ORIENTAL JOURNAL OF
COMPUTER SCIENCE & TECHNOLOGY
An International Open Free Access, Peer Reviewed Research Journal
Published By: **Oriental Scientific Publishing Co., India.**
www.computerscijournal.org

ISSN: 0974-6471
December 2016,
Vol. 9, No. (3):
Pgs. 186-193

Testing of Multithreaded Code under Deterministic and Predictable Environment

MOHAMMED ALGHAMDI[2], MOHAMMED ALRIFAI[2], KHALIL ALSULBI[2], WADEE ALHALABI[3] and MOUSTAFA MAHMOUD YOUSRY[1]*

[1]Department of Production Engineering, Faculty of Engineering, Alexandria University, Egypt.
[2]Master of Computer Science, King Abdulaziz University, Saudi Arabia.
[3]Professor of Computer Science, King Abdulaziz University, King Abdulaziz University, Saudi Arabia.
*Corresponding author E-mail: eng_moustafaa@hotmail.com; Tel.: +201008227507

http://dx.doi.org/10.13005/ojcst/09.03.04

(Received: December 15, 2016; Accepted: December 25, 2016)

ABSTRACT

This report focuses on the execution of multithreaded programs and finding bugs and errors in those programs. Testing is done to determine if the code written runs correctly or not. The report also covers comparison of traditional testing tools with the new and efficient systematic testing tool called CHESS. The repost explains in detail about the testing technique of CHESS including how it identifies and handles bugs in multithreaded programs. The various experiments performed using different outputs have also been discussed and their respective results have also been shown in order to determine the behavior of CHESS tool when it is provided random inputs. Using this input did not lead to non-deterministic test and the execution time increases exponentially.

Keywords: Multithreaded, CHESS, Software testing, Operating system, Concurrently.

INTRODUCTION

Multithreading is the ability of an operating system to run programs concurrently that are divided into sub parts or threads. It is similar to multitasking but instead of running on multiple processes concurrently, multithreading allows multiple threads in a process to run at the same time. Threads are more basic and smaller unit of instruction. Hence multithreading can occur within a single process. Multithreading can also be defined as a combination of microprocessor design and machine code which allows computer instructions to be carried out concurrently and the results to be combined in right logical order. Programs can execute multiple tasks simultaneously by incorporating multithreading. The real purpose of multithreading is to help in proper and resource effective utilization of the hardware and software resources. Multithreading provides concurrency as it enables many programs to run in parallel and execute simultaneously thus saving time and providing efficiency (Ball T. (2011)).

Operating systems are running multiple threads at one time in background, for example, logging file changes, indexing data and managing the operating systems. At the same time, web browsers also support multithreading. Users can open multiple web pages running animations in different tabs in a web browser concurrently. Multiple threads running simultaneously don't affect each other as long as the CPU has enough power to run all of them. Multithreading adds stability to the programs and prevent it from crashing. All threads run independently. So if an error is encountered by a thread, it should not affect rest of the program. It allows better utilization of the processor and other system resources (Blumofe R. (1996)).

The paper examines how the chess system works to effectively to ensure the effective testing of the concurrent software. The paper will go into details on the methodology chapter and the discussion part to discuss more on the multithreaded scenario, findings and discussion.

Proposed Work

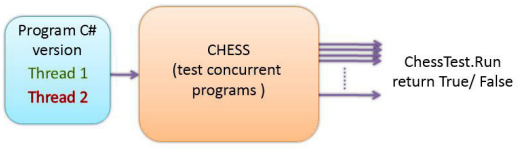
The execution of multithreaded programs is pretty much complicated and tricky because of high probability of encountering unpredictable and erratic interference among concomitantly running programs. Writing, testing and debugging multithreaded programs is, therefore, not an easy task and requires a lot of hardwork, input and attention of the developer. Certain challenges are faced while testing multithreaded programs such as the lack of control over which schedule has to be executed each time a program is run. Another challenge is systems having non-deterministic scheduler which causes the system to be unable to predict output as there are so many outcomes for each input, thus the system fails to predict and generate the accurate output (Rinard M. (2001)).

