2014 IEEE International Symposium on Software Reliability Engineering Workshops

Combining Black Box Testing with White Box Code Analysis: A Heterogeneous Approach for Testing Enterprise SaaS Applications

Stefano Rosiello*, Amish Choudhary†, Arpan Roy‡ and Rajeshwari Ganesan§
*Department of Computer and Systems Engineering, Federico II University of Naples, Naples, Italy.
†Infosys Ltd., Product Research and Development Unit, Bangalore 560100, India.
‡Infosys Labs, Dependability Center of Excellence, Bangalore 560100, India.
§EdgeVerve Systems Limited, Systems Engineering Group, Bangalore 560100, India.
Email:*st.rosiello@studenti.unina.it, †Amish_Choudhary@infosys.com, ‡Arpan_Roy02@infosys.com,
§Rajeshwari_Ganesan@edgeverve.com

***Abstract*—Faulty enterprise applications may be producing incorrect outputs or performing below service expectations due to code vulnerabilities that do not show up in standard code analyzers (e.g., CAST [1]). A tester can figure out such functionality or performance issues by black box functional testing and fault injection. Then based on the specific test scenario, targeted white box code analysis can be done to figure out the code errors causing the application functionality or performance issue. In this paper, we use such a heterogeneous testing approach that combines black box testing with white box code analysis for testing an enterprise licensing application (ELA). We describe experiments designed to uncover functionality and performance issues in ELA and then explore the corresponding code errors causing the issues. We find that our approach is effective in faster detection and fixing of application performance and functionality errors than simple white box code analysis.**

***Keywords*—enterprise application, code analysis, process kill, functional testing, synchronization errors**

I. Introduction

Enterprise applications are bound by strict service level agreements of response time and availability. Underperforming applications respond slowly or are sometimes rendered unavailable. In this paper, we propose a framework for testing enterprise applications that combines black box testing with white box code analysis. This approach has several advantages over simple white box code analysis. In white box code analysis, usually first a code vulnerability is discovered and then its corresponding functionality or performance issue (if any) is unearthed by targeted fault injections. On the other hand, in our approach we first use black box testing to pinpoint application functionality or performance issues. Then based on the specific functionality or sub-functionality error, we do targeted source code analysis to figure out code errors that may be causing the issue. Usually this approach enables faster detection and fixing of application performance and functionality issues than simple white box code analysis. More specifically, in this paper

- we discuss a testing strategy where first black box testing is applied to pinpoint specific application functionality issues and then if need be, white box code analysis is done to find the causes of those issues,
- we discuss the experiments we designed to test our sample application, an enterprise licensing application (ELA) and the errors discovered during each test and
- finally we explore the cause of the errors discovered by targeted analysis of the application source code.

The rest of this paper is organized as follows. Our testing strategy is introduced in Section II along with some related work. In Section III, we discuss the architecture of the application under study. In Section IV we discuss our experiments, the observed errors and methods of mitigating said errors. Finally, we discuss the signficance of the results we obtained in Section V.

II. Testing of Enterprise Applications and Related Work

In order to develop robust applications, it is necessary to test all the logical paths in the code before deploying the software to production.

A. Application Testing: Black box vs. White box Approach

White box testing of an enterprise application can be done using both code analysis tools and fault injection tools (software implemented fault injection or SWIFI) . SWIFI is about reproducing software errors and consequently the failures that would have been produced by subsequent hardware faults. Software code analysis reveals points of vulnerabilities in the source code such as: (i) point where a COTS component is called, (ii) point where a database is called, (iii) point where a message queue is updated etc.. This is followed by fault injection using fault models such as *(i) Process Crash:* Code is injected into the application under test such that while trying to execute it, the client JVM crashes rendering the service unavailable. *(ii) Session Crash:* Code is injected into the application under test such that when this code is activated only one session of the application crashes and *(iii) Delay:* Code is injected into the application under test such that when this code is activated, the lines after this code are executed after a certain delay causing subsequent SLA violations. After injection, it is checked if the application output is coherent.

