How Good is Static Analysis at Finding Concurrency Bugs?

Devin Kester, Martin Mwebesa, Jeremy S. Bradbury
Software Quality Research Group
Faculty of Science (Computer Science)
University of Ontario Institute of Technology
Oshawa, Ontario, Canada
devin.kester@mycampus.uoit.ca, {martin.mwebesa, jeremy.bradbury}@uoit.ca

***Abstract*—Detecting bugs in concurrent software is challenging due to the many different thread interleavings. Dynamic analysis and testing solutions to bug detection are often costly as they need to provide coverage of the interleaving space in addition to traditional black box or white box coverage. An alternative to dynamic analysis detection of concurrency bugs is the use of static analysis. This paper examines the use of three static analysis tools (FindBugs, JLint and Chord) in order to assess each tool's ability to find concurrency bugs and to identify the percentage of spurious results produced. The empirical data presented is based on an experiment involving 12 concurrent Java programs.**

***Keywords*-static analysis, concurrency, data race, deadlock, empirical software engineering.**

I. Introduction

The widespread adoption of multicore processors has lead to an increase in demand for concurrent software that can exploit the performance gains possible from utilizing more than just a single core. For example, the Java programming language, which has primarily been used to write sequential programs is now often used to write multithreaded programs using synchronization, locks, semaphores, barriers and exchangers. All of these language features aim to enhance the sharing of data between threads in a Java program.

Although using concurrency can be beneficial with respect to performance it can also be extremely difficult to do correctly. The difficulty with concurrent programming has been discussed since before multicore technology became mainstream. The importance of this problem is reflected in the following quote *"Industry needs help from the research community to succeed in its recent dramatic shift to parallel computing. Failure could jeopardize both the IT industry and the portions of the economy that depend on rapidly improving information technology"* [1]. One of the major issues with concurrent programming is that it can lead to kinds of bugs that are not possible in sequential programs. For example, a concurrent program that shares data across multiple threads can contain a bug that leads to a data race or deadlock. In a language like Java, the interleaving space of a concurrent program consists of all possible thread schedules [2]. Detecting concurrency bugs like data races or deadlocks is a challenging software quality assurance problem because a particular bug may only exhibit itself in one or a few interleavings of a program - such bugs are often referred to as heisenbugs [3].

Although concurrency bugs are difficult to detect there are tools that can significantly enhance the ability to find possibly obscure concurrency bugs. One category of these tools are static analysis tools like FindBugs [4], [5], JLint [6], [7], RacerX [8], RELAY [9] and Chord [10], [11], [12]. Static analysis tools are typically considered to be fast since execution of the program is not required. However, the use of these tools has a potential for spurious analysis results which can be time consuming to identify. Despite this shortcoming, the effectiveness of static analysis tools in finding concurrency bugs can be clearly witnessed in the case of Coverity, a very popular static analysis tool used in industry [13]. Another category of tools are dynamic analysis tools which include testing and model checking tools like ConTest [14], [15], CalFuzzer [16], CHESS [17] and Java PathFinder [18], [19]. The dynamic analysis tools all involve executing the program and in general are immune from spurious results. Most dynamic concurrency bug detection tools include a mechanism for executing different thread interleavings in order to provide increased confidence in the quality of the source code.

An important factor to consider when discussing static and dynamic analysis tools is the increasing size and complexity of programs under analysis. The size of concurrent software offers a major challenge to the effectiveness of both static and dynamic tools for detecting concurrency bugs [20]. The successfully use of tools like Coverity and FindBugs in industry provides evidence that static analysis tools can scale to larger industrial software systems.

The goal of our research is to focus on the static analysis class of bug detection tools. In particular, we are interesting in answering the following questions:

- *How effective are existing static analysis tools at detecting concurrency bugs?*
- *What is the rate of false positives (spurious results) in existing static analysis tools?*

In order to answer the above questions we have conducted an experiment which compares three existing tools that are all capable of detecting concurrency problems in Java source

Tool	Description	Input	Output	Interface	Kinds of static analysis performed?	User defined properties or analysis?	False positives?
FindBugs	General purpose static analysis tool that includes concurrency patterns	Java bytecode	text, html, xml, xdoc	command-line, GUI, Eclipse plugin	pattern matching, local data flow analysis	Yes – user defined patterns (written in Java)	Yes
JLint	Static analysis tool that includes detection of deadlocks, race conditions and wait-nosync	Java bytecode	text	command-line, Eclipse plugin (Lint4J)	dataflow analysis, lock graph analysis	No	Yes
Chord	Concurrency-specific analysis tool	Java bytecode	Html	command-line	thread-escape analysis, alias analysis, lock analysis, call graph analysis	Yes – user defined dynamic analysis (written in Java)	Yes

Table I: Summary of Bug Detecting Tool Properties

code. The three tools are FindBugs, JLint and Chord. In our experiment we use a set of 12 test programs to assess the effectiveness of detecting actual concurrency bugs and we measure the rate of false positives produced by the tools. In the next section we will provide a brief overview of each tool included in our study. In Section III we explain our experimental setup and in Section IV we present the results of our research. Threats to validity are described in Section V and related work is described in Section VI. Finally in Section VII we present our conclusions and future work.

