OSCAR - A Java Noise Injection Framework*

Filipe de Luna[1], Jeremy S. Bradbury[2], and João M. Lourenço[1]

[1] NOVA University Lisbon — FCT & NOVA LINCS, Portugal
`f.luna@campus.fct.unl.pt` `joao.lourenco@fct.unl.pt`
[2] Ontario Tech University, Canada
`jeremy.bradbury@ontariotechu.ca`

Abstract. Concurrent programs offer much better and scalable performance, at the cost of being notoriously harder to design and to assess their correctness. Noise injection presents itself as a strategy to mitigate the negative effects of non-determinism in concurrent software testing, by increasing the diversity (coverage) in the observed interleavings of a concurrent program. In this paper we propose a novel open-source noise injection framework for the Java programming language (OSCAR), together with a novel taxonomy for categorising new and existing noise injection heuristics. When noising some synthetic benchmark programs with different heuristics, we observed that OSCAR is highly effective in increasing the coverage of the interleaving space, and that the different heuristics provide diverse trade-offs on the cost and benefit (time/coverage) of the noise injection process.

Keywords: Noise Injection · Java Bytecode · Instrumentation · Concurrency · Error Detection · Static Analysis

1 Introduction

As multicore processors become ubiquitous, software needs to be designed and optimised to make use of the potential performance gains stemming from the use of concurrency. Concurrent programs come with added performance and efficiency, through the coordinated usage of additional threads to execute independent sub-tasks, at the cost of added complexity and an overall increase in nature and number of errors, stemming from their non-deterministic nature. Unlike sequential programs, which only depend on their input, concurrent programs also depend upon temporal events, meaning their number of possible interleavings rises exponentially with respect to the number of threads and instructions [3].

Standard program testing techniques quickly proved themselves ineffective against such a massive number of possible interleavings. Especially given that some of these interleavings may have a very low probability of occurrence, being highly dependent on the (non-deterministic) thread scheduler [1]. Essentially

* Supported by Research Grants PTDC/CCI-COM/32456/2017 & LISBOA-01-0145-FEDER-032456.

meaning that if an erroneous were to occur, it would be incredibly difficult to reproduce, and if it were not, it would give the tester an illusion of correctness.

One approach to designing testing tools able to reliably tackle such massive amounts of interleavings, is to leverage the *probe effect* [8] to our advantage. The *probe effect* is the observable difference in a concurrent program's behaviour upon having delays introduced into its routine. *Noise injection* is a technique that leverages on the probe effect to introduce delays into specific program locations, using heuristic-based policies, to optimise the influencing of the scheduler. By its nature, noise injection is a confidence (i.e., best-effort) technique, and it cannot be guaranteed to trigger every possible interleaving.

This paper proposes and evaluates a novel shared-memory-oriented noise injection framework for concurrent software written using the Java programming language. This framework, named OSCAR (Open-source Static Concurrency AnalyseR), injects noise into Java bytecode by means of intermediate code instrumentation, relying on the Soot [14] Java bytecode instrumentation framework. Being fully open-source, OSCAR will fill an existing void in the availability of noise injection frameworks for concurrent software written in Java. Additionally, this paper proposes a novel taxonomy for categorising noise injection heuristics and a strategy to evaluate interleaving distances.

In this paper, OSCAR is used to inject noise into Java concurrent programs and the impact of this noise is then measured and evaluated. The results show that, overall, OSCAR can be reliably used to explore the interleaving space of a concurrent Java program.

2 Noise Injection Heuristics

As a concurrent program's complexity increases, so does the number of possible interleavings, rendering a systematic and exhaustive state space exploration approach (as proposed in Java PathFinder [15]) infeasible [5]. Noise injection heuristics solve this problem by precisely noising the locations which are most likely to disturb the scheduling [7], thus triggering additional interleavings. Some of the biggest challenges in the field of noise injection include: continuously improving existing heuristics, creating novel heuristics and, finally, finding the right heuristic for a given application, as a one-size-fits-all heuristic does not exist [11]. These noise injection heuristics are made up of two components—noise placement and noise seeding—, which together define how to generate the noise abd where to place it [5].

