Race Catcher™

US and International Patents Issued and Pending.

**Automatically Pinpoints Concurrency Defects
in Multi-threaded JVM Applications with 0% False Positives.**

Whitepaper Introducing Race Catcher™

List of JVM Languages:
http://en.wikipedia.org/wiki/List_of_JVM_languages

Following Java, JRuby and Jython are the most popular presently. The list of 28 languages presently has 50+ JVM-powered implementations.

Race Catcher™ performs dynamic analysis of the bytecode. It supports all of the JVM-powered languages.

The worst bugs are those which we do not know we have - those that do not generate exceptions, logs or crash your system, but are the silent and intermittent conditions of unpredictable results.

They are often impossible to be reproduced in a debugging environment, and even if reproduced, they are not detectable using traditional testing techniques. They live within your code like time bombs. Except when they explode, they are not disarmed but wait for the next time to do damage again and again.

THINKING SOFTWARE INC.
Email: contact@ThinkingSoftware.com

» Thinking Software, Inc. offers the Race Catcher™, which when combined with your JVM environment, provides superior advantages, making it the perfect companion for your multi-threaded JVM-powered applications.

Overview:

Reliability of your business critical software is absolutely essential to the success of your business and must never be compromised.

CPU silicon transistor density has reached its peak. However, processing speeds are continually increasing. Those advances in computing speeds are due, not to increasing silicon density, but the advent of Multi-core CPU technologies.

In order to cope with the proliferation of multi-core machines and leverage that technology to remain ahead of the competition, software developers must design new applications using multi-threading techniques.

The challenge of, and the need for creating multi-threaded applications will only increase with time. This trend will create a higher prevalence of concurrency defects, which can easily slip through traditional testing techniques.

As a consequence of multi-threading, a new non-trivial type of error conditions called thread contentions, consisting of race conditions and deadlocks, has come to the forefront. Dealing with these conditions is becoming increasingly important.

after years of testing

Even after many years of testing applications often host l· large [illegible] conditions.

NIST estimates that over two days of developer's time (17.4 hours [1]) is spent fixing an average bug, but a race condition is not your average bug.

Microsoft published a study where race conditions took from days to month's to diagnose, and where 30% of manual fixes were incorrect.

Race Catcher™ pinpoints experienced race conditions with 100% accuracy and 0% false positive rate.

[1] Source: National Institute of Standards and Technology - The Economic Impacts of Inadequate Infrastructure for Software Testing http://www.nist.gov/director/planning/upload/report02-3.pdf

About Thinking Software:

Thinking Software, Inc. has developed the Software Understanding Machine® (SUM) - a technology that delivers to the software industry higher reliability processing and offers a dramatic reduction in expenses for software testing, debugging and maintenance.

Three Postulates of the Software Understanding Machine® (SUM)

Postulate One: Error free software does not exist: the only proof of software correctness is through testing.

Postulate Two: Exhaustive testing is not feasible: future inputs can be tested only in the future.

Postulate Three: The only way to get truly close to error free software is through constant run-time analysis of software applications using a sophisticated Dynamic Code Analyzer — the kind of analyzer implemented in the Software Understanding Machine® (SUM).

For more information, visit www.ThinkingSoftware.com

THINKING SOFTWARE INC.

Email: contact@ThinkingSoftware.com

Race Conditions & Deadlocks

The potential for race conditions can be eliminated if all the methods within an application are synchronized. However, all of the benefits of multiprocessing then disappear.

By synchronizing, we reduce the risk of race conditions but increase the risk of deadlocks.

» Race Conditions:

A race condition occurs when two or more threads have unordered access to a shared memory location and at least one access is for "Write".

An access for "Write" is when a thread modifies the value of a memory location. An access for "Read" is when a thread simply obtains the value of the memory location.

» Deadlocks:

Deadlock refers to a specific condition when two or more processes are waiting for each other to release a resource, or more than two processes are waiting for resources in a circular chain.

Deadlock is a common problem in multiprocessing where many processes share a specific type of mutually exclusive resource known as a software, or soft lock[1]

Race Catcher™ will notify you of a deadlock at the time the deadlock occurs.

Race Catcher™ will pinpoint the code causing the deadlock

» Unlike the option provided by the JVM:

A deadlock is a condition that is not silently passing (like a race condition) – when you see your application not progressing, you can use the combination of keys <Ctrl> + <Break> (in Windows OS) or <Ctrl> + <\> (In Linux OS) to see the state of all the JVM threads.

However, this requires one to constantly watch the screen. Race Catcher™ will notify you of a deadlock at the time the deadlock occurs. Additionally, and unlike the JVM option, Race Catcher™ will analyze the code causing the deadlock, as opposed to simply listing all the threads being in a state of wait, when threads are in that state often as a consequence of the other threads state.

[1] Source: wikipedia entry on "Deadlocks" - The full aricle can be read here: http://en.wikipedia.org/wiki/Deadlock

» The Dangers of False Positives

False positive analysis results harm productivity. False positives reduce trust in the tool.

