2))∈esc

For our running example, LogManager.manager (l2,l3) and Logger class (l1), being static fields, clearly escape everywhere and so both tuples d1 and d2 satisfy escaping deadlock.

Estimation of avoidance deadlock: For a tuple (ta, la1, la2, tb, lb1, lb2) to be a deadlock our parallel condition must be satisfied: can different threads abstracted by ta and tb simultaneously reach la2 and lb2, respectively?

The motivation for checking this condition is twofold. First, it eliminates each tuple (t,*,*,t,*,*) where t abstracts at most one thread in any execution. The most common example of such an abstract thread is ([], mmain) but it also applies to any thread class allocated at most once in every execution. The second motivation is that even if different threads abstracted by ta and tb may be able to reach la2 and lb2, respectively, the thread structure of the program may forbid them from doing so simultaneously, namely, threads ta and tb may be in a "parent-child" relation, causing la2 to happen before lb2 in all executions. We approximate these two conditions using a may happen-in-parallel analysis that computes relation map which contains each tuple (t1, (o,m), t2) such that a thread abstracted by t2 may be running in parallel when a threadabstracted by t1 reaches method m in context o. Our may happen-in-parallel analysis is simple and only models the program's thread structure, ignoring locks and other kinds of synchronization (fork-join, barrier, etc). Our parallel condition is thus:

avoidanceDeadlock (ta, la1, la2, tb, lb1, lb2) if
(ta, la2, tb)∈mhp/\(tb, lb2, ta)∈mhp

For our running example, clearly nothing prevents t1 and t2 from running in parallel, so tuples d1 and d2 satisfy avoidanceDeadlock.

Estimation of nonsafe deadlock: In Java, a thread can re-acquire a lock it already holds. This nonsafe lock acquisition cannot cause a deadlock. Thus, for a tuple (ta, la1, la2, tb, lb1, lb2) to be a deadlock ournon-nonsafe condition must be satisfied: Can a thread abstracted by ta acquire a lock at la1 it does not already hold and while holding that lock, proceed to acquire a lock at la2, it does not already hold (and similarly for tb, lb1, lb2)? Soundly, identifying nonsafe locks requires must-alias analysis. Must-alias analysis, however is much harder than mayAlias analysis. Instead, we use our mayAlias analysis itself to unsoundly check that whenever a thread abstracted by t acquires a lock at l1 and while holding that lock, proceeds to acquire a lock at l2, then the lock it

acquires at l1 or 2 may (soundness requires must) be already held by the thread, a property approximated by unmindfult:

unmindful(t, l1, l2) iff l1 = l2∨
(∀L1: (t→l1_L1 =⇒mayAlias({l1, l2}, L1)))∨
(∀L2: (l1→l2_L2 =⇒mayAlias({l2}, L2)))

Intuitively, the first conjunct checks that the locks acquired at l1 and l2 may be the same. The second conjunct checks that when a thread abstracted by t reaches up to but not including l1, the set of locks L1 it holds may contain the lock it will acquire at l1 or 2. The third conjunct checks that when the thread proceeds from l1 and reaches up to but not including l2, the set of locks L2 it holds may contain the lock it will acquire at l2. Next, we use the unmindful predicate to approximate our non-mindful condition as follows:

nonMindfulDeadlock (ta, la1, la2, tb, lb1, lb2) if
¬unmindful(ta, la1, la2)/\¬unmindful (tb, lb1, lb2)

The above approximation itself is sound but the approximation performed by the unmindful predicate it uses is unsound; thus, a tuple that does not satisfy nonMindfulDeadlock is not provably deadlock-free. For our running example, the two locks acquired by either thread do not alias and no locks are acquired prior to the first lock or between the first and second lock in either thread, so tuples d1 and d2 satisfy nonMinduful Deadlock.

Estimation of unmindful Deadlock: One approach to preventing deadlock is to acquire a common guarding lock in all threads might deadlock. Thus, for a tuple (ta, la1, la2, tb, lb1, lb2) to be a deadlock our nonsafe condition must be satisfied: can threads abstracted by ta and tb reach la1 and lb1, respectively, without already holding a common lock?

