Dl-Check: Dynamic Potential Deadlock Detection Tool for Java Programs

Nikita Koval[1](✉), Dmitry Tsitelov[2], and Roman Elizarov[2]

[1] Computer Technology Department, ITMO University, St. Petersburg, Russia
koval@rain.ifmo.ru
[2] dxLab, Devexperts, St. Petersburg, Russia
{cit,elizarov}@devexperts.com

Abstract. Deadlocks are one of the main problems in multithreaded programming. This paper presents a novel approach to detecting potential deadlocks at run-time. As opposed to many dynamic methods based on analyzing execution traces, it detects a potential deadlock immediately at the point of the lock hierarchy violation which happens first during execution, so all the run-time context is available at the point of detection. The approach is based on the target program instrumentation to capture lock acquisition and release operations. An acyclic lock-order graph is maintained incrementally, so cycles are detected during graph maintenance. The presented algorithm is based on topological order maintenance and uses various heuristics and concurrent data structures which improve performance and scalability. Experimental results show that Dl-Check is efficient for large-scale multithreaded applications.

Keywords: Potential deadlock detection · Lock-order graph
Dynamic analysis · Java

1 Introduction

Since the beginning of the multicore era, developers have started writing multithreaded programs in order to use new hardware more efficiently [1]. One of the most common problems in multithreaded programming is a deadlock. There are two types of deadlocks [2–4]: resource deadlocks and communication deadlocks. The resource deadlock occurs if each thread of the set tries to acquire the lock held by another thread from this set. The communication deadlock happens if each thread of the set is waiting for a signal from another thread from this set. This paper focuses on resource deadlocks.

Several authors [5–8] use static analysis to detect potential deadlocks, which analyzes source code without its execution and can guarantee that the program is deadlock-free. However, this approach produces a lot of false positives. For example, in the experiment of Williams et al. [6] more than 100'000 potential deadlocks were reported while only 7 of them were real deadlocks. The second approach is model checking, which is used to analyze all possible program executions. It is used in several works [9,10] and it can also guarantee that the program has no possible deadlocks. In comparison with static analysis, model checking can guarantee the absence of false positives, but it is more complicated to implement and requires a lot of computational resource. Consequently, this approach is less suitable for large programs. The third approach is dynamic analysis, when two different techniques are applied. The first technique is based on collecting execution traces and analyzing them after the program has been executed. This technique is used in several tools such as JCarder [11], MulticoreSDK [4], and ConLock [12]. The second technique detects a potential deadlock immediately when it happens like in the VisualThreads [13] tool. The main advantage of this technique is immediate access to all context information, such as stack traces, at the point of detection.

In this paper, we present an algorithm based on dynamic analysis and immediate detection of potential deadlocks at run-time. Alike many authors, we define potential deadlock as a lock hierarchy violation and use the lock-order graph to detect it [7,11,13–16]. However, we present a more scalable approach to maintain lock-order graphs. The main idea of this approach is to keep the acyclic part of the lock-order graph and to perform incremental topological order maintenance.

The paper is organized in the following way. Section 2 presents the algorithm with pseudo code listings. Implementation details are discussed in Sect. 3. Then Sect. 4 shows experimental results for the performance overhead. Finally, Sect. 5 gives the paper conclusion.

2 Algorithm

Definition 1. *Lock u is* ***acquired before*** *lock v ($u \rightarrow v$) if at some point lock v is acquired under lock u in the same thread.*

Definition 2. *The* ***lock hierarchy*** *is a partial order on locks such that for every lock u, which is acquired before lock v, u comes before v in the ordering:*

$$\forall u, v : u \rightarrow v \Rightarrow ord(u) < ord(v)$$

Note that lock hierarchy exists only when all pairs of locks are acquired in the same order in all executions.

The lock hierarchy is a primary method to avoid deadlocks in complex programs [17,18], so this paper determines **potential deadlock** as a lock hierarchy violation.

