28TH ANNUAL
PACIFIC NW
SOFTWARE
QUALITY
CONFERENCE

OCTOBER 18TH – 19TH, 2010

ACHIEVING QUALITY IN A COMPLEX ENVIRONMENT

Conference Paper Excerpt from the

CONFERENCE PROCEEDINGS

THE LAST 9% UNCOVERED BLOCKS OF CODE – A CASE STUDY

Cristina Daniela Manu
cmanu@microsoft.com

Pooja Nagpal
ponagpal@microsoft.com

Donny Amalo
donnya@microsoft.com

Roy Patrick Tan
roytan@microsoft.com

Abstract

Code coverage is an efficient technique for understanding what code has been exercised by an existing test bed, yet it is often debated how much effort should be invested in increasing coverage. This paper is a case study of our code coverage effort during the product development cycle for a component of the .NET Framework 4. At the end of the cycle, we had a focused initiative to increase the code coverage from 91% to 100%. In order to measure the effectiveness of increasing code coverage, we devised a few metrics based on the number and importance of the bugs found, the time invested, the increase in code coverage and bug yield in comparison with other test activities. Our results showed that:

1. It is not prohibitively expensive to achieve effective[1]-100% code coverage.
2. There is no significant increase in the number of bugs found with higher code coverage.

In this paper, we discuss what parts of our development process (such as the propensity of testers to develop more complex scenarios first) may have contributed to this lack of bugs.

Biography

Cristina Manu is a Software Development Engineer in Test at Microsoft, in the Technical Computing group, working on testing libraries for supporting parallel applications. Before joining Microsoft, she was a lecturer at "Politehnica University of Bucharest" teaching mathematics. She holds a PhD degree in mathematics from University Of Bucharest.

Donny Amalo is a Software Development Engineer in Test at Microsoft in the Technical Computing group working on testing libraries for supporting parallel applications.

Pooja Nagpal is a Software Development Engineer in Test at Microsoft in the Technical Computing group working on testing libraries for supporting parallel applications.

Roy Tan is a Software Development Engineer in Test at Microsoft, in the Technical Computing group, working on testing libraries for supporting parallel applications. He earned his PhD in Computer Science at Virginia Tech in 2007.

[1] We define "effective code coverage" as the coverage of all the reachable blocks of code.

1. Introduction

Towards the end of the product development cycle for the Parallel Extensions to the .Net Framework 4 (7), our test team was able to reach 91% code coverage. We wanted to investigate what was left in the uncovered areas to make sure we added tests for it. Further, we wanted to understand the test investment required for increasing code coverage from 91% to 100% and what benefits would there be beyond discovering new bugs.

Managers often struggle with the decision about how many resources to allocate for increasing code coverage values. Several studies (e.g., (1) (2) (3)) have claimed that code coverage is a good technique to measure test effectiveness. In fact, several studies claim that test suites with higher code coverage have higher bug finding rates. A study by Hutchins et al. (1), for example, found that as code coverage goes from 90% to 100%, test suites get exponentially better fault detection rates. Similar work by Frankl and Iakounenko (4) and Andrews et al. (5) also claimed exponentially increasing bug finding ability as code coverage gets closer to 100%.

On the other hand, some research has also found that increasing code coverage is prohibitively expensive, as in the same study above by Andrews et al. (5), and including research done at Microsoft and Avaya (6). This paper is a case study of our code coverage efforts to cover the last 9% of uncovered code.

We show that increasing code coverage from 91% to effectively 100% need not be prohibitively expensive. However, we also show that find increasing code coverage did not find an exponentially increasing number of bugs.

