Prediction of Coverage of Expensive Concurrency Metrics Using Cheaper Metrics

Bohuslav Křena, Hana Pluháčková, Shmuel Ur, and Tomáš Vojnar

IT4Innovations Centre of Excellence, FIT, Brno University of Technology, Czech Republic
{krena, ipluhackova, vojnar}@fit.vutbr.cz
shmuel.ur@gmail.com

Abstract. Testing of concurrent programs is difficult since the scheduling nondeterminism requires one to test a huge number of different thread interleavings. Moreover, a simple repetition of test executions will typically examine similar interleavings only. One popular way how to deal with this problem is to use the noise injection approach, which is, however, parametrized with many parameters whose suitable values are difficult to find. To find such values, one needs to run many experiments and use some metric to evaluate them. Measuring the achieved coverage can, however, slow down the experiments. To minimize this problem, we show that there are correlations between metrics of different cost and that one can find a suitable test and noise setting to maximize coverage under a costly metrics by experiments with a cheaper metrics.

1 Introduction

With the current massive use of multicore processors, concurrent programming has become widespread. Such programming is, however, far more challenging since apart from errors that one can cause in sequential code, there is a number of synchronization-related errors specific for concurrent code. What is worse, such errors are easy to cause but difficult to find since they often manifest under some very specific conditions only. Therefore, advanced approaches for finding such errors are highly needed.

A traditional, yet still dominating approach to finding errors is *testing*. However, for testing to be effective in the context of concurrent code, a special care must be taken to cope well with the nondeterminism of thread scheduling.

First, to steer the testing process, various *coverage metrics* are often used. When testing concurrent code, traditional coverage metrics (such as statement coverage) are not sufficient as they do not reflect how well the concurrent behaviour of the program under test has been exercised. Instead, one needs to use specialised metrics, such as coverage of concurrently executing instructions [4], synchronisation coverage [21], or coverage of internal states of dynamic analysers while chasing for various concurrency-related bugs [15]. Sometimes, maximizing coverage under several metrics at the same time is even used since they characterize different aspects of concurrent behaviour.

To maximize coverage under a chosen concurrency coverage metric (or a combinations of such metrics), the space of possible thread schedules has to be properly examined. For that, simple repetitive execution of the same tests in the same environment does not help much [9]. Indeed, despite the scheduling is nondeterministic, some schedules may prevail. To deal with this problem, one can use the approach of *noise injection* [7, 9] which influences the thread scheduling by injecting different kinds of noise (e.g.,

context switches or delays) into a program execution. However, there are many different heuristics for *generating noise* and for deciding *where to place it*, which are, moreover, heavily parametrized. This, in turn, leads to a need of solving the so-called *test and noise configuration setting* (TNCS) problem, which consists in finding the right parameters of noise generation together with the right test cases and suitable values of their possible parameters [11].

If the TNCS problem is not solved properly, the usage of noise can even decrease the obtained coverage [9]. However, solving the TNCS problem is not an easy task. Sometimes, its solution is not even attempted, and purely random noise generation is used. Alternatively, one can use genetic algorithms or data mining [11, 12, 1]. These approaches can outperform the purely random approach, but finding suitable test and noise settings this way can be quite costly. The aim of this paper is to make the cost of this process cheaper.

The approach which we propose builds on the facts that (1) maximizing coverage under different metrics may have different *costs*, and that (2) one can find *correlations* between test and noise settings that are suitable for maximizing coverage under different metrics. Moreover, such correlations may link even metrics for which the process of maximizing coverage is expensive but which are highly informative for steering the testing process and metrics for which the process of maximizing coverage is cheaper but which are less efficient when used for steering the testing process. We confirm all these facts through a set of our experiments. In particular, we identify the correlations by building a *predictive model* between several expensive metrics (under which one may want to simultaneously maximize coverage) and several cheap metrics.

