JOHANNES KEPLER
UNIVERSITY LINZ

Author
Florian Schwarcz

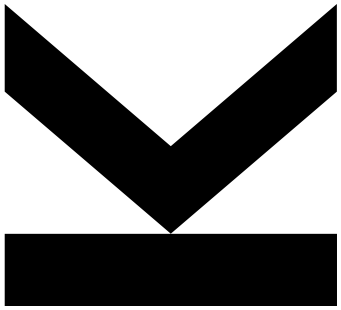
Submission
Institute for System Software

Thesis Supervisor
Prof. Dr. Dr. h.c. Hanspeter Mössenböck

Assistant Thesis Supervisor
Dr. Gergö Barany (Oracle Labs)

September 2024

FEEDBACK DIRECTED FUZZING FOR THE GRAALVM COMPILER



Bachelor's Thesis

to confer the academic degree of

Bachelor of Science

in the Bachelor's Program

Informatik

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Str. 69
4040 Linz, Austria
www.jku.at
DVR 0093696

Abstract

Large and complex software systems like compilers have to be tested thoroughly to ensure correctness and robustness. Although handwritten tests, e.g. unittests, cover the most important test scenarios, edge cases might be overlooked or ignored.

Fuzzing is a method that helps covering previously unconsidered test scenarios by generating random input that is fed into the system under test. In GraalVM's fuzzing project, daily fuzzing jobs try to detect compiler bugs that found their way through the extensive gate tests.

Currently, most of the code generator's parameters are hard-coded, causing its output space to be quite limited. Feedback-directed fuzzing should be employed to tweak either the code generator's configuration or the generated code itself based on some metric, e.g. the compiler's code coverage, to explore more test scenarios than would otherwise be possible.

In this thesis, we present a way of using GraalVM's optimization log in combination with a genetic algorithm to reach this goal. During the development process, we found and reported nearly 50 bugs.

Kurzfassung

Große und komplexe Softwaresysteme, beispielsweise Compiler, müssen gründlich getestet werden, um Korrektheit und Robustheit sicherzustellen. Obwohl hand-geschriebene Tests wie Unit-Tests die meisten Testszenarien abdecken, werden Grenzfälle oft übersehen oder ignoriert.

Fuzzing ist eine Methode, die hilft, bisher unbeachtete Testszenarien abzudecken, indem zufällige Inputs dem getesteten System zur Verarbeitung gegeben werden. Im Falle des Fuzzing-Projektes der GraalVM werden täglich Fuzzing-Prozesse durchlaufen, um Bugs im Compiler zu entdecken, die einen Weg durch die umfangreichen Tests gefunden haben.

Aktuell sind die meisten Parameter des Codegenerators Konstanten, wodurch die Diversität der generierten Programme eingeschränkt ist. Feedback-getriebenes Fuzzing soll eingesetzt werden, um entweder die Parameter des Codegenerators oder den generierten Code selbst anhand einer bestimmten Metrik zu ändern, beispielsweise der Code-Abdeckung. Dadurch werden mehr Testszenarien erforscht als ansonsten möglich wäre.

In dieser Arbeit präsentieren wir einen Weg, wie das Optimierungs-Log der GraalVM in Kombination mit einem genetischen Algorithmus genutzt werden kann, um dieses Ziel zu erreichen. Während des Entwicklungsprozesses wurden fast 50 Bugs gefunden und gemeldet.

Contents

1 Introduction

Compilers are an essential part of every software development toolchain, and their correctness and robustness are expectations every developer sets. However, they can only be truly fulfilled using formal verification, which is possible [1, 2], but hard for software systems of such complexity. Bugs in compilers can cause different symptoms: In the best case, the compiler crashes or runs indefinitely, signaling to the developer that there is a problem. A missed optimization is undesirable but not critical in most cases as it only impacts the performance or size of the generated machine code. The worst case scenario is a silent miscompilation. Developers might spend hours trying to find a bug in their source code when the actual error happened during compilation. Additionally, a miscompilation can be a security concern, even for open-source projects [3, 4].

Standard methods of quality control, e.g. unit-testing, guarantee correct behavior only to a certain extent. As those tests are handwritten, they test cases that the developer is already aware of and probably implemented correctly anyway. Easily overlooked cases, e.g. edge cases, potentially undermine the compiler's correctness if they are not handled correctly in the code. Compiler fuzzing (Section 2.4.1) is a testing method that helps detecting bugs in overlooked scenarios. It works by generating random code that the compiler must process.