II. Static Analysis Tools

In our experiment we selected three static analysis tools to compare: FindBugs [4], [5], JLint [6], [7] and Chord [10], [11], [12]. A high level comparison of the tools is presented in Table I. We selected these tools because they all vary in terms of the kind of analysis used – thus providing a interesting subset of all of the techniques available. We will now discuss the static analysis techniques used in FindBugs, JLint and Chord.

A. FindBugs

FindBugs is a program which uses static analysis to inspect Java bytecode for occurrences of bug patterns – a code instance that might be an error. Bug patterns can arise from a number of different situations such as: misuse of language features; misunderstood API methods; misunderstood invariants during maintenance; and simple typographical mistakes [4].

FindBugs uses the Byte Code Engineering Library (BCEL) and the ASM bytecode framework to analyze, create and manipulate Java class files in order to find bug pattern matches in Java programs [4]. From a user perspective, FindBugs can be utilized in several different ways including as a stand-alone command-line tool, a stand-alone GUI tool and as a plugin for the Eclipse development environment. FindBugs provides the user with very well structured and well defined output and allows for bug pattern warnings to be generated in a variety of formats including HTML, which was used in our study.

By design, FindBugs is a general purpose static analysis tool and can find many different kinds of bug patterns. We focus on the category of multithreaded bug patterns (see Table II) and do not consider other kinds of patterns in our study.

Code	Description
DC	Possible double check of field
DL	Synchronization on Boolean could lead to deadlock
DL	Synchronization on boxed primitive could lead to deadlock
DL	Synchronization on interned String could lead to deadlock
DL	Synchronization on boxed primitive values
Dm	Monitor wait() called on Condition
Dm	A thread was created using the default empty run method
ESync	Empty synchronized block
IS	Inconsistent synchronization
IS	Field not guarded against concurrent access
JLM	Synchronization performed on Lock
LI	Incorrect lazy initialization of static field
LI	Incorrect lazy initialization and update of static field
ML	Synchronization on field in futile attempt to guard that field
ML	Method synchronizes on an updated field
MSF	Mutable servlet field
MWN	Mismatched notify()
MWN	Mismatched wait()
NN	Naked notify
NP	Synchronize and null check on the same field.
No	Using notify() rather than notifyAll()
RS	Class's readObject() method is synchronized
RV	Return value of putIfAbsent ignored
Ru	Invokes run on a thread (did you mean to start it instead?)
SC	Constructor invokes Thread.start()
SP	Method spins on field
STCAL	Call to static Calendar
STCAL	Call to static DateFormat
STCAL	Static Calendar
STCAL	Static DateFormat
SWL	Method calls Thread.sleep() with a lock held
TLW	Wait with two locks held
UG	Unsynchronized get method
UL	Method does not release lock on all paths
UL	Method does not release lock on all exception paths
UW	Unconditional wait
VO	A volatile reference to an array doesn't treat the array elements as volatile
WL	Sychronization on getClass rather than class literal
WS	Class's writeObject() method is synchronized but nothing else is
Wa	Condition.await() not in loop
Wa	Wait not in loop Multithreaded correctness

Table II: Multithreaded patterns detected by FindBugs [4]

B. JLint

JLint is similar to FindBugs in that it statically inspects Java bytecode and performs syntax and semantic verification. The JLint tool was originally written by Konstantin Knizhnik and was further extended by Cyrille Artho [6], [7]. We use the extended version of JLint in our study. We have several interface choices for JLint and we have decided to use the command-line interface of JLint in our research.

JLint performs both local and global data flow analysis along with building lock dependency graphs for class dependencies. Data flow analysis allows JLint to calculate possible local variable values and catch redundant or suspicious calculations. More importantly for this study, the use of lock dependency graphs allows JLint to detect situations which can cause synchronization errors [6]. Furthermore, JLint uses semantic verification via data flow analysis and lock dependency graphs to produce warnings under various synchronization categories, such as: deadlock; raceCondition; and waitNoSync. These categories can be further subdivided into specific warnings that are identified by JLint [6].

C. Chord

Chord is a newer static analysis tool that was designed and built specifically to detect concurrency bugs including data races [21] and deadlocks [12]. There are 4 kinds of static analysis used together in Chord to detect problems with concurrency:

1) *Call-graph analysis (also known as multi-graph analysis):* It makes representations of calling relationships between sub-routines in the program being analyzed.
2) *Alias analysis:* It determines if an object is accessed by more than one thread. Two threads are considered aliased if they point to the same referenced object.
3) *Thread-escape analysis:* It determines whether a referenced object can be restricted to a specific thread without escaping to another thread.
4) *Lock analysis:* It analyzes locks in the code to ensure that the locking and unlocking of critical sections in the code has been done correctly.

The above analysis techniques were included in Chord because they are both scalable and precise with respect to the detection of concurrency bugs. Prior to using the above analysis techniques Chord determines the analysis scope – the reachable classes and methods. Analysis scope can be detected statically or dynamically however we have chosen to use the default static analysis technique known as rapid type analysis (rta).