Unfortunately the standard testing techniques are not efficient and reliable as they just cover some fraction of schedules and many schedules are left untested in such traditional testing methods. This drawback of traditional testing can lead to a software bug called Heisenbugs which is an unusual kind of bug that changes its behavior or vanish when it is observed or studied. This bug is of very unpredictable time and it may disappear or change its form when attempts are made to debug it. These types of bugs appear rarely and are very difficult to handle and debug (Farchi E. (2003)). The basic purpose of this work is to provide different inputs to a multithreaded program and run large number of threads on CHESS to test how Chess handles multithreaded codes and what kind of results are generated.

Tool Used

The tool that is most commonly used for testing parallel execution of multiple programs in a predictable and progressive manner is called Checker for System Software (Chess). It is a systematic tool designed for testing software's having concurrent multithreaded programs. The Chess testing tool tests software's that carry out programs concomitantly in order to check if the programs are running in a right manner or not and also to determine whether correct output is being generated or not. This testing tool employs model checking methods to produce all possible interleaving results of a particular situation. Chess is capable of testing large number of programs that are executing concurrently and is able to detect many unexpected and unfamiliar bugs in a system which otherwise may remain unnoticed by standard testing techniques.

Chess repetitively executes a multithreaded program and ensures providing a predictable and deterministic schedule. It also ensures covering all schedules and even more so that all bugs and errors in the programs are identified quickly. The tool works in an iterative fashion such that a program is executed repeatedly and in each execution, a different thread schedule is followed. It works in loop where all iterations of the loop takes a different interleaving and is repeatable. Whenever a bug's presence is identified in a program, Chess constantly keeps on producing erroneous execution repeatedly which makes the presence of bug in the program more obvious and thus it becomes easier to debug. Many software's use



Chess for testing purposes and it is incorporated in the test frameworks of basic code of many programs.

A Multithreaded Scenario

Let's consider the scenario of a bank account involving implementation of a multithreaded program. The program consists of a class named Test Account which contains a method called Run. The Run method is to test another class called Account in a multithreaded fashion. Consider an instance of class Account with value $10. Then consider a child thread in which $2 are being withdrawn from the account. The main thread starts the child thread and two operations are being performed concurrently on the account; one is withdrawing $2 from the account and the other is depositing $1 in the account. The main thread then waits for the child thread to complete. Since the two operations that are withdrawal and deposit of money are executing in parallel so when we will withdraw $2 from the account, $8 will be left in the account but at the same time we are also depositing $1 in the account therefore the expected amount to be present in the account at the end is $9 but it is not so. The program is giving an incorrect result which says that the account will be having $11 at the end of execution because of some error in the threading code. The reason of generation of this incorrect result is the complexity of multithreaded programs. In the given scenario, what actually happening is that the main thread is executing both operations; withdraw and deposit in parallel. The main thread starts the Withdraw operation of child thread and this operation reads the current value of the account that is $10 and stores this value in temp. The main thread also executes the Deposit operation simultaneously in which $1 is added to the account. Since the child thread is not completed yet so the account's value is still $10 and deposit of $1 to the account changes to the account value to $11. The control then returns to the child thread and using the value stored in temp that is $10, $2 are deducted from it and the child execution leads to an inaccurate value of the account balance $8 which is actually supposed to be $9.

This error would not have occurred if the child thread had locked the account so that some other thread does not interfere and access the account meanwhile and would not have been able change the value in Account. The presence of this error lead to an instable value in account. Now let's apply both traditional testing method and chess testing method on this scenario so that we can 3 compare results of both testing methods and conclude which one is better.