2.1 Noise Placement and Seeding

The noise placement component of a noise injection heuristic is responsible for defining which program locations are the most suitable for noising (i.e., most likely to affect scheduling), along with when (i.e., in which runs of the program) the noise at these locations will be triggered [11]. An example of a perfectly valid noise placement strategy would be to insert a bounded randomised amount of noise before and/or after every access to an atomic block.

Noise seeding, includes choosing a noise type to be generated, its frequency and its intensity [9]. Frequency is defined as the probability function of noise being triggered at each chosen location. Intensity (also called "strength" [5]) is characterised by how much the system itself is impacted by the noise.

Although all noising techniques essentially have the same purpose, to influence the scheduler to disturb a running program's behaviour, some are more effective at this task than others [3]. Additionally, many noise heuristics rely on a specific type of noise for their implementation, meaning noise types are not interchangeable, with an example being the Barrier heuristic proposed in [4].

However, the use of any of these noise injection primitives will result in some level of performance degradation, attributed to the additional thread delays and context switching is forced upon the scheduler, even in cases where a *sleep* has a nil value, or a *yield* is ignored by the scheduler.

Some of the most notable examples of noise types for Java include:

Sleep: Inserting a `Thread.sleep(t)` statement, temporarily suspending the execution for an interval of time `t`, releasing the processor to execute another thread's task;

Yield: Inserting a `Thread.yield()` statement, which will hint the scheduler of a thread's willingness to *yield* its use of a processor. The effect of this operation is not deterministic, as the scheduler is free to ignore it;

Wait: Inserting a `lock.wait()` statement, which suspends a thread's execution until it is awaken by another thread;

Priority: Inserting `Thread.priority(p)` statements, forcing the scheduler to prioritise the execution of threads with a higher priority value, in detriment of the ones with a lower priority value.

Research has shown that, overall, the *sleep* primitive is the most effective primitive at triggering additional interleavings, consequently resulting in improved error detection [3].

2.2 Noise Injection Heuristic Taxonomy

As part of our research, we developed a taxonomy for noise injection heuristics, with the express purpose of achieving coherence between the heuristics proposed by different researchers. We categorise the heuristics into the following categories:

Synchronisation-block-based: For this heuristic, noise can be injected before, after and/or inside synchronised blocks (i.e., enclosed by a lock or atomic register).

Read/Write-based: In this heuristic, noise is injected before and/or after shared-variable accesses. This heuristic is particularly helpful for finding common write-read and read-write data race scenarios [5]. An algorithm can be used to detect shared variables, so that only the accesses to these variables are tracked and noised. Since shared variables are registered, this heuristic can be used while focusing the testing process on specific, previously suspicious variables. Research has shown that restricting noised locations to shared variable accesses will increase error detection probability [4];

Function-based: Insert noise either inside of, before and/or after function calls. This technique can also be used to noise specific monitored functions;

Thread-based: Inserting noise that is only triggered for a specific thread;

Pattern-based: Noise is injected in program locations where the code exhibits a particular pattern, such as a repeated access to the same variables in a method, as proposed in [5].

Random: A mix of all the other heuristics, with noise being triggered randomly when arriving at locations relative to each heuristic. This technique was shown to perform poorly actually masking concurrency errors [10].

It is important to keep in mind that this is not an exhaustive list and that none of these noise localisation and seeding heuristics are mutually exclusive, allowing the use of a hybrid solution that employs a harmonious mixture of various localisation and seeding heuristics. Lastly, research has shown that no heuristic is best for every use case, with every situation requiring the tester's careful consideration [12]. This task has been formalised as the TNCS problem [9].

3 OSCAR

The main drive for undertaking this project was to further research in the field of noise injection in Java, investigating proposed noise injection heuristics, understanding how to apply them to Java concurrency primitives, proposing novel variations of these heuristics, and researching methods to evaluate their impact. To do so, however, we are faced with several challenges, including: how to design a generalised instrumentation solution; which methods/statements/primitives to instrument; where noise should be localised; how to handle edge-cases in instrumentation; what are the pitfalls of noise injection; and how to measure and evaluate the impact of noising, e.g., coverage of the interleaving space.