Using the above facts, we suggest to *optimize the testing process* in the following way. Given some expensive but informative metrics, one may find suitable values of test and noise parameters for maximizing coverage under these metrics by experimenting with coverage under some cheap metric (or a combination of such metrics) and then use this setting for testing with the expensive metrics. We show on a set of experiments that this approach can indeed increase the efficiency of noise-based testing.

Our contribution is thus threefold: (1) An experimental categorisation of various concurrency-related metrics to cheap and expensive ones according to the price of maximizing coverage under these metrics. (2) The observation and experimental confirmation of correlations between test and noise settings suitable for testing under metrics of different cost. (3) The idea of exploiting the above facts for more efficient noise-based testing of concurrent programs and its experimental evaluation.

Related Work. There exist many ways how to verify concurrent programs such as systematic testing [13], coverage-driven testing [22], or various kinds of static analysis and model checking [16]. Compared with these techniques, noise-based testing, considered in this paper, is probably the most light-weight. In our previous works [11, 12, 1], we focused on solving the TNCS problem via genetic algorithms and data mining. Here, we propose an orthogonal optimisation based on solving the TNCS problem for expensive concurrency metrics by using cheaper ones, which is justified by existence of a predictive model between the expensive and cheap metrics. Prediction is used in various other areas of software testing, e.g., to predict bug severity [17] or to link concurrency-related code revisions with the corresponding issues and characterize bugs [5]. None of these works, however, builds on prediction in a similar way as this work.

2 Preliminaries

In this section, we briefly introduce noise injection, concurrency coverage metrics, as well as the benchmark programs and experimental setting used in the rest of the paper.

2.1 Noise Injection

The main principle of noise injection when testing concurrent programs is to influence the scheduling of concurrently executing threads by inserting various delays, context switches, temporary blocking of some threads, and/or additional synchronization into the execution. Two main questions to be resolved when applying noise injection are *how* to generate noise and *when* to generate it, which are referred to as the so-called *noise seeding* and *noise placement* problems. For solving these problems, a number of heuristics has been proposed [9]. In this work, we, in particular, use noise seeding and noise placement heuristics implemented within IBM ConTest [6] or on top of it.

2.2 Concurrency Coverage Metrics

A number of concurrency coverage metrics has been proposed in the literature. We build primarily on a selection of those discussed in [15], including both metrics concentrating on various generic aspects of concurrency behaviour as well as on metrics focusing on behavioural aspects relevant when chasing for various synchronisation defects. The former are represented by the *ConcurPairs*, *Synchro*, *WSynchro*, and *HBPair* metrics while the latter by *Avio*, *Avio**, *GoodLock*, *WEraser**, *GoldiLockSC**, and *Datarace*. Below, we briefly characterize those of these metrics that we use the most.

Coverage tasks of the *ConcurPairs* metric [4] consist of pairs of program locations that are checked to be reached in the given order and a Boolean flag indicating whether a context switch happened in between. Coverage tasks of the *Avio* metric are based on the Avio atomicity violation detector [18] and track which pairs of locations of one thread were interleaved with which locations reached in another thread. The *Avio** metric is the same as *Avio* up its tasks are enriched with an abstract identification of the involved threads (reflecting, e.g., their type).

The *WEraser** and *GoldiLockSC** metrics are based on coverage of internal states of the Eraser [19] and GoldiLocks [8] data race detectors (the latter with the so-called short-circuit checks), both of them extended with an abstract identification of the involved threads. The *Datarace* metric measures the number of data race notifications raised by the GoldiLock detector.

2.3 Benchmarks and Experimental Setting

The experimental results presented below are based on the following 10 multithreaded benchmark programs written in Java: *Airlines* (0.3 kLOC), *Cache4j* (1.7 kLOC), *Animator* (1.5 kLOC), *Crawler* (1.2 kLOC), *Elevator* (0.5 kLOC), *HEDC* (12.7 kLOC), *Montecarlo* (1.4 kLOC), *Rover* (5.4 kLOC), *Sor* (7.2 kLOC) and *TSP* (0.4 kLOC). More details about these benchmarks can be found in [1, 2]. All these benchmarks are available on the Internet[1]. All our experiments were performed using the IBM ConTest tool [6] on a machine with Intel Xeon E3-1240 v3 processors at 3.40GHz, 32GiB RAM, running Linux Debian 3.16.36, and using OpenJDK version 1.8.0_111.