III. Experimental Setup

We will now discuss the experimental setup of our research. Specifically, we will define the purpose of our experiment, the selection of example programs used in our comparison, and the experimental procedure we followed.

A. Purpose

Recall that we have two research goals for this study: we want to determine the effectiveness of using the three static analysis concurrency bug detecting tools and we want to measure the amount of spurious results produced by each tool. In general, we would like to be able to generalize our results to better understand the benefits of using static analysis to find concurrency bugs.

In addition to our primary purpose we will also try to provide some insight from a user perspective on the tools themselves. In particular:

- usefulness of tool report output; and
- tool usability in terms of the general ease of use.

B. Example Program Selection

In order to satisfy our goals and determine the effectiveness of FindBugs, JLint and Chord we needed to select a set of example programs that would be independent and provide an unbiased comparison of the three tools. After considering several possible sources for example programs we decided to use 12 example programs provided by the developers of two dynamic analysis tools: Java PathFinder and ConTest. To the best of our knowledge these programs were not used by any of the developers of the static analysis tools evaluated in our study.

Java PathFinder is an open source software model checking tool developed by NASA [19]. We selected the following 6 programs (each with a single documented concurrency fault) from the examples that were included with the Java PathFinder source distribution:

- two different implementations of a classic deadlock error *(deadlock.d1, deadlock.d2)*
- two different implementations of the dining philosophers program, each exhibiting starvation/deadlock faults *(diningPhilosophers.dp1, diningPhilosophers.dp2)*
- two different implementations with race conditions *(race.r1, race.r2)*

ConTest is a concurrency testing tool developed by researchers at IBM's Haifa Lab [14]. Unlike Java PathFinder which includes its own scheduler to explore different thread interleavings, ConTest inserts random delays before and after synchronization points in order to increase the chance that a different thread interleaving will be explored each time the program is executed. The developers of ConTest also maintain an IBM Concurrency Benchmark [22] which was the source of our other 6 example programs. Again each of these programs exhibit a single documented concurrency fault, with the exception of one program, an airline ticket sales application, in which two faults were present. The programs selected from the IBM Concurrency Benchmark were:

Warning Detected	Priority	Description of Warnings	Warning Location	Remarks	Actual Bug?
Java PathFinder example: deadlock.d1					
No warnings	-	-	-	-	**-**
Java PathFinder example: deadlock.d2:					
SWL	MED MED	SWL: Sleep with lock held Deadlock2$1.run() calls Thread.sleep with lock held →could result in poor performance or deadlock Deadlock2$2.run() calls Thread.sleep with lock held →could result in poor performance or deadlock	Deadlock2.java [line 34] Deadlock2$1.run() Deadlock2.java [line 55] Deadlock2$2.run()	Deadlock2$1.run() synchronizes resource1 and pauses before synchronizing resource2. This pause allows for resource 2 to be acquired by another process causing deadlock. Bug indirectly found. Deadlock described by bug is not valid.	**Yes**
Java PathFinder example: diningPhilosophers.dp1					
No	LOW	No: Using notify rather than notifyAll in diningPhilosophers.dp1.Fork.release(int) → only wakes one thread, may not wake the thread waiting for condition that is satisfied	Fork.java [line 53] Fork.release(int)	Notify wakes one thread allowing it to access the fork to grab it. All conditions are waiting for fork so this is invalid bug	**No**
Java PathFinder example: diningPhilosopthers.dp2					
No warnings	-	-	-	-	**-**
Java PathFinder example: race.r1					
Wa	MED	Wa: Wait not in loop in race.r1.Event.wait_for_event() → if monitor uses multiple conditions, the monitor caller waits for may not be one intended	OldClassic.java [line 80] Event.wait_for_event()	Wait() called on monitor which does have multiple conditions. signal_event and wait_for_event are unsynchronized	**Yes**
Java PathFinder example: race.r2					
No warnings	-	-	-	-	**-**

Table III: Detailed FindBugs results for the Java PathFinder example programs

- three programs with deadlocks *(account, deadlock, deadlockexception)*
- three programs containing weak reality synchronization faults which are caused by improper synchronization *(airlineTickets, allocationvector, boundedbuffer)*

We verified the presence of the documented concurrency faults in our example programs by testing each program using both ConTest and Java PathFinder. This additional analysis allows us to increase our confidence in our assessment of the effectiveness of the three static analysis tools.

C. Experimental Procedure

Our experimental procedure involved three primary steps:

1) *Dynamic analysis preprocessing.* As mentioned in Section III-B we confirmed that the concurrency bugs documented in our example programs were indeed present using both Java PathFinder and ConTest. During this step we were also looking to identify any additional bugs that may have been overlooked and not documented by the authors of the programs.
2) *Static analysis using FindBugs, JLint and Chord.* After confirming that the example programs contained real concurrency faults we wrote scripts to automate the analysis of the 12 example programs with each of our three tools. All tools were run using their default settings as described in the user documentation.
3) *Assessment of the static analysis output produced by FindBugs, JLint and Chord.* Finally, after automatically analyzing all of our programs with each tool we considered each warning produced by the tools and attributed them to a known bug or identified them as spurious results. The assessment step was conducted by hand and the details and rational for our assessment will be discussed in Section IV.