The need for creating a novel noise injection framework became apparent given the scarcity of solutions available for concurrent Java software. The most notable framework for this language, *ConTest* [3], is no longer available or undergoing active development. As such, we propose OSCAR as a novel open-source shared-memory-oriented static analysis framework, relying on bytecode instrumentation to inject noise into concurrent Java programs.

OSCAR's main objective, as a framework, is to not only be used for standalone testing, but also as a building block for other, more opinionated, testing tools. The choice of Java as the target language is due to its portability and wide adoption. The Java instrumentation done by OSCAR is at the bytecode level and supported by the Soot framework [14], allowing a more fine-grained level of access when injecting noise into Java's built-in concurrency constructs. Through Soot, OSCAR supports Java bytecode up to version 9.

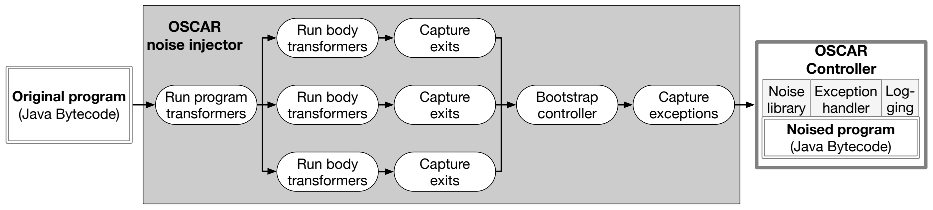


Fig. 1: OSCAR's architecture.

3.1 Architecture

In Figure 1, we present the architecture of OSCAR's instrumentation engine. OSCAR takes a (compiled) Java Bytecode program as input, and runs a series of instrumentation steps, resulting in a new transformed/noised version of the same program. The following Sections describe the implementation of each of the components described in Figure Figure 1.

OSCAR Noise Injector The first step is to transpile the Java program's bytecode to the more readable Jimple alternative intermediate representation, which alleviates many of the difficulties stemming from Java bytecode analysis. Each instrumentation step of the noise injector, is implemented as a Soot Transformer, either for every Java method body or for the whole program.

After Soot has transpiled the program bytecode to the Jimple format, the OSCAR instrumentation engine begins, relying on callgraph exploration to serially execute a series of non-parallelizable Soot transformers. Ideally every instrumentation would be done resorting to body transformers, which are ran in parallel, allowing for superior performance. But, since not all instrumentation logic is parallelisable, some transformers have to be implemented via program transformers.

The next step is to initiate a parallel pipeline for every method body, executing every body transformer in order, followed by the method body transformer responsible for capturing every exit call. Then, the OSCAR Controller logic is bootstrapped to the main/initial method. Lastly, the original program's code is surrounded by a try/catch block, allowing OSCAR to catch any exceptions thrown by the instrumented program.

OSCAR Controller OSCAR's instrumentation process creates a new instrumented program which contains the Controller bootstrapped to the original program. The Controller is OSCAR's dynamic noising engine which allows for the control of noise type, noise intensity, active noise heuristics, output methods and other configurable parameters, when running the new, noised program. The Controller logic can be broken down into the following components:

Noised Program: The original program's main method is replaced with a wrapper which passes arguments to the bootstrapped Controller. The Controller can parse arguments passed into the program to parameterise its noise injection routine. Every instrumentation statement is implemented as a call to a Controller method, which contains the necessary logic. This approach allows for reusing and more easily maintaining the code responsible for the analysis of the noised program.

Noising Library: Every single noise statement is implemented as a call to the `noise` method of this Controller. Noise instructions are statically instrumented into the program's bytecode, while the noise itself is triggered dynamically, as the program runs. The amount of noise triggered upon reaching a noising location is random, with configurable bounds.

The `noise` method takes as a parameter an hardcoded *UUID*, uniquely representing a program trace location, along with the noise category and placement types, so that they can be enabled or disabled on a per case basis.

Tracing/Logging: Since the `noise` statements are already inserted at heuristically determined key synchronisation locations, the tracing piggybacks on the noising statement and can be configured to be triggered before and/or after the noising logic.