Testing using Traditional Method

Traditional testing methods do not go through all possible schedules of multithreaded programs and are nondeterministic which means they may end up producing different random outputs for the same input thus leading to an instable result. When multiple threads are running concurrently, their executions interrupt each other and the intensity of interruption depends on the processing speed of the system on which they are being carried out and also on the state of memory and cache. In case of single processor system, a particular time slot is allocated to each thread by thread scheduler. When time slot of a particular thread is expired, resources are preempted from that thread and its execution is suspended till it gets the next time slot for complete execution and the execution of the next thread is started. This type of preemption of resources from thread and suspension of its execution can happen anywhere in the code of threads. The allocation of time slots to threads by thread scheduler is not accurate and can lead to nondeterministic scheduling of multiple threads.

When traditional testing method is applied to the previously described scenario, we observe that they are unable to predict correct results and unable to find bugs in multithreaded programs since they do not cover all thread schedules. They do not guarantee trying all schedules even if the test is run forever that means they would not execute all schedules ever no matter what because even the operating system scheduler does not provide guarantee of covering all schedules.

Testing using Chess

There is a Chess Scheduler in Chess which is called by default at the start and end of execution of each thread. Whenever a thread starts, Chess makes a call to the Chess Scheduler which simple serves to provide delays in the program so

that some gaps can occur between the starting and ending of two threads in order to avoid their interference with each other but this technique is not an efficient as it does not provide deterministic scheduling (Musuvathi M. (2007)). Deterministic scheduling is guaranteed by selective blocking of threads in which Chess blocks threads in such a way that only one thread is running at a time while the others are blocked so that they do not execute at that particular moment. This provides a serialized execution of threads that is one after the other and ensures no interference of threads with each other and successfully eliminates non-deterministic schedules that were being generated by the operating system schedulers and hardware. Chess monitors the actions performed by thread which is running at the moment such as system calls, synchronizations etc and thus keep a check on when to block this thread and start executing the next one. This approach, however, reduces concurrency which can be overcome by running multiple instances concomitantly having each instance to explore different section of the same test schedule. Therefore the Chess Scheduler uses wait and release operations to organize sequence of threads with smooth and uninterrupted execution. The wait operation keeps a thread blocked until some other thread performs the release operation. In a multithreaded program, Chess carries out the child thread first while keeping the main thread blocked which remains on waiting state until the child thread is executed completely. Then after execution of child thread, Chess releases the main thread and blocks the child thread through wait instruction. Thus Chess testing technique provides deterministic scheduling.

Consider running chess on the program discussed previously that has a TestAccount class which in turn has a Run method. The run method does not contain any parameter, it simply runs the test and produces result in Boolean form that is true if the test is successful and vice versa. The run method is executed repeatedly by the Chess testing technique and in each execution of the run; a different thread schedule is used. The repeated executions using different thread schedules by Chess enables identification of those thread schedules which can cause bugs and errors in the program. Chess supports deterministic way so we

Table 1: The Execution time to find multithreading error in second using 2 threads, 15 execution steps and 3 tests

	Original Paper value	Random values 10-20	Random values 100-200	Random values 1000-2000	Random values 10000-20000
test1	0.31	0.46	0.47	0.47	0.47
test2	0.46	0.46	0.188	0.63	0.47
test3	0.46	0.62	0.234	0.78	0.63

Table 2: The Execution time to find multithreading error in second using 4 threads and 42 execution steps and 39 tests

	Original paper values	Random values 10-20	Random values 100-200	Random values 1000-2000	Random values 10000-20000
test1	0.47	0.62	0.62	0.46	0.63
test2	0.62	0.94	0.62	0.62	0.63
test3	0.62	0.109	0.78	0.109	0.78
test4	0.78	0.125	0.93	0.109	0.78
test5	0.78	0.156	0.109	0.124	0.94
test6	0.93	0.156	0.109	0.14	0.94
test7	0.109	0.172	0.124	0.14	0.11
test8	0.109	0.172	0.14	0.156	0.11
test9	0.125	0.187	0.156	0.171	0.125
test10	0.125	0.187	0.187	0.171	0.141
test20	0.187	0.265	0.327	0.234	0.406
test30	0.234	0.406	0.561	0.28	0.5

will get the same output for a particular thread schedule each time we run it.

