Avoiding Deadlocks using Stalemate and Dimmunix

Surabhi Pandey
IIIT-Bangalore
surabhi.pandey@iiitb.org

Sushanth Bhat
IIIT-Bangalore
sushanth.bhat@iiitb.org

Vivek Shanbhag
IIIT-Bangalore
vivek.shanbhag@gmail.com

ABSTRACT

The execution of a concurrent Java program can deadlock if its threads attempt to acquire shared locks in cyclic order. The JVM permits such behaviour. Research has demonstrated that such deadlocks can be predicted through static analysis. It is also known that a tool like Dimmunix helps to avoid deadlocks whose deadlock patterns (fingerprints) are known. The current work combines both approaches: conducting static analysis to predict possible deadlocks and provide their corresponding fingerprints to Dimmunix. These fingerprints forewarn Dimmunix of all deadlock possibilities rather than it *learn* about them one at a time. For our experiments we use 8 deadlock programs that were developed based upon deadlock predictions from static analysis of the entire JRE by a tool called Stalemate. We design a process to generate Dimmunix fingerprints from deadlock predictions.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Experimentation, Reliability

Keywords

global static analysis, deadlock prediction & avoidance

1. INTRODUCTION

There are many static analysis approaches that can predict deadlocks in Java libraries [4, 3] and programs [2]. The approach described in [3] predicts sets of reachable call stacks that may deadlock if realised concurrently. Dimmunix [1] is a tool that actively prevents *known* deadlocks from remanifesting. Dimmunix must witness a program execution that leads to deadlock. In the process it records the *deadlock fingerprint* which is read and used in subsequent executions to avoid *that* deadlock. So that it can be used during subsequent executions, *dynamic* aspects of the deadlocked state are abstracted away when recording the *fingerprint*. It turns out, the format and information content of the Dimmunix fingerprint is such that it can be generated through static analysis. In the current work we use Stalemate [5], an improvement over [3]. Generating fingerprints through static analysis is a way to forewarn Dimmunix of all deadlock possibilities rather than it have to *learn* about them, one at a time until it is completely *vaccinated*. Thus making is unnecessary for the application to ever deadlock.

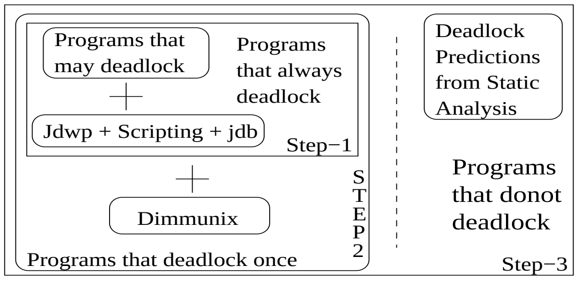
Dimmunix: Its Java prototype comprises of a Java agent, and the *deadlock avoidance strategy (das)*. The Java agent installs a bytecode transformer which employs the instrumentation API to engineer bytecode of classes loaded during application execution, inserting calls to *das* at appropriate points: before and after every `monitorenter` and also before every `monitorexit`. *das* maintains an *Resource Allocation Graph (RAG)* whose nodes represent locks & threads and directed edges represent the acquired / requested relations amongst them. If the execution reaches an unknown deadlock, the RAG is used to construct a deadlock fingerprint that is recorded. Subsequent executions read the fingerprints, and based on how the RAG evolves, *das* may block the acquisition of an (otherwise) available lock by some thread, to avoid a known deadlock.

Stalemate: Deadlock predictions by this tool [5] are of the form $< T_1, ...T_i, ...T_k > \equiv Ts_1, ...Ts_i, ...Ts_k$, where $T_i, \forall 1..k$ are types. In general, Ts_i is a set of possible thread stack fragments along each of which (at least) two locks are acquired in a nested fashion: the first being an object of type T_i, and the last of type $T_{(i+1)\ mod\ k}$. A 2-cycle prediction is interpreted as: given an application whose one thread realises a member of Ts_1 and another thread realises a member of Ts_2 concurrently and if they share the locks, then they could deadlock. Stalemate publishes few Java programs that deadlock as predicted by the static analysis of the JRE. The current work uses 8 of these programs.

2. STALEMATE + DIMMUNIX

Starting with the 8 programs from [5] that sometimes deadlock, our three step approach proceeds as follows: In step 1 we externally cause the JVM to select amongst only such thread-interleavings that always leads execution to deadlock. In step 2 we incorporate Dimmunix into the framework and verify that when program execution does deadlock,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE '14, May 31 - June 7, 2014, Hyderabad, India
Copyright 14 ACM 978-1-4503-2768-8/14/05 ...$15.00.


Programs that may deadlock
Programs that always deadlock
Jdwp + Scripting + jdb
Step−1
Dimmunix
STEP2
Programs that deadlock once
Deadlock Predictions from Static Analysis
Programs that donot deadlock
Step−3

Dimmunix records the deadlock fingerprint and also that all subsequent executions do not similarly deadlock. Step 3 describes the Dimmunix fingerprint structure, and how Stalemate may generate deadlock fingerprints for Dimmunix so deadlock is avoided in all executions of the programs.