978-1-4799-7377-4/14 $31.00 © 2014 IEEE
DOI 10.1109/ISSREW.2014.113

In white box testing, code analysis comes first and then faults are injected to study the failures at the vulnerable points. However in black box testing, first faults are injected and then subsequent failure behavior is observed. Then based on the failure behavior, software code is analyzed for specific bugs. Hence, in white box testing there are two phases: golden run (fault-free run of the application) and resilience test (run with injected faults). Black box testing has only one phase: resilience testing. For our purposes, we use two methods of black box testing an application:

1. Functional Testing: We send a payload composed of a heterogeneous set of requests to the application. Then we check the database before and after the sending of requests to verify number of successful requests of each type.

2. Fault Injections: We crash one of the system processes (e.g., the primary DBMS process for the application) during the test run and then we observe how the incoming requests to the database are handled.

B. Related Work

Code analysis, fault injection and functional testing are common techniques for studying the failure behavior of the application. White box code analysis tools such as CAST [1] needs the source code of the application under test exposed to its APIs. On the other hand, performance testing tools such as Apache JMeter [2]) perform black box stress tests by applying different loads on the application. Fault injection can be:

- hardware fault injection where fault is injected by contact (e.g., a socket is inserted into the circuit and complex logic faults are injected using this additional hardware) or fault is injected without contact (e.g., heavy ion radiation is used to cause faults without any phyiscal contact to system) [3] or
- software fault injection where different injectors have different novel aspects for injecting faults. For instance, Xception [4] uses hardware debugging registers, Ferrari [5] uses traps and Orchestra corrupts messages [6] to inject faults. Hardware and software fault injection can be combined. For instance, the fault injection tool NFTAPE [7] allows for both hardware fault injections using simple fault signals such as bit flips, bit inversion, delays (fault effects demonstrated over Myrinet LAN [8]) and a software implemented fault injector (SWIFI) that injects faults at points of code vulnerabilities using more complex fault models such as session crash, process crash (demonstrated over a image processing application [8]).

We avoid using any such proprietary software for our testing needs (our scripts are standalone and specifically crafted for the application we are testing). In this paper, we present a case study that uses a unified approach combining all three above mentioned application testing approaches. We show how a combination of black box fault injection and load testing combined with white box code analysis can bring out faults that could not have been discovered by any one of these approaches alone.

C. Our Heterogeneous Testing Approach

In order to setup the test process, it was necessary for us to analyze the requirements of the system under test to identify:

- the architecture and its components,
- the system's purpose and its high level functionalities,
- the features of the workload and
- the system's interfaces (exposed to the users, to other systems and between the architectural components).

The purpose of the above analysis is to:

1) Identify a component for testing with the targeted workload
2) Identify the interfaces and the functionality to build the test scenarios. Assign priorities to different functionalities to create a test plan.
3) Identify a criteria for each functionality by which we can distinguish faulty results from correct results. To identify correct functional behavior, it might be useful to try to identify some execution invariants.
4) Identify a way to inject faults into the system from the external environment (i.e. operating system).

We can split our testing process into two phases:

PHASE I: Pure functional testing is done to ensure that the system meets its requirements.

1) apply the workload to the system and collect the response.
2) use the functionality criteria to check if the system meets its requirements and behaves correctly and
3) if the output reveals functional defects, perform **code analysis** to detect the cause of defect and fix it. Then proceed to $PHASE\ II$.

PHASE II: Fault injections for assessing the system's behavior in presence of faults (after ensuring that the system is behaving correctly).

1) Apply the same workload used in $PHASE\ I$.
2) While applying the load, inject the fault from the OS into the component under test.
3) Observe the system behavior and analyze the logs during the fault of one of its components.
4) Use the functionality criteria to check if the system is in coherent status after the application of load.
5) If not coherent, fix with or without **code analysis**.