The GraalVM compiler (Section 2.3), although being tested extensively, is not an exception to these problems. Daily compiler fuzzing jobs try to find bugs that did not cause any of the gate tests to fail, with some success. However, there is room for improvement. The parameters of the random code generator are hard-coded, hence the diversity of generated test cases is limited.

In this thesis, we extend GraalVM's compiler fuzzing to be feedback-directed. A prominent example of a feedback-directed fuzzer is *american fuzzy lop* (AFL)[5], a general fuzzer that uses code coverage as feedback to directly mutate input data in a set of test cases, increasing coverage iteratively. Measuring code coverage, however, is not 'free'. We conducted preliminary experiments using JaCoCo to collect coverage information during a GraalVM compiler fuzzing run to measure the introduced overhead. In these experiments, we generated about 17% fewer tests compared to a run without coverage collection. Also, code coverage does not include potentially useful context information such as which compiler optimization was applied at what point in the compilation.

Having these requirements (low overhead, context) in mind, we propose using the compiler's optimization log as feedback. In this thesis, we use a simplified version of this log which counts how often every optimization was applied during a fuzzing run.

Collecting this information is practically free of overhead, and we have domain-specific information. Our goal is to use this log as feedback to mutate the code generator's parameters. This results in the generator creating more diverse code, increasing the probability of triggering previously undetected bugs. We use a genetic algorithm (Section 2.6) to drive this process.

In this thesis, we first explain necessary background concepts in Section 2. Then, we have a look at the code generator and its abilities in Section 3 to get an overview of the parameters we can mutate based on the optimization log. As using all parameters would increase the search space drastically, we explain in Section 4 how we select only the most important parameters. Section 5 is the core of this thesis, where we show how to implement our feedback-directed fuzzing. Lastly, we evaluate our implementation in Section 6.

In the context of this thesis, we authored a paper for the 3rd International Fuzzing Workshop [6]. Hence, some parts of this thesis, although not copied, will be very similar to the content of our paper.

2 Background

2.1 Interpreted languages

Most object-oriented programming languages like Java or C# are not actually compiled to native machine code, but to an intermediate language. Java source code, for example, is compiled to *bytecode*, which is then interpreted by a Java runtime environment (JRE) containing a Java Virtual Machine (JVM) [7] in most instances. A major benefit of an interpreter-based JRE is that a Java program can be "written once, run anywhere" [8] as only the interpreting backend is platform-specific. However, interpretation comes at the cost of a significant speed and memory overhead. To mitigate this overhead, just-in-time (JIT) [9] compilation is used.

2.2 HotSpot VM

One such JIT compiling JVM is the *HotSpot VM* [10]. It finds a trade-off between interpreting bytecode and dynamically compiling so-called *hot spots*, i.e. parts of a program that are executed repeatedly based on collected profiling information. Such hot spots are compiled to native code using a JIT compiler if the performance benefits outweigh the compilation effort. This code is then executed directly on hardware instead of the usual interpretation. Profiling information collected during runtime enables speculative optimization of such code sections by making assumptions about the program state and checking them in the compiled code. Should these assumptions turn out to be false, *deptimization* causes the program to return to bytecode interpretation.

2.3 GraalVM

GraalVM builds upon the HotSpot VM by adding a new JIT compiler. The GraalVM compiler constructs an intermediate representation (IR) [11] from bytecode as a sea-of-nodes, combining a data flow graph (DFG) and a control flow graph (CFG) in static single-assignment (SSA) form [12]. This graph has three levels of abstraction, where compilation starts at the highest level, gradually lowering it until a platform-specific low-level IR (LIR) can be built. Currently, there are over 60 different optimizations available that transform the IR. GraalVM not only supports JIT compilation, but also ahead-of-time (AOT) compilation to generate platform-specific machine code in advance. Startup-time, performance and memory usage greatly benefit from this as the overhead caused by interpretation vanishes.

Optimization log. For debugging purposes, the GraalVM compiler can generate an optimization log that contains an entry for every optimization applied to the IR. These entries also contain domain-specific information about the compiler phase at the time when the optimization as applied and the position in the source code. In this thesis, we will use a simplified version of the optimization log for feedback to direct our fuzzer as the compiler generates it with nearly no overhead compared to the full optimization log.

Listing 1 shows an example of an entry in the full optimization log.