Some static analysis tools claim 20% false positives while analyzing race conditions. However during the testing of these tools by our team, some of them really produced 77% false positives. Even the best known dynamic analysis tools claim results to be 15% - 20% false positives.

Race Catcher™ dynamic analysis provides 0% false positives results, and does not miss any of the race conditions experienced by the application.

» Race Conditions - Analyzing Collective Experience

Race Catcher™ allows analysis of the collective experiences of multiple machines running the same application.

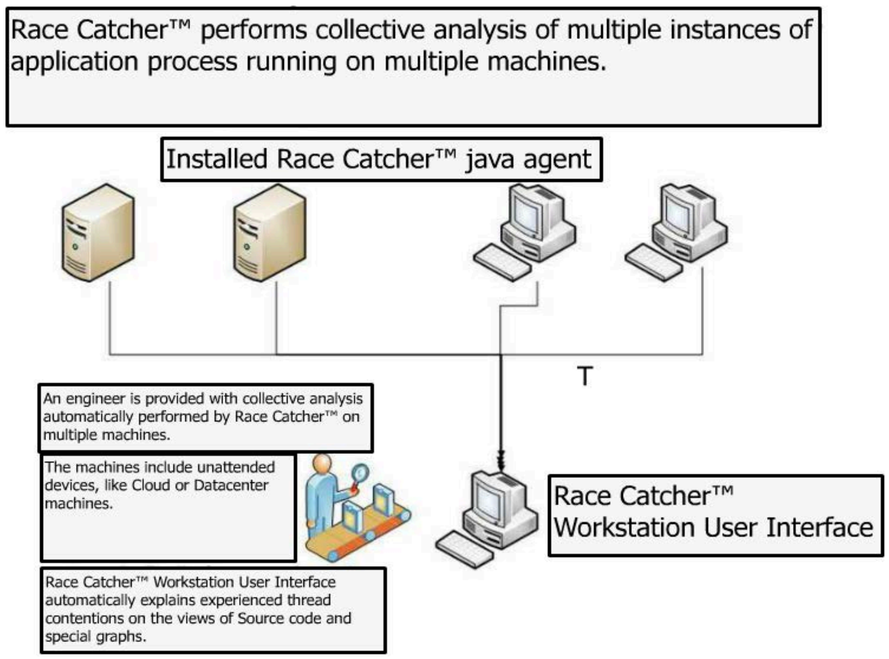
ARM-CM stands for Application Reliability Management via Collaborating Machines.

Since race conditions are intermittent conditions of unpredictable results, and since applications with race conditions are running in a “happy” mode (not crashing, generating exceptions or error logs), analyzing the collective experience of applications running in “ARM-CM Enabled mode” is very crucial.

Race Catcher™ agents can be installed on the machines in the field. Race condition and dead lock analysis results can be communicated to the machine running the Race Catcher™ UI and made available to the tech support engineers or QA depart-ment of the application software vendor.

The fixed version can then be pushed back to the field machines, even before most of them ever experience the fault. In this configuration, the machines in the field act as QA machines.

» Collective Analysis with Race Catcher™:


Race Catcher™ performs collective analysis of multiple instances of application process running on multiple machines.
Installed Race Catcher™ java agent
T
An engineer is provided with collective analysis automatically performed by Race Catcher™ on multiple machines.
The machines include unattended devices, like Cloud or Datacenter machines.
Race Catcher™ Workstation User Interface
Race Catcher™ Workstation User Interface automatically explains experienced thread contentions on the views of Source code and special graphs.

Principles of Race Catcher™

The term "false positive" - in the case of program analysis an error is reported where no error exists - and in the term "false negative" - in the case of program analysis, an error is missed - were first introduced in statistical analysis in the 19th century.

Think of a doctor giving an errant - false positive -diagnosis of some condition to a patient where the condition, in fact, does not exist - and what it does to the patient - and what it does to the doctor's - in this case tool's - reputation).

» "All Automatic" philosophy of SUM

Race Catcher™ is designed to operate automatically. This is required for consistently accurate results, wide adoption and the absence of the learning curve.

» No source code access requirement

Since Race Catcher™'s complete analysis is done on the bytecode, it does not need the source code to be provided by the user in order to operate. This kind of auto-mation is necessary for two reasons:

- The likelihood of human error, when pointing to a wrong source code
- The time and effort required to locate and link the automatically generated project to the original source code.

» No requirements for special environment or integration

In order to be completely automatic, a tool must not require any kind of inte-gration with a specific Interactive Development Environment (IDE). Race Catcher™ is neutral to the environment used to build the application, provided the application is executed by the JVM.

» Why Race Catcher™ is an addition to the JVM

The only one requirement for using Race Catcher™ is that the application executes bytecode.

» Multithreading Contentions are generally not permitted

(with the rare exception when Race Conditions are encoded intentionally.)
In such rare cases, the reporting of the thread contentions is expected.