III. Case Study: Enterprise Licensing Application (ELA)

In this section, we describe the architecture of the application under study and then we describe the test environment (environment in which the application is deployed).

A. Enterprise Application Under Study

The enterprise licensing application (ELA [9]) is an application that helps manage licenses of products sold and used by clients worldwide. The service can concurrently serve a large number of users. The enterprise licensing application can have one of three architectural modes (as shown in Figure 1):

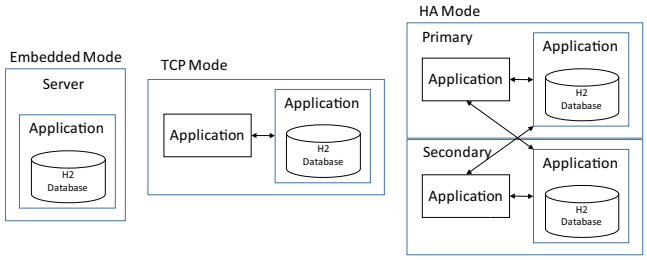

Fig. 1. Application Architectural Modes

- **Embedded Mode:** Here the H2 persistent SQL database containing the product licensing information is embedded in the same Java VM as the license management service. Additional features include the use of an in-memory query caching system
- **TCP Server Mode:** The H2 persistent SQL database is located in a VM different from the application VM. The application VM communicates with the database over a TCP connection.
- **High Availability Mode:** In the High Availability (HA) mode, two instances of the application VM and the database VM are created on two different machines - primary and secondary to allow for better fault tolerance. In this clustered application mode, there are three different functional modes: (i) normal HA mode where the primary machine is working and the secondary as redundant spare, (ii) failover HA mode where the primary has failed and the secondary is working and (iii) failback HA mode where the primary machine is in recovery and the secondary machine is working. Data stored in the primary machine is synchronously stored in the secondary machine so that the secondary can take over at any point when the primary has failed.

The license file specific to each client of a certain product is stored in the database in AES encrypted format. Content of license file can be exposed over SSL in xml format. The file contains information such as product name, no. of users and no. of machines (with the IP addresses of machines currently using a license). A license management analyst can (i) upload a license file, (ii) update an existing license file or (iii) delete a license file. For each client, APIs exposed allow for operations such as (i) registering a user, (ii) validating a user, (iii) getMetricsInfo for each user and (iv) deregistering a user. For our testing purposes we use an additional operation involving a random mix of the first four operations. Databases can be on disk or in-memory with failover and failback modes possibles (between a primary database and a secondary replicated database). Failures in this licensing application may result in use of unlicensed application, unavailability of licensed application and SLA violations (all resulting in loss of revenue). A sample application architecture is shown in Figure 2.

B. Test Environment

The test environment is the license server application deployed in VMs hosted in the following server.

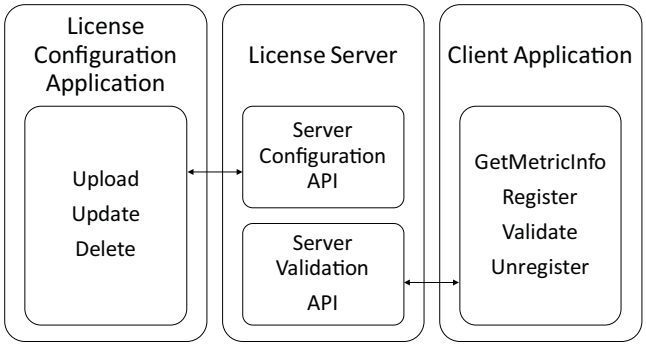

Fig. 2. Application Architecture

- Intel(R) Xeon(R) CPU E7-4830 @ 2.13GHz -8CORE
- 8GB Ram
- Red Hat Enterprise Linux Server release 6.2
- OpenJDK64-Bit Server VM
- H2 Database Engine v. 1.3.176
- Virtualized using VMWare ESX 5.1