[1] http://www.fit.vutbr.cz/research/groups/verifit/benchmarks/

3 Distinguishing Cheap and Expensive Metrics

We now explain our way of distinguishing cheap and expensive metrics, i.e., metrics for which collecting coverage is cheaper or more expensive, respectively.

For the classification of the cost of the metrics, we first ran a series of 1000 test runs of each of our benchmark programs without collecting any coverage. These tests were, however, run already in the ConTest environment, using its random noise setting, which already slows the programs down. This way, we obtained the so-called *bottom case*. The running time of the tests in the bottom case was around 93 seconds for one execution when averaging over all our case studies.

Second, for each metric, we performed 100 test runs while collecting coverage under the given metric, again using ConTest with random noise injection. We then compared the time needed for the bottom case with the times of the experiments with each single metric. We classify metrics into three groups: cheap metrics, expensive metrics, and others (i.e., metrics with medium slowdown). In particular, we mark metrics with the slowdown between 10 % and 30 % as cheap metrics and those with the slowdown 50 % and more as expensive metrics.

Experimental Results The obtained classification is shown in Table 1 and used in the further experiments.

Table 1. Cheap and expensive metrics.

	Slowdown in %	***Metrics***
Cheap metrics	$10\,\% \leq x < 30\,\%$	Avio, Avio*, Concurpairs, HBPair, GoodLock, ShvarPair*, Synchro, WSynchro
Expensive metrics	$50\,\% \leq x$	Datarace, WEraser*, GoldiLockSC*

4 Discovering Correlations between Cheap and Expensive Metrics

Next, we aim at automatically finding correlations between metrics that will allow us to find suitable test and noise settings for testing under expensive metrics by experimenting with cheaper ones. Due to multiple metrics are often used in testing of concurrent programs (each of them stressing somewhat different aspects of the behaviour), we, in fact, aim at correlations between sets of expensive metrics and sets of cheap metrics.

For the above, one can use *multi-variable regression* on the cumulative coverage of the different metrics obtained from multiple test runs (i.e., coverage based on a union of the sets of coverage tasks covered in the different runs). However, we, instead, decided to use the so-called *lasso* (least absolute shrinkage and selection operator) algorithm [14, 10] to build a *predictive model* between cheap and expensive metrics. The algorithm selects suitable cheap metrics and constructs their linear combination capable of predicting a given expensive metric, hence showing correlation among the metrics. In our experience, this approach gives more stable results than normal correlation.

In more detail, we use the lasso algorithm to search for a combination of cheap metrics which has a high partial correlation coefficient with a chosen expensive metric. The algorithm iteratively increases the partial correlation and selects a subset of cheap

metrics with the highest partial correlation. The obtained predictive model then looks as follows: $expMetric = \beta_0 + \beta_1 * cheapMetric^1 + \cdots + \beta_n * cheapMetric^n$.

Note that the above model predicts a single expensive metric based on several cheap ones. However, we said that, in general, we aim at maximizing coverage under *several* expensive metrics based on settings suitable for several cheap metrics. To cope with this, we propose to replace the role of the single expensive metric in the above model by using a *fitness function* representing a weighted combination of the chosen expensive metrics (as often done in genetic algorithms).

Such a combination can have the following form:

$$fitness = \frac{expMetric^1}{expMetric^1_{max}} + \cdots + \frac{expMetric^n}{expMetric^n_{max}}.$$

Here, $expMetric^i$ is the cumulative coverage under the i-th metric obtained in the given series of test runs with the same test and noise setting while $expMetric^i_{max}$ is the maximum of all cumulative coverage values under the given metric in all so far performed experiments even with different test and noise settings (this way of approximating the maximum is used since there is no exact way of computing it).