In the bank account scenario, Chess provides a sequence of wait and release operations so that the when one thread is being carried out; the other thread waits and does nor execute unless the first thread releases the resource it was holding. In this way, concurrently executing operations do not interrupt each other and can work smoothly producing a stable output.

When Chess finds a bug in the program, it saves the depiction of the thread schedule in disk, which caused this bug. Then following the same thread schedule for running a program, chess can reproduce the bug. In case of failure of a test, the thread schedule stored in disk by Chess can be used to regenerate that schedule which actually caused error and lead to failure. This schedule can then be debugged and whenever the test fails, Chess will check this schedule and debug it in order to fix the program. This provides efficiency as the debugging of a multithreaded program is reduced and instead of debugging all thread schedules, only the error-causing schedule is debugged.

EXPERIMENT

We carried out number of experiments using different number of threads in each experiment in order to determine the behavior pattern of results and to check out how Chess handles and identifies bugs in multithreaded programs. We

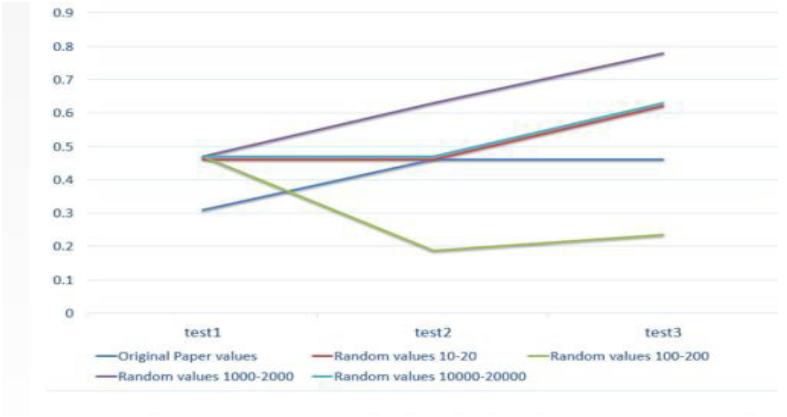


Fig. 1: The Execution time to find multithreading error in second using 2 threads, 15 execution steps and 3 tests

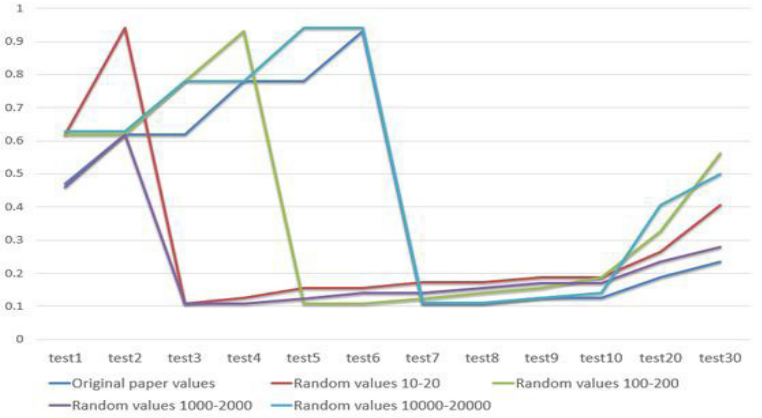


Fig. 2: The Execution time to find multithreading error in second using 4 threads and 42 execution steps and 39 tests

added more threads in the multithreaded program by using random function in C# testing code. This can be done by simply adding the C# statement random. NextDouble to the original testing input code.

In the experiment, there are attempts to add more threads using the random functions in C# testing code from the original paper. The experimentation is possible through the injection of the C# statement randomly. From there, doubling of the original testing code for inputs is necessary to ensure accuracy. Then, there is the multiplication of the number of experiments using a companion of four amounts of the available threads and four ranges of some random values. Ultimately, the companion produces sixteen experiments.