The Case for Dynamic Code Analysis

Known today dynamic analysis tools, such as Profilers, Bounds Checkers and APM (Application Performance Management) tools are relatively simple in nature, unlike a dynamic code analyzer like Race Catcher™.

Analyzing race conditions statically is defined to be a NP-Hard problem.

"While a method for computing the solutions to NP-complete problems using a reasonable amount of time remains undiscovered, computer scientists and programmers still frequently encounter NP-complete problems. An expert programmer should be able to recognize an NP-complete problem so that he or she does not unknowingly spend time trying to solve a problem which so far has eluded generations of computer scientists. Instead, NP-complete problems are often addressed by using approximation algorithms." [2]

"In computer science and operations research, approximation algorithms are algorithms used to find approximate solutions to optimization problems. Approximation algorithms are often associated with NP-hard problems; since it is unlikely that there can ever be efficient polynomial time exact algorithms solving NP-hard problems, one settles for polynomial time sub-optimal solutions. Unlike heuristics, which usually only find reasonably good solutions reasonably fast, one wants provable solution quality and provable run time bounds. Ideally, the approximation is optimal up to a small constant factor (for instance within 5% of the optimal solution)." [3]

» Static Analysis tools

Examples of static code analyzers are: compilers, Javadoc, and other tools that analyze source code in order to convert it into something else such as an executable or documentation.

A sufficiently complex functionality, like diagnosing of Race Conditions, requires more complex implementation.

If a primary static analysis tool is to consider all possible permutations of a process execution, the number of permutations that must be considered would be very large.

Analyzing race conditions statically is defined to be a NP-Hard problem.

"NP-hard (non-deterministic polynomial-time hard), in computational complexity theory, is a class of problems that are, informally, at least as hard as the hardest problems in NP" [1]

» To analyze Race Conditions, statically approximation algorithms are used

Even when using approximation algorithms, a large portion of such analysis will inevitably be "false positive", i.e., a significant percentage of these permutations will never be encountered during the code execution.

Additionally, due to the approximation used, the results will contain "false negatives", i.e., some of race conditions that will be experienced in practice will not be diagnosed.

» Race Catcher™ is not a Static Analysis tool, it does not use approximation.

Race Catcher™ will not cause any "false positive" analysis while examining actual experienced race conditions. And, it will not miss any of the experienced race conditions either, so there will be no "false negatives". In both cases, it can be trusted to perform as designed without any approximation, and deterministically.

[1] Source: http://en.wikipedia.org/wiki/NP-hard
[2] Source: http://en.wikipedia.org/wiki/NP-complete
[3] Source: http://en.wikipedia.org/wiki/Approximation_algorithm

Conclusion

Multi-core hardware increasingly affects the need for multi-threaded application development, which in turn affects the complexity of software. The challenge of, and the need for creating multi-threaded applications will only increase with time.

This trend creates an increasingly prevalent class of software defects, namely thread contentions or concurrency defects that can easily slip the traditional testing techniques resulting in new application reliability issues.

These reliability defects are specially hard to deal with due to the intermittent nature of the conditions that trigger them. This means that, often, they are not present during the traditional testing phase.

The conditions described above are – deadlock – a state of indefinite wait, and - race condition – a condition of unpredictable results, present when multiple threads are accessing the same memory location in undetermined order, while at least one of these threads is changing the memory location's content.

Race Catcher™ catches all manifested Race Conditions & diagnoses all manifested Deadlocks.

Thinking Software Inc.

provides a tool with very specific advantages. The main advantage is in the ability to treat this tool as an **addition to your JVM**.

- » Opens the "black box" of code going through JVM.
- » Requires no integration with development environments, alteration or recompilation of existing code – the only requirement is that the application executes bytecode.
- » Automatically catches all manifested (experienced) race conditions & diagnoses all manifested deadlocks with 0% false positive precision.

Race Catcher™ is a perfect companion to multi-threaded JVM applications.

Functional Comparison to other technologies:

	Other Static Analysis Tools	Other Dynamic Analysis Tools	**Race Catcher™**
Requires IDE Integration:	YES	YES	**NO**
Source Code Required:	YES	YES	**NO**
False Positive Rate %:	20% **+**	14% **+**	**0%**
Requires Domain Experience:	YES	YES	**NO**
UI opens Black Boxes:			**YES**

About Thinking Software:

Thinking Software, Inc. – has developed the Software Understanding Machine® (SUM) - a technology that delivers to the software industry higher reliability processing and offers a dramatic reduction in expenses for software testing, debugging and maintenance.

Three Postulates of the Software Understanding Machine® (SUM)

Postulate One: Error free software does not exist: the only proof of software correctness is through testing.

Postulate Two: Exhaustive testing is not feasible: future inputs can be tested only in the future.

Postulate Three: The only way to get truly close to error free software is through constant run-time analysis of software applications using a sophisticated Dynamic Code Analyzer — the kind of analyzer implemented in the Software Understanding Machine® (SUM).

For more information, visit **www.ThinkingSoftware.com**

THINKING SOFTWARE INC.

Email: contact@ThinkingSoftware.com