Soundly, identifying guarding locks, like mindful locks, needs a must-alias analysis. We once again use our may-alias analysis to unsoundly check whether every pair of threads abstracted by ta and tb may (soundness requires must) hold a common lock whenever they reach la and lb,respectively, a property approximated by safe:

safe(ta, la, tb, lb) iff∀L1, L2:
(ta→la_L1/\tb→lb_L2) =⇒ mayAlias(L1, L2)

Then, we use the guarded predicate to approximate our nonsafe condition as follows:

nonsafeDeadlock (ta, la1, la2, tb, lb1, lb2) if
¬safe(ta, la1, tb, lb1)

The above approximation itself is sound but the approximation performed by the guarded predicate it uses is unsound; thus, a tuple that does not satisfy nonsafeDeadlock is not necessarily deadlock-free. For our running example, as we saw for nonMindfulDeadlock, no locks are acquired prior to the first lock, so tuples d1 and d2 satisfy nonsafeDeadlock.

RESULTS AND DISCUSSION

Experiments: We evaluated Deadlock Detector (DD) on a suite of multi-threaded Java programs comprising over the processor. The suite includes the multi-threaded benchmarks from the Java Grande suite (moldyn, montecarlo and raytracer); from ETH, a traveling salesman problem implementation (tsp), a successive over-relaxation benchmark (sor) and a web crawler (hedc); a website download and mirror tool (weblech); a web spider engine (jspider); W3C's web server platform (jigsaw) and Apache's FTP server (ftp). The suite also includes open programs for which we manually.

Wrote harnesses; apache's database connection pooling library (dbcp); a fast caching library (cache4j); the JDK4 logging facilities (logging) and JDK4 implementations of lists, sets and maps wrapped in synchronized collections (collections).

The experiments were performed on a 64-bit Linux server with two 2GHz Intel Xeon quad-core processors and 8 GB memory. DD, however, is single-threaded and 32 bit and hence utilizes only a single core and at most 4 GB memory. The '0-CFA' and 'k-obj.' columns give the size of final deadlocks after one and two iterations of our algorithm final deadlocks is empty or starts to grow and DD terminates, after at most two iterations for all our benchmarks. The first iteration uses a k-object-sensitive analysis that is essentially a 0-CFA-based analysis. The difference between the two columns, most notable for hedc, weblech, jspider, ftp and dbcp, is the number of extra false positives that would be reported by a 0- CFA-based analysis over a k-object-sensitive one. All previous static deadlock detectors we are aware of employ a 0-CFA-based analysis or an even more imprecise CHA-based analysis; moreover, they exclude checking one or more of our six necessary conditions (Fig. 2 and 3).

Performance analysis: The performance measurement of the dynamic deadlock detection technique is given below.