The trace information that OSCAR outputs, consists of an ordered set of strings which represent the order of arrival to trace locations. These strings are each a tuple constituted by a non-deterministic thread IDs and the trace location *UUID*.

Exception Handler: Since the program can crash or exit unexpectedly during its execution, all calls to `System.exit` are captured and the program's main method is wrapped in a try/catch block that captures every exception type. This mechanism allows for a graceful exiting, which is invaluable for a user to understand the reason for this abrupt exit. Such understanding can be derived from the interleaving trace, which is composed of information originating from calls to the Controller's `noise` method.

Listing 1 depicts a simple program which launches a series of threads and increments their value inside a *synchronized* block, before and after being processed by OSCAR's instrumentation engine. The final instrumented executable contains noise statements in before and after the thread launch, before the thread's routine and before and after the *synchronized* block. Additionally, it is possible to see how OSCAR wraps the original main method in order to provide a graceful exit mechanism.

3.2 Features

According to [3], the *sleep* primitive has been shown to be the most effective at triggering additional interleavings, when executing an instrumented program multiple times. However, this efficacy comes at the cost of a significant slowdown of the execution [7]. As such, besides *sleep*-based heuristics, OSCAR also supports *yield*-based heuristics, which cause almost no execution slowdown. This will allow the tester to leverage the strengths of both noise types.

```java
static Counter counter = new Counter();

public static void main(args) {
  for (int i = 0; i < args[0]; i++) {
    Thread t = new Thread(Main::add);
    t.start();
  }
}

public static void add() {
  synchronized (counter) {
    Counter.add();
  }
}
```

Before OSCAR's instrumentation

```java
static Counter counter = new Counter();

public static void main(args) {
  try {
    main_wrapped(Controller.start(args));
  } catch (Exception e) {
      Controller.exception(e);
  }
  Controller.end(e);
}
public static void main_wrapped(args) {
  for (int i = 0; i < args[0]; i++) {
    Controller.noise(BEFORE_THREAD_START, UUID);
    Thread t = new Thread(Main::add_bootstrap);
    t.start();
    Controller.noise(AFTER_THREAD_START, UUID);
  }
}
public static void add_bootstrap() {
  Controller.noise(BEFORE_THREAD_ROUTINE, UUID);
  add();
}
public static void add() {
  Controller.noise(BEFORE_SYNC_BLOCK, UUID);
  synchronized (counter) { counter.add(); }
  Controller.noise(AFTER_SYNC_BLOCK, UUID);
}
```

After OSCAR's instrumentation

Listing 1: OSCAR noise instrumentation example.

In OSCAR, noise categories do not have a direct correlation to Java primitives, but instead group a set of related placement locations to more closely represent the heuristics in the proposed taxonomy. For example, "Synchronization-based" is a category and "After Synchronized method call" a specific location. Currently, OSCAR supports the noising of the following Java primitives:

Synchronized method calls: noise is injected before and/or after method calls;

Synchronized blocks: noise is injected before the start and/or after the end of synchronized blocks;

Reentrant locks: noise is injected before and/or after the `lock` and `unlock` operations;

Threads: noise is injected before and/or after the thread is launched and before the thread's routine begins. Supports Runnable objects, lambdas and method references as thread arguments.

The following parametrisation is currently implemented and available from the OSCAR Controller's CLI:

Noise type: the noise intensity is randomised between a range defined by upper lower bounds. It is also possible to completely disable the noise injection;

Noise intensity: the noise intensity is always randomised between an upper and a lower bound which can be set. It is also possible to completely disable the noise injection. Its value represents the duration in milliseconds or the number of repetitions, for the *sleep* and *yield* noise types, respectively;

Heuristic selection: the user is free to select which noise categories and placements will be active in each run;

Controller trace output: it is possible to output the trace with the interleavings of a run to either a file or the console. In order to avoid the performance deterioration of file I/O accesses, an option for in memory trace collection with file output at program exit, is also available.