Step 1: Given a Java program that sometimes deadlocks, we want to deadlock its every execution. In general, for an n-thread n-lock deadlock program, if the JVM interleaves the execution of its n threads such that each of them acquires one lock, then all such executions will proceed to deadlock. Since the 8 programs deadlock as predicted, from their stack predictions we select execution points at which only one of the two locks would have been acquired. For the JVM to allow only such interleavings, we replace `java` by `jdb` to oversee the execution of the program. `jdb` allows program execution to be stopped at breakpoints and witnessed. Its threads can be suspended and resumed. We have developed a script-assisted framework that forces the above choice of thread-interleavings upon the JVM. Every such execution within this framework does deadlock.

Step 2: When Dimmunix is allowed to witness the first program execution in this framework, its deadlock detection and deadlock fingerprint recording capability is triggered, resulting in an update to the history file. Subsequent executions of the same programs do not deadlock since Dimmunix prevents it. We are able to verify / demonstrate both these modes of operation of Dimmunix.

Question: Is it possible to generate the deadlock fingerprint, given the deadlocking program and its corresponding deadlock prediction fetched from static analysis? The answer is yes. The details are in Step 3 below.

Step 3: This step ascertains that deadlock predictions from Stalemate can be converted to Dimmunix fingerprints.

Deadlock Predictions from Stalemate: A stack trace predicted by Stalemate is a *hyphen separated* sequence of code fragment identifiers. A *Code fragment* may be either an entire method body or just a `synchronized` statement block. The entire method body is identified using its method signature. A `synchronized` statement block is identified using a *#-delimited* concatenation of the containing method signature, the lock type, and an integer value (for uniqueness).

Stalemate uses method signatures to identify execution points in the code. Whereas, Dimmunix fingerprints use the *ClassNm.MethodNm (fileNm:LineNum)* format. Given (say) that `A.m()` invokes `B.n()`, one can fetch the Dimmunix fingerprint corresponding to this invocation of `B.n()` as follows: looking up the bytecode index of the `invoke`ation of `B.n()`, in the `LineNumberTable` corresponding to the method body for `A.m()` fetches the required *LineNum.* The class header for `A`.class contains the *fileNm. ClassNm.MethodNm* is `A.m`.

Dimmunix fingerprints: The Dimmunix history file records one deadlock fingerprint per line, for every uniquely reachable deadlock witnessed. For an n-thread deadlock, the fingerprint comprises of a *semicolon separated* serialisation of the fragments of *participating thread* stacks that lead up to the acquisition of the first *participating lock* for that thread. The serialisation of the thread stack is a *comma separated* list of stack trace elements. The *stack trace element* format, as described above, is identical to that in the output of the JVM's deadlock detection component.

3. RESULTS

All the 8 deadlock programs passed Step 1. Six of them passed step 2. The manual step 3 of verifying the rewriting of stack traces using the corresponding 'LineNumber Table' also succeeds for these 6 programs.

API	Program	BugId	step1	step2	step3
MIDI:	RtsTc	8010771	Pass	Pass	Pass
Musical	TcDp1	8031702	Pass	Pass	Pass
Instrument	TcDp2	8031698	Pass	Pass	Pass
Digital I/f	RtsTcDp	8031699	Pass	Pass	Pass
Corba	TcPdh	8024334	Pass	Pass	Pass
	TcEdh	Awaited	Pass	Pass	Pass
Annotation	JlcAt	6588239	Pass	Fail	Fail
Security	RcCt	7118809	Pass	Fail	Fail

The Annotation & Security API deadlocks fail step 2 because Dimmunix is unable to instrument `java.lang.Class` and `java.util.Hashtable`, respectively. Adding deadlock fingerprints into the Dimmunix history file based upon the corresponding predictions also does not help it to avoid the deadlocks: Since those classes are not instrumented, their methods acquire and release locks without updating the RAG, and bypassing *das* component of Dimmunix. More than 425 JRE-classes (`java.lang.Class` & `java.util.Hashtable` included) are loaded by the JVM even before the Java agent is *preloaded.* As a result, deadlocks involving their methods cannot be avoided by this approach.

4. CONCLUSIONS

Dimmunix explores an effective approach to deal with known deadlocks in applications. We demonstrate how its usefulness can be enhanced by providing it fingerprints fetched from static analysis. The two failure cases expose the subtle limitations of the engineering approach employed by its current implementation.

Exploring the scalability of the Dimmunix approach to deadlock exploits predictable through static analysis of legacy libraries (like JRE) can be useful. Thus making it unnecessary to continue viewing them as bugs that require fixing.

5. REFERENCES

[1] D. T. Horatiu Jula, C. Zamfir, and G. Candea. Deadlock immunity:enabling systems to defend against deadlocks. In *USENIX OSDI*, December 2008.

[2] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *ICSE*, 2009.

[3] V. K. Shanbhag. Deadlock-detection in java-library using static-analysis. In *15th APSEC*, 2008.

[4] A. Williams, W.Thies, and M.Ernst. Static deadlock detection for java libraries. In *ECOOP*, 2005.

[5] https://github.com/vivekShanbhag/Stalemate