Results and Findings

The following results were obtained after re-running the threads again and again using random inputs. As table 1,2,3,4 figure 1,2,3,4 shows the observation was that the behavior pattern remains the same and the same output is generated for each run irrespective of the number of times a thread is executed.

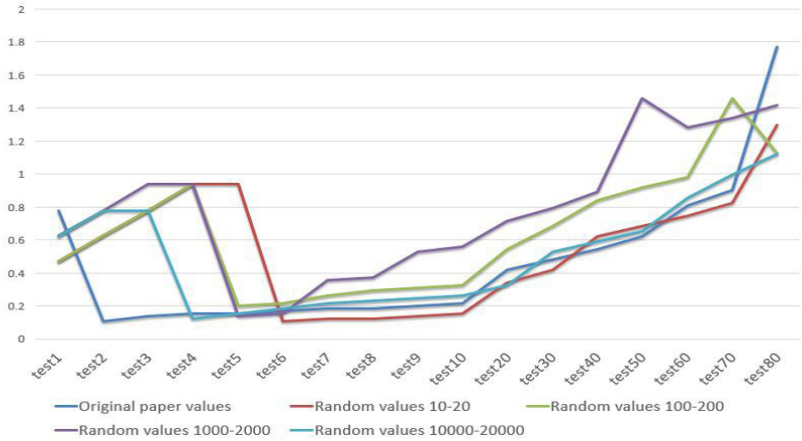


Fig.3: The Execution time to find multithreading error in second using 8 threads and 78 execution steps and 87 tests

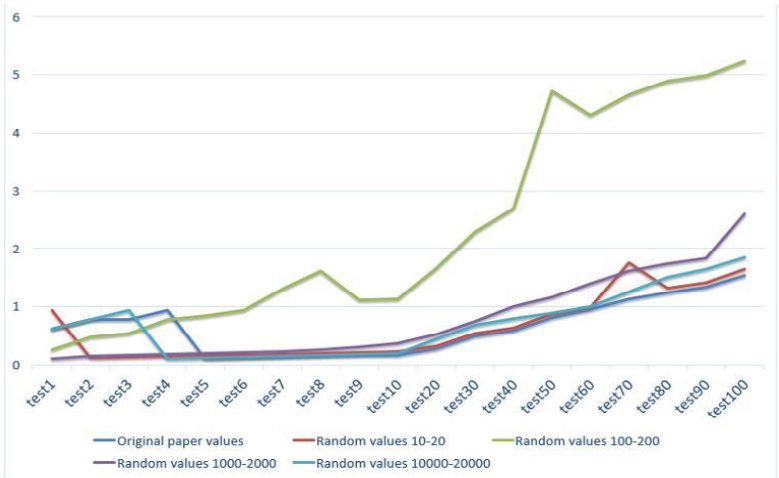


Fig.4: The Execution time to find multithreading error in second using 16 threads and 150 execution steps and 183 tests

Table 3: The Execution time to find multithreading error in second using 8 threads and 78 execution steps and 87 tests

	Original paper values	Random values 10-20	Random values 100-200	Random values 1000-2000	Random values 10000-20000
test1	0.78	0.47	0.47	0.63	0.62
test2	0.11	0.63	0.63	0.78	0.78
test3	0.141	0.78	0.78	0.94	0.78
test4	0.156	0.94	0.94	0.94	0.125
test5	0.156	0.94	0.203	0.141	0.156
test6	0.172	0.11	0.219	0.156	0.187
test7	0.188	0.125	0.266	0.359	0.218
test8	0.188	0.125	0.297	0.375	0.234
test9	0.203	0.141	0.312	0.531	0.25
test10	0.219	0.156	0.328	0.562	0.265
test20	0.422	0.344	0.546	0.718	0.328
test30	0.484	0.422	0.687	0.796	0.53
test40	0.546	0.624	0.843	0.89	0.593
test50	0.624	0.687	0.921	1.46	0.655
test60	0.812	0.749	0.983	1.28	0.858
test70	0.905	0.827	1.46	1.342	0.998
test80	1.77	1.3	1.124	1.42	1.123