Class blacklisting: it is possible to blacklist specific classes and/or namespaces to be ignored by OSCAR in the instrumentation process;

4 Evaluation

In this section, OSCAR's functionalities will be validated through the analysis of the interleaving information obtained from batch testing two sample programs. The test bench consisted of a Linux KVM instance, limited to a single core and 4 GB of RAM, running Ubuntu 22.04, on an AMD Ryzen 7 3700X 16-core processor and 24 GB of RAM. We opted for a single-core system due to the necessity of creating an environment where the *yield* primitive would have an effect, a choice which was also made in [3].

During evaluation, both of the identifiers of each tuple in a trace are mapped to unique symbols, to avoid collisions and facilitate readability and analysis. It is important to note, however, that this technique ignores the thread spawn order, as it assigns new IDs to threads and interleavings based on order of their appearance, which allows us to ignore a plethora equivalent interleavings. An example of equivalent interleavings, represented by sets of (thread ID, trace location) tuples, with two threads t_1 and t_2 executing the same code containing two instrumentation points a and b, is $\{(t_1, a), (t_1, b), (t_2, a), (t_2, b)\}$ and $\{(t_2, a), (t_2, b), (t_1, a), (t_1, b)\}$. As such, the results will show a lower number of different interleavings than would otherwise be expected. After a trace is converted to a string representation, a heuristic can determine the string distance between interleavings. This metric allows us to compare interleaving likeness.

We extract two main metrics from OSCAR's trace information:

Number of unique interleavings: This metric represents the number of x non-equivalent interleavings in y runs, where $x \leq y$. A higher number of unique interleavings directly correlates with a higher amount of the program's state space being explored. However, it is important to notice that interleavings are defined only by the trace locations captured by OSCAR, meaning equivalent interleavings using this metric may actually be different if we consider the full set of program states;

Interleaving distance: This metric measures the distinctness of the observed interleavings and uses Levenshtein distance [13], the number of modifications which must be made to a string to transform it into another. An interleaving is encoded as a string of Unicode characters representing a sequence of

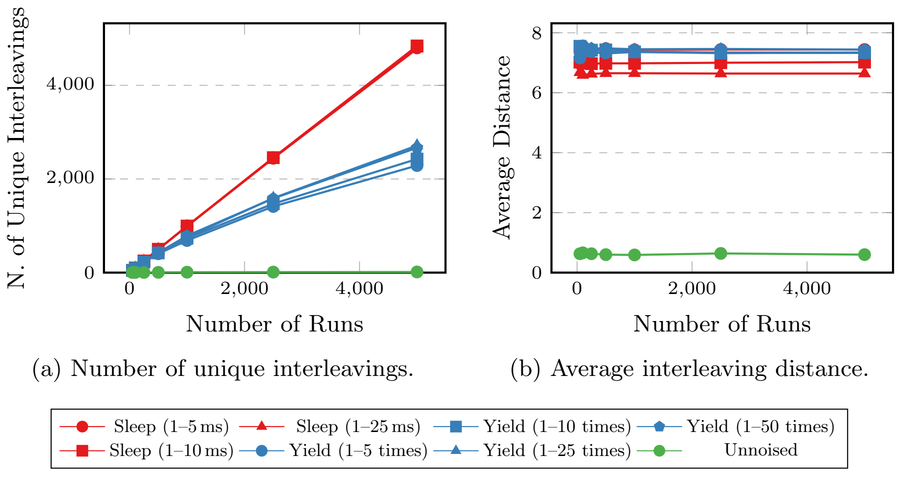


Fig. 2: Results from the `PrintID` program.

thread ID/noised location pairs. A higher distance between two interleavings indicates they are more distinct.

