A SURVEY REPORT ON DEADLOCK DETECTION TECHNIQUES

Anurag Sao
cs22m018
Chinmay Badjatya
cs22m036
Swetha R.
cs22m089

ABSTRACT

This survey report presents our study of several deadlock detection techniques, both static and dynamic; implemented for languages such as Java, C/C++ and GoLang. We'll briefly discuss the techniques introduced by 6 different research papers along with their advantages and limitations. We also present a comparative analysis of the techniques covered.

1. INTRODUCTION

- Concurrent programming allows us to take advantage of multi-core processors and parallelise tasks, improving program efficiency. However, this comes with its own set of problems that affect the program quality. Deadlocks make up one such category of problems.
- Deadlocks occur when two or more processes are waiting for a communication call or a resource to be released from another process and end up waiting forever. Deadlocks occur when the following conditions are satisfied in a concurrent program setting - Cyclic wait among processes, no preemption among processes, hold and wait, and mutual exclusion.
- Deadlocks have traditionally been dealt with using the following classes of methods - Deadlock Detection, Deadlock Avoidance, Deadlock Prevention and Deadlock Ignorance. In this text we take a look at several deadlock detection techniques.
- Following are the papers covered over the course of this survey, in the given order - "Using Runtime Analysis to Guide Model Checking of Java Programs"[1], "Detecting Potential Deadlocks with Static Analysis and Run-Time Monitoring"[2], "Dynamic Deadlock Analysis of Multi-threaded Programs"[3],"A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks"[4], "Sherlock: Scalable Deadlock Detection for Concurrent Programs"[5] and "Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis"[6].

2. BACKGROUND

- Deadlock detection techniques allow us to diagnose a program in execution and identify if a deadlock exists (dynamic techniques) or analyze the high level code using techniques such as program analysis, model checking, and formal verification techniques (static techniques); allowing us to identify potential deadlocks before having actually executing the program.
- While static analysis techniques allow us to identify every possible deadlock scenario in a program, they are often hard to implement and often report a high number of false positives. Dynamic analysis identify deadlocks by analyzing a program in execution or a program trace and report the possibility of a deadlock. While dynamic techniques may be more precise, they may also report false negatives.

3. LITERATURE SURVEY

3.1. GoodLock

Havelund et al., in their paper "Using runtime analysis to guide model checking of java programs"[1], introduce a novel dynamic deadlock detection technique named Goodlock. The algorithm detects potential resource based deadlocks in the program using trace obtained from single execution of the program. Goodlock does not detect communication based deadlocks occuring due to use of wait/notify etc.

3.1.1. Algorithm Description

- In the first step input program is executed to generate a trace which is then analyzed to generate a Lock-Tree structure for each thread. Lock tree for a thread represents the nested pattern in which locks are acquired by that thread i.e. if we are analyzing a node on the tree representing a lock, before taking this lock the thread would have acquired all the locks in the parent nodes upto the root of the tree.

- In the next step, Lock trees of each pair of threads are analyzed to see if a lock cycle which can generate a deadlock exists by comparing the nodes starting from root of the lock tree and processuing its children iteratively.

- To reduce false positives the algorithm does not report deadlock cycles which are protected by Gate Locks which act as protective locks taken first by participating threads which prevent lock cycles further down the tree from leading to any potential deadlocks.

- The algorithm detects gate locks by marking the nodes processed as it moves from root to child in previous step and does not consider nodes below marked nodes when looking for lock cycles.

3.1.2. Limitations

- Does not detect deadlocks which may be formed due to lock cycles between more than 2 threads.

- Since Lock Trees are derived from a single run of the program, it may still miss some critical Lock acquisition patterns of participating threads which did not occur during that particular execution.

- Does not verify whether for a Deadlock candidate a valid program input and schedule exists/is feasible which may really lead to Deadlock for that candidate, hence reports a high number of false positives.