Table 4: The Execution time to find multithreading error in second using 16 threads and 150 execution steps and 183 tests

	Original paper values	Random values 10-20	Random values 100-200	Random values 1000-2000	Random values 10000-20000
test1	0.63	0.94	0.265	0.11	0.63
test2	0.78	0.125	0.484	0.156	0.78
test3	0.78	0.14	0.546	0.172	0.94
test4	0.94	0.156	0.78	0.188	0.11
test5	0.11	0.172	0.842	0.203	0.125
test6	0.125	0.172	0.936	0.219	0.141
test7	0.141	0.187	1.3	0.234	0.156
test8	0.156	0.203	1.61	0.266	0.172
test9	0.172	0.218	1.108	0.312	0.188
test10	0.172	0.234	1.123	0.375	0.203
test20	0.281	0.328	1.654	0.531	0.453
test30	0.5	0.546	2.28	0.749	0.687
test40	0.593	0.64	2.699	0.999	0.796
test50	0.812	0.874	4.72	1.155	0.89
test60	0.952	0.983	4.306	1.389	0.999
test70	1.124	1.76	4.649	1.607	1.248
test80	1.233	1.31	4.883	1.732	1.498
test90	1.326	1.404	4.992	1.826	1.638
test100	1.529	1.638	5.242	2.6	1.841

CONCLUSIONS

Chess is capable of exploring all schedules that may range up to thousand or more and thus it is able to find errors in a program while other testing tools do not check all schedules and will therefore be unable to detect bugs in a program. Chess testing mechanism can be applied by inserting the ChessSchedulerClass at the beginning and end of each thread in the original test code. This class enables efficient checking of bugs in each thread. Chess is capable of blocking threads and providing serialized execution of threads so that they do not interfere with each other and do not produce invalid outputs.

After performing the various experiments and from the results obtained, we claim that CHESS is a proficient multithreading testing tool that can find bugs in multithreaded programs quickly and efficiently. Random input values were given to the tool and the same thread was executed repeatedly with those random values but it did not lead to any non-deterministic output. The test rather ran in a productive fashion and successfully detected bugs in the code thus CHESS provides predictive and progressive testing for multithreaded codes. There is a significant increase in the in the number of executions causing the experiments to take long than expected. Therefore, the results represent typical considerations in the multithreading scenario.

REFERENCES

1. Ball, T., Burckhardt, S., de Halleux, P., Musuvathi, M., & Qadeer, S., "Predictable and Progressive Testing of Multithreaded Code," *IEEE Software*, **28**(3), (2011).
2. Blumofe, R., Joerg, C., Kuszmaul, B., Leiserson, C., R, all, K., & Zhou, Y., "Cilk: An efficient multithreaded runtime system," *Journal Of Parallel And Distributed Computing*, **37**(1), 55-69, (1996).
3. Farchi, E., Nir, Y., & Ur, S. (2003), "Concurrent bug patterns and how to test them," 7.
4. Jagannath, V., Gligoric, M., Jin, D., Rosu, G., & Marinov, D. (2010), "IMUnit: improved multithreaded unit testing," 48-49.
5. Musuvathi, M., & Qadeer, S., "CHESS: Systematic stress testing of concurrent software," *Springer*, 15-16, (2007).
6. Musuvathi, M., & Qadeer, S. (2007), "*Iterative context bounding for systematic testing of multithreaded programs*," **42**(6), 446-455.
7. Rinard, M., "Analysis of multithreaded programs," *Springer*, **1**-19, (2001).
8. Sen, K., Rocsu, G., & Agha, G., "Detecting errors in multithreaded programs by generalized predictive analysis of executions," *Springer*, 211-226, (2005).
9. Sen, K., Rosu, G., & Agha, G., "*Runtime safety analysis of multithreaded programs*," **28**(5), 337-346, (2003).
10. Souza, S., Brito, M., Silva, R., Souza, P., & Zaluska, E. (2011), "Research in concurrent software testing: *a systematic review*," 1-5.