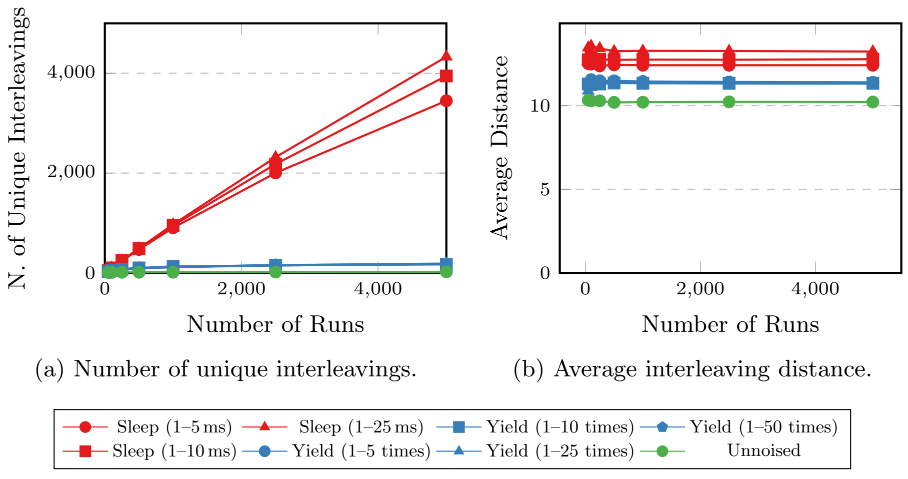
For both programs, the noising locations before and after threads are started were disabled, as they have a tendency to serialise execution. This effect makes state space exploration much harder in programs where threads with similar routines are sequentially started (e.g., in a loop), while simultaneously sharply rising the program run time.

4.1 `PrintID` Program

The first program used to evaluate OSCAR is `PrintID` (see Figure 2). `PrintID` spawns a series of threads, with each thread printing its ID and then joining the main thread. In `PrintID`, each thread's routine consisted of a single instruction, which meant the program executed extremely fast. However, even with a small amount of instructions, it is important not to choose too high a number of threads, as the number of possible interleavings will grow exponentially. The number of threads chosen for this first test was eight.

In our evaluation, we found that setting the *sleep* intensity in the 1–5 ms range was sufficient for triggering thousands of different interleavings in `PrintID`, with higher numbers showing diminishing returns. Furthermore, *yield* showed a much lower efficacy at generating a higher number of unique interleavings. From Figure 2(a) it is possible to conclude that injecting noise with OSCAR highly raised the overall number of witnessed interleavings in the `PrintID` program.

While the number of unique interleavings grew with *yield* intensity, this is more easily attributed to the processing delays incurred by successive calls to the *yield* function. Figure 2(b) shows the average distance between generated interleavings. All the values are somewhat similar, with the exception of the



(a) Number of unique interleavings. (b) Average interleaving distance.

Fig. 3: Results from the `Account` program.

unnoised runs which, as expected, show a much lower average distance, as there are very few unique interleavings. Interestingly, in this scenario, *yield* is shown to be better at generating interleavings which are more distinct from each other. This can be attributed to the relatively high number of threads compared to the physical cores and the context switching that is forced upon the scheduler. Overall, the 1–5 ms *sleep* was the most effective noise seeding for this test.

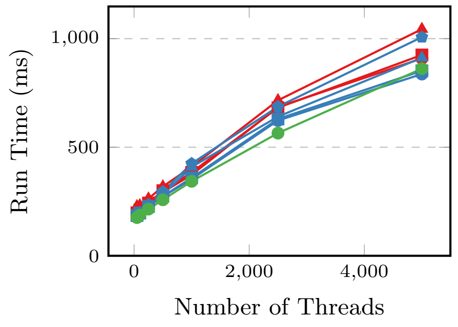
Since the state space grows linearly up to 5000 runs, it is likely that a considerable amount of additional interleavings exist. This understanding can be leveraged to a tester's advantage, allowing one to grasp the dimension of the program's state space.

4.2 `Account` Program

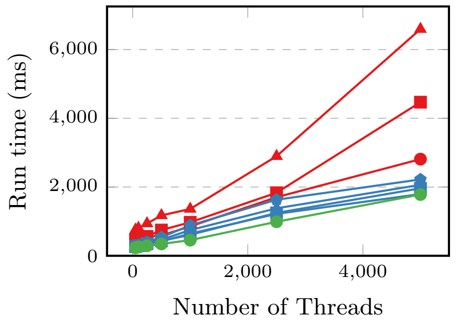
The second program that was tested was the `Account` program, in which a series of spawned threads simulate bank transactions between each other. In contrast to `PrintID`, which only included one noised Java primitive (threads), this program contains three primitives (threads, *synchronized* methods and *synchronized* blocks), in multiple program locations. This results in a much bigger state space, as a consequence of all the additional trace locations. To mitigate the state space explosion, the number of spawned threads for this program was set to two.

