28TH ANNUAL
PACIFIC NW
SOFTWARE
QUALITY
CONFERENCE

OCTOBER 18TH – 19TH, 2010

ACHIEVING QUALITY IN A COMPLEX ENVIRONMENT

Conference Paper Excerpt from the

CONFERENCE PROCEEDINGS

Testing Concurrency Runtime via a Stochastic Stress Framework

Atilla Gunal, Rahul V. Patil
Technical Computing, Microsoft
atgunal@microsoft.com, rpatil@microsoft.com

Abstract

The non-linear interaction of many software components makes quality assurance a complex problem even for traditional serial code. Concurrency and interaction from multiple threads add an additional dimension to software complexity. This extra dimension introduces unique bug types such as deadlocks, live-locks, and race conditions.

In this paper, we will describe how a solid stress framework, complete with integrated structured randomization and methodical meddling of temporal properties makes practical software quality assurance possible. Specifically, we discuss the methods and practices applied to provide solid assurance to a critical commercial component – the native Concurrency Runtime [3] stack from Microsoft. First, by applying random distributions in individual tests and integrating such individual tests via a statistically fair scheduler, we describe how to cope with traversing the seemingly infinite interaction patterns. Second, we will expose how such testing helps identify hangs stemming from deadlocks and live-locks. Third, we will talk about injecting randomization to the temporal properties of the software system methodically, and how that can be used to assure that we find bugs with reasonable probabilistic expectation. We will conclude with a brief survey of the effectiveness of our stochastic stress framework over other tools.

Biography

Atilla Gunal is a software development engineer in test at Microsoft Technical Computing Group. He has been thinking, implementing and improving Concurrency Runtime testing strategies since 2007. Before Microsoft, he has worked in the defense industry in Turkey for 7 years in positions ranging from Software Engineer to Systems Engineer and Technical Project Management. Atilla holds a BSc degree from Istanbul Technical University Mathematics Engineering and a Master's degree in System Analysis. He has relocated to Redmond to join Microsoft as he was pursuing his Computer Engineering PhD degree in Istanbul. His interests are test automation, concurrency testing, modeling with math and applications with software.

Rahul V. Patil is a senior QA lead for Microsoft's new native Concurrency Runtime technologies. As a QA lead in charge of a brand new concurrency platform, Rahul has had to address reporting quality on new risks introduced by non-determinism inherent in concurrent software. He also holds a Master's degree in Software Engineering and has been working at Microsoft for the past 6 years, testing various SDKs and platforms.

1. Introduction

Even with the improvements in software engineering [9], the process of developing software is still complex and error-prone. A report from Standish Group [1] indicates that software quality has improved from 1994 to 2006. However, the failure rate of software projects is still a significant 46%. Holzmann [2] argues that for traditional serial software it is impossible to provide input data with all possible variations for a complete testing. The situation is even more complex when the input is coming from multiple threads where the relative ordering of events yields different run-time behavior. With multi-core CPUs becoming prominent, parallel applications will become more prevalent and new quality risks will become a significant concern.

In this paper, we describe how we used a stochastic stress framework that aims to overcome the complexities of testing concurrent software. We will start with a brief description of the Concurrency Runtime, then define a model for testing highly concurrent software and talk about its implementation characteristics. We will focus on three parts of the model that make it an effective bug detector:

i) Ability to integrate random distributions in test code that represent user scenarios.
ii) Ability to combine multiple tests that act on shared software under test to achieve complex scenarios.
iii) Ability to randomize thread executions for finer grain interleavings.

We will conclude with a survey of bugs found with the stress framework together with a comparison of different approaches for concurrency testing.

We assume that the reader has basic knowledge about parallel programming using threads and is familiar with typical concurrency bugs. Reader is encouraged to refer to [7] for a discussion on concurrency bugs.

2. Software under test: Concurrency Runtime

The Concurrency Runtime is a C++ library that helps developers apply parallel programming in their applications under the Visual Studio 2010 development environment. The overall architecture is shown in Figure 1.

The resource manager is a process wide entity that arbitrates threads as a resource depending on the load of the schedulers. The task schedulers are long lived entities that apply smart heuristics to maximize throughput of parallel tasks. The Agents Library as well as the Parallel Patterns Library (PPL) provide an Application Programming Interface (API) that allows users to express concurrency and describe tasks in a simple way (an example is the parallel_for API that can easily parallelize a serial for loop automatically).

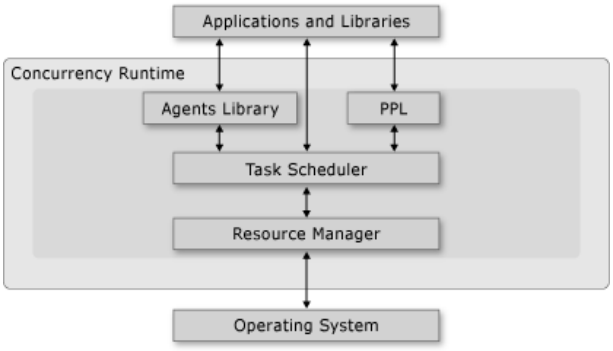

Figure 1 - Component stack of the Concurrency Runtime

The goal for Quality Assurance (QA) with respect to concurrency risks is to ensure that the runtime is free of:

- Adverse race conditions that lead to application malfunctions or crashes.
- Deadlocks that arise out of bad synchronization schemes.
- Task starvation that arises out of bad scheduling heuristics.

There are two important aspects of the runtime for our discussion around risks to concurrency testing:

i) Synchronization used within the runtime

The Concurrency Runtime has to ensure thread safety of its internal states while maintaining the goal of maximizing throughput. To achieve that, its developers applied synchronization in the following order of preference:

a) Do not use synchronization if the race is harmless (benign)
b) Use interlocked operations and memory barriers for synchronization
c) Use multiple fine grain locks
d) Use coarse grain locks

In the order of occurrence, a) has best performance with highest QA risk, while d) has the worst performance with lowest QA risk. Hence, as QA we treated the Concurrency Runtime as a huge lock free algorithm and designed tests that are able to traverse this pool of race conditions.

ii) Temporal and dynamic interaction of runtime components

The scheduler and resource management pieces of the runtime are long running, living entities of the process. They traverse many states in their lifetime, triggered by the temporal and dynamic interaction of various components and user actions. Many defects may be unique to traversing to certain states with a specific temporal precondition. Thus, it is very important to have a framework that is capable of traversing this huge state space with seemingly infinite temporal preconditions.

3. Model for Concurrency Testing

In this section, we will introduce a Concurrency Testing Model (CTM) to work with the challenges of testing concurrent software. CTM can be defined as a set of Stress Tests (ST) implemented on top of a Stress Framework (SF).

$$CTM = \{SF, \{ST_i\}\} \quad i = 1,2, \dots, TestCount$$

We can further decompose the Stress Framework (SF) into the following elements:

- Randomizer (R)
- Individual Stress Test Structure (ISTS)
- Stress Test Combiner Structure (STCS)
- Stress Test Driver (STD)
- Runtime Randomizations (RR)

$$SF = \{R, ISTS, STCS, STD, RR\}$$

A block diagram of the proposed CTM can be seen in Figure 2. The following paragraphs describe each component in more detail.

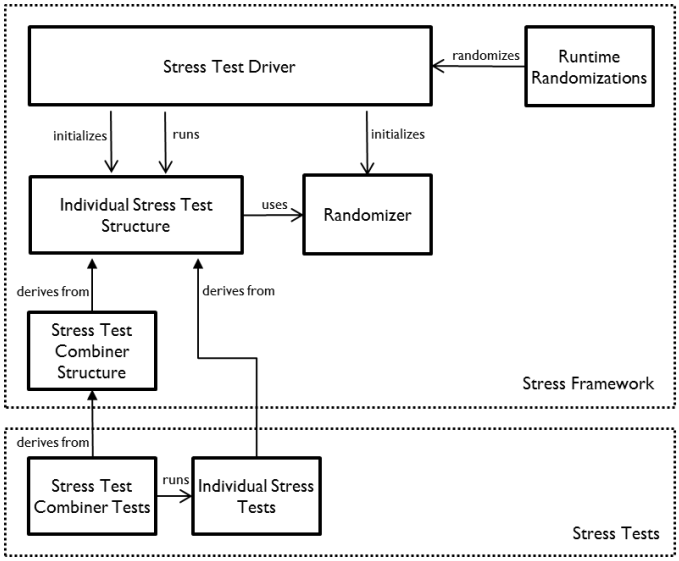

Figure 2 - Block diagram of the Concurrency Testing Model

3.1. Randomizer

Running the same static test over and over has a very small potential of exposing different concurrency bugs. Burckhardt et al. [5] for example, observed that bug finding capability of tests that depend on external factors only such as the OS scheduler or the load of the operating system instance is observed to be very low. Thus, it is essential to have an element in the model that varies the execution path. In our model that element is the randomizer which helps test developers to generate test parameters from random distributions that represent the scenario of the individual test. The parameters can be inputs to the software under test or even the control logic decisions of what action to take next.

3.2. Individual Stress Test Structure (ISTS)

ISTS defines the interface that each individual test has to implement. The interface makes it possible to implement tools such as test drivers to create, initialize, cleanup and run until cancellation.

3.3. Stress Test Driver

Given an individual test that follows the ISTS interface, the driver can execute that test until cancelled. The driver can also be used to implement common verifications and debugging support such as:

i) Hang/Starvation/Deadlock detection
The driver will monitor the individual test execution and report a hang if the test does not return in a predefined amount of time.

ii) Memory leak detection
The driver will monitor tests for memory leaks.

iii) Unhandled exception handler
The driver will capture the state of software under test upon an unhandled exception during an execution of the test. Typically exceptions result from state corruption due to underlying races.

3.4. Stress Test Combiner Structure (STCS)

This component targets concurrency bugs that can only be exercised as a result of running multiple individual stress tests simultaneously from multiple threads (i.e. by exploring the thousands of states due to the dynamic interaction of the different components, resulting from many user actions). An implementation of an STCS takes many individual tests and runs them simultaneously. STCS is only possible because there is a well-known ISTS. STCS, itself implements the ISTS. This makes the design of stress test driver simpler by being agnostic of the test type.

3.5. Stress Tests

Stress tests can be grouped into two categories:

- Tests that implement the ISTS interface (Individual Stress Tests)
- Tests that implement the STCS interface (Stress Test Combiner Tests)

3.5.1. Individual Stress Tests

Individual stress tests are scoped tests that focus on a specific use case. These can also make up the building blocks of more complex tests. Individual stress tests can further be examined in two categories:

i) Stateless

Stateless tests have a repository of objects of software under test and associated actions on objects. The actions are executed according to a random distribution. Such tests are defined as:

$$Stateless = \left\{\{O_i, A_{ij}\}\right\} i = 1,2, \dots, ObjectCount, j = 1,2, \dots, ActionCount$$

Where O_i denotes the object under test and A_{ij} denotes the actions applicable on i^{th} object.

Stateless tests do not maintain state between action executions, nor do they care about semantics of sequences of such actions. They may operate on the same object for more complex state transitions for the manipulated O_i.

It is essential to note the tight coupling of actions with their parameters and a corresponding distribution to walk different paths.

An example of a stateless test applied in Concurrency Runtime is the parallel_for stress where parallel_for is called with uniformly distributed numbers within the domain of parallel_for index type. Here the object is an instance of parallel_for class and the action is the () operator. Note that test iterations don't store state and do not remember the previous inputs to the parallel_for () operator.

ii) Scenario based / State full

Scenario based tests have a predefined structure on how to manipulate the object under test. The structure can be viewed as a set of Action Groups (AG) where each group manipulates the Internal State (IS) of the test which is passed to the next group.

$$AG = \left\{\{O_i, A_{ij}, IS\}\right\} i = 1,2, \dots, ObjectCount, j = 1,2, \dots, ActionCount$$

$$Scenario\ based = \{AG_k\}\ k = 1,2, \dots, GroupCount$$

Stateless tests can be seen as a specialization of scenario based where $k = 1$ and $IS = \{\}$. Scenarios end up targeting the semantics of sequences of actions.

An example of a scenario based test is a tree traversal scenario where nodes of a tree are traversed via parallel constructs of the Concurrency Runtime such as; parallel_for, parallel_invoke, task_group, etc... Here, IS is the tree data structure, O_i are the parallel constructs and A_{ij} are actions like scheduling a task, cancelling a parallel_for.

3.5.2. Stress Test Combiner Tests

Individual stress tests are not able to trigger race conditions that require more than one scenario running at the same time. It is important to have a combiner test that takes multiple individual stress tests and integrates them to compose more complex scenarios. Most likely there will be one combiner test for a group of related objects of the software under test. Combiner tests can also maintain a pool of related objects and pass them to the individual tests to have them operate collectively on same objects. This will result in more complex state transitions by reusing existing individual tests.

3.6. Runtime Randomizations

It is important to note that for lock free software such as the Concurrency Runtime, the threads can execute for their full quantum without blocking. Therefore even with randomization embedded into the tests there can be blind spots where the execution is not interrupted. In order to overcome this issue, runtime randomizations are introduced. The idea behind the Runtime Randomization is to introduce even more interleavings by randomizing the thread schedules of a given process.

4. CTM Implementation for Concurrency Runtime

In this section we will describe the implementation characteristics of the proposed CTM for the Concurrency Runtime.

4.1. Randomizer

Randomizer is implemented as a library that provides configurable randomization support to the stress tests. Each test has to define the random distribution type and parameters in a configuration file, that best models the expected customer usage. There are four different types of distributions supported:

i) Uniform distribution
This will generate numbers homogeneously in the range of $[Min, Max]$

ii) Gaussian distribution
This will generate numbers according to the Gaussian curve characterized by Mean and $\mathrm{Standard\ Deviation}$.

iii) Exponential distribution
This will fit an exponentially decreasing curve from Min to Max and generate random numbers accordingly.

iv) Weighted distribution
This will generate numbers from an $[\mathrm{Min_i}, \mathrm{Max_i}]$ interval homogenously where selecting an interval has the probability

$$\mathrm{P_i}\ , \mathrm{i} = 1,2, \dots, \mathrm{IntervalCount}$$

where

$$\sum_{i=1}^{\text{IntervalCount}} P_i = 1$$

Thus Weighted Distribution (WD) can be defined as:

$$\text{Interval}_i = \{\text{Min}_i, \text{Max}_i, P_i\}\ i = 1,2, \dots, \text{IntervalCount}$$

$$\text{WD} = \{\{\text{Interval}_i\}\}\ i = 1,2, \dots, \text{IntervalCount}$$

Note that weighted distribution can be used to approximate any random distribution:

Let $P(x)$ be the probability density function of a random variable x, where $x \in [\min, \max]$. The $[\min, \max]$ can be partitioned into $\{[\min_i, \max_i]\}$ such that for any $x_0, x_1 \in [\min_i, \max_i]$ $(\text{Eq. } 1)$ holds.

$$|P(x_0) - P(x_1)| < \text{Threshold} \qquad (\text{Eq. } 1)$$

P_i would be then defined as:

$$P_i = \int_{\min_i}^{\max_i} P(x)dx$$

Note that Threshold as defined in $(\text{Eq. } 1)$ determines the precision of the approximation; the smaller it is the better approximation is.

4.2. Individual Stress Test Structure

Individual stress tests are derived from a TestBase class for which the driver uses to initialize, start, cancel and clean-up the test. Each test implements the following:

Exported _cdecl APIs:

- TestBase* CreateInstance(): To create the stress test
- Void DestroyInstance(TestBase*): To destroy the stress test

Virtual methods:

- OnTestStart(): To initialize the stress test.
- TestMain(): To define what the test will do. This method is executed repeatedly until the test is canceled. It is expected that this method returns before the hang detection configuration parameter for the test. Otherwise test is considered as hung, the process state will be dumped and process will be exited.
- OnTestCancelled(): To handle test cancellation. Test is expected not to add more work and finish its tasks as quickly as possible upon cancellation.
- OnTestEnd(): To clean up test resources

4.3. Individual Stress Test Combiner Structure

Stress Test Combiner Structure is encapsulated in a class (MeltingPotBase) that each combiner derives from. The base class handles the common functionalities as follows:

- Loading a collection of individual stress tests at start up
- Scheduling a subset of tests simultaneously on multiple threads

- Collecting statistics about each test
- Reporting statistics upon failure

MeltingPotBase also has virtual methods that the combiner implementations can customize to do specific actions at individual stress load / unload, start and stop time.

4.3.1. The scheduling algorithm

MeltingPotBase implements a fair scheduler to decide which set of stress tests to execute in parallel and how long. Regardless of a test being short running or long running, the standard deviation of the total execution time of each test remains low. This is to prevent long running tasks from getting an uneven share of execution time over short running tests. The algorithm keeps track of average execution time of each test (AverageTestDuration) and creates a weighted distribution (WD) made up of intervals that represent the probability of selecting an individual test as follows:

$$\mathrm{WD} = \left\{\{\mathrm{Interval_i}\}\right\}\ \mathrm{i} = 1,2,\dots,\mathrm{TestCount}$$

$$\mathrm{Interval_i} = \{\mathrm{i}, \mathrm{i}, \mathrm{P_i}\}\ \mathrm{i} = 1,2,\dots,\mathrm{TestCount}$$

where

$$\mathrm{P_i} = \frac{\frac{1}{\mathrm{AverageTestDuration_i}}}{\sum_{\mathrm{j=1}}^{\mathrm{TestCount}} \frac{1}{\mathrm{AverageTestDuration_j}}}\ \mathrm{i} = 1,2,\dots,\mathrm{TestCount}$$

The algorithm will select N tests by generating N numbers from the WD where each number corresponds to the i^{th} test. The goal is to have all tests converge to the same execution time. Therefore for a faster convergence the algorithm runs the selected tests $\mathrm{C_i}$ number of times where:

$$\mathrm{C_i} = \frac{\max(\mathrm{AverageTestDuration_j})}{\mathrm{AverageTestDuration_i}}$$

Here are the algorithm steps:

- Create a weighted distribution (WD) inversely proportional to average run duration of each test
- Generate N unique numbers from the WD that indicates the N tests to be executed
- Attach each test to a thread
- Execute N tests in N threads C_i number of times
- Update statistics
- If cancelled exit otherwise continue from 1st step

4.4. Stress Test Driver

The implementation of the stress test driver is generic for all software systems under test. It implements the capabilities outlined under 3.3 Stress Test Driver.

4.5. Stress Tests

4.5.1. Individual Stress Tests

The Concurrency Runtime stress tests are composed of 27 individual tests where 25 of them can be combined with others. All tests use random distributions (there are 145 distributions in total). The test scope, count, and functionality is designed after decomposing the runtime into components,

understanding each component, their relation with each other together with the scenarios that could trigger interesting race conditions. Understanding functional specifications, design documents, source code and collaborating with developers are good practices to follow for a better test design.

4.5.2. Stress Test Combiner Tests

There is only one STCS implementation for the Concurrency Runtime implementation of the model which is called the Melting Pot. Melting Pot targets integration around the scheduler which is the most critical component of the runtime. It maintains a pool of schedulers and populates individual tests with an instance of a scheduler from the pool.

4.6. Runtime Randomizations

The runtime randomization (as described in section 3.6) applied for the Concurrency Runtime testing is called the Thread Randomizer. The randomizer will basically sleep for a configurable duration, suspend a number of target test execution threads, resume them and then go back to sleep. The number of threads to suspend and the duration of the sleep follow a random distribution and can be configured. This configuration enables different interleavings without modifying test behavior. Thread Randomizer is a simple external process that randomizes the thread scheduling without any modification to SUT or the tests. The idea is similar to [4,8] where instead, we use a uniform distribution to decide injection points.

It can be shown that the probabilistic guarantee of finding a bug of depth one is in the order of $O\left(\frac{1}{N}\right)$ where N (N>1) is the number of threads in the process (see Appendix I – Thread Randomizer bug finding probability proof).

5. Results

We will compare the results of applying concurrency testing model with other methodologies used for testing the runtime. In addition, the bugs found by our methodology will be examined to describe the effectiveness of techniques used. Finally a comparison with similar concurrency testing methodologies will be provided.

It is also important to note that the Concurrency Runtime team has no reported Watson bug (customer reported crash database) which is another indication of the effectiveness of reliability testing done.

5.1. Survey of Bugs Found

5.1.1. Overall Efficiency of Bug Finding Activities

We will lay out the efficiency of the stress testing by ranking it among other bug finding activities. The test methodologies that Concurrency Runtime has applied are as follows:

- Feature Testing (aka Functional Testing)
  Functional tests implemented by the test team
- Stress
  Tests that are implemented by using the methodology described in this paper
- Adhoc Testing
  End to end scenario based tests
- Code Reviews
- Unit Testing
  Functional tests implemented by the developer team

- Performance Testing
- Other
  Other buckets like bugs coming from Microsoft internal usage, security, etc…

The number of bugs for each methodology can be seen in Figure 3.

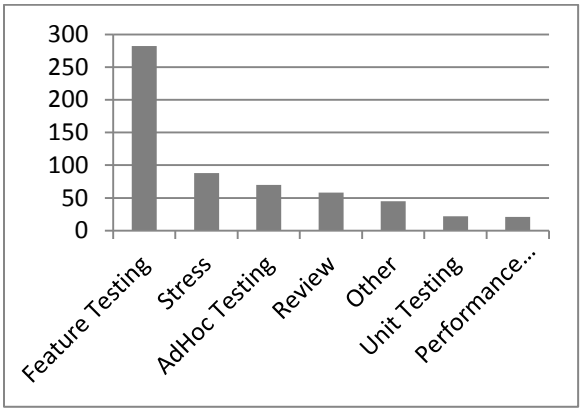

Figure 3 - Number of bugs for per testing methodology

One way to look at efficiency is the bugs found per lines of code written. We will evaluate the efficiency of top two bug finding methodologies; feature testing and stress.

Table 1 - Evaluating top two bug finding activity efficiencies

Bug Finding Methodology	Number of Bugs Found	Lines of code
Feature Testing	282	163552
Stress	88	63751

Data in Table 1 reveals the efficiency per 1000 lines of code as:

Efficiency of feature testing: 1.72
Efficiency of stress testing: 1.38

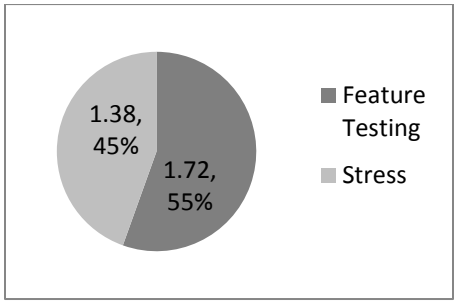

Figure 4 – Efficiency of top two bug finding methodologies

This indicates that even though the number of bugs found with feature testing is more than triple the number of bugs found with stress testing, the efficiency of the methodologies are comparable.

It is important to note that the bug types that are found by the stress tests are much more complex in nature with respect to reproducibility which we have not taken into consideration in evaluating the

efficiency. Another very critical point to make here is that the stress tests are typically developed after the functional tests have verified that the core testing is done.

5.1.2. Type of Bugs Found by Stress Tests

The analysis of bugs found can be another factor in evaluating the concurrency testing methodology. The results for the Concurrency Runtime experience can be found in Figure 5.

Access violations are a classic symptom of harmful race conditions. Live-locks and deadlocks are classic symptoms of incorrect synchronization and scheduling. The developer and test asserts1 are a symptom of incorrect state transition. By noting the variety of bugs that Stress finds, we can also assert its effectiveness.

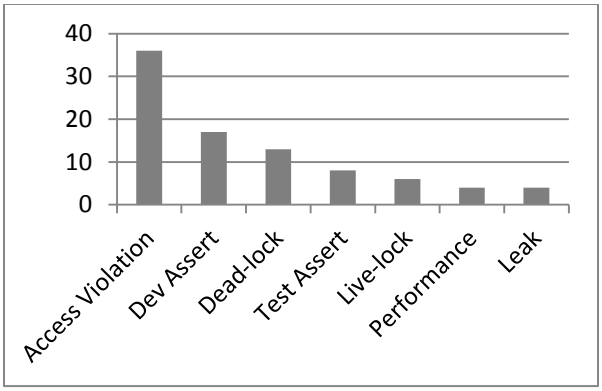

Figure 5 – Type of bugs found by stress tests

The performance and leak buckets also appear in the analysis. Even though stress does not target performance verification, performance characteristics of the software under test may end up making the test take too long under heavy load and not complete in a predefined amount of time. Such cases are detected as hangs by the stress framework but further analysis yields to underlying performance issue. On the other hand, certain leak bugs are exposed due to race conditions or after running a long duration which is very hard to catch on environments without targeting stress testing.

5.1.3. Effectiveness of Stress Tests

We have made a distinction between individual stress tests and the melting pot that combines individual tests. We were looking at melting pot as the strategy to traverse various race conditions and exercise various scenarios that are not possible by running the individual tests. The following graph is a validation of that strategy that melting pot on its own has detected almost 1/3 of total stress bugs (The analysis is based on the 'repro steps' field in the bug database. If repro steps refer to the melting pot, the bug is marked as detected by melting pot). This is even more important since the individual tests were first stabilized outside of the melting pot environment. Only after the individual test runs successfully is it added to the melting pot mix.

The second biggest effective test is the tree traversal test case which is a unique combination of stressing all parallel constructs of the Concurrency Runtime under a tree data structure traversal user scenario. Tree traversal also involves verification which makes it even more powerful bug detector. For a detailed effectiveness of stress tests see Figure 6. Melting pot and tree traversal indicates that composition of tests or features in order to exercise more complex scenarios has a greater chance of detecting bugs.

[1] There is a natural tension between using random distributions and being able to inject verifications. The more randomness is introduced the less verification can be done.

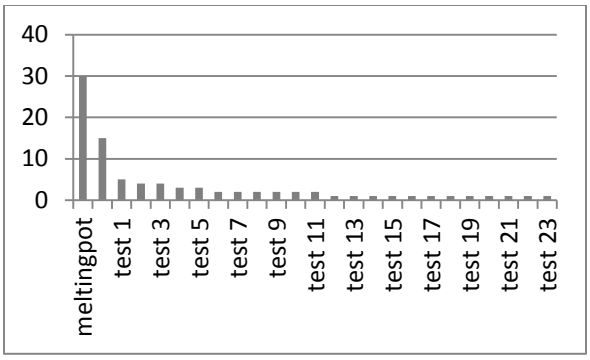

Figure 6 – Effectiveness of stress tests

5.2. Comparison to Other Concurrency Testing Methodologies

5.2.1. Repetitive Functional Test Execution

One way for finding concurrency bugs is reusing functional verification tests and running the tests over and over until cancelled. Such approaches also have capabilities to run tests from multiple threads. We applied this approach to existing functional tests of the Concurrency Runtime. Only the tests that were capable of running with other tests at the same time were harnessed. Tests were selected randomly and ran in multiple threads. Even though the implementation of this approach is easy due to excessive re-use of existing tests, the number of bugs found remain as two which suggests that the methodology is not enough on its own.

The low detection ratio can be explained by two features missing in such an approach:

i) Limited randomization
The functional tests are deterministic in nature; there is minimal tolerance to flakiness in functional tests. Thus the only way to exercise different interleavings is to rely on the noise related with the system load or the OS thread scheduler.

ii) Limited shared runtime state
The individual functional tests do not know about other tests in their environment. They do not share state explicitly. They act on their own instances of objects and also clean-up resources every time the test completes. This prevents more complex states from being reached as the software under test resets every time a new test starts.

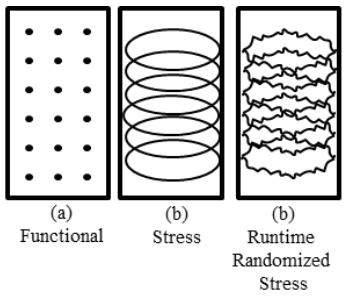

Figure 7 – Depicting functional vs. stress test coverage

In order to visualize the difference between repetitive functional test execution and stress, let us assume that Concurrency Runtime usage scenario space is a two dimensional space[2]. We interpret the coverage of a functional test within this space as a dot (Figure 7a) because of its deterministic input and execution logic. On the other hand, due to the introduction of well-defined random distributions into the test code, we interpret the coverage of stress tests as a collection of regular dots (Figure 7b) and as a result of thread randomization, we interpret runtime randomized stress test coverage as a collection of irregular dots (Figure 7c).

5.2.2. Systematic Concurrency Testing - CHESS

CHESS [6] is a tool which systematically explores the schedules that a concurrency program can exercise. Running a test, which does not introduce randomizations by itself, CHESS will control the scheduling of threads and will exercise a different path each time the test executes. CHESS will also record the scheduling decisions made so that if a specific schedule hits a bug, it can be reproduced. Due to the exponential growth capacity of the search space, CHESS will limit its systematical search with a bound on the number of preemptions it makes on the threads [6].

For software as complex as the Concurrency Runtime, we have observed that systematic approach of CHESS is infeasible due to the length of time to complete testing. Thus, a stochastic walk in our framework provided more practical coverage and proved to be a better tool to predict quality.

5.2.3. Randomizing Thread Schedules - Cuzz

Cuzz [5] is an implementation of runtime randomization introduced in paragraph 3.6. Cuzz introduces the concept of bug depth as the minimum number of scheduling constraints needed to expose a bug. Cuzz will randomize thread schedules and guarantee to find a bug of depth ‘d’ for a test having ‘k’ scheduling points and n threads as 1 / (n x kd-1) [5]. Here the scheduling points are instructions for which thread preemption has a potential of exposing a bug. Examples of such scheduling points are interlocked operations, critical section operations, etc…

Cuzz and Thread Randomizer have similar goals in that both of them aim to randomize the thread executions so that race conditions are exercised. Cuzz will do this by injecting delays at interesting scheduling points. On the other hand, Thread Randomizer will randomly suspend/resume any thread with no bias to scheduling points. From this perspective, Cuzz has a better probability of finding a bug. On the other hand, Cuzz can have blind spots.

After the public release of the Concurrency Runtime, we have been running stress with Cuzz. We have not yet discovered a bug, which is another indication that the concurrency testing methodology described in this paper is firm.

6. Conclusion

In this paper we have described a framework to cope with the challenges of testing concurrent software. Our model is composed of data driven random distribution support, individual stress tests with parameters coming from random distributions, integration of such individual tests for more complex state transitions and runtime randomizations for finer grain interleavings. By providing real data from the bug database of the Concurrency Runtime and comparing our methodology with other concurrency testing methodologies, we demonstrated the effectiveness of the methodology.

We believe that applying data driven random distributions into test executions, combining existing tests on shared state to explore even more complex scenarios, and introducing thread scheduling randomizations in order to probabilistically guarantee finding concurrency bugs arising from different interleavings is an effective way of testing highly concurrent software.

[2] The assumption is for visualization purposes of our interpretation.

7. Appendix I – Thread Randomizer bug finding probability proof

Let us define the bug depth as the number of ordering constraints needed to be met to end up with a given bug. This is essentially the definition given in [5]. For a simple scenario let us visualize this as follows. Note that two threads are enough to have this bug occur.

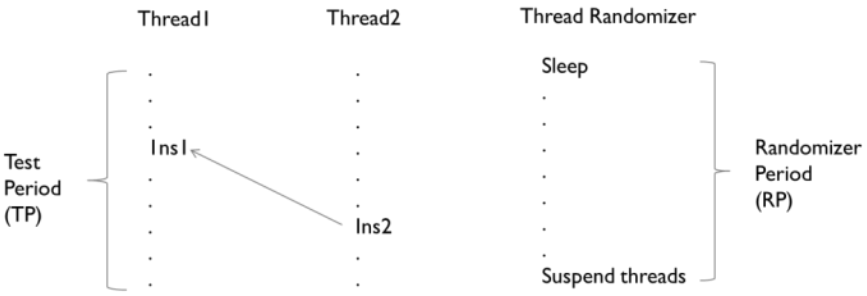

Figure 8 - Thread Randomizer typical runtime behavior for a test with two threads

We will model the test doing the following in a loop:

- Create and start two threads
- Wait for the threads to end

Note that threads are free to do whatever action they want to perform. We will define each execution of the loop as test iteration and the duration of each test iteration as TP. We will denote MINTP as the minimum duration of any possible iteration and MAXTP as the maximum duration of any possible iteration.

Meanwhile Thread Randomizer will do the following in a loop:

- Sleep for its period (RP)
- Wake up
- Suspend threads for Suspend Duration (SD) amount of time

We will define each execution of this loop as randomizer iteration.

For a bug depth of one we will assume that the Thread Randomizer will suspend one thread at a time. Since Thread Randomizer is capable of getting its parameter values from random distributions, we will define RP values coming from a uniform distribution on interval $[\mathrm{MINRP}, \mathrm{MAXRP}]$ and SD values coming from a uniform distribution on interval $[\mathrm{MINSD}, \mathrm{MAXSD}]$.

Note that the bug will occur if Ins1 (instruction 1) happens before Ins2 (instruction 2) (See Figure 8). For this bug to happen under thread randomization:

i- Thread1 needs to be suspended before executing Ins1
ii- Suspension must be long enough for Thread2 to execute Ins2

Since Thread Randomizer is configured to suspend at minimum MINSDms we will assume that Thread Randomizer cannot detect bugs if the duration between Ins1 and Ins2 is longer than MINSD. If MINSD is not enough to hit the mentioned bug then the Thread Randomizer's configuration can be modified to increase the MINSD. Thus we will assume that (ii) is met by definition.

Let us define IPP as the number of instructions that can be executed on the test machine in TPMAXms. Given that Thread Randomizer is unbiased (it interrupts the test process after sleeping randomly between $[\mathrm{MINRP}, \mathrm{MAXRP}]$) each test process instruction has equal chance of being interrupted. Then the probability of inserting a sleep before an instruction at k^{th} location would be $\mathrm{P} = \frac{\mathrm{k}}{\mathrm{IPP}}$. Let $\mathrm{k1}$ be the index of

Ins1 with respect to Thread1's executed instruction set. If there are N threads, then probability of selecting Thread1 and injecting before Ins1 would be:

$$P(Thread1) = \frac{k1}{IPP} \times \frac{1}{N}$$

Since Thread Randomizer can miss some test iterations completely while sleeping, we need to take into account the probability of selecting test iteration i. The probability of hitting test iteration within Thread Randomize iteration is the ratio of test iteration to randomizer iteration.

$$P(TestIteration_i) = \frac{TP}{RP + SD}$$

A lower bound would be:

$$P(TestIteration_i) > \frac{MINTP}{MAXRP + MAXSD} \quad (Eq.2)$$

Note that the following must be satisfied to have $P(\text{TestIteration}_i) < 1$:

$$MAXRP + MAXSD \geq MINTP$$

Then probability of finding the bug in test iteration i would be:

$$P(bug) = P(Thread1) \times P(TestIteration_i)$$

$$= \left(\frac{k1}{IPP} \times \frac{1}{N}\right) \times P(TestIteration_i) \qquad (Eq.3)$$

Lemma 1: Given the fact RP and SD are derived from a uniform random distribution, $k1$ can be estimated as $\frac{IPP+1}{2}$.

Proof: Since RP and SD values are uniformly distributed, the rank of $k1$ with respect to Thread Randomizer iteration will be a uniformly distributed between 1 and IPP. Possible values for $k1$ become:

$$k1 \, \epsilon \, K = \{1, 2, \dots, IPP\}$$

where each value has probability of $\frac{1}{IPP}$. Let us visualize this as shown in Figure 9.

Thus the estimated value of $k1$ becomes:

$$E(k1) = \sum_{k \in K} \frac{1}{IPP} \times k = \frac{1}{IPP} \times (1 + 2 + \cdots + IPP) = IPP \times \frac{IPP + 1}{2IPP} = \frac{IPP + 1}{2}$$

Replacing $E(k1)$ in Eq.3, we have:

$$P(bug) = \left(\frac{\frac{IPP + 1}{2}}{IPP} \times \frac{1}{N}\right) \times \; P(TestIteration_i)$$

$$= \left(\frac{IPP + 1}{2NIPP}\right) \times \; P(TestIteration_i)$$

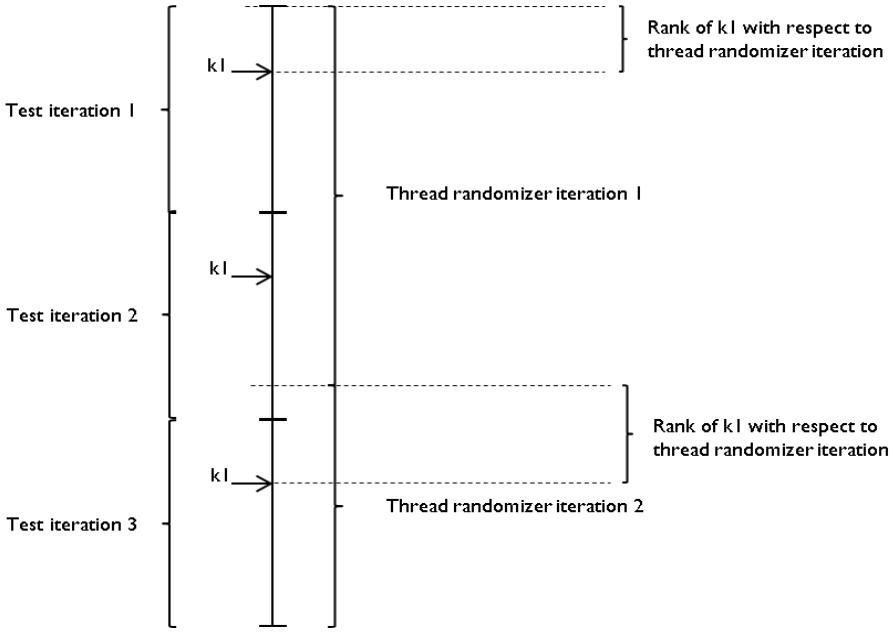

Figure 9 - Runtime visualization of thread randomization

Since we are interested in a lower bound for the probability we can lower it by dropping the positive terms. Thus we have:

$$P(bug) > \left(\frac{IPP}{2NIPP}\right) \times P(TestIterationi)$$

$$= \frac{1}{2N} \times \frac{TP}{RP + SD} \quad (Eq.4)$$

Replacing Eq.2 in Eq.4:

$$> \frac{1}{2N} \times \frac{MINTP}{MAXRP + MAXSD}$$

Thus the probability of detecting the bug of depth one in a process having N threads is in the order of $O(\frac{1}{N})$.

This indicates that running the test for $\frac{1}{P(bug)}$ times will reveal the bug.

References

[1] Narciso Cerpa and June M. Verner. “Why did your project fail?” Communications of the ACM, December 2009, Volume 52 Issue 12.

[2] Gerard J. Holzmann. “The logic of bugs.” SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering, November 2002.

[3] Concurrency Runtime, http://msdn.microsoft.com/en-us/library/dd504870(VS.100).aspx

[4] Y. Ben-Asher, Y. Eytani, E. Farchi, S. Ur. “Producing scheduling that causes concurrent programs to fail.” PADTAD '06: Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging, July 2006.

[5] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, Santosh Nagarakatte. “A randomized scheduler with probabilistic guarantees of finding bugs.” ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems, March 2010.

[6] Madan Musuvathi. “Systematic concurrency testing using CHESS.” PADTAD '08: Proceedings of the 6th workshop on Parallel and distributed systems: testing, analysis, and debugging, July 2008.

[7] Rahul V. Patil, Boby George. “Tools and Techniques to Identify Concurrency Issues.” MSDN Magazine, June 2008. http://msdn.microsoft.com/en-us/magazine/cc546569.aspx

[8] Yarden Nir-Buchbinder, Shmuel Ur. “Multithreaded unit testing with ConTest.” http://www.ibm.com/developerworks/java/library/j-contest.html

[9] Li Jiang, Armin Eberlein. “An analysis of the history of classical software development and agile development.” Proceedings of the 2009 IEEE International Conference on Systems, Man, and Cybernetics. October 2009, Page(s): 3733 - 3738

Acknowledgements

We would like to thank Mohamed Ameen Ibrahim for sharing his bug trend analysis on the Concurrency Runtime, Steve Gates for providing investigation results of applying CHESS on the Concurrency Runtime as well as his review on the paper, Roy Tan for his review, the Concurrency Runtime Test Team for implementing tests on top of the model and the Concurrency Runtime Developer Team for their support in designing test cases.