Definition 3. ***Lock-order graph*** *is a graph, where every vertex is associated with a lock while edge (u, v) means that at some point lock u is acquired before lock v.*

Lemma 1. *Lock hierarchy is a suborder of all possible topological orders on the lock-order graph.*

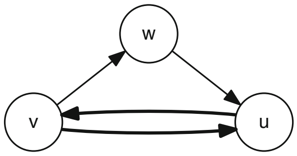
According to Lemma 1, the approach presented in this paper maintains topological order on the lock-order graph and reports a potential deadlock when topological order cannot be satisfied. However, this method differs from searching cycles in the lock-order graph.

2.1 Minimization Principle

In Fig. 1 you can see an example of the code which produces two cycles in the lock-order graph. This lock-order graph is presented in Fig. 2. However, cycle $\langle v, w, u\rangle$ has lock w which is not related to the error. The real error lies in the fact that locks v and u are acquired in the wrong order, which corresponds to cycle $\langle v, u\rangle$. Consequently, two or more cycles in the lock-order graph can correspond to the same potential deadlock.

```
new Thread(() -> {
  synchronized(v) {
    synchronized(w) {
      synchronized(u) { }
    }
  }
}).start();
new Thread(() -> {
  synchronized (u) {
    synchronized (v) { }
  }
}).start();
```

Fig. 1. This example produces two potential deadlocks



$$\begin{tikzcd} & w \arrow[dr] & \\ v \arrow[ur] \arrow[rr, bend right=5] & & u \arrow[ll, bend right=5] \end{tikzcd}$$

Fig. 2. The lock-order graph produced by the code in Fig. 1. It has two cycles: $\langle v, w, u\rangle$ and $\langle v, u\rangle$. However, only $\langle v, u\rangle$ is really helpful

Definition 4. *If insertion of an edge creates several cycles in the lock-order graph, the first cycle is* ***more useful*** *than the second one on condition that the fist cycle is shorter. The ones which are shortest (may be more than one) are* ***useful****.*

This paper proposes the following rule. If a new edge creates several cycles in the lock-order graph, only one *useful* cycle is to be produced. This principle is called **minimization**. For instance, edge (u, v) creates two cycles in the lock-order graph in Fig. 2, however, the most important cycle is $\langle v, u\rangle$, as discussed before, and only this cycle is to be produced according to the proposed rule.

2.2 Capturing Lock Acquire and Release Operations

Lock acquire and release operations shall be captured to create a lock-order graph. In this paper acquisition and release of lock l by thread τ are denoted as LA_l^τ and LR_l^τ respectively. During each of these operations, a multiset of locks held by the current thread should be updated. This multiset for thread τ is denoted as $LOCKSET_\tau$. Algorithm 1 captures lock acquire and release operations through invocation of `AfterLA` and `BeforeLR` procedures. It is worth mentioning that these procedures are invoked under the acquired lock l. For each lock instance, the algorithm associates a node in the lock-order graph and thereafter manipulates this node only. Here is the base capturing algorithm without graph maintenance logic, which is discussed further.

Algorithm 1. Lock acquire and release analysis

```
 1: Nodes // Associates locks with nodes
 2:
 3: procedure AfterLA($l, \tau$)
 4:     $v \leftarrow$ GetNode($l$) // Get a node associated with the lock
 5:     // Add edges to the lock-order graph and look for new cycles
 6:     $cycles \leftarrow$ AddEdges($v$, $LOCKSET_\tau$)
 7:     // Add the node to the multiset associated with the current thread
 8:     $LOCKSET_\tau$.add($v$)
 9:     print($cycles$) // Print potential deadlocks
10: end procedure
11:
12: procedure BeforeLR($l, \tau$)
13:     $v \leftarrow$ GetNode($l$) // Get a node associated with the lock
14:     // Remove the node from the multiset associated with the current thread
15:     $LOCKSET_\tau$.remove($v$)
16: end procedure
17:
18: function GetNode($l$)
19:     return Nodes.computeIfAbsent($l$, CreateNode($l$))
20: end function
```