From Figure 3, it is possible to visualise the results for the `Account` program testing with OSCAR. The first observable phenomenon is the fact that *sleep* noise seeding performance eclipsed both the *yield* and unnoised results. The number of unique interleavings grows almost linearly for the *sleep*'s 1–25 ms vaariant, suggesting the state space can grow further. It becomes evident that longer programs highly benefit from more intense noise seeding strategies.

Figure 3(b) shows the expected ordering of the average distance of interleavings for each noise seeding strategy, however with the average distance not



(a) `PrintID` program.



(b) `Account` program.

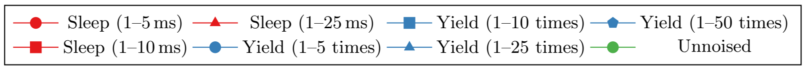


Fig. 4: Average program run times (10 runs).

varying as much between as it did `PrintID`. Unlike `PrintID`, which is expected to be executed linearly, when absent of noise, the `Account` program contains a considerable amount of *synchronized* blocks, which highly raise the probability of the occurrence of context switches.

While results are highly dependent of not only the tested program, but also the test environment and configurations, making such comparisons unfruitful, this effect can be attributed to the fact that there is a much longer trace, which causes overall average differences in interleavings to be much higher. For the `Account` program, the full trace for a given run is composed of 26 tracing points, compared to `PrintID`'s 16 points. This means that the observable difference of 4 points in the average distance in the `Account` program, between *sleep* and unnoised runs, is still quite significant, when contextualised.

4.3 OSCAR Run Time Impact

Noise injection, by its nature, will always be expected to have a considerable impact on the overall run time of a program. From Figure 4 it is possible to assess the impact of OSCAR's noise injection into the `PrintID` and `Account` programs. The results show that the run time when using *sleep*-based heuristics increases exponentially, in relation to the number of threads and noising locations, while for *yield*-based heuristics that increase is linear. Since `PrintID` only contains one noising location, the overall impact of *sleep* noising is minimal, while for the `Account` progrm it becomes relevant. As such, the *yield* primitive, though shown to be a worse performer, may become a better option when testing very complex programs under time constraints.

In summary, the evaluation of OSCAR with the `PrintID` and `Account` programs demonstrate that OSCAR is a viable and effective tool for exploring the state space of Java programs, through the use of noise injection. Additionally, the *yield* primitive, although demonstrably worse than *sleep* at triggering new interleavings, can be used in scenarios where time constraints exist.

5 Related Work

ConTest [3] is the most well known noise injection framework and, consequently, the main inspiration for OSCAR. The framework relies on source code instrumentation, which although more simple, is by definition more limited than OSCAR's bytecode instrumentation. ConTest breaks down every assignment in the program into two separate concurrent events—*before* and *after*—, allowing control of the program to be passed on to ConTest when assignments are made. This approach is somewhat similar OSCAR's, which inserts calls to the controller in noising regions. Like OSCAR, ConTest allows for noise injection before and/or after concurrency events [5]. Unlike the current version of OSCAR, ConTest includes a replay component based on an adapted version of the *DejaVu* [2] deterministic replay algorithm. This is an important mechanism that can allow a tester to accurately reproduce the exact program trace that led to the concurrent error. The current OSCAR trace mechanism, could be further adapted to implement such a feature as well.

Lastly, ConTest also has a fault detection mechanism, which checks for the program's correctness on a given test. Since OSCAR already captures program exceptions and system exits, it would be feasible to implement such a mechanism.

ANaConDA [6] is another notable noise injection framework for concurrent C/C++ software. Its designers mention that ANaConDA was partly inspired by ConTest and, as such, may be viewed as its C/C++ counterpart, albeit resorting instrumenting the binary code dynamically instead of the source code statically, allowing for a generic solution that can be used for any piece of software that is compiled to binary code. However, this approach is not compatible with Java programs, which OSCAR intends to target.