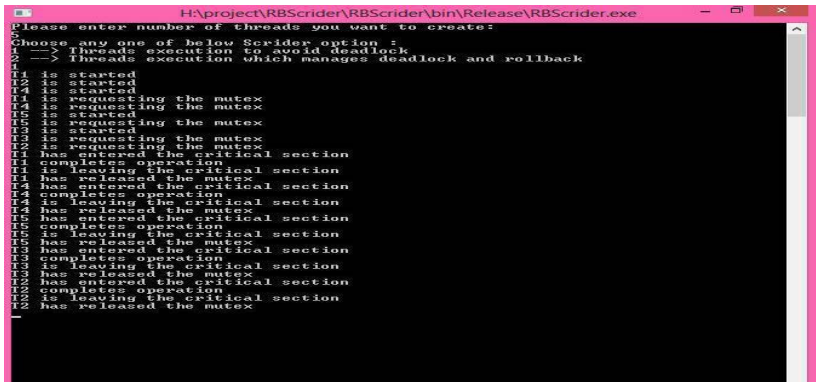

Fig. 2: Execution of threads

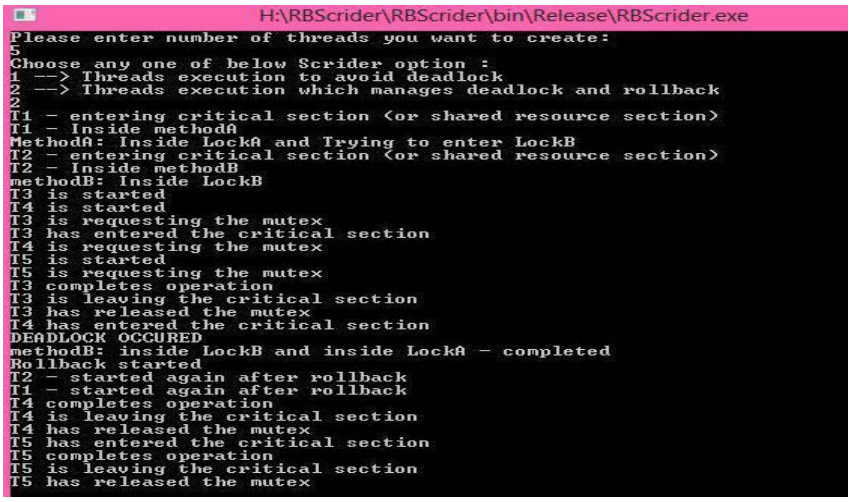

Fig. 3: Execution of deadlock detection using DD

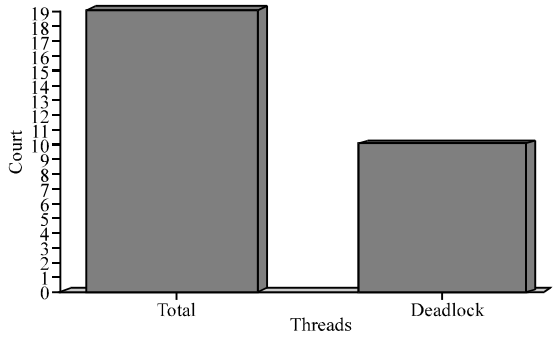

Fig. 4: Performance analysis of deadlock detection

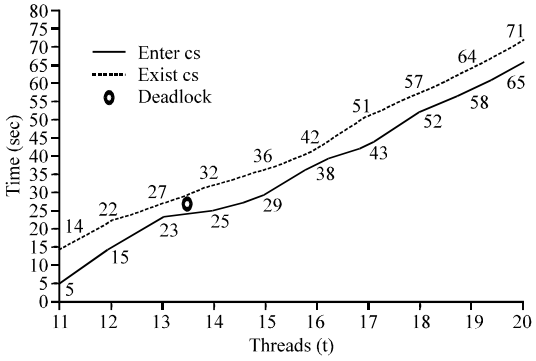

Fig. 5: Performance analysis using scrider

The diagram shows how many threads are running in the thread map in the name of total threads and how many deadlock are solved based on priority level in a show in deadlocked threads. Based on our program it solves the nearly 10 deadlocked threads (Fig. 4 and 5).

The proposed system maintains a deadlock map that holds the information about all the threads running in a program. This will reduce the deadlock occurrences based on the random priority assignment. The deadlock monitor controls the overall process of program execution.

CONCLUSION

This study addresses the efficient access of shared memory by the multiple threads execution using deadlock monitor. Deadlock monitor can be used to monitor the thread process regularly. The monitor will reduce the deadlock occurrences. Deadlock available and avoidance will be used for dynamic deadlock detection method. Also, it reduces the cost of accessing memory and improves the efficiency of multithreaded applications.

REFERENCES

Agarwal, R., S. Bensalem, E. Farchi, K. Havelund and Y. Nir-Buchbinder *et al*., 2010. Detection of deadlock potentials in multithreaded programs. J. Res. Dev., Vol. 54. 10.1147/JRD.2010.2060276.

Bodden, E. and K. Havelund, 2010. Aspect-oriented race detection in Java. IEEE Trans. Software Eng., 36: 509-527.

Chen, K.Y., J.M. Chang and T.W. Hou, 2011. Multithreading in Java: Performance and scalability on multicore systems. IEEE Trans. Comput., 60: 1521-1534.

Flanagana, C. and S.N. Freund, 2006. Type inference against races. Sci. Comput. Program., 64: 140-165.

Flanagana, C. and S.N. Freund, 2008. Atomizer: A dynamic atomicity checker for multithreaded programs. Sci. Comput. Program., 71: 89-109.

Hasanzade, E. and S.M. Babamir, 2012. An artificial neural network based model for online prediction of potential deadlock in multithread programs. Proceedings of the 16th CSI International Symposium on Artificial Intelligence and Signal Processing, May 2-3, 2012, Shiraz, Iran, pp: 417-422.

Joshi, P., C.S. Park, K. Sen and M. Naik, 2009. A randomized dynamic program analysis technique for detecting real deadlocks. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, June 15-21, 2009, Dublin, Ireland.

Mohan, A. and P.S. Kumar, 2014. Deadlock maps: A dynamic deadlock detection for multithreaded programs. Asian J. Inform. Technol., 13: 356-362.

Radoi, C. and D. Dig, 2013. Practical static race detection for Java parallel loops. Proceedings of the International Symposium on Software Testing and Analysis, July 15-20, 2013, Lugano, Switzerland, pp: 178-190.

Wang, Y., S. Lafortune, T. Kelly, M. Kudlur and S. Mahlke, 2009. The theory of deadlock avoidance via discrete control. Proceedings of the 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 21-23, 2009, New York, USA., pp: 252-263.