2.3 Topological Order Maintenance

This subsection describes the implementation of `AddEdges` function (Algorithm 2). As described above, the presented algorithm uses incremental topological order maintenance to find lock hierarchy violations. To maintain topological order incrementally the algorithm suggested by Marchetti-Spaccamela et al. [19,20] is used. Their solution does not reposition all nodes in the graph, but only the ones in the currently affected region. For instance, after adding edge (u, v), where $ord(u) > ord(v)$, only the edges with the order between $ord(v)$ and $ord(u)$ should be repositioned. However, this approach needs a data structure which associates a topological order with the node in the graph. An algorithm to maintain such structure is presented in Subsect. 3.4.

In `AfterLA` procedure, multiple edges can be added if the number of already acquired locks by the current thread is more than one. However, it is possible to maintain topological order in a single iteration if these edges do not create cycles. In practice, it tends to be true. According to Algorithm 2, it is assumed that the lock, associated with the node v, is acquired. Firstly, if the order of all nodes in $LOCKSET_{\tau}$ is lower than the order of v, then all edges from these nodes to v can be added according to the current topological order. To check this fact the synchronization with topological order modification process is required. For this purpose, the algorithm uses a read-write lock. It acquires a read lock for read-only operations (e.g. atomically reading topological order values for several nodes) and modifying operations with nodes associated with the currently acquired lock only (e.g. adding an edge from the associated node to another one). The write lock is acquired during topological order maintenance. If the current topological order is violated, it should be fixed according to the new edge (u, v), where u is the node with the smallest topological order from $LOCKSET_{\tau}$. If this operation is successfully done, then v is correctly ordered with all nodes from $LOCKSET_{\tau}$ because nodes in $LOCKSET_{\tau}$ are ordered already. Otherwise, if the topological order cannot be maintained, then the algorithm tries to add each new edge separately and detects cycles.

In case adding edge (u, v) leads to a topological order violation, the shortest path from v to u is to be found. This shortest path is a cycle to be reported further. To solve this issue the BFS (breadth-first search) algorithm [21] is used.

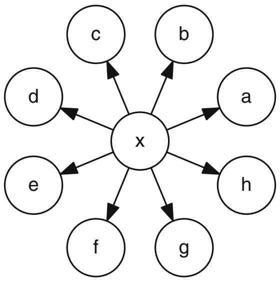


Fig. 3. Snowflake pattern of lock-order subgraph in many programs

The last part in topological order maintenance is the decision on the initial value of the topological order for new nodes. Our experience shows that many programs (like Apache Derby [22], IntelliJ Idea [23], dxFeed [24] and others) have a common pattern in the lock-order graph like a subgraph in Fig. 3 with one long-lived lock in the center and many hundreds of short-lived locks around it. Thus, initializing new nodes with the next highest order immediately produces correct topological order after adding the corresponding edges.