The paper is structured into the following sections:

1. Description of the project and of the test strategies used during the product development cycle.
2. Description of the metrics used for measuring the code coverage.
3. Description of the code coverage efforts.
4. Results and Conclusions

2. The Project

Parallel Extensions to .NET Framework 4 (7) is a set of library additions to the .NET Framework 4 that simplifies concurrent and parallel programming. It consists of three main components:

Task Parallel Library (8) – this provides a rich task-based model for asynchronous and parallel programming. It defines continuation patterns and allows for custom scheduling and task-attachment relationships, while providing support for efficient joins along with cancellation and exception handling models. It also provides Parallel constructs including Parallel.For, Parallel.ForEach, and Parallel.Invoke.

PLINQ (9) – this is a parallel implementation of LINQ to Objects (10);

Data Structures for Coordination (11) – these are a set of important types that round out the Parallel Extensions, supporting necessary patterns and practices for adding parallelism to your applications. Important groups include thread-safe collections and synchronization primitives.

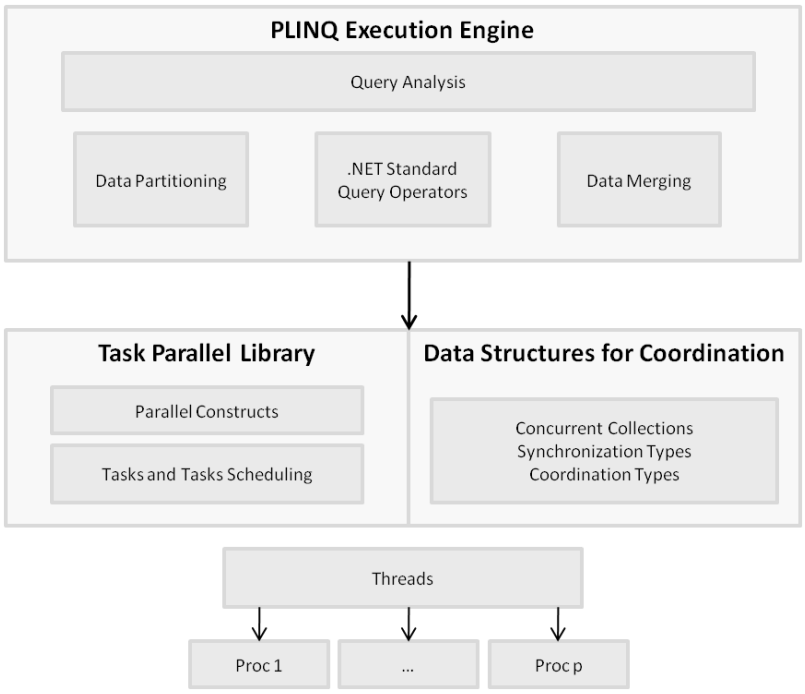

Figure 1:.NET Framework 4 Parallel Extensions Architecture

Classes	405
Functions	2397
LOC	55728
Blocks of Code[2]	16889

Table 1: Size of the Parallel Extensions product code

3. Test Strategies used for getting to 91% code coverage

Within the Parallel Extensions team, tests are developed simultaneously with the product code. These tests are a combination of tests written by the developer and tests written by the tester. By the time the product code is checked in, a single run of the test suite would have 100% pass rate. However, testing does not stop at the initial check-in of the product code. More thorough functional tests are typically developed as the product development cycle continues. In addition, we develop many other types of tests, including performance, stress, and security tests. For the purposes of this document, however, we only measure the code coverage [3] of *functional* and *fault injection* tests.

The functional tests employed by our team are tests that verify the intended behaviors of a software component. The functional tests comprise of a mix of manually written and automatically generated unit

[2] A block, also known as a basic block, is a set of contiguous instructions (code) in the physical layout of an executable that has exactly one entry point and one exit point. Calls, jumps, and branches mark the end of a block. A block typically consists of multiple machine-code instructions.
[3] Block code coverage numbers were captured through our in-house code coverage tool based on Vulcan binary analysis framework (14)

tests. Tests meant to check the integration between our components are also considered part of the functional test suite that gets run every day. In a typical test run, we have approximately 50,000 pass/fail results for our functional tests.