D. Experimental Environment

The experimental environment chosen for this study was a single laptop machine containing a dual core Intel T7200 2.0GHz processor with 2GB of RAM.

IV. Results

This section describes the results of our study comparing the static analysis tools: FindBugs, JLint and Chord. For each tool we will provide a discussion of the output and an analysis of the results. We will also consider the tool's limitations and the usability of the tool.

A. Analysis using FindBugs

Tool output and analysis results. After running FindBugs on all 12 example programs from both Java PathFinder and the IBM Concurrency Benchmark, a total of 38 possible bugs were detected – this does not include the general purpose bug warnings which were discarded. Of the 38 bug warnings, only 12 of the warnings were matches to actual

Warning Detected	Priority	Description of Warnings	Warning Location	Remarks	Actual Bug?
IBM Concurrency Benchmark: account					
IS	MED	IS: Inconsistent synchronization of Account.amount; locked 63% of time -Mixed synch/unsynch access -one locked access by class own method -unsych access no more then 1/3 → method intended to be thread safe is not synchronized	Account.java Account.amount Unsyncrhonized access: Account.java [line 25] Main.java [line 78,81]	Account.java [line 25] shows unsynchronized access to account.amount Shows interleaving of access which is not locked/thread safe	**Yes**
IBM Concurrency Benchmark: allocationvector					
No warning	-	-	-	-	**-**
IBM Concurrency Benchmark: boundedbuffer					
IS (x3)	MED	IS: Inconsistent Synchronization -Buffer._BUFSIZE -Buffer._bufArr -Buffer._consoleOut → unsynchronized access of thread safe fields	Buffer.java	The accesses to these methods only occur during construction, do not need to be synchronized (thread safe)	**No**
No (x2)	LOW	No: Using Notify rather than NotifyAll() -Buffer.deq(String) -Buffer.enq(String) → only wakes one thread, may not wake correct thread waiting on condition	Buffer.java Line 149 Line 126	When using Notify(), a single arbitrary thread is waken. For the boundedbuffer problem, this may wake the incorrect thread waiting to be activated on the condition of needing consumers or needing producers. This can result in a deadlock	**Yes**
IBM Concurrency Benchmark: deadlock					
SC	MED	SC: new deadLock(String, String) invokes Thread.start() → likely to be wrong if the class is ever extended/subclassed, since the thread will be started before the subclass constructor is started.	deadLock.java [line 39] deadLock(String,String) calls Thread.start()	Simple program, main creates new class which creates threads and runs them inside constructor. Run methods writes between two files concurrently	**No**
IBM Concurrency Benchmark: deadlockexception					
No warning	-	-	-	-	**-**
IBM Concurrency Benchmark: airlineTickets					
SC	MED	SC: new deadLock(String, String) invokes Thread.start() → likely to be wrong if the class is ever extended/subclassed, since the thread will be started before the subclass constructor is started.	airlineTickets.java [line 61] airlineTickets(String,String) calls Thread.start()	Simple program, main creates new class which creates threads and runs them inside constructor. Run methods writes between two files concurrently	**No**

Table IV: Detailed FindBugs results for the IBM Concurrency Benchmark programs

multithreaded bugs present in the programs. Tables III and IV describe each of the 12 multi-threaded warnings, as well as a discussion of each warning based on a manual code inspection and analysis.

Table V shows a summary of the FindBugs results across all of the programs. From the table we can see that, of the 12 multi-threaded warnings provided by FindBugs, only 6 were determined via manual inspection to be real bug detections. Also, of the 13 known concurrency bugs exhibited by the 12 programs, only 30.77% of the bugs were successfully identified by the FindBugs tool. The FindBugs analysis of each test program took between 7 and 14 seconds of CPU time.

Tool limitations. It was found that FindBugs had a difficult time determining deadlock conditions unless synchronized methods were used inconsistently. When programs use synchronization inconsistently, FindBugs did a good job of reporting the inconsistent usage, as well as determining where the inconsistency occured and how much of the time a resource was locked. FindBugs also was able to identify incorrect usages of wait and notify statements which were used in synchronized blocks to wait for and obtain access to an object's lock.

Total Programs Run	12
Total Warnings	39
Total Multithreaded Warnings	12
Total Warnings Exhibiting Real Bugs	6
Total Known Bugs Successfully Found	4
Total Known Bugs Not Found	9
Effectiveness of Finding all Known Bugs	30.77%
Effectiveness of Warnings being Real Bugs	50.00%

Table V: Summary of results for all example programs analyzed using the FindBugs Tool

FindBugs was not able to detect any of the starvation bugs located in the dining philosopher examples. Due to the nature of static analysis tools it also could not recognize any incorrect program interleavings which resulted in incorrect output being displayed by the program.