Listing 1: Example of an optimization entry in the optimization log. Taken from [6].

```
"phaseName": "UseTrappingNullChecksPhase",
"optimizations": [ {
    "optimizationName": "UseTrappingNullChecks",
    "eventName": "NullCheckInsertion",
    "position": {
        "HashCodeTest.hashCodeSnippet01(Object)": 1
    }
} ]
```

2.4 Fuzzing

In 1988 Barton P. Miller from the University of Wisconsin-Madison suggested "The Fuzz Generator", a generator that produces random character streams [13]. These streams were then relayed to various UNIX utilities to see if they break or get stuck. In the experiments that followed, about a third of the tested utilities crashed or hung at some point, with one instance even crashing the operating system. Since then, this simple principle has proven to be very effective for testing complex input-processing systems [14, 15].

2.4.1 Compiler fuzzing

Compiler fuzzing is the application of the fuzzing principle on compilers. A random code generator produces source programs in a certain language that a compiler must process. As purely random character streams probably never create valid source code, their generated inputs will not even pass the parser. Hence, primitive fuzzing is ineffective as it reaches no interesting parts of the compiler such as optimizations. The generated code must be syntax-compliant and semantics-compliant, e.g. declarations of variables must happen before a usage. In short, generated source code must be correct enough to be successfully compiled if the compiler works as specified. Also, the code should should not contain any language constructs with undefined behavior.

2.4.2 Guided compiler fuzzing

Fuzzing as explained before falls under *black-box* testing or *functional* testing: The test only observes the system under test (SUT) from the outside through its interface without knowing how the system works internally [14, 16]. If some information about the SUT is available, e.g. a specific compiler optimization's prerequisites, the code generator can be adjusted to support the generation of language constructs that raise the chance of this optimization to be applied at some point during compilation. In this case, it is called *gray-box* testing as some internal knowledge about the SUT is taken advantage of.

But while this might improve fuzzing effectiveness, the information must be available before the execution of a fuzzing campaign. When fuzzing a compiler, some information such as code coverage can only be collected at run time. Some fuzzing projects use this information to dynamically "guide" the code generator into a direction that explores new parts of the system [14]. In this thesis, we use the GraalVM compiler's *optimization log* to guide the generator at run time.

2.5 GraalVM's fuzzing project

2.5.1 Architecture

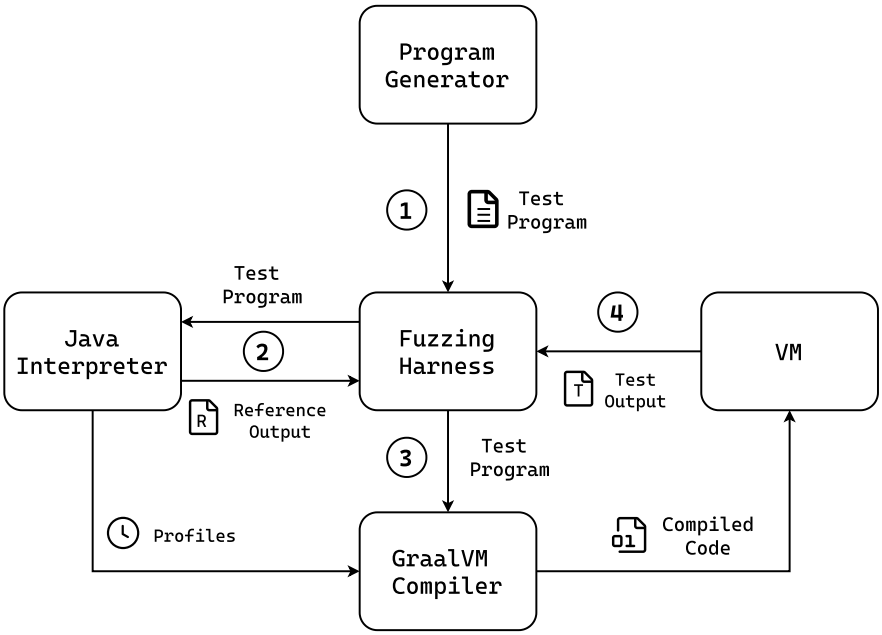


Figure 1: Overview of the GraalVM fuzzing framework. Taken from [6].

As of writing this thesis, GraalVM's compiler is fuzzed daily by a handful of fuzzing jobs, each executing a fuzzing campaign for one hour. A fuzzing campaign consists of a chosen test harness which processes Java code generated by a chosen code generator. Figure 1 shows how all components cooperate. The test harness used in this thesis is the `FuzzingGraalCompileTest` in combination with the `LivenessDrivenCodeGenerator` (see Section 3). This harness implements differential testing to not only find compiler crashes, but also silent miscompilations. Differential testing works by executing the same generated code ① in two different ways and matching the outputs. First, the harness runs the code solely in the interpreter. Unless the generated code contains endless loops (which must be avoided), this run always terminates and produces some reference output ②. After the interpreted run, the harness calls the GraalVM compiler to compile the generated methods until a specific call depth is reached ③. As everything happens within the same JVM instance, the compiler can use profiling information collected by the interpreter for speculative optimizations. While compiling, crashes (i.e., abnormal termination of the compilation with an uncaught exception or an assertion failure) can occur which cause the test to fail and be marked accordingly. If the compilation was successful, the harness runs the compiled main method and again records the output ④. In the last step, the harness compares the actual test output to the reference output and checks for differences, which are a sign of silent miscompilations. For this to work, all random number generators use the same seed to prevent nondeterminism. Additionally, the `LivenessDrivenCodeGenerator` overrides `Object.hashCode`, `Object.toString` and `Object.equals` with calls to a helper library as the default implementations of these methods are inherently nondeterministic.

2.5.2 Code generation

GraalVM's fuzzing infrastructure already provides some code generators, most importantly the `LivenessDrivenCodeGenerator`, which we use for guided fuzzing in this thesis. The basic principle of this code generator is avoiding *dead* code, i.e. code that the compiler can remove early on [17]. For more details on the inner workings see Section 3.

2.6 Genetic algorithms

A genetic algorithm (GA), as described by Holland in 1975 [18], mimics the natural dynamics of evolution to solve optimization problems. In nature, a *population* is a set of *individuals*, each having invisible properties (*genotype* or *chromosome*) that define its visible properties (*phenotype*). Repeatedly, some selected individuals reproduce to gener-

ate children for a new population (the next *generation*). Children inherit the mixed and mutated invisible properties of their parents. As the visible properties are dependent on the invisible properties, the children's visible properties are also similar to their parents'. Natural selection ensures that individuals have a better chance of reproducing if their visible properties fit the environment better than their competition's visible properties.

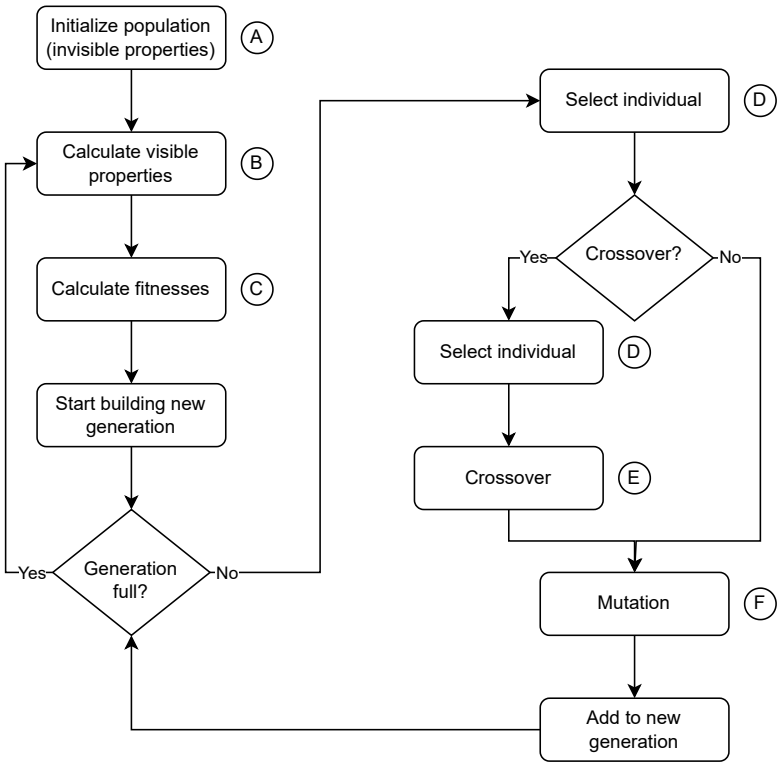


Figure 2: Workflow of our GA.

A GA is the application of this principle to optimization problems. Figure 2 shows the workflow of our GA implementation described in Section 5 using general descriptions. Individuals are solution candidates with some (to the selection process) invisible properties Ⓐ that can be translated to visible properties Ⓑ. In the beginning, the initial generation is a set of random solution candidates. Each solution candidate receives a *fitness* score Ⓒ depending on how optimal its visible properties are. Using the fitness, the algorithm selects Ⓓ solution candidates who then reproduce, combining Ⓔ and mutating Ⓕ their invisible properties to form a new generation of solution candidates. By benefitting solution candidates with a higher fitness (i.e. better visible properties), the

next generation is more likely to have a higher average fitness. This process repeats until it reaches a stopping criterion such as convergence of the average fitness or a certain number of generations. In some applications, the process can run indefinitely.

We identify five components one must define to implement a GA:

A The invisible properties of an individual.

B The function transforming invisible properties to visible properties.

C The fitness function that calculates a score depending on the visible properties.

D The selection function that selects individuals, favoring those with a high fitness.

E The crossover function that combines two individuals into one.

F The mutation function that transforms properties.

3 Code generation in the GraalVM fuzzer

The effectiveness of compiler fuzzing is strongly tied to the capabilities of the code generator. It must support a variety of different language constructs while still generating code that is syntax-compliant, semantics-compliant and has no unspecified behavior. However, there are additional requirements. The generated code must exercise interesting parts of a compiler such as optimizations. Code that is just semantics-compliant often lacks complexity where in extreme cases, the compiler can replace an entire method by a constant, making more interesting optimizations inapplicable. One instance of this problem is *dead* code, or dead assignments [19].

In this thesis, we work with GraalVM's fuzzing infrastructure, which provides a random Java code generator that avoids generating dead assignments. The following sections will describe in detail how this generator works.

3.1 Liveness driven code generation

Imagine the following: You just finished reading a chapter of J.R.R. Tolkien's fantasy book "The Lord of the Rings". Suddenly in the next chapter, a set of completely new characters is introduced. They do not interact with anything described in the previous chapters, but reference some of it. The new characters go on one of their adventures, which is told from beginning to end in this chapter. After that, the usual story of The Lord of the Rings continues without referencing the new characters or their adventure ever again.

This extra chapter can be seen as redundant, i.e. it can be removed from the book without causing any change on how the book ends. The same principle can be found in code generated by a simple random code generator: If it generates statements purely randomly, it might produce code segments that do not affect a method's return value. An example of this are assignments to local variables that remain unused until their next assignment or the end of the scope. Such assignments are called *dead* and a compiler detects and removes them at an early stage in most cases. This removal can 'kill' other assignments, resulting in a cascade of removals. Listing 2 shows example code containing dead assignments. Reducing code size early on also reduces the probability of interesting optimizations being applied to the code, which makes fuzzing less effective.

Listing 2: Example for a dead assignment. It can be removed without side effects.

```
static int method0(boolean param1) {
    int var0;
    int var1 = 1; // dead after removing 5
    if (param1) {
        var0 = var1 + 1; // dead
    }
    var0 = method1();
    return var0;
}
```

Liveness-driven random program generation [17] avoids this problem by ensuring liveness of local variable assignments. It works by generating a code block bottom-to-top, starting with a non-void `return` statement or a local field assignment. When the code generator generates an expression, it adds all used variables to a live variable set. Assignments to variables will only be generated if this set contains the assignee, which guarantees that there is a usage somewhere in the subsequent statements. After an assignment, the assignee is removed from the set. Listing 3 shows a snapshot during the creation of a method to visualize this process.

Listing 3: Snapshot during code generation. The code is generated bottom-to-top and assignments require usages below.

```
// read the code from bottom to top
static int method0() {
    ...
    // at this point we can assign to var0, var2 or var3
    // live set: {var0, var2, var3}
    var1 = var2 * var3.class0_method1();
    // live set: {var0, var1}
    return var0 * var1 + 3;
}
```

Similarly, method parameters also require usages. At any point in an expression, the generator can create a parameter which is added to the method's parameter list.

GraalVM's fuzzing project has its own implementation of a liveness-driven code generator that supports the most common Java language constructs. Also, it supports some special code patterns catered to specific optimizations, e.g. *fold* `for` loops and *map* `for` loops for vectorization.

3.2 Language constructs

Every test consists of a *program* containing the main class `Test`. Within this class, we generate subclasses and enums, where each structure can only use the ones created before to simplify the generation while avoiding most cases of indirect recursion. After that, we generate some `static` methods as well as a `main` method that calls the `static` methods in a short loop and prints the results. These static methods can use all generated subclasses and enums.

3.2.1 Classes and enums

Each class or enum has a bounded random number of methods which we generate in the same way as the static methods in `Test`. To make sure that fields are not created unnecessarily, we only create them when generating field assignments or usages within methods of the class. For every local field we decide randomly whether we initialize it in the constructor or in an initializer block. It works similarly for `static` fields, which we initialize inline or in a `static` block.

3.2.2 Methods

As described before, we create the statements in methods bottom-to-top to ensure liveness of all assignments. A parameter of each method generation is its maximum number of statements, where for each statement the generator can choose from the following options:

- **Assignment to random local live variable:** Generates an assignment of an expression to a random live variable. We remove the variable from the live set before adding all free variables in the expression (which can include the assignee again).
- **If statement:** Generates conditions and statement lists (recursively) for each branch of an `if else` statement. Every statement list has its own live variable set, which starts as a copy of the current live set. The resulting live set is the union of the returned live sets of all branches and all free variables appearing in the conditions.
- **Fold for loop:** Generates a `for` loop that transforms and then accumulates elements of an array or the values of a simple loop counter into a variable. When using an array, the loop can either have a counter from 0 to the array's length minus 1, or an enhanced for-loop iterates directly over the elements.

- **Map for loop:** Generates a `for` loop that transforms and then stores elements of an array in the corresponding positions of another array. Both arrays can be the same object.

- **While loop:** Generates a `while` loop with a loop counter to prevent endless loops. This counter might be usable in the loop, but must not be assignable to avoid endless loops. The fact that the loop is bounded can be hidden to the compiler using special compiler directives applied to the statement increasing the counter.

  Liveness handling is a bit more complicated for general loops as described by Barany [17]. We want to generate assignments to variables whose values are used in later loop iterations. This is impossible when the loop body adheres to the same liveness principle as other code blocks because assigned variables would have to be used in the same iteration. Thus, we add a random set of variables to the live set before generating the loop body. To ensure that there is at least one usage, at the top of the loop we generate an assignment that uses all generated variables that are still live. The resulting live set is the union of the extended live set with the free variables of the loop's condition.

- **Synchronized block:** Generates a `synchronized` block by generating a statement list and wrapping it with a `synchronized` statement containing an expression of type `object`. All free variables of this expression are added to the live set returned by the statement list.

- **Field assignment:** Generates an assignment similarly to the random local live variable assignment, but uses either a local field of a live object variable or a `static` field of a `class`. Free variables occurring in the expression are added to the live set.

- **Method call:** Generates a call to a `void` method. This method can either be a `static` method of some class or a member method of some live object variable. All free variables contained in the argument expressions are added to the live set, if the method is not `static` also all free variables contained in the callee expression are added.

- **Early return:** Generates an early `return` statement. It must be the last statement of an `if else` branch to avoid unreachable code below it. The new live set is the set of free variables in the expression, similarly to the initial live set.

- **Loop exit:** Generates a `break` or `continue` inside a loop. It must be the last statement of an `if else` branch to avoid unreachable code below it. The live set remains unchanged.

3.2.3 Expressions

Each expression generation starts by defining the type that the result must be assignable to. Then, we randomly select one of the expression types that can produce the required type. If a selected expression type relies on creating subexpressions (e.g. a binary expression), we do this recursively. Some expressions have to obey certain conditions that we pass to the generator, in which case it generates expressions in a retry loop until the conditions are met. An improvement of this time-consuming mechanism is the subject of future work (Section 7.1.1). The generator can choose from the following expression types:

- **Initialization** Generates an initialization of a variable, which is either a constant value or a call to the `FuzzerUtils` helper class that generates a random value at runtime.
- **Local variable usage:** Generates a usage of a local variable. This variable can already exist or be created at this point if the defined maximum number of variables has not been reached yet.
- **Parameter usage:** Generates a usage of a method parameter if the expression is created in a method. This method parameter can already exist or be created at this point if the defined maximum number of method parameters has not been reached yet.
- **Binary expression:** Generates a binary arithmetic expression with a bitwise, numeric or boolean operator applied to two subexpressions.
- **Unary expression:** Generates a unary expression with a bitwise, numeric or boolean operator applied to a subexpression.
- **Ternary expression:** Generates a ternary expression of the form `boolean subexpression ? subexpression : subexpression`.
- **Comparison:** Generates a numeric comparison applied to two subexpressions or an `instanceof`, `== null` or `!= null` check with a subexpression returning an object.

- **Library method call** Generates a call of a library method, which can either be a static method or require a subexpression that returns an instance of the class providing the method. For each parameter, another subexpression is generated.
- **Generated method call** Generates a call of a method of a class or enum that has already been generated, similarly to library method calls.
- **Field access** Generates an access to a `static` field of a class or a local field of some object returned by a subexpression.

3.3 Parameters

When the `LivenessDrivenCodeGenerator` chooses a statement type or expression type, the probabilities are not distributed uniformly. The generator provides parameters for the probability of each option, allowing different probability distributions. Weighted randomized decisions happen not only in this case, but throughout the whole generation process to allow for a better guiding of the code generation. Listing all parameters (about 130) would be of little value, so we only provide a brief overview of the ones we think are the most important and/or easy to understand:

- Probability distribution for statement types (ten separate parameters for every option).
- Probability distribution for expression types (ten separate parameters for every option).
- Probability of a numeric constant being an 'edge case', i.e. close to the limit of the datatype's value range.
- Probabilities of a class: (1) inheriting from another class, (2) being `final` and (3) implementing `Cloneable`.
- Probabilities of a method: (1) overriding another method, (2) being `final`, (3) being `synchronized`, (4) returning `void`, (5) returning a boxed value (6) returning an array and (7) containing calls to other methods.
- Probabilities of a `for` loop being an enhanced `for` loop or containing more code than just the folding or mapping.
- Probabilities of a `for` loop counter decreasing instead of increasing.

- Probabilities of a local field being initialized in an initializer block or a `static` field being initialized in a `static` block.
- Maximum statement nesting depth.

These parameters are especially relevant for this thesis as we mutate them to steer code generation into a direction that increases fuzzing effectiveness. More details follow in Section 4 and Section 5.1.

4 Parameter Selection

GraalVM's `LivenessDrivenCodeGenerator` provides a lot of parameters, some of which are able to affect generated code more than others. For example, greatly raising the probability of generating `while` loops changes the shape of the code more than raising the probability of a `class` implementing `Cloneable`. Every parameter that we mutate to direct the code generator increases the size of the search space. Therefore, we must find out which parameters have the greatest influence on the generation and choose those as our mutable parameters.

To do so, we conducted experiments to find out the correlations between the input parameters and the output we plan to use for guiding, namely GraalVM's optimization log. This log summarizes which optimizations were executed at which time during compilation. In this thesis, we use a compact version of the optimization log that only logs the number of applications of each optimization. We are not interested in whether the correlations are positive or negative, i.e. if increasing a certain parameter has a positive or negative influence on the counts of triggered optimizations. Preventing some optimizations can lead to others being triggered more often, therefore, any kind of correlation can be beneficial.

4.1 Setup

The general sequence of events in these experiments starts by taking a (sub)set of the generator's parameters. We randomly mutate them using the mutation function we will later use for the genetic algorithm to ensure the validity of all parameter types (see Section 5.6). However, for these experiments we applied the function multiple times and also increased the amount that a parameter can be increased or decreased. We store the mutated parameters on the file system and then start a fuzzing run with these parameters. One run generates 1000 tests, after that we aggregate the compact optimization logs of the fuzzing processes and also store them. Then, we mutate the parameters again and the process starts over.

A difference to normal fuzzing runs is that differential testing (see Section 2.5.1) is turned off, i.e. there are no interpreted and native runs, but only compilations as we are only interested in the triggered optimizations during compilation. This massively speeds up every test run.

Compilation errors or timeouts can still occur though, therefore after every run, we check if there are test cases marked as `failed` or `timeout` that we should back up.

4.2 Results

Figure 4 on page 18 and Figure 5 on page 19 show the results of one experiment using 24 different parameters. Some correlations are clearly visible, positive as well as negative ones. We can see obvious positive correlations between the probability of generating a `synchronized` block and various lock optimizations, e.g. `LockElimination_LockCoarsening`. The big strip in the middle, which indicates strong negative correlations, shows that a high probability of a local variable usage in an expression suppresses many optimizations, but this need not be a bad thing. When some optimizations are not triggered as often, others might appear more frequently if they are not affected negatively by the parameter, e.g. all `EnterpriseLockElimination` optimizations.