In addition to checking functional behavior, we also need to check whether our components behave correctly when faced with certain catastrophic failures. For example, our software component can be hosted in an AppDomain[4] , which can be unloaded at any time by the host application. We need to make sure that our software component shuts down gracefully in such cases (i.e., the software does not orphan threads or throw inappropriate exceptions). We check this behavior using tests that forcefully inject faults during execution. We identified approximately 50 such scenarios that were tested on a weekly basis.

As part of our test process, we occasionally have dedicated quality "pushes," where the test team focuses on a certain type of testing effort. Examples included a stress push, a performance push, etc. The data in this document is the outcome of two code coverage pushes. These pushes occurred towards the end of the product development cycle. At the start of the code coverage efforts, we already had 91% code coverage; the objective was to obtain 100% code coverage.

4. Measurements

To measure the value of test activities during certain time period, we formulated the following metrics:

Return of Test Effort (RTE), the sum of bugs found per day weighted by severity. In our case, the weight increased exponentially with the severity of the bug.

$$RTE = \frac{\sum_{k=1}^{maxSeverity} BugsOfSev(k) * (2)^{(maxSeverity+1-k)}}{TimeInvested}$$

Where:

TimeInvested = time invested per person

BugsOfSev(k) = the number of bugs of severity[5] k found during the current time period

Return of Coverage Efforts (RCE), the number of new blocks covered per day

$$RCE = \frac{\Delta(oldCC, newCC)}{TimeInvested}$$

Where:

Δ*(oldCC,newCC)* = currentCoveredBlocksValue – previousCoveredBlocksValue

Bug Finding Efficiency (BFE), the ratio of the number of bugs found by a specific test activity compared to other test activities during the time period

$$BFE(timeInterval) = \frac{CCBugCount(timeInterval)}{TotalBugCount(timeInterval)}\%$$

[4] An AppDomain is a feature that allows .NET programs to have isolation boundaries within a process.
[5] Severity of a bug is represented in our case by an integer value (between one and four) with one being the highest. A bug of severity one is usually a blocking bug or a high impact bug that needs to be fixed in maximum 24 hours.

Where:

CCBugCount(timeInterval) = the number of bugs found by code coverage activities during the time interval

TotalBugCount (timeInterval) = the number of bugs found by all test activities during the time interval

5. Results

We tried to cover the remaining 9% by breaking down the work into two test exercises specifically targeted to increase the code coverage numbers. We will refer to these two exercises as **First Code Coverage Push** and **Second Code Coverage Push**.

5.1.1 Goals

The **First Code Coverage Push** was focused only on *interesting* test holes. A test hole was considered to be *interesting* if it had one or more of the following characteristics: an uncovered block with high cyclomatic complexity (12), code that involves direct calls to public APIs, or code paths along core customer scenarios.

The main goal of the **Second Code Coverage Push** was to achieve coverage of all the blocks left uncovered after the First Code Coverage Push. Our approach was as follows:

1. Review each block of uncovered code.
2. Add the tests necessary to cover all the reachable code paths. If any path could not be reached, document the reason.
3. Identify the dead code and advocate having it removed.

The two code coverage efforts were done four months apart; during this interval a number of 136 more test cases were added and 334 were removed without any change in the overall block coverage value. The state of the test bed before we started the two code coverage pushes is summarized in the tables below.

#tests	block coverage	uncovered public methods
3649 test cases	91%	2.3%

Table 2: Data before the First Code Coverage Push

#tests	block coverage	uncovered public methods
3537 test cases	92%	0.2%

Table 3: Data before the Second Code Coverage Push

5.2 Collected Data

In both code coverage pushes, we collected the necessary data to calculate the RTE, RCE and BFE using the formulas defined in the Measurements Section. The data is summarized in the tables below.

code coverage push	#tests	block coverage	#of new covered blocks	uncovered public methods
#1	3735 test cases	92%	169	0.6%
#2	3829 test cases	97%	845	0.1%

Table 4: Tests and coverage at the end of each push

The primary reason we could not achieve higher than 97% coverage is the presence of dead code in our product. For a detailed explanation of the uncovered blocks, refer to section 5.4.