3.2. Generalized GoodLock

Agarwal et al., in their paper "Detecting Potential Deadlocks with Static Analysis and Run-Time Monitoring"[2], address the first limitation of the above discussed *GoodLock* algorithm by extending it to a generalized algorithm that can detect deadlocks during concurrent execution of any number of threads. Like GoodLock before it, the algorithm detects deadlocks caused only by locks. Deadlocks caused by the other synchronization constructs like wait/notify and barriers are not handled.

3.2.1. Algorithm Description

A run-time lock graph similar to the one used in *GoodLock* is created from an execution of the program. To handle multi-thread deadlock sequences, additional edges called *inter edges* are added, connecting nodes belonging to different lock trees, both acquiring the same lock. They define a *valid cycle* as a directed cycle D in the graph such that

1. No two consecutive edges in D are inter edges. This reduces the cycle length.

2. Nodes belonging to the same thread form a single subsequence in D, i.e., for each thread T_i, there exists a subsequence P of D such that P contains only nodes of the thread T_i and no nodes of $D - P$ belong to T_i.

A valid cycle is a representation of a potential deadlock, with the thread subsequences depicting circular-wait, the mechanism of locks implementing mutual-exclusion and no-premption, and the order of nodes in the valid cycle defining a possible schedule that would result in a hold-and-wait situation.

An altered DFS algorithm traverses from the root nodes of the lock graph to detect possible valid cycles. The original GoodLock algorithm's concept of gate locks is also handled by checking separately if nodes belonging to different threads in the cycle share a gate lock. A valid cycle without gate locks is declared as a potential for deadlock.

3.2.2. Combination with deadlock typing

The paper further discusses static deadlock type checking as a means of prevention of deadlocks. Deadlock typing imposes lock levels to classes and define rules of lock acquirement. Since the structure of the code is well defined, programs written using lock typing do not run into (many types of) deadlocks. But the focus is on runtime deadlock detection since deadlock types restrict the expression of programs into runnable type-adhering code. The authors device a type inference algorithm that infers typing from static code and combine its inferences with GoodLock to reduce search complexity.

3.3. Dynamic Methods to Detect Valid Deadlock Cycles

In the paper "Dynamic Deadlock Analysis of Multi-threaded Programs"[3] Havelund et al. introduce a dynamic deadlock detection algorithm that akin to Sherlock[5] analyzes a single execution trace.
The algorithm identifies resource deadlocks occurring due to nested synchronized blocks in Java programs. However, there are no checks for communication based deadlocks. The techniques presented are neither sound nor complete but they scale well for large programs and are very effective in practice. While the original implementation works for Java programs, the algorithm itself is usable for most languages providing concurrency support.

3.3.1. Background

Like GoodLock[1] this paper also tries to reduce the false positives reported in the presence of Gate Locks. However, unlike GoodLock, the algorithm is not limited to 2 thread deadlocks. Moreover, using a cyclic graph model, the algorithm also prevents reporting deadlocks due to code segments that

cannot run in parallel, which was not addressed in the generalized GoodLock algorithm[2].
Following categories of deadlocks are handled here -

- Single Threaded Cycles - Prevents false positives reported due to re-entrant locks
- Guarded Cycles - Prevents false positives reported due to presence of gate locks
- Thread Segmented Cycles - Cycles in thread segments that have some HB relation are not reported as deadlocks.

3.3.2. Algorithm Description

A single execution of the program is used to obtain a trace σ of lock and unlock events where each entry is of the form $lock(lineno., thread, obj)$ or $unlock(lineno., thread, obj)$.
A graph of lock events is built using the trace with an edge $(l1, l2)$ denoting that lock l1 was taken before l2, by some thread t. Deadlocks in such a graph are reported on the presence of a cycle.
A lock context $C_L(\sigma, i)(t)$ is defined for a thread 't' as the set of locks it already holds at position 'i' in the given trace and this information added to the lock graph gives us the "guarded lock graph". It is formally defined as follows -

$G_L = (L, W, R)$
L : The set of locks (L_σ)
W : The set of labels, each of the form (thread_id, lock context)
R : Edge information of the form $(l_1, (t, g), l_2)$
where, l_1, g, $l_2 \epsilon\, C_L(\sigma, i)(t)$

A cycle reported is considered a valid cycle if there is no common thread in the labels of the edges - eliminating single threaded cycles - and no two edges in the cycle show two threads holding the same guard locks.