Algorithm 2. AddEdges function

```
 1: $E$ // Lock-order graph edges
 2: $CE$ // Lock-order graph cyclic edges
 3: $RWLock$ // Read-Write lock for manipulations with graph
 4:
 5: function AddEdges($v$, $LOCKSET_\tau$)
 6:     $newEdges \leftarrow \{ (u,v) \mid u \neq v \wedge u \in LOCKSET_\tau \wedge (u,v) \notin E \cup CE\}$
 7:     if $newEdges = \varnothing$ then
 8:         return $\varnothing$
 9:     // Try to add all new edges without changing topological order
10:     $RWLock$.readLock()
11:     if $\forall\ (u,v) \in newEdges$: $ord(u) < ord(v)$ then
12:         $E \leftarrow E \cup newEdges$
13:         $RWLock$.readUnlock()
14:         return $\varnothing$
15:     $RWLock$.readUnlock()
16:     // Maintain topological order
17:     $RWLock$.writeLock()
18:     $rightmost \leftarrow u : (u,v) \in newEdges \wedge \forall\ (w,v) \in newEdges$: $ord(w) \leq ord(u)$
19:     if MaintainTopologicalOrder($rightmost$, $v$) then
20:         $E \leftarrow E \cup newEdges$
21:         $RWLock$.writeUnlock()
22:         return $\varnothing$
23:     // Topological order cannot be maintained, find cycles
24:     $cycles \leftarrow \{\}$
25:     while $newEdges \neq \varnothing$ do
26:         $(u,v) \leftarrow newEdges$.remove() // Get and remove node from newEdges
27:         if MaintainTopologicalOrder($u$, $v$) then
28:             $E \leftarrow E \cup (u,v)$
29:         else
30:             // Cycle detected
31:             $cycles \leftarrow cycles \cup$ ShortestPath($v$, $u$)
32:             $CE \leftarrow CE \cup (u,v)$
33:     $RWLock$.writeUnlock()
34:     return $cycles$
35: end function
```

2.4 Algorithm Complexity

This subsection bounds the complexity of the proposed algorithm. The `BeforeLR` procedure only gets an associated node in $O(1)$ on the average using hash-tables and removes it from $LOCKSET_\tau$ in $O(B)$ at worst, where B is the number of acquired locks in the current thread. The bound of `AfterLA` procedure consists of two parts. Firstly, if acquisition of lock l does not produce new cycles (the typical case), it works in $O(B+|V|+|E|)$, where $|V|$ is the number of nodes in the lock-order graph and $|E|$ is the number of edges in it. According to the statistics, which was collected for analyzed programs, $|E| \approx 4 \cdot |V|$. It is worth noting, that if a program uses the same lock instances, the associated with them nodes can be already ordered before and LA_l^T operation bounds to $O(B)$. Otherwise, if acquiring lock l creates a cycle, BFS could be invoked B times at worst, and the total complexity is $O(B(|V| + |E|))$.

2.5 Limitations

There is likelihood that a new edge will create two independent cycles, as shown in Fig. 4. In practice, such situations are improbable, and commonly the error lies in acquiring lock u before lock v. Anyway, this approach guarantees that at least one potential deadlock is reported in case lock hierarchy is violated.

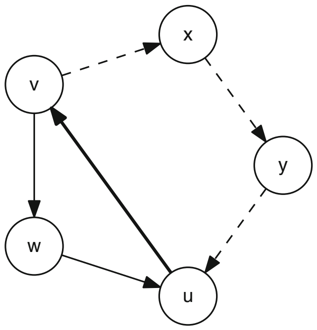


Fig. 4. Edge (u, v) creates two independent cycles: $\langle v, w, u\rangle$ and $\langle v, x, y, u\rangle$, but only the first cycle is produced according to the *minimization* principle

Another problem is that *useful* cycle may not be detected. This issue can occur if an already found cycle has been minimized. For instance, Fig. 5 shows that if edge (u, v) creates cycle $\langle v, z, u\rangle$, then adding edge (v, u) does not create a shorter cycle $\langle v, u\rangle$. In real-world programs the length of cycles in the lock-order graph is almost always lower than five [25], so the cycle minimization logic is not included in the presented algorithm. However, there is a simple way to minimize cycles without a significant performance impact. By the moment edge (u, v) creates a cycle, path $v \rightsquigarrow u$ already exists. Therefore, minimizing this cycle applies to minimizing path $v \rightsquigarrow u$. This can be achieved with the help of BFS algorithm.

2.6 Single Threaded and Guarded Cycles

Many authors treat single threaded and guarded cycles as safe [11,14–16]. *Single threaded cycle* refers to a cycle which is created from locks acquired only in the single thread. *Guarded cycle* refers to a cycle which is guarded by a gate lock “taken higher” up by all involved threads. However, ignoring the lock hierarchy violation can lead to a potential deadlock after code refactoring. Thus, this paper considers all lock hierarchy violations.