FindBugs found multiple false positive concurrency bugs patterns. For each bug pattern detected, FindBugs provides

Warning Description	Warning Location	Remarks	Actual Bug?
Java PathFinder example: deadlock.d1			
..\ConcurrencyBugExamples\deadlock\d1\deadlock\d1\Deadlock.java:40: Loop 1: invocation of synchronized method deadlock/d1/Deadlock.foo() can cause deadlock.	Deadlock.java Line 40	Deadlock occurs when Deadlock.foo() is called	**Yes**
Java PathFinder example: deadlock.d2			
No warning	-	-	**-**
Java PathFinder example: diningPhilosophers.dp1			
No warning	-	-	**-**
Java PathFinder example: diningPhilosophers.dp2			
No warning	-	-	**-**
Java PathFinder example: race.r1			
No warning	-	-	**-**
Java PathFinder example: race.r2			
..\ConcurrencyBugExamples\race\r2\race\r2\Racer.java:6: Method race/r2/Racer.run() implementing 'Runnable' interface is not synchronized.	Racer.java Line 6	Runnable interface not synchronized, identifies the interleaving of commands that should be synchronized	**Yes**

Table VI: Detailed JLint results for the Java PathFinder example programs

Warning Description	Warning Location	Remarks	Actual Bug?
IBM Concurrency Benchmark: account			
No warning	-	-	**-**
IBM Concurrency Benchmark: airlineTickets			
..\BugBenchmark\airlineTickets\airlineTickets\bug.java:95: Method airlineTickets/bug.run() implementing 'Runnable' interface is not synchronized.	bug.java Line 95	Bug.run() is not synchronized. This lack of synchronization allows a ticket to be sold by a thread and then another ticket from a different thread to be sold (over max tickets) before the flag for stopping sales is set by the first thread.	**Yes**
IBM Concurrency Benchmark: allocationvector			
No warning	-	-	**-**
IBM Concurrency Benchmark: boundedbuffer			
..\BugBenchmark\boundedbuffer\boundedbuffer\Buffer.java:116: Method wait() can be invoked with monitor of other object locked. ..\BugBenchmark\boundedbuffer\boundedbuffer\BufferNotify.java:284: Call sequence to method boundedbuffer/Buffer.enq(java.lang.Object) can cause deadlock in wait(). ..\BugBenchmark\boundedbuffer\boundedbuffer\Buffer.java:138: Method wait() can be invoked with monitor of other object locked. ..\BugBenchmark\boundedbuffer\boundedbuffer\BufferNotify.java:302: Call sequence to method boundedbuffer/Buffer.deq(java.lang.String) can cause deadlock in wait().	Buffer.java Line 116 BufferNotify.java Line 284 Buffer.java Line 138 BufferNotify.java Line 302	Program uses Notify, rather than NotifyAll method which causes incorrect thread to wake upon a condition.	**Yes**
IBM Concurrency Benchmark: deadlock			
..\BugBenchmark\deadlock\deadlock\deadLock.java:61: Method deadlock/deadLock.run() implementing 'Runnable' interface is not synchronized.	deadlock.java Line 61	deadlock.run() is not synchronized. This method is not needed to be synchronized. The bug occurs in the write method due to incorrect synchronization of file locks	**No**
..\BugBenchmark\deadlock\deadlock\deadLock.java:75: Field 'hash' of class 'deadlock/deadLock' can be accessed from different threads and is not volatile.	deadlock.java Line 75	Field hash can be accessed from different threads but is not part of the synchronization errors displayed in the program. The 'hash' field was created to allow for the monitoring of a deadlock situation between file writing	**No**

Table VII: Detailed JLint results for the IBM Concurrency Benchmark programs

Total Programs Run	11
Total Warnings	31
Total Multithreaded Warnings	9
Total Warnings Exhibiting Real Bugs	7
Total Known Bugs Successfully Found	4
Total Known Bugs Not Found	8
Effectiveness of Finding all Known Bugs	33.33%
Effectiveness of Warnings being Real Bugs	77.78%

Table VIII: Summary of results for all example programs analyzed using the JLint Tool

a description of when each warning may cause a bug. Therefore the user need to determine whether or not the program is actually running in the way FindBugs describes and whether or not the bug pattern identifies a real bug. For example, FindBugs provided the warning *SC: new Class_constructor() invokes Thread.start()*. The description of this warning states that a bug is likely to occur if the class is ever extended or subclassed. For each instance that this bug was detected during this study, this was never found to be the case and a real bug was not detected.

Tool usability. We used the command line version of FindBugs and our usability observations are based only on this version. Overall, we found the warnings report to be very well structured. The report first displayed project information and metrics, detailing a list of the classes analyzed, followed by the number of packages, classes, and lines of code in the project, as well as the total number of potential bugs found. FindBugs next displayed a list of the kinds of bug patterns that were found. The bug pattern categories are provided as hyperlinks, thus allowing a user to proceed directly to the specific bug warning information if necessary. Next, FindBugs displayed a bug warning summary, describing the number of bug patterns found for each category, before displaying all of the detailed information discovered for each potential bug. This detailed information was sorted by the kind of bug pattern that each warning is an instance of. Finally, the report concludes with a description of each kind of bug pattern that was found, along with a possible scenario that may have caused the warning.