C. Test Parameters

During $PHASE\ I$ of our heterogeneous testing approach, we do load testing where a workload (consisting of a number of register and deregister requests) is applied to the licensing application and the output is observed. We estimate the correct output (output from a fault free application) and then compare it with the real output (output of the actual faulty application). The workload simulates a number of users, a fixed number of requests per user, a specific request type (or a random sequence of requests) and a maximum duration T upto which test will run. The server response to each request can be: (i) success, (ii) failure or (iii) service unavailable. For each operation, a summary report containing the thread ID, operation type, start and stop timestamps, duration and response code is generated. Additionally at the server side, the following information is studied: (i) initial metric value (e.g., initial number of registered users), (ii) number and types of operations performed, (iii) number of 'success' responses, (iv) final expected metric value and (v) final actual metric value. During $PHASE\ II$ of our testing process, we do black box fault injection where we kill different processes and the subsequent effect on system response using above parameters. Code analysis is done manually during both $PHASE\ I$ and $PHASE\ II$ to find and fix application bugs causing faulty behavior.

IV. Experiments: Testing of the Enterprise Licensing Application

In this section, we describe sequentially the different test scenarios for black box testing of our application. For each test scenario, we describe the workload, the expected outcomes, the observed outcomes, reproducibility of the error and the cause of error and performance impact associated with each error.

A. Test Scenario I: High Concurrent Workloads

In the first case, the application is loaded with a large number of concurrent requests from the user and the output is studied for anomalies.

Phase I (Functional Testing): The server contains a License file for 500 different users. Initially there are 283 registered users in the database. A concurrent workload is generated for 50 users with 10 requests per user. The distribution for the requests is 110 register requests, 132 unregister requests, 125 validate requests and 133 GetMetricInfo requests. At the end of the test, the count of the number of users registered in the database should be 261 (283 initial + 110 registration – 132 deregistrations). But the observed number of users in the database was seen to be 263. This issue could be reproduced under the same load but the final number of users in the database is random.

Finding (Incorrect Read-Write Synchronization): We concluded that since this happens only on concurrent workloads, the application code contained some critical sections which are not properly protected from concurrent access. On inspecting the code, we discovered the possible cause of error to be use of incorrect synchronization mechanism for read/write lock. Read/write lock was only being used in read lock mode to protect a critical section instead of a mutual exclusion lock between read and write. This is a simple readers/writers problem. In a read/write lock, read locks prevent a write during an ongoing read but allow for multiple concurrent reads during an ongoing write. This makes $dirty\ reads$ possible. Since the readers in a Readers/Writers problem can access the critical section concurrently during a write, there is no synchronization at all.

Fix: The type of lock was changed from ReentrantReadWriteLock [10] to ReentrantLock [11]. The corresponding changes in code is shown in Table I.

Performance Impact: We observed that changing the lock type caused the system throughput to drop from 3170 requests per minute to 828 request per minute on average. Moving the I/O code outside the critical section for a fine-grain synchronization, caused the throughput to increase to 2730 request per minute on average. After making above changes, on applying the same load there was no mismatch in the expected number of registered users and actual number of users registered.

B. Test Scenario II: Database Unavailable in TCP mode

Some errors are specific to the architectural modes of the application. In TCP Mode, the application is connected to the database system through TCP/IP stack.

1) Sub-scenario 1: In this case, we kill the database process and study the system response.

Phase II (Fault Injection): A $POSIXSIGKILL$ signal is sent to DBMS process during the test run.

kill -9 $(ps aux | grep LicenseServer | grep -v grep | awk '{print }')

This is to simulate a crash on DBMS's machine as well as a network problem. A concurrent high workload is generated for 50 users and 1000 requests per user. The request type distribution consists of 12425 Register requests, 12484 Unregister requests, 12581 Validate and 12510 GetMetricInfo requests.