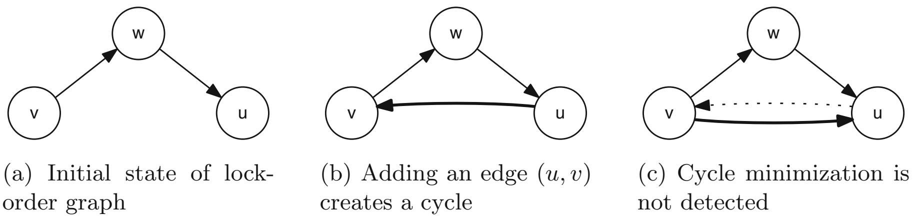

w
v
u
(a) Initial state of lock-order graph
w
v
u
(b) Adding an edge (u, v) creates a cycle
w
v
u
(c) Cycle minimization is not detected

Fig. 5. Example of cycle minimization problem

3 Implementation

3.1 Instrumentation

The presented algorithm is implemented in Dl-Check tool [26] as a Java agent. For capturing lock acquire and release operations it modifies Java byte code via ASM framework [27,28] at certain points during class loading:

- `monitorenter` and `monitorexit` instructions. `AfterLA` procedure is inserted after `monitorenter` instruction and `BeforeLR` procedure is inserted before `monitorexit` instruction, so analysis is invoked under the acquired lock;
- `synchronized` methods. `AfterLA` and `BeforeLR` procedures are inserted as the first and the last action of the method, accordingly;
- `java.util.concurrent.locks.Lock`. `AfterLA` procedure is inserted after `lock` and after successful `tryLock` methods invocations, `BeforeLR` procedure is inserted before `unlock` method invocations.

3.2 Memory Management

Java has a garbage collector for memory management purposes. Thus, when a lock, associated with specified node in the lock-order graph, is collected by the garbage collector, this node should be removed from the lock-order graph to avoid memory leaks. For this purposes WeakReference [29] is used for almost all internal data structures.

3.3 Multiset of Held Locks

According to the statistics, which was collected for analyzed programs, locks are acquired and released in "last in, first out" order in almost all cases. Thus, the multiset is implemented as a stack with support of removing from the middle. This way, almost all operations with this multiset work in $O(1)$.

Algorithm 3. Node life-cycle

```
1: Buffer// Buffer for new nodes
2: OrdInv// Associates topological order with nodes
3:
4: function CreateNode($l$)
5:     node← new Node($l$)
6:     Buffer.push(node)
7:     if Buffer.size() > MAX_BUFFER_SIZE then
8:         RWLock.writeLock.lock()
9:         // Compress OrdInv if needed and flush new lock nodes buffer
10:        CompressOrdInv()
11:        FlushBuffer()
12:        RWLock.writeLock.unlock()
13:    return node
14: end function
15:
16: procedure FlushBuffer
17:    while Buffer$\neq \varnothing$ do
18:        node← Buffer.pop()
19:        allocate(node, OrdInv.size())
20: end procedure
21:
22: procedure CompressOrdInv
23:    if OrdInv.deadNodes() > MAX_DEAD_NODES then
24:        order ← 0
25:        for node | node $\in$ nodes $\wedge$ isAlive(node) do
26:            OrdInv.remove(node)
27:            allocate(node, order)
28:            order ← order+1
29: end procedure
30:
31: // Modification for AddEdges procedure
32: function AddEdges($v$, $LOCKSET_\tau$)
33:    ...
34:    // If edge with tail from Buffer should be added then
35:    // compress OrdInv if needed and flush new lock nodes buffer
36:    if $\exists$(u, v) $\in$ newEdges: $u \in$ Buffer then
37:        RWLock.writeLock()
38:        CompressOrdInv()
39:        FlushBuffer()
40:        RWLock.writeUnlock()
41:    ...
42:    RWLock.writeLock() // Acquired for topological order maintenance
43:    // Compress OrdInv if needed and flush new lock nodes buffer
44:    CompressOrdInv()
45:    FlushBuffer()
46:    ...
47: end function
```