Note that the "#tests" in Table **4** is not equal to the difference between the tests before and after the code coverage push. This is because some tests were deleted during other test activities.

code coverage push	# bugs (severity 1)	# bugs (severity 2)	# bugs (severity 3)	# bugs (severity 4)	time invested (person days)	RTE
#1	0	4	0	0	10	3.2
#2	0	4	0	0	22	1.5

Table 5: Calculation of RTE

#Bugs in each column are the total of

1. bugs found by review of uncovered code during the code coverage push
2. bugs found by new tests during their lifetime (during and after the push)

code coverage push	∆ (oldCC, newCC)	time invested (person days)	RCE
#1	169	10	16.9
#2	845	22	38

Table 6: Calculation of RCE

BFE gives us an idea of the percentage of bugs found by code coverage activities compared to the total bugs found. To calculate BFE, we only look at the bugs found during the time period of the code coverage push.

code coverage push	total bugs found	bugs found by code coverage activities	BFE
#1	29	3	10.34%
#2	7	1	14.2%

Table 7: Calculation of BFE

From the table above, we can see that a large number of bugs are found by test activities other than code coverage investigations. To understand the value of each test activity, we categorized these bugs based on the activity that found them.

code coverage Push	customer reported	ad-hoc testing	code review	code coverage	regression	security	stress
#1	8 (27%)	7 (24%)	7 (24%)	3 (10.34%)	2 (6.89%)	1 (3.44%)	1 (3.44%)
#2	0	3 (42.9%)	0	1 (14.2%)	3(42.9%)	0	0

Table 8: Bug classification based on test activities

5.3 Analyzing the Data

To understand better the value of the two pushes, we looked at the proposed metrics: RTE, RCE and BFE in addition to the Bug Distribution.

5.3.1 RTE

In order to understand if the RTE from the code coverage pushes is good or not, we compared the combined code coverage RTE of the two pushes to the overall RTE of the product development cycle. The RTE for overall product development cycle using the same RTE formula defined in the Measurements section is **2.7.** This value is calculated using the time for overall product development cycle that include other test activities, like infrastructure architecture and development, test documentation development and review, etc. in addition to bugs finding activities and test case development.

# bugs (severity 1)	# bugs (severity 2)	# bugs (severity 3)	# bugs (severity 4)	time invested (person days)	RTE
0	8	0	0	32	2