Experimental Results We have decided to experiment with finding suitable test and noise settings maximizing simultaneously coverage under all the three identified expensive metrics, i.e., $GoldiLockSC^*$, $WEraser^*$, and $Datarace$. The first step is to construct the fitness function combining these three metrics for the lasso algorithm. For that, we generated 100 random test and noise settings, ran 5 tests with each of the configurations, cumulating the coverage obtained in these runs. Finally, we took the maximum values of the cumulated coverage from the 100 experiments. This way, we obtained the following fitness function:

$$fitness = \frac{GoldiLockSC^*}{1443} + \frac{WEraser^*}{3862} + \frac{Datarace}{267}.$$

Second, we used the lasso algorithm with forward regression as implemented in the *glmnet()* function from *glmnet* package [14] of the R-project tool to obtain the predictive model. We created the predictive model from a cumulation of results from five runs on all the considered case studies.

In the forward lasso algorithm, one can choose how many cheap metrics one wants for the prediction. It is because the algorithm starts with an empty model, and, in each step, it adds one cheap metric to the previously built prediction model. Thus, one can see which cheap metrics form the model in each iteration. For our case, we choose to predict three expensive metrics by only two cheap metrics. In the future, it could be interesting to compare prediction using two, three, or four cheap metrics. We suppose that using more cheap metrics for the prediction could be more precise but also more time consuming.

Using the above approach, we obtained the following predictive model:

$$fitness = 2.9e-01 + 2.2e-06 * ConcurPairs + 1.8e-03 * Avio^*.$$

This predictive model and also the above mentioned fitness function are used in all further experiments described below.

5 Using Correlations of Metrics to Optimize Noise-based Testing

Once the predictive model is created and we know which set of cheaper metrics can be used to predict coverage under a given (set of) expensive metrics, this knowledge can be used to optimize the noise based testing process. In particular, we can try to find suitable test and noise settings for the given expensive metrics by experimenting with the cheap ones. The experiments can be controlled using a genetic algorithm [11, 12], or data mining on the test results can be used [1], all the time evaluating the performed experiments via the chosen cheap metrics, or, more precisely, through the predictive model built. In the simplest case, one can perform just a number of random experiments with different test and noise settings and choose the settings that performed the best in these experiments wrt the predictive model. This is the approach we follow below to show that our approach can indeed improve the noise-based testing process.

Experimental Results We randomly generated 100 test and noise configurations and executed 5 test runs with each of them for each of our case studies while collecting coverage under the selected cheap metrics (leading to 500 executions for each case study). We cumulated results within the 5 executions of one configuration and then worked with the obtained cumulative value. We chose 20 configurations with the best results wrt the derived predictive model. These 20 configurations were used for further test runs under the three considered expensive metrics. Each of the chosen 20 configurations was executed 200 times, leading to 4000 test executions under the three expensive metrics for each case study. Finally, to compare the efficiency of this approach with the purely random one, we also performed 4500 test runs with random test and noise settings while directly collecting coverage under the expensive metrics for each of the case studies. Hence, both of the approaches were given the same number of test runs.

In Table 2, we compare the random approach with our prediction-based approach. In particular, we aim at checking whether the proposed approach can help to increase the obtained coverage of the expensive metrics when weighted by the consumed testing time. From the table, we can see that this is indeed the case: the coverage over time increased in most of the cases. The average improvement of the obtained cumulative coverage over the testing time across all our case studies ranges from 46 % to 62 %.

Finally, Figure 1 (left) compares how the obtained cumulative coverage, averaged over all of our case studies, increases when increasing the number of performed test runs under the purely random noise-based approach and under our optimized approach. Our approach wins despite it has some initial penalty due to using some number of test runs to find suitable test and noise parameters via cheap metrics. The right part of the figure then compares the average time needed by the two approaches over all the case studies. Again, the optimized approach is winning.