3.4 Node Life-Cycle

Creating a new node requires topological order initialization and storing in *OrdInv*, so this operation should be executed under the write lock. However, according to the snowflake pattern in Fig. 3, many nodes have only one incoming and no outgoing edges, so topological order maintenance is not required for them. Adding such nodes without acquiring the write lock allows to process them without blocking and improves the scalability of the algorithm. To achieve this goal a temporary lock-free buffer for new lock nodes is used. While the node is stored in this temporary buffer, its order is equal to ∞ and it is not stored in *OrdInv*. Thus, when a new edge (u, v) is added and u is in the temporary buffer all nodes from it should be stored in *OrdInv* with initialized order. To avoid memory leaks this buffer has a maximum capacity and *OrdInv* is cleaned up periodically to remove nodes, which have been collected by the garbage collector. Algorithm 3 has a pseudo code for maintaining such temporary buffer and *OrdInv*. Note that *OrdInv* also changes due to topological order maintenance.

4 Evaluation

Dl-Check is evaluated on a variety of Java multithreaded benchmarks and compared with JCarder [11] and MulticoreSDK [4] tools. Apart from the fact that these instruments use another technique, there is no available instrument which detects potential deadlocks immediately at run-time. Table 1 lists the benchmarks used in the experiment. All benchmarks except Fine-Grained are real-world programs. The experiment runs on a machine with two Intel® Xeon® E5-2630 v4 @ 2.20 GHz processors and 128 GiB RAM under Java HotSpot version 1.8.0_92.

Table 1. Benchmark programs and their descriptions

Benchmark	Description
derby 1, 2	Benchmark for Apache Derby [22] from SpecJVM2008 benchmark suite [30]. Runs with 4 and 40 threads respectively
Fine-Grained 1, 2	Benchmark for fine-grained locking [17] from Dl-Check [26]. Runs with 10 threads for 100 and 10'000 locks respectively
luindex	Benchmark for Apache Lucene [31] from DaCapo benchmark suite [32]
avrora	Benchmark from DaCapo benchmark suite [32] which simulates a number of programs run on a grid of AVR microcontrollers
h2	Benchmark from DaCapo benchmark suite [32] which simulates a banking application

Table 2. Memory usage

Benchmark	Baseline	Dl-Check	JCarder	MulticoreSDK
derby 1	$\sim$ 1.85 GiB	$\sim$ 1.85 GiB	$\sim$ 1.12 GiB	$\sim$ 1.36 GiB
derby 2	$\sim$ 205 MB	$\sim$ 220 MB	$\sim$ 250 MB	$\sim$ 230 MB
Fine-Grained 1	$\sim$ 69 MB	$\sim$ 111 MB	—[1]	—[1]
Fine-Grained 2	$\sim$ 140MB	$\sim$ 188MB	—[1]	—[1]
luindex	$\sim$ 17 MB	$\sim$18 MB	$\sim$17 MB	$\sim$23 MB
avrora	$\sim$22 MB	$\sim$23 MB	$\sim$23 MB	$\sim$20 MB
h2	$\sim$190 MB	$\sim$190 MB	$\sim$190 MB	$\sim$190 MB

[1]JCarder and MulticoreSDK cannot be used with Fine-Grained benchmarks because of bugs in the bytecode instrumentation