Table 9: Calculation of Combined RTE of the two code coverage pushes

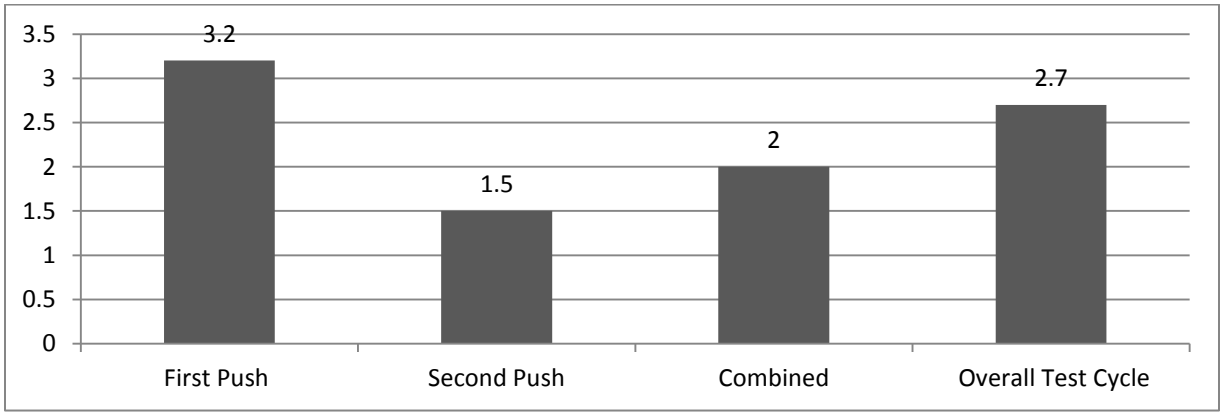

Figure 2: RTE comparison

From the results above we can conclude that the return on investment of the combined code coverage push is not as high as the rest of the product development cycle. However the first coverage push is somewhat better than the overall test cycle.

5.3.2 RCE

The RCE values calculated using the formula defined in Measurements section represent an estimation of the blocks covered per day. We use this value to estimate how easy to cover the last percentage of uncovered blocks is in comparison with the previous ones.

∆ (oldCC, newCC)	**time invested (person days)**	**RCE**
1014	32	31.6

Table 10: Calculation of Combined RCE of the two code coverage pushes

We compare these values with the RCE for the overall product development cycle. The RCE for the overall product development cycle is **6.7**. The comparison graph below shows that a focused code coverage push yields a higher coverage despite the fact that it is executed over the last blocks of code.

An explanation of this behavior might be the fact that during the regular product development cycle testers usually do not focus only on increasing the code coverage values. They execute a broad set of testing activities in order to assure a high quality test bed that will support the current and next product's versions.

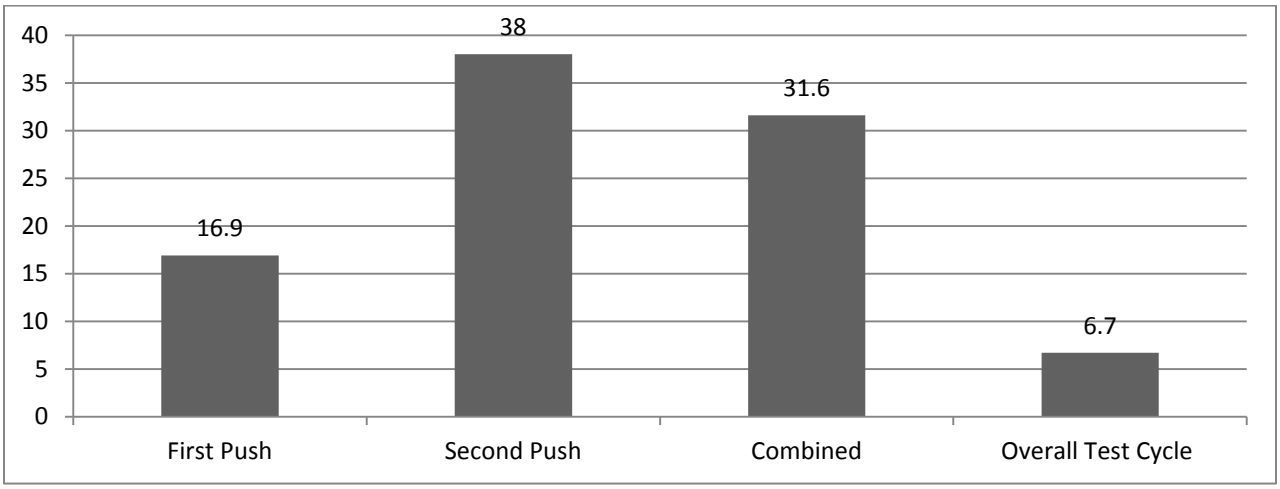

Figure 3: RCE comparison

5.3.3 BFE

BFE was a measure to understand how effective the code coverage activity was for finding bugs when compared to other test activities. We have only compared the efficiency of the two coverage pushes to understand which code coverage strategy was better in terms of bug finding capability.