In order to eliminate segmented cycles, start and join events are added to the trace σ. Segments are code portions defined using the ordering of start and join operations. A segmentation context $C_S(i, \sigma)(t)$ is defined as the current segment of the thread 't' and directed segmentation graph is used to establish MHP relations among code segments. Ultimately, the authors define a "Segmented and Guarded Lock Graph" -

$G_L = (L_\sigma, W, R)$ where -
L is defined as before
W is a set of labels each of the form - $(s_1, (t, g), s_2)$ where s_1 is the segment the source lock was taken in and s_2 the segment for target lock.
R is the edge set defined as - $l_1, (s_1, (t, g), s_2), l_2)$. A cycle is considered a valid deadlock cycle only if -

$\forall e_1, e_2 \epsilon R$ such that $e_1 \neq e_2$,
$thread(e_1) \neq thread(e_2)$,
$guards(e_1) \cap guards(e_2) = \phi$
and there exists no MHP ordering among the segments of e_1 and e_2.

3.3.3. Implementation details

This algorithm has been used in the JavaPathExplorer(JPax) tool which was used in 3 NASA case studies to analyze code both in Java and C++. The algorithm might miss some true positives in rare cases when locking patterns in the execution trace don't paint the complete picture. But such cases were reported to be rare.

3.3.4. Advantages and Limitations

The algorithm discussed in this work is more precise than the GoodLock versions[2], [1] discussed above. However, since the algorithm is ultimately based on analysis of a single execution trace, the completeness of the algorithm is not guaranteed.

3.4. Deadlock Fuzzer

In the paper "A Randomized Deadlock Detection Technique for finding Real Deadlocks"[4], Pallavi et al. introduce Deadlock Fuzzer, a technique which uses a variant of Goodlock[1] called iGoodlock as a base to generate potential deadlock candidates amongst which it tries to find and return real deadlocks and the program schedule which may lead to those deadlocks. Since it uses Goodlock as its base, this technique detects only resource based deadlocks.

3.4.1. Algorithm Description

- The first stage of the technique uses a variant of Goodlock algorithm – iGoodlock which generates potential deadlock candidates alongwith relevant context information used in next stage of program to find valid program schedules.
- The second stage of the technique takes as input a deadlock candidate and runs a biased random thread scheduler on the input program to generate a schedule which can lead to the input deadlock cycle with high probability.
- If the technique finds any deadlock candidate with a valid program schedule to arrive at that deadlock, it outputs that deadlock as a real deadlock with corresponding schedule which may lead to that deadlock.

3.4.2. Limitations

- Since Deadlock fuzzer uses Goodlock[1] as a base to detect potential deadlock candidates, if Goodlock's analysis is imprecise and misses any candidate then it will also be missed by Deadlock fuzzer.
- Due to thread scheduler being random, Deadlock fuzzer's results on an input program may contain variations in reported deadlocks across different runs.

3.5. Sherlock

Eslamimehr et al present *Sherlock*[5], a dynamic deadlock detection algorithm for the Java language. The algorithm uses $GoodLock$[1] to rapidly generate a set of deadlock candidates. For each deadlock, the algorithm tries to come up with a set of inputs and a schedule that would result in a deadlock based on concolic execution. The algorithm flags the candidate as a real life deadlock only if it finds fitting inputs and schedule, and hence never makes false positives.