B. Analysis using JLint

Tool output and analysis results. After running JLint on all the test programs, except the deadlockexception program which caused an error in JLint, a total of 31 possible bugs were detected. Of the 31 bug warnings, only 9 of the warnings described possible multithreaded bugs (determined from bug warning descriptions). Tables VI and VII describe each of the 9 multithreaded warnings, as well as some remarks about the warnings based on a manual inspection of the code and a comparison with the Java PathFinder and Contest results.

Table VIII shows summary statistics of the 9 multi-threaded warnings provided by JLint. Of the 9 warnings, 7 (over 75%) were positive bug findings. Also, of the 12 known concurrency bugs in the 11 example programs, JLint only successfully identified 33.33% of the bugs. The results show that JLint yields less false positive results than FindBugs; however, this static analysis tool still only finds a small number of the known concurrency bugs presented in the example programs. The JLint analysis of each test program took less than 1 second of CPU time.

Tool limitations. From the study results, it was found that similar to FindBugs, JLint had a hard time detecting most deadlock bugs in the example programs. JLint was not effective for determining deadlocks and concurrency errors in this study. For example, JLint uses lock graphs in the analysis, however, it was still unable to detect the synchronization error in two implementations of the classic dinning philosopher's problem. JLint did successfully catch the incorrect usage of wait and notify statements from the boundedbuffer example program and the warning description was more detailed than FindBugs, making it much easier to recognize this warning as a real bug.

Tool usability. Our discussion of usability is based on our experience using the command-line version of JLint. In general, JLint produced descriptive warning messages that made it easy to determine the cause of the warnings. However, complete descriptions for each warning could only be found by referencing the JLint User Manual. The output provided by JLint was not well structured, making this tool difficult to use when analyzing many classes or when many bug warnings were generated.

C. Analysis using Chord

Tool output and analysis results. After running Chord on all of the example programs a total of 8 warnings and 31 different thread schedules were produced. All 8 of the warnings described possible data race bugs and none of the warnings referred to possible deadlocks. Tables IX and X describe the warnings and comment on their accuracy based on a manual code inspection.

Table XI shows summary statistics of the 8 multi-threaded warnings produced by Chord for our 12 example programs. All 8 of the warnings were identified as referring to actual data race bugs, however, of the 13 know concurrency bugs only 4 were detected (30.77%). When compared with Find-Bugs and JLint, the results show that Chord finds a similar number of the known concurrency bugs but without the false positives. The Chord analysis of each test program took longer than the analysis using FindBugs or JLint. Analysis CPU times for Chord varied between 1 minute 50 seconds and 2 minutes 14 seconds.

Tool limitations. From the study results, it was found that similar to FindBugs and JLint, Chord had a hard time detecting deadlock bugs. In fact, Chord did not find a

Warning Description	Remarks	Actual Bug?
Java PathFinder example: deadlock.d1		
No warning	-	**-**
Java PathFinder example: deadlock.d2		
No warning	-	**-**
Java PathFinder example: diningPhilosophers.dp1		
No warning	-	**-**
Java PathFinder example: diningPhilosophers.dp2		
No warning	-	**-**
Java PathFinder example: race.r1		
Dataraces detected (x3)	Detects data races on following 3 fields: race.r1.Event.count, race.r1.FirstTask.count, race.r1.SecondTask.count. In total it produces 16 different schedules that cause races on one of the above fields.	**Yes**
Java PathFinder example: race.r2		
Datarace detected	Detects datarace on field race.r2.Racer.d and produces a single schedule.	**Yes**

Table IX: Detailed Chord results for the Java PathFinder example programs

Warning Description	Remarks	Actual Bug?
IBM Concurrency Benchmark: account		
No warning	-	**-**
IBM Concurrency Benchmark: airlineTickets		
Dataraces detected (x2)	Detects dataraces on 2 fields: airlineTickets.bug.Num_Of_Tickets_Sold and airlineTickets.bug.StopSales. In total 3 schedules are produced for the first field and 1 schedule for the second field.	**Yes**
IBM Concurrency Benchmark: allocationvector		
No warning	-	**-**
IBM Concurrency Benchmark: boundedbuffer		
Dataraces detected (x2)	Detects dataraces on 2 fields: boundedbuffer.BufferNotify._finish and boundedbuffer.Buffer._active. There are 6 schedules for the first field and 4 schedule for the second field.	**Yes**
IBM Concurrency Benchmark: deadlock		
No warning	-	**-**
IBM Concurrency Benchmark: deadlockexception		
No warning	-	**-**

Table X: Detailed Chord results for the IBM Concurrency Benchmark programs

single deadlock bug in the example programs. Naik, et al. in their paper discussing deadlock detection using Chord, give a detailed discussion about the current implementation's limitations [12]. For example, currently Chord is capable of detecting deadlocks that involve only two threads which explains why many of the deadlocks in our examples went undetected.