Table 3. Throughput, op/s

Benchmark	Baseline result	Dl-Check		JCarder		MulticoreSDK	
		result	slowdown	result	slowdown	result	slowdown
derby 1	319.37 ± 2.6	208.94 ± 6.08	1.53	32.94 ± 0.73	9.7	43.75 ± 1	7.3
derby 2	2298.47 ± 10.72	1303.48 ± 76.29	1.76	17.93 ± 0.18	128.2	19.04 ± 0.61	120.71
Fine-Grained 1	23.62 ± 6.7	5.51 ± 0.93	4.29	—[1]	—[1]	—[1]	—[1]
Fine-Grained 2	39.72 ± 1.92	2.2 ± 0.46	18.05	—[1]	—[1]	—[1]	—[1]
luindex	1.21 ± 0.07	1.18 ± 0.08	1.03	0.97 ± 0.02	1.25	0.03 ± 0.001	40.33
avrora	0.24 ± 0.01	0.23 ± 0.01	1.04	0.19 ± 0.01	1.26	0.18 ± 0.03	1.33
h2	0.13 ± 0.004	0.05 ± 0.001	2.6	0.05 ± 0.002	2.6	0.04 ± 0.001	3.25

[1]JCarder and MulticoreSDK cannot be used with Fine-Grained benchmarks because of bugs in the bytecode instrumentation

As Table 2 shows, all tools introduce additional memory overhead. However, for Fine-Grained benchmarks Dl-Check needs to store only $O(V+E)$ additional information, where $|E| \approx \frac{|V|}{2}$ instead of approximation as $|E| \approx 4 \cdot |V|$.

Table 3 shows the performance metrics for every analyzing tool. The number of benchmark operations per second (the higher the better) is used as the base metric and the average slowdown factor (the less the better) is presented additionally. As this table shows, the performance impact is more significant. JCarder and MulticoreSDK trace lock acquire and release operations during run-time and analyze the collected data off-line. Thus, the impact of tracing has been measured for them. The table shows that JCarder and MulticoreSDK make some benchmarks more than 100 times slower. Despite checking for deadlocks during run-time, Dl-Check slows down real programs threefold at most and shows high scalability. The fine-grained locking is the most aggressive benchmark for the presented algorithm, so for a big number of locks, the slowdown is significant. It occurs because topological order maintenance is often invoked on the large region.

5 Conclusion

This paper presents an efficient and scalable algorithm to detect potential deadlocks immediately at run-time. As a result of this work, Dl-Check tool for Java has been implemented. The experiments have shown that Dl-Check has a small impact on performance and memory usage for real-world programs and shows better results than other tools using dynamic analysis.

Introducing a lock grouping feature [4,14,25] and contracts to describe lock acquisition rules for specified parts of the analyzed program is planned for the future. The current version of Dl-Check is available on GitHub: http://github.com/Devexperts/dlcheck.

References

1. Sutter, H.: The free lunch is over: a fundamental turn toward concurrency in software. Dr. Dobb's J. **30**(3), 202–210 (2005)
2. Knapp, E.: Deadlock detection in distributed databases. ACM Comput. Surv. **19**(4), 303–328 (1987)
3. Singhal, M.: Deadlock detection in distributed systems (1989)
4. Da Luo, Z., Das, R., Qi, Y.: MulticoreSDK: a practical and efficient deadlock detector for real-world applications. In: Proceedings - 4th IEEE International Conference on Software Testing, Verification, and Validation, ICST 2011, pp. 309–318 (2011)
5. Artho, C., Biere, A.: Applying static analysis to large-scale, multi-threaded Java programs. In: Proceedings of the Australian Software Engineering Conference, ASWEC, pp. 68–75, January 2001
6. Williams, A., Thies, W., Ernst, M.D.: Static deadlock detection for Java libraries. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 602–629. Springer, Heidelberg (2005). https://doi.org/10.1007/11531142_26
7. Agarwal, R., Wang, L., Stoller, S.D.: Detecting potential deadlocks with static analysis and run-time monitoring. In: Ur, S., Bin, E., Wolfsthal, Y. (eds.) HVC 2005. LNCS, vol. 3875, pp. 191–207. Springer, Heidelberg (2006). https://doi.org/10.1007/11678779_14
8. Naik, M., Park, C.-S., Sen, K., Gay, D.: Effective static deadlock detection. In: Proceedings of the 31st International Conference on Software Engineering
9. Mazzanti, F., Spagnolo, G.O., Della Longa, S., Ferrari, A.: Deadlock avoidance in train scheduling: a model checking approach. In: Lang, F., Flammini, F. (eds.) FMICS 2014. LNCS, vol. 8718, pp. 109–123. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10702-8_8
10. Antonino, P., Gibson-Robinson, T., Roscoe, A.W.: Efficient deadlock-freedom checking using local analysis and SAT solving. In: Ábrahám, E., Huisman, M. (eds.) IFM 2016. LNCS, vol. 9681, pp. 345–360. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33693-0_22
11. JCarder – dynamic deadlock finder for Java (2010). http://www.jcarder.org
12. Cai, Y., Wu, S., Chan, W.K.: ConLock: a constraint-based approach to dynamic checking on deadlocks in multithreaded programs. In: Proceedings of the 36th International Conference on Software Engineering - ICSE 2014, pp. 491–502 (2014)
13. Havelund, K.: Using runtime analysis to guide model checking of Java programs. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 245–264. Springer, Heidelberg (2000). https://doi.org/10.1007/10722468_15