3.5.1. Algorithm

The algorithm uses the following datatypes:

- $Program$ is the Java 6 program to be analysed
- $Input$ is a vector of input values fed to the program
- $Lock$ is a Java 6 object
- $Event$ denotes a statement ($threadId \times statementLabel$)
- $Schedule$ is a sequence of $Event$s
- $Link$ denotes a thread holding a lock while requiring another lock ($threadId \times (statementLabel \times Lock)$ $\times (statementLabel \times Lock)$)
- $Cycle$ is a $Link$ set
- $Deadlock$ is a representation of a deadlock with a $Cycle$ depicting the circular-wait, and the corresponding inputs and schedule ($Cycle \times Input \times Schedule$)

A deadlock candidate is a sequence of events that could potentially lead to a deadlock. A schedule that contains all these events in sequence is a proof of a real life deadlock. The algorithm defines an **execute** function that, for a given schedule and input, checks if the execution of the program results in a deadlock. If not, the function then performs concolic execution of the program to record constrains that would result in the control flow reaching more of the deadlock events (in sequence). It solves the constraints and gives an alternative (better) input sequence.

The algorithm also defines a **permute** function that takes a schedule and alters it to increase the likelihood of a deadlock occuring (corresponding to the deadlock candidate). This is done by solving four types of constraints:

1. α_π represents the program order and happens-before relations imposed by thread forks and joins
2. β_π imposes write-read ordering
3. $\Psi_{(V,E)}$ is the set of constraints on the order of lock acquirement
4. δ_c is the ordering imposed by the deadlock candidate event sequence

where π is an execution trace of the program, c is the deadlock candidate, and (V, E) is lock order graph.

The algorithm performs a series of alternating calls to **execute** and **permute** until a deadlock is detected, or the functions fail to find a better input/schedule.

3.5.2. Implementation

Soot was used to implement the execute function on top of Lime concolic execution engine [7]. They run the algorithm on 22 well known benchmarks of varies sizes ranging from a few thousands to millions of lines of code. The algorithm found 86 previously unknown deadlocks in the benchmarks. The $permute$ function scales to long schedules. Hence the algorithm was able to detect deadlocks occurring after millions of execution steps unlike most algorithms that focus on smaller ranges.

3.5.3. Limitations

- Since Goodlock[1] is used to get the initial set of deadlock candidates, it cannot analyse possible deadlocks missed by GoodLock, neither can it handle deadlocks caused by synchronization constructs like wait/notify.

3.6. Deadlock Detection Using Global Session Graph Synthesis

Ng et al. in their paper "Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis"[6] introduce a static analysis tool - "dingo-hunter" - to identify potential communication based deadlocks at compile time. Go uses a channel based concurrency model based on communicating sequential processes (CSP). The authors make use of "session types" - a formal typing discipline for communication among parallel tasks - goroutines. The analysis for finding potential deadlocks is performed on "Global Session Graphs", described in a later section.

3.7. Background

Concurrency in Go is implemented using lightweight tasks running in parallel and sharing the same address space. These goroutines communicated with each other via typed $channels$. The channels are typed FIFO queues which are synchronous by default.

Session Types - Originating from type theory, session types

are used to ensure correctness in communication in concurrent programs. A concurrent program is a closed communicating system called a session. Each "local type" corresponds to a single instance of a goroutine. A local session type is a control flow graph where each node is one of the following -

- *Channel* ch T - create a channel "ch" of type T
- *Send* ch - send to a channel (denoted by !)
- *Recv* ch - receive from a channel (denoted by ?)
- *Close* ch - close channel
- *Label* - a jump label

CFSM - Communications FSMs are finite state machines where transitions between states are characterized by send or receive operations. The CFSM models used in this works can be categorized as Communication FSM and Goroutine FSMs.
Global Session Graph - A global session graph is derived from the Goroutine and Channel FSMs. The global graph consists of communication nodes showing possible communication among two Goroutine FSMs or choice nodes which cause a split in the graph as a decision needs to be made.

Generalized Multiparty Compatibility - GMC specifies a set of sound and complete conditions for the construction of a global graph. Systems satisfying GMC are safe and deadlock free. GMC specifies 2 conditions -

- Representability - Each trace and choices in CFSM should be reflected in the Global Graph
- Branching Property - Whenever there's branching in the global graph, a unique machine's branch is taken and this decision is propagated to other machines.