Tool usability. Chord was executed from the command-line and produced very detailed html reports summarizing possible concurrency bugs. In the case of data races, warnings are categorized by fields or objects where the data races were found. Each warning within the reports has accompanying thread schedules and provides hyperlinks that allow the user to go directly to the potential bug in the source code.

Total Programs Run	12
Total Warnings	8
Total Multithreaded Warnings	8
Total Warnings Exhibiting Real Bugs	8
Total Known Bugs Successfully Found	4
Total Known Bugs Not Found	9
Effectiveness of Finding all Known Bugs	30.77%
Effectiveness of Warnings being Real Bugs	100.00%

Table XI: Summary of results for all example programs analyzed using the Chord Tool

V. Threats to Validity

We have designed our experiment with the aim to minimize the impact of threats to validity. Specifically, our decisions were based on minimizing issues related to internal validity, external validity, construct validity, and conclusion validity.

Despite our best efforts there are still potential threats to the validity of our experiment which need to be discussed. One potential threat to external validity and our ability to generalize our results is based on the static analysis tools we have included in our study and the example programs which we have utilized. With respect to the static analysis tools, our conclusions about detecting concurrency bugs using static analysis may not hold for tools that were not included in this study. We selected FindBugs, JLint and Chord with the intention of finding a representative subset of tools however further experiments with additional concurrency bug detection tools need to be conducted in order to make strong assertions regarding the ability of static analysis as a concurrency bug detection technique. With respect to the example programs, we have selected a set of 12 programs from two different sources, both of which are independent from each other and the static analysis tools. However, we have to acknowledge the possibility that the example programs may not be representative of concurrency programs in general since this is extremely difficult to measure.

VI. Related Work

A number of other empirical studies have been done to determine the effectiveness of various static analysis tools with respect to concurrency bug detection. A detailed comparison of different ways in which concurrency static analysis is carried out was conducted by Chamillard [23]. Corbett has conducted an empirical study of 3 static analysis techniques with respect to deadlock detection [24]. Both of these studies were conducted almost 15 years ago and we view our research as adding to the evidence provided by this earlier work. Another related study is the work of Wojcicki and Strooper who conducted a controlled experiment examining static analysis tools in combination with code inspection in order to detect concurrency bugs [25].

There are also a number of surveys and experiments that compare static analysis techniques in general [26], [27] and static analysis for other problems such as buffer overflow [28] and more [29]. Furthermore, static analysis tools have not only been compared with each other but they have also been compared to non-static analysis tools [30].

Comparisons between various static analysis tools have not only been used to determine under what conditions a tool or technique is more effective than another but, these comparisons have also been used to help create benchmarks in bug detection. For example, the work by Shan Lu, et al. classified various static analysis tools (and dynamic analysis tools and model checker tools) by their strengths and weaknesses to help create the BugBench benchmark [31].

VII. Conclusions

This paper compares three static analysis tools: FindBugs, JLint and Chord. The goal of the experiment is to compare the effectiveness of each tool at detecting known concurrency bugs as well as to measure the percentage of spurious results produced by each tool.

Overall, the results of our experiment were mixed and no tool was able to find more concurrency bugs than the others. For the 12 example programs, FindBugs and Chord were able to identify only 4 of the 13 known concurrency bugs. JLint was able to identify 4 of the 12 known concurrency bugs on 11 of the example programs[1].

The program that produced the lowest percentage of spurious results was Chord with 0% while FindBugs and JLint produced 50% and 78% respectfully. We believe that Chord was able to minimize the percentage of spurious results due to the effective use of multiple types of static analysis within the tool. This leads to one of the conclusions of our experiment: *Our experimental results confirm the benefits of using multiple static analysis techniques in combination in order to reduce the number of spurious results.*

The use of multiple analysis techniques to enhance the detection of concurrency bugs has been previously researched in the literature [32], [33] and more research is needed to determine how static analysis tools can be used in combination to increase their effectiveness at detecting bugs while decreasing the number of false positives. In the future we would like to study which static analysis techniques are the most complementary. We intend to conduct further experiments with a wider selection of programs and additional static analysis tools such as RacerX and RELAY.

Acknowledgment

The authors would like to thank the Natural Sciences and Engineering Research Council of Canada (NSERC) for funding this research.

References

[1] K. Asanovic, R. Bodik, J. Demmel *et al.*, "A view of the parallel computing landscape," *Communications of the ACM*, vol. 52, no. 10, 2009.

[2] Y. Eytani, E. Farchi, and Y. Ben-Asher, "Heuristics for finding concurrent bugs," in *Proc. of the 1st International Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD 2003)*, 2003.

[3] M. Musuvathi, S. Qadeer, T. Ball *et al.*, "Finding and reproducing heisenbugs in concurrent programs," in *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*, 2008.

[4] "FindBugs - find bugs in java programs," Web page: http://findbugs.sourceforge.net/ (last accessed May 07, 2010).

[5] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Not.*, vol. 39, no. 12, pp. 92–106, 2004.