6 Conclusion and Future Work

We have proposed an approach that uses correlations between cheap and expensive concurrency metrics to optimize noise-based testing under the expensive metrics by finding suitable values of test and noise parameters for such testing through experiments with the cheap metrics. Our experiments showed that such an approach can improve noise-based testing. In the future, we would like to combine the proposed optimization with previously proposed optimizations of noise-based testing via genetic optimizations

Table 2. A comparison of random and prediction-optimized noise-based testing.

	GoldiLockSC*		WEraser*		Datarace	
CaseStudies	Random	Predict	Random	Predict	Random	Predict
Airlines	9.46	**22.42**	74.92	**182.59**	0.28	**0.72**
Animator	817.82	**1451.35**	233.20	**291.42**	0.35	**0.46**
Cache4j	0.93	**2.62**	4.14	**10.98**	0.03	**0.10**
Crawler	54.93	**88.69**	351.85	**547.41**	1.90	**2.86**
Elevator	**297.09**	286.30	**756.72**	733.91	**2.31**	2.23
HEDC	**27.50**	19.93	**67.37**	48.73	**0.50**	0.36
Montecarlo	4.24	**5.19**	9.03	**11.35**	0.02	**0.03**
Rover	37.62	**62.89**	174.14	**292.18**	**0.08**	**0.08**
Sor	3.19	**7.16**	4.93	**12.69**	**0.00**	**0.00**
TSP	**1.86**	1.40	**15.36**	11.74	**1.14**	0.86
Average Impr.		**1.62**		**1.59**		**1.46**

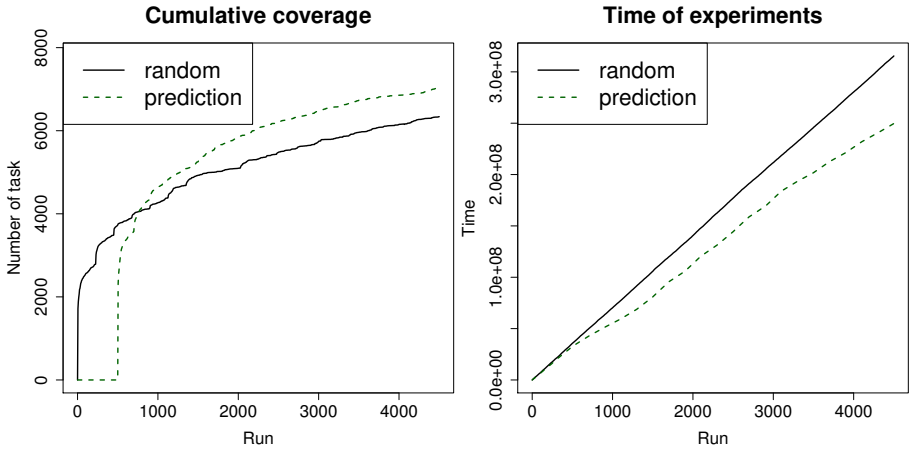

Fig. 1. Cumulative coverage (left) and testing time (right) for an increasing number of test runs.

and/or data mining. Moreover, it is interesting to generalize the idea of finding suitable noise settings maximizing coverage under an expensive metric via experiments with a cheap one to a context of dealing with other kinds of cheap and expensive analyses some of whose parameters may also be correlated.

Acknowledgement. The work was supported by the Czech Science Foundation (project 17-12465S), the internal BUT project FIT-S-17-4014, and the IT4IXS: IT4Innovations Excellence in Science project (LQ1602).

References

1. R. Avros, V. Hrubá, B. Křena, Z. Letko, H. Pluháčková, S. Ur, T. Vojnar, and Z. Volkovich. Boosted Decision Trees for Behaviour Mining of Concurrent Programs. In *Proc. of MEMICS'14*, NOVPRESS, 2014.