3.7.1. Algorithm Description

Figure 1 outlines the algorithm used to build global session graphs.

- **Deriving Local Session Types** - Go's inbuilt SSA IR generator is used to first convert the source code in a simpler format, then the communication statements are used to derive a set of local types for all goroutines.
- **Building CFSMs** - The local types are translated into Channel and Goroutine FSMs. The nodes (communication statements) are translated to events in FSMs. There are no direct one-to-one connections among the FSMs themselves but there does exist a one-to-one link among CFSMs and Go channels. However, instead of simply treating channels as static connectors for CFSMs, the authors model Go channels as FSMs too - in order to represent their state changes.

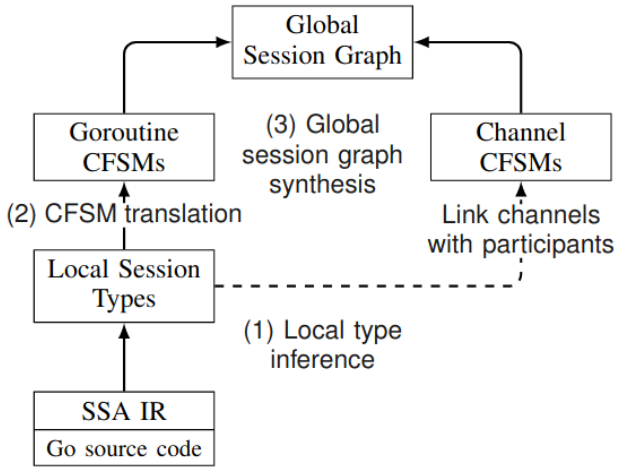

Fig. 1. Workflow of dingo-hunter tool-chain

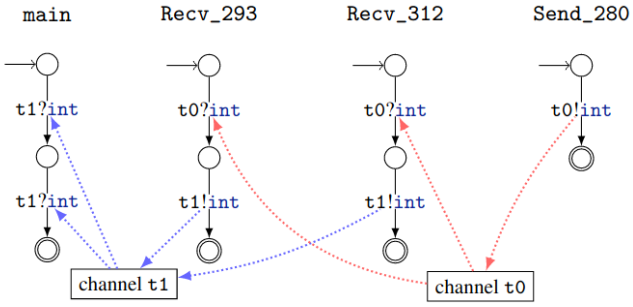

Fig. 2. Goroutine FSMs demonstrated in [6]

- **Global Graph Synthesis** - Having acquired both Goroutine CFSMs and Channel CFSMs, the global graph is built by generating all possible combinations of synchronous labelled transitions of the composed CFSMs. Send and recieve operations in the same channel can be matched and synchronized as a single node in the global graph. Figures 2 and 3 paint a clearer picture.
  Figure 2 shows that channel t0 receives one integer type while two sends are performed. Since Channels are FIFO type queues, this poses a problem. One of the receives is not going to be fulfilled. And since operations on channels are blocking, this leads to a deadlock. This situation is shown by the presence of a branch in the global graph in figure 3.

Upon successful construction of global graph, GMC conditions are checked and if no violations are found, the program is said to be deadlock free.

3.7.2. Implementation Details

The static analysis tool was evaluated using 4 case studies and the "htcat" concurrent HTTP download tool from Heroku.

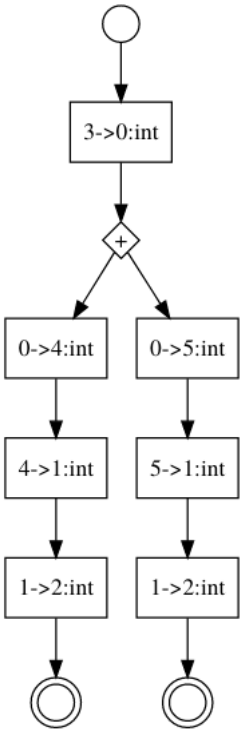

Fig. 3. Global graph showing communications corresponding to Figure 2 [6]

Upon analysis, 19 CFSMs (11 goroutines, 8 channels) were obtained from the Go program, and was reduced from 9632 local session graph nodes extracted.