6 Conclusion

In this paper, we have presented OSCAR, a new noise injection framework for the Java programming language. We have evaluated that programs which had noise injected into their routines by OSCAR were verified to consistently generate more interleavings. Furthermore, OSCAR also generated interleavings that were consistently more different from each other. The results show that OSCAR can be reliably used as a noise injection framework for the development of error detection or coverage analysis tools. Additionally, we found that the *yield* primitive, although less effective is more efficient and can be a valid option in time-constrained scenarios.

In the future, we plan to develop a more accurate representation of interleaving distance. The Levenshtein method, although valid, is not ideal, as it only examines the difference between the strings that represent the interleaving and does not take into account the actual differences in interleavings themselves, such as the distance between statements. Additionally, it would be interesting to have a method which gives more weight not only to how more “distant” in the trace any given modifications is, but also to how many modifications are done in total. This means an interleaving distance of 4, for example, should be more than simply twice as significant than a distance of 2. Another possible future improvement to OSCAR, would be to add a replay mechanism, allowing specific interleavings to be reexecuted. This feature is highly desirable, as it offers a means to simulate determinism and check if a fix for an error found for an incorrect interleaving has effectively made that same error disappear, thus providing a means for avoiding the dreaded “heisenbugs”.

References

1. Artho, C., Havelund, K., and Biere, A.: High-level data races. Software Testing, Verification and Reliability 13(4), 207–227 (2003)
2. Choi, J.-D., and Srinivasan, H.: Deterministic replay of Java multithreaded applications. In: SPDT ’98. ACM Press (1998). DOI: 10.1145/281035.281041
3. Edelstein, O., Farchi, E., Nir, Y., Ratsaby, G., and Ur, S.: Multithreaded Java program test generation. In: ACM Press (2001). DOI: 10.1145/376656.376848
4. Eytani, Y., Farchi, E., and Ben-Asher, Y.: Heuristics for finding concurrent bugs. In: Proceedings International Parallel and Distributed Processing Symposium. IEEE Comput. Soc (2003). DOI: 10.1109/ipdps.2003.1213514
5. Fiedor, J., Hrubá, V., Křena, B., Letko, Z., Ur, S., and Vojnar, T.: Advances in noise-based testing of concurrent software. 25(3), 272–309 (2014). DOI: 10.1002/stvr.1546
6. Fiedor, J., and Vojnar, T.: ANaConDA: A Framework for Analysing Multi-threaded C/C++ Programs on the Binary Level. In: Qadeer, S., and Tasiran, S. (eds.) Runtime Verification, pp. 35–41. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
7. Fiedor, J., and Vojnar, T.: Noise-based testing and analysis of multi-threaded C/C++ programs on the binary level. In: ACM Press (2012). DOI: 10.1145/2338967.2336813
8. Gait, J.: A probe effect in concurrent programs. 16(3), 225–233 (1986). DOI: 10.1002/spe.4380160304
9. Hrubá, V., Křena, B., Letko, Z., Ur, S., and Vojnar, T.: Testing of Concurrent Programs Using Genetic Algorithms. In: Search Based Software Engineering, pp. 152–167. Springer Berlin Heidelberg (2012). DOI: 10.1007/978-3-642-33119-0_12
10. Krena, B., Letko, Z., Tzoref, R., Ur, S., and Vojnar, T.: Healing data races on-the-fly. In: ACM Workshop on Parallel and Distributed Systems: Testing and Debugging, pp. 54–64 (2007)
11. Letko, Z.: Analysis and Testing of Concurrent Programs. Information Sciences & Technologies: Bulletin of the ACM Slovakia 5(3) (2013)
12. Letko, Z.: Analysis and Sophisticated Testing of Concurrent Programs. (2010)
13. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. In: Soviet physics doklady, pp. 707–710 (1966)
14. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V.: Soot. In: CASCON ’10. ACM Press (2010). DOI: 10.1145/1925805.1925818

15. Visser, W., Păsăreanu, C.S., and Khurshid, S.: Test Input Generation with Java PathFinder. SIGSOFT Softw. Eng. Notes 29(4), 97–107 (2004). DOI: 10.1145/1013886.1007526