[1]The deadlockexception program, from the IBM Concurrency Benchmark, has been excluded because it produced an error in JLint and the analysis could not be completed.

[6] "Jlint," Web page: http://jlint.sourceforge.net/ (last accessed May 07, 2010).

[7] C. Artho, "Finding faults in multi-threaded programs," Master's thesis, Institute of Computer Systems, Federal Institute of Technology, Zurich/Austin, 2001.

[8] D. Engler and K. Ashcraft, "RacerX: effective, static detection of race conditions and deadlocks," in *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, 2003, pp. 237–252.

[9] J. W. Voung, R. Jhala, and S. Lerner, "RELAY: static race detection on millions of lines of code," in *Proc. of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC-FSE '07)*, 2007, pp. 205–214.

[10] "jChord - a static and dynamic program analysis framework for Java," Web page: http://code.google.com/p/jchord/ (last accessed May 07, 2010).

[11] M. Naik and A. Aiken, "Conditional must not aliasing for static race detection," *ACM SIGPLAN Notices*, vol. 42, no. 1, 2007.

[12] M. Naik, C.-S. Park, K. Sen, and D. Gay, "Effective static deadlock detection," in *Proc. of the 31st International Conference on Software Engineering (ICSE 2009)*, 2009, pp. 386–396.

[13] A. Bessey, K. Block, B. Chelf *et al.*, "A few billion lines of code later: using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, 2010.

[14] "ConTest - A Tool for Testing Multi-threaded Java Applications," Web page: http://www.haifa.ibm.com/projects/verification/contest/ (last accessed May 07, 2010).

[15] O. Edelstein, E. Farchi, Y. Nir *et al.*, "Multithreaded Java program test generation." *IBM Systems Journal*, vol. 41, no. 1, pp. 111–125, 2002.

[16] P. Joshi, M. Naik, C.-S. Park, and K. Sen, "CalFuzzer:an extensible active testing framework for concurrent programs," in *Proc. of the 21st International Conference on Computer Aided Verification (CAV'09)*, 2009, pp. 675–681.

[17] M. Musuvathi, "Systematic concurrency testing using CHESS," in *Proc. of the 6th workshop on Parallel and distributed systems: testing, analysis, and debugging (PADTAD 2008)*, 2008.

[18] K. Havelund and T. Pressburger, "Model checking Java programs using Java PathFinder," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 2, no. 4, 2000.

[19] "Java PathFinder," Web page: http://babelfish.arc.nasa.gov/trac/jpf (last accessed May 07, 2010).

[20] K. Stobie, "Too Darned Big to Test," *Queue*, vol. 3, no. 1, 2005.

[21] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for Java," *ACM SIGPLAN Notices*, vol. 41, no. 6, 2006.

[22] Y. Eytani and S. Ur, "Compiling a benchmark of documented multi-threaded bugs," in *Proc. of the 2nd International Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD 2004)*, 2004.

[23] A. T. Chamillard, "An empirical comparison of static concurrency analysis techniques," Ph.D. thesis, University of Massachusets Amherst, 1996.

[24] J. C. Corbett, "Evaluating deadlock detection methods for concurrent software," *IEEE Transactions on Software Engineering*, vol. 22, no. 3, pp. 161–180, 1996.

[25] M. A. Wojcicki and P. Strooper, "Maximising the information gained from a study of static analysis technologies for concurrent software," *Empirical Software Engineering Software Engineering*, vol. 12, pp. 617–645, 2007.

[26] P. Emanuelsson and U. Nilsson, "A comparative study of industrial static analysis tools," *Electronic Notes in Theoretical Computer Science (ENTCS)*, 2008.

[27] N. Rutar, C. B. Almazan, and J. S. Foster, "A comparison of bug finding tools for Java," in *Proc. of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)*, 2004, pp. 245–256.

[28] M. Zitser, R. Lippmann, and T. Leek, "Testing static analysis tools using exploitable buffer overflows from open source code," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 6, 2004.

[29] F. Wedyan, D. Alrmuny, and J. Bieman, "The effectiveness of automated static analysis tools for fault detection and refactoring prediction," in *Proc. of the International Conference on Software Testing Verification and Validation, 2009 (ICST '09)*, 2009, pp. 141–150.

[30] D. Engler and M. Musuvathi, "Static analysis versus software model checking for bug finding," in *Proc. of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI04) (Invited paper)*, 2004, pp. 191–210.

[31] S. Lu, Z. Li, F. Qin *et al.*, "BugBench: Benchmarks for evaluating bug detection tools," in *Proc. of the Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[32] J. S. Bradbury, J. R. Cordy, and J. Dingel, "Comparative assessment of testing and model checking using program mutation," in *Proc. of the 3rd Workshop on Mutation Analysis (Mutation 2007)*, 2007, pp. 210–219.

[33] J. Chen and S. MacDonald, "Towards a better collaboration of static and dynamic analyses for testing concurrent programs," in *Proc. of the 6th workshop on Parallel and distributed systems (PADTAD '08)*, 2008, pp. 1–9.