3.7.3. Advantages and Limitations

- This tool has an advantage over Go's runtime deadlock detector. The inbuilt tool detects a deadlock only when all goroutines running have been stalled. However, if the deadlock is local, it goes undetected.
- Being a static analysis tool, it can report all potential deadlocks at compile time, improving software quality.
- The paper reports this tool being tested only on a single large program. To our knowledge, this tool has not been tested extensively.

4. CONCLUSION

This report covers a brief summary and comparison of 6 deadlock detection techniques, both static and dynamic. The papers covered give deadlock detection algorithms for multiple languages i.e. C++, Java and Go. Some improvements that can be implemented over the current techniques are -

- Havelund's paper[3] while using MHP analysis to identify segments that can't run in parallel, doesn't take communication calls into account. A more complete MHP analysis can be implemented for more sound deadlock detection.
- The generalized Goodlock algorithm [2] can make use of MHP relations from a simple analysis and generate the lock graph with inter edges only between nodes that may happen in parallel. This could bring down the number of cycles significantly, reducing the false positive cases.
- Since DeadlockFuzzer[4] and Sherlock[5] both base their analysis on potential deadlock candidates, the use of a more complete and precise algorithm as the base will automatically improve their performance. Havelund's later work appears to be a good replacement for GoodLock.
- Every trace based algorithm would be benefitted if a symbolic trace representation that preserves the meaning of programs is devised. This would ensure that the algorithm is not dependent on a single instance of execution of the program.
- Since the current version of Goodlock used as the base for other techniques[4][5] uses just a single run of program to generate deadlock candidates, we can run Goodlock multiple times to explore more candidates but to ensure it is more explorative in each run, we can incorporate concolic execution used in execute stage of Sherlock[5] in Goodlock also to drive execution in each run towards previously unexplored branches so that the coverage is more exhaustive.

A literature review involving a wider range of languages and frameworks such as Rust, OpenMP, Scala etc. can be performed further.

5. REFERENCES

[1] Klaus Havelund, "Using runtime analysis to guide model checking of java programs," in *SPIN Model Checking and Software Verification*, Klaus Havelund, John Penix, and Willem Visser, Eds., Berlin, Heidelberg, 2000, pp. 245–264, Springer Berlin Heidelberg.

[2] Rahul Agarwal, Liqiang Wang, and Scott D. Stoller, "Detecting potential deadlocks with static analysis and runtime monitoring," in *Hardware and Software, Verification and Testing*, Shmuel Ur, Eyal Bin, and Yaron Wolfsthal, Eds., Berlin, Heidelberg, 2006, pp. 191–207, Springer Berlin Heidelberg.

[3] Saddek Bensalem and Klaus Havelund, "Dynamic deadlock analysis of multi-threaded programs," in *Hardware and Software, Verification and Testing*, Shmuel Ur, Eyal

Bin, and Yaron Wolfsthal, Eds., Berlin, Heidelberg, 2006, pp. 208–223, Springer Berlin Heidelberg.

[4] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik, "A randomized dynamic program analysis technique for detecting real deadlocks," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2009, PLDI '09, p. 110–120, Association for Computing Machinery.

[5] Mahdi Eslamimehr and Jens Palsberg, "Sherlock: Scalable deadlock detection for concurrent programs," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, New York, NY, USA, 2014, FSE 2014, p. 353–365, Association for Computing Machinery.

[6] Nicholas Ng and Nobuko Yoshida, "Static deadlock detection for concurrent go by global session graph synthesis," in *Proceedings of the 25th International Conference on Compiler Construction*, New York, NY, USA, 2016, CC 2016, p. 174–184, Association for Computing Machinery.

[7] "Lime concolic execution engine," http://www.tcs.hut.fi/Software/lime.