Finding ("Out of Memory" Error): It is expected that the server should discard new requests when the DBMS is down or should send an error response to clients (immediately after DBMS crash or after a timeout). It was observed that the server remains waiting for DBMS connection indefinitely (HANG) without sending any response to clients. In high workload situations like this, the server will eventually go $Out\ Of\ Memory$. Note the applied load above is higher than $Test\ Scenario\ I$ to look for an $Out\ Of\ Memory$ error. This fault can be reproduced anytime when TCP mode is on and the database server is unavailable.

2) Sub-scenario 2: In this case, we apply a workload on the ELA application. Then we kill and restore the database process and study the system response.

Phase II (Fault Injection): As in the last test case, a POSIX SIGKILL signal is sent to the DBMS process during the test run. After a while the database is restored. A concurrent workload is generated with 50 users and 10 requests for each user. The request type distribution is 110 Register requests, 132 Unregister requests, 125 Validate and 133 GetMetricInfo requests.

Finding (Serving aborted clients): The server kept queuing new requests in memory without processing them. When DBMS comes up again the server starts processing all queued requests even if the clients have already aborted. This results in inconsistencies at the client end. Note, the load is not as high as Sub-scenario 1 as we are not looking for exhaustive errors like $Out\ Of\ Memory$. Generally it is expected that in such situations the server should discard new requests when the DBMS is down or should send an error response to clients (immediately or after a timeout) during DBMS unavailability.

Fix: For both errors/sub-scenarios, discarding of new requests or switching to the secondary machine in HA mode would have solved the problem. However in order to get rid of this error, the ELA team removed support for TCP Mode as it was useful only in the early developmental stages for accessing DBMS from an external application. DBMS is always embedded in the application VM in the new architecture.

C. Test scenario III: HA Mode Cluster Coherency

In high availability (HA) Mode, initially the TCP server mode architecture is replicated on two different machines (primary and secondary). The cluster synchronization is handled completely by H2 DBMS Cluster Mode.

Phase II (Fault Injection): A concurrent high workload is generated for 50 users at 10 requests per user. Initially there are 283 registered users in the database. The distribution for the requests is 110 register requests, 132 unregister requests, 125 validate requests and 133 GetMetricInfo requests. A POSIX SIGKILL signal is sent to DBMS process in the primary machine during the test run. Results should suggest a coherent failover mode i.e., all requests served with a positive response to the client should be recorded in the secondary database and request for which the client received no response shouldn't be recorded in the secondary DB.

TABLE I. TEST SCENARIO I: SYNCHRONIZATION ERROR

BEFORE	AFTER

private final static ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
@Override
public void run() { // Worker Thread
lock.readLock().lock(); // Start Critical Section
operation = getRequestFromClient();
if ("1".equals(operation)) // registration
result = lb.register(clientRequest);
else if ("2".equals(operation)) // validation
result = lb.validateLicense(clientRequest);
else if ("3".equals(operation)) // Unregister
result = lb.unRegister(clientRequest);
else if ("4".equals(operation)) // getMetricDetails
result = lb.getMetricDetails(clientRequest);
else
result = "Not valid request";
sendResultToClient(result);
lock.readLock().unlock(); // End Critical Section
}
}
	private final static Lock lock = new ReentrantLock();
	
	@Override
	public void run() { // Worker Thread
	operation = getRequestFromClient();
	lock.lock(); // Start Critical Section
	if ("1".equals(operation)) // registration
	result = lb.register(clientRequest);
	else if ("2".equals(operation)) // validation
	result = lb.validateLicense(clientRequest);
	else if ("3".equals(operation)) // Unregister
	result = lb.unRegister(clientRequest);
	else if ("4".equals(operation)) // getMetricDetails
	result = lb.getMetricDetails(clientRequest);
	else
	result = "Not valid request";
	lock.unlock(); // End Critical Section
	sendResultToClient(result);
	}
	}