14. Agarwal, R., Bensalem, S., Farchi, E., Havelund, K., Nir-Buchbinder, Y., Stoller, S.D., Ur, S., Wang, L.: Detection of deadlock potentials in multithreaded programs. IBM J. Res. Dev. **54**(5), 3:1–3:15 (2010)
15. Bensalem, S., Havelund, K.: Dynamic deadlock analysis of multi-threaded programs. In: Ur, S., Bin, E., Wolfsthal, Y. (eds.) HVC 2005. LNCS, vol. 3875, pp. 208–223. Springer, Heidelberg (2006). https://doi.org/10.1007/11678779_15
16. Harrow, J.J.: Runtime checking of multithreaded applications with visual threads. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 331–342. Springer, Heidelberg (2000). https://doi.org/10.1007/10722468_20
17. Herlihy, M.: The art of multiprocessor programming (2006)
18. Meisel, J.: Multithreaded programming. EE Eval. Eng. **46**(12), 12–17 (2007)
19. Marchetti-Spaccamela, A., Nanni, U., Rohnert, H.: Maintaining a topological order under edge insertions. Inf. Process. Lett. **59**(1), 53–58 (1996)
20. Pearce, D.J., Kelly, P.H.J.: A batch algorithm for maintaining a topological order. In: Conferences in Research and Practice in Information Technology Series, vol. 102(1), pp. 79–87 (2010)
21. Cormen, T.H.: Introduction to Algorithms. MIT Press, Cambridge (2009)
22. Apache Derby (2016). https://db.apache.org/derby
23. IntelliJ IDEA the Java IDE (2016). https://www.jetbrains.com/idea/
24. dxFeed Market Data (2016). http://www.dxfeed.com/
25. Chan, W.K.: Magiclock: scalable detection of potential deadlocks in large-scale multithreaded programs. IEEE Trans. Softw. Eng. **40**(3), 266–281 (2014)
26. Dl-Check – tool for finding potential deadlocks in Java programs (2016). https://github.com/Devexperts/dlcheck
27. Bruneton, E.: ASM 4.0-A Java bytecode engineering library (2011). http://download.forge.objectweb.org/asm/asm4-guide.pdf. Accessed 18 May 2013
28. Kuleshov, E.: Using the ASM framework to implement common Java bytecode transformation patterns. Aspect-Oriented Software Development (2007)
29. Monson, L.: Caching & weakreferences. JAVA Dev. J. **3**(8), 32–36 (1998)
30. Shiv, K., Chow, K., Wang, Y., Petrochenko, D.: SPECjvm2008 performance characterization. In: Kaeli, D., Sachs, K. (eds.) SBW 2009. LNCS, vol. 5419, pp. 17–35. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-540-93799-9_2
31. Apache Lucene (2010)
32. DaCapo benchmark suite (2009). http://dacapobench.org/