2. R. Avros, V. Hrubá, B. Křena, Z. Letko, H. Pluháčková, S. Ur, T. Vojnar, and Z. Volkovich. Boosted Decision Trees for Behaviour Mining of Concurrent Programs. Extended version of [1], under submission, 2017.
3. S. Bensalem and K. Havelund. Dynamic Deadlock Analysis of Multi-Threaded Programs. In *In Proc. of HVC'05*, LNCS 3875, Springer, 2005.
4. A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur. Applications of Synchronization Coverage. In *Proc. of PPoPP'05*, ACM Press, 2005.
5. P. Ciancarini, F. Poggi, D. Rossi, and A. Sillitti. Mining Concurrency Bugs, *Embedded Multi-Core Systems for Mixed Criticality Summit*, CPS Week, 2016.
6. O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java Program Test Generation. *IBM Systems Journal*, 41:111–125, 2002.
7. O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for Testing Multi-threaded Java Programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, Wiley, 2003.
8. T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A Race and Transaction-aware Java Runtime. In *Proc. of PLDI'07*, ACM Press, 2007.
9. J. Fiedor, V. Hrubá, B. Křena, Z. Letko, S. Ur, and T. Vojnar. Advances in Noise-based Testing of Concurrent Software. In Software Testing, Verification and Reliability, 25(3):272–309, Elsevier, 2015.
10. T. Hastie, R. Tibshirani, and J. Friedman. The Elements of Statistical Learning. Data Mining, Inference, and Prediction. Springer, 2009.
11. V. Hrubá, B. Křena, Z. Letko, S. Ur, and T. Vojnar. Testing of Concurrent Programs Using Genetic Algorithms. In *Proc. of SSBSE'12*, LNCS 7515, Springer, 2012.
12. V. Hrubá, B. Křena, Z. Letko, H. Pluháčková, and T. Vojnar. Multi-objective Genetic Optimization for Noise-based Testing of Concurrent Software. In *Proc. of SSBSE'14*, LNCS 8636, Springer, 2014.
13. G.-H. Hwang, H.-Y. Lin, S.-Y. Lin, and Ch.-S. Lin. Statement-Coverage Testing for Concurrent Programs in Reachability Testing, In *Journal of Information Science and Engineering*, 30(4):1095–1113, 2014.
14. G. James, D. Witten, T. Hastie, and R. Tibshirani. An Introduction to Statistical Learning with Applications in R, Springer, 2013.
15. B. Křena, Z. Letko, and T. Vojnar. Coverage Metrics for Saturation-based and Search-based Testing of Concurrent Software. In *Proc. of RV'11*, LNCS 7186, Springer, 2012.
16. B. Křena and T. Vojnar. Automated Formal Analysis and Verification: An Overview In International Journal of General Systems, 42(4):335–365, Taylor and Francis, 2013.
17. J. Kwanghue, D. Amarmend, Y. Geunseok, L. Jung-Won, and L. Byungjeong. Bug Severity Prediction by Classifying Normal Bugs with Text and Meta-Field Information. *Advanced Science and Technology Letters*, 129, Mechanical Engineering, 2016.
18. S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *Proc. of ASPLOS'06*, ACM press, 2006.
19. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-threaded Programs. In *Proc. of SOSP'97*, ACM press, 1997.
20. R. Tibshirani. Regression Shrinkage and Selection via the Lasso. In *Journal of the Royal Statistical Society, Series B*, 58(1):267–288, 1996.
21. E. Trainin, Y. Nir-Buchbinder, R. Tzoref-Brill, A. Zlotnick, S. Ur, and E. Farchi. Forcing Small Models of Conditions on Program Interleaving for Detection of Concurrent Bugs. In *Proc. of PADTAD'09*, ACM Press, 2009.
22. J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: A Coverage-driven Testing Tool for Multithreaded Programs. In *Proc. of OOPSLA'12*, ACM Press, 2012.