Findings (Crash before/after output): From our test, we found that the number of requests acknowledged to the client was different from the number of such requests committed to the secondary database. The fault could be reproduced anytime in HA mode when the primary DBMS fails with random results. The error arises mostly due to the fact that status of the synchronization between the primary and secondary machines is not clarified in HA mode architecture design. Separate ACKs are not returned to the application after synchronization between primary and secondary. Specifically the two errors possible in HA mode are:

1. Crash before output: In HA mode, if the primary machine crashed between applying changes to primary database and updating the primary cache in HA mode, the secondary machine may or may not have been updated. But the client can only contact the database cluster, cannot contact and update secondary database on his own. In new HA architecture, if primary database updation is ACKed, secondary database updation can be done by contacting the secondary database separately.

2. Crash after output: In HA mode, the client contacts the database cluster as a whole. The client can't contact the secondary database separately. So if primary crashes after the client receives acknowledgement of the database commit, there are no guarantees that the secondary is updated (whether secondary update has happened depends on an in-database synchronization mechanism). In new HA architecture because of the sequence of ACKs, if primary crashes after the client receives acknowledgement of commit, secondary update is guaranteed.

Fix: Use of a simple ACK-based (acknowledgement) synchronization mechanism gets rid of this error. A new architecture consisting of Hazelcast Distributed Cache to synchronize the updation of the primary and secondary databases. Workflow of the primary and secondary database commits using hazelcast cache is as in Table II.

With this architecture, during the normal HA mode, both spares are making the same changes to their respective databases. The DBMS works in embedded mode in both the primary and the spare. This new architecture can tolerate the following situations: (i) A primary crash between steps 1 and 2 (before the response) results in no response to the client. So the client can resend the request to the secondary spare. The secondary is updated and coherent with all changes if the primary crashes after 7 (after the response). However this failure doesn't impact the system throughput (i.e., performance).

D. Additional Errors: Safe MultiThreading

These errors were discovered just by $white\ box\ code\ inspection$. These errors refer to multiple threads creating multiple instances of the same object and using them, when ideally only one singleton instance of an object needs to be created and shared among multiple threads.

1) Singleton LicenseBusiness Object Error: Different threads were creating different instances of the LicenseBusiness object instead of sharing one common instance. The LicenseBusiness object singleton instance creation is not synchronized. On inspecting the code, it was found that multiple threads were creating multiple instances of the model. The solution is to implement thread-safe singleton pattern. Out of the multiple instances of the LicenseBusiness object, all the threads used the instance created last. All other instances were left in memory unused. Hence, this issue did not affect system throughput. The current setup is an 8 core machine. This means that (ideally) 8 threads can check the null condition at the same time. However in our tests with an 8 core machine, it was found that only 2 threads check the null condition at the same time. Hence this error did not create any performance issues. The corresponding change in code is shown in Table III.

2) Singleton LicenseCache Object Error: While inspecting the code, it was also found that multiple instances of the License Cache Model are generated by different threads. The License Cache Model Singleton instance creation is not synchronized. Multiple threads can create different instances of the model. This was because of the multiple instances of cache created, the threads only used two of the cache object instances created. The other cache instances, though created were stored in memory unused. So the issue did not affect system throughput. To resolve the issue, a thread-safe singleton LicenseCache object creation was implemented. The corresponding change in code is shown in Table IV.

TABLE II. WORKFLOW FOR DATABASE UPDATION

Primary Cache	Distributed Cache (Hazelcast)	Secondary Cache
1. Apply changes to Database		
2. Put request in cache	3. Send notification to all active spares	4. Execute request from cache
		5. Send an ACK to a distributed cache
	6. Send an ACK to primary	
7. Send response to client		