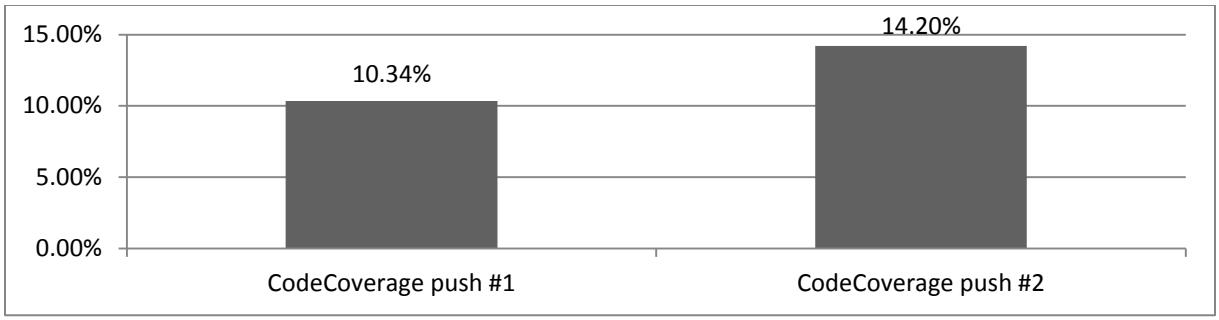

Figure 4: Code Coverage BFE

5.3.4 Bug classification

We studied the accumulated bug distribution from the two pushes to understand the value of the code coverage push when compared to other test activities. We want to highlight that this distribution is specific to the end of our product development cycle and may have been different at different points in the product development cycle. Our goal was to understand what test activities are effective when we already have high code coverage.

customer reported	ad-hoc testing	code review	code coverage	regression	security	stress
8	10	7	4	5	1	1

Table 11: Accumulated Bug classification based on test activities

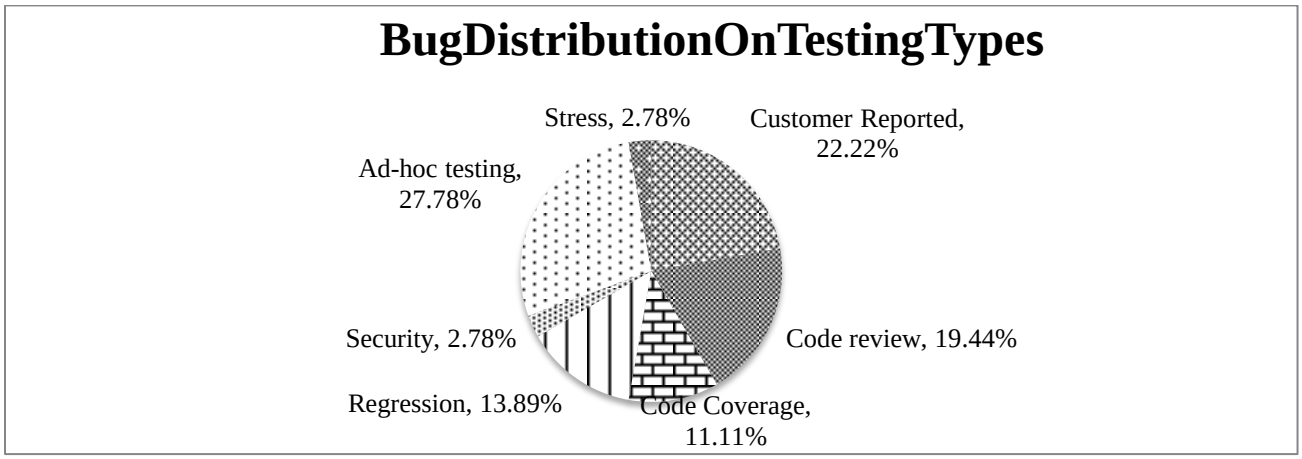

Figure 3: Bug Distribution

5.4 Why is it hard to achieve 100% coverage?

This is a general question and the answer can adjust for different products. Our purpose was to document our experience hoping that some of our observations will be applicable to other products. Below are enumerated the reasons that affected our efforts of achieving 100% code coverage.

1. *Dead Code*

Dead code should not be left in a product. Any code identified as “dead” should be removed for improving code maintainability. Removing the dead code should not have any effect over the product’s behavior if done with care and while removing it no other code paths are affected. For large products any code change requires a test pass because new bugs may be inadvertently introduced in the product along with the changes. At the time of writing this paper, our product was close to shipping, such that only code changes for high severity bugs were accepted.

2. *Nature of the product*

As specified above, Parallel Extensions is a library for supporting multi-threading applications.

Unlike their sequential counterparts, our algorithms might not take the same execution path every time. Hence developing tests that can cover all the possible thread scheduling is difficult. For the hard-to-reach execution paths, we had to use specific concurrency testing tools like CHESS (13).

3. *Stress Conditions*

Some paths could only be reached by long running tests together with the right interleaving. The time taken to execute these scenarios could span as long as a few days. We analyzed these paths and decided that it was not worth the effort to measure the code coverage for them.

Given all these factors, we were able to achieve an effective 99.76% code coverage. We made sure the remaining 0.24% was covered either by using CHESS, writing long running stress tests, or by inspection of the code.

total blocks	covered blocks	dead code blocks	missed blocks because of concurrency/stress
16889	16480 (97.58%)	368 (2.18%)	41 (0.24%)

Table 12: Analysis of uncovered portions of the code in our case

6. Related Work

There have been several studies related to determining how code coverage relates to the strength of the test suite, and the cost of that test suite. Hutchins et al. (1) randomly picked a set of tests and measured their code coverage against the number of seeded faults found. They showed that for test suites of the same size, the test suite with higher code coverage has substantially higher fault detection rates; specifically as coverage goes from 90% to 100%. Hutchins et al. assert that even 100% code coverage does not find all bugs.

Frankl and Iakounenko (4) published similar work, using European Space Agency programs as the software under test. They found an even more stark exponential increase in fault detection as code coverage increases. Andrews et al. (5) used a different approach: first showing that detecting mechanically seeded faults (mutation testing) is a reasonable approximation of detecting real-world bugs, then showing the relationship between code coverage and killed mutants. They found that for difficult to find bugs, the relationship between code coverage and fault detection is exponential, with a sharp increase in the last 10-20%. Andrews et al. also measured the number of test cases needed to achieve coverage, and show an exponential relationship between the number of test cases needed, and the code coverage achieved. They suggest that achieving 100% coverage may be prohibitively expensive.

All three studies, above imply that the last 10% of coverage will find the most bugs, a result that does not match our experience. One explanation could be that in the above lab experiments, the coverage is computed by running randomly selected tests from a known test suite. In the industrial practice of testing, however, test cases are not randomly selected. Usually, a test team will plan-out a set of tests that they believe will find the most bugs. Thus, it is possible that testers develop tests for difficult-to-find bugs first, and test parts that are known to be of high quality less. If this is the case in the industrial context, the cost of going from 90% code coverage to 100% may not be as expensive as Andrews and his colleagues suggest.

Mockus et al. (6) presented two case studies (one each from Microsoft and Avaya) that try to relate code coverage with post-release bugs. They found that high code coverage is associated with low post-release bugs when controlled for pre-release code churn. The authors speculate that testers put more effort into testing components that have a higher propensity to fail, and that higher code churn is indicative of a higher risk of a failure in that component. They also found that classes with higher code coverage had exponentially higher cost. They conclude that getting better than 90% code coverage may not be feasible in most cases. However, we have to consider that if testers tend to cover the most difficult code paths first, the last uncovered sections that are left may be *easier* to test.

7. Conclusions

Code Coverage tools are great when used along with other test activities. Investigations of the code coverage data encourage testers to invest in white box testing which can lead to discovering more scenarios to test. The dead code can be found earlier in the product development cycle and removed promptly in the same release.

Reaching 100% block coverage is without any doubt the best goal to aim for; however with 97% block coverage and the rest 3% reviewed, our confidence in the quality of our product is very high. Even if the eight bugs found represent a small number comparative with the number of bugs found as a result of the other test activities, we found as well a couple of test issues that will improve the testing process going forward. One such issue was the systematic miss of parameter validation for several of our APIs. The other one is the fact that we missed query testing over data sources of specific count, which leads to an entirely different execution path. The dead code found will be removed in the next product development cycle, which will result in increasing the maintainability of the source code.

Despite the current research in the field (1) (4) (5) (6) , our results show that achieving effective 100% code coverage code coverage is affordable. A possible reason may be that testers in general test the complex parts first, and so the remaining areas are not as hard to cover. At the same time, our research showed that the bug value does not increase exponentially as the code coverage increases. However, using the data from the first coverage efforts we can conclude that investigating the code sequences with high complexity could yield a better Return of Test Efforts.

As a result, our recommendation is to prioritize the test work that still needs to be completed before concentrating all the efforts to increase the coverage to 100%. The formulas defined in this paper may be used during the product development cycle to help in estimating how much effort should be put for a targeted code coverage activity. In our case, the customer scenarios are the top priority.

Acknowledgements

We would like to thank our test lead, Shaun Miller, for supporting the project. We would also like to thank Stephen Toub, Boby George and Chris Dern for reviewing drafts of this paper.

References

1. *Experiments of the effectiveness of dataflow - and controlflow-based test adequacy criteria.* **Hutchins, M., Foster, H., Goradia, T., and Ostrand, T.** Sorrento, Italy : IEEE Computer Society Press, Los Alamitos, CA, 1994.

2. *The effect of code coverage on fault detection under different testing profiles.* **Xia Cai, Michael R. Lyu.** 4, s.l. : ACM SIG Software Engineering Notes, 2005, Vol. 30.

3. *Achieving software quality with testing coverage measures.* **Joseph R. Horgan, Saul London. Michael R. Lyu.** 9, s.l. : IEEE Computer Society, 1994, Vol. 27.

4. *Further empirical studies of test effectiveness.* **Frankl, P. G. and Iakounenko, O.** Lake Buena Vista, Florida, United States : SIGSOFT '98/FSE-6. ACM, New York, NY, 1998.

5. *Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria.* **Andrews, J. H., Briand, L. C., Labiche, Y., and Namin, A.** 8, s.l. : IEEE Trans. Softw. Eng., Vol. 32.

6. *Test Coverage and Post-Verification Defects: A Multiple Case Study.* **Mockus, A., Nagappan N., and Dinh-Trong, T.** Orlando, FL : ACM-IEEE Empirical Software Engineering and Measurement Conference (ESEM), 2009.

7. Parallel Extensions to .NET 4. [Online] http://msdn.microsoft.com/en-us/concurrency/default.aspx.

8. Task Parallel Library. [Online] http://msdn.microsoft.com/en-us/library/dd460717(v=VS.100).aspx.

9. Parallel Linq. [Online] http://msdn.microsoft.com/en-us/library/dd460688(v=VS.100).aspx.

10. LINQ to Objects. [Online] http://msdn.microsoft.com/en-us/library/bb397919.aspx.

11. *Data Structures For Parallel Programing.* [Online] http://msdn.microsoft.com/en-us/library/dd460718(v=VS.100).aspx.

12. *A complexity measure.* **McCabe., T. J.** 4, s.l. : IEEE Trans. on Software Engineering, 1976, Vol. 2.

13. CHESS . [Online] http://research.microsoft.com/en-us/projects/chess/.

14. **A. Srivastava, A. Edwards, and H. Vo. Vulcan.** *Binary transformation in a distributed environment.* s.l. : Technical ReportMSR-TR-2001-50, Microsoft Research Technical Report, 2001.