TABLE III. SINGLETON LICENSEBUSINESS OBJECT ERROR

BEFORE	AFTER

private static LicenseBusiness licenseBusiness = null;
 	
public static LicenseBusiness getLicenseBusiness(){
if (licenseBusiness == null){
licenseBusiness=new licenseBusiness();
licenseCache=LicenseCache.getLicenseCache();
if(Configuration.hamode!=null &&
“true”.equalsIgnoreCase(Configuration.hamode)){
try{
HazelcastUtil.getHazelCastInstance();
}
catch(Exception exception){
System.out.println("Hazelcast Cannot be started as port is already in use"); // Unregister
}
}
}
return licenseBusiness;
}
	protected static final LicenseBusiness licenseBusiness = new LicenseBusiness();
	protected LicenseBusiness(){
	logger.debug("LicenseBusiness Constructor: ");
	licenseCache = LicenseCache.getLicenseCache();
	if(Configuration.hamode!=null &&
	“true”.equalsIgnoreCase(Configuration.hamode)){
	try{
	HazelcastUtil.getHazelCastInstance();
	}
	catch(Exception exception){
	System.out.println(“Hazelcast Cannot be started as port is already in use”);
	}
	}
	}
	public static LicenseBusiness getLicenseBusiness(){
	return licenseBusiness;
	}

TABLE IV. TEST SCENARIO IV: LICENSECACHE OBJECT SYNCHRONIZATION ERROR

BEFORE	AFTER

private static LicenseCache licenseCache = null;
 	
public static LicenseCache getLicenseCache() {
if(licenseCache == null) {
logger.debug("New LicenseCache object created");
licenseCache = new LicenseCache();
}
return licenseCache;
}
	protected static final LicenseCache licenseCache = new LicenseCache();
	protected LicenseCache() {
	logger.debug("LicenseBusiness Constructor: ");
	}
	public static LicenseCache getLicenseCache()
	return licenseCache;
	}

V. CONCLUSION

White box code analyzers can parse through millions of lines of code and shortlist only a couple of hundred code vulnerabilities. We used CAST code analyzer [1] to analyze the code of ELA. CAST uncovered coding issues that include (i) performance issues such as expensive calls in loops, (ii) architectural design issues such as avoiding cyclical calls and inheritance among parameters, (iii) security issues spanning input validation and object-level dependencies and others. However, not all of these errors directly contribute to erroneous application output or application underperformance. In order to find only the errors that affect application functionality/performance, we performed black box functional testing and fault injections on ELA. Then if need be, we did targeted analysis of the application source code to figure out the specific code errors causing the functionality or performance issue. Hence, we conclude that combining black box testing with white box code analysis can help address critical errors in enterprise applications more efficiently. Our approach is also useful in cases where the application source code is not available to the tester or if the application uses any third party COTS components (source code for COTS components is usually unavailable to the application tester).

REFERENCES

[1] http://www.castsoftware.com/products/application-intelligence-platform.

[2] http://jmeter.apache.org/.

[3] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, “Fault injection techniques and tools,” *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[4] J. Carreira, H. Madeira, and J. Silva, “Xception: Software fault injection and monitoring in processor functional units,” *Dependable Computing and Fault Tolerant Systems*, vol. 10, pp. 245–266, 1998.

[5] G. Kanawati, A. Kanawati, and J. Abraham, “Ferrari: A flexible software-based fault and error injection system,” *IEEE Transactions on Computers*, vol. 44, no. 2, pp. 248–260, 1995.

[6] S. Dawson, F. F Jahanian, and T. Mitton, “Orchestra: A probing and fault injection environment for testing protocol implementations,” in *Proc. IPDS*. IEEE, 1996, p. 56.

[7] http://www.armored-computing.com/nftapeoverview.html.

[8] D. Stott, B. Floering, D. Burke, Z. Kalbarczyk, and R. Iyer, “Nftape: a framework for assessing dependability in distributed systems with lightweight fault injectors,” in *Proc. IPDS*. IEEE, 2000, pp. 91–100.

[9] http://www.infosys.com/engineering-services/service-offerings/Pages/software-cloud-mobile-enablement.aspx.

[10] http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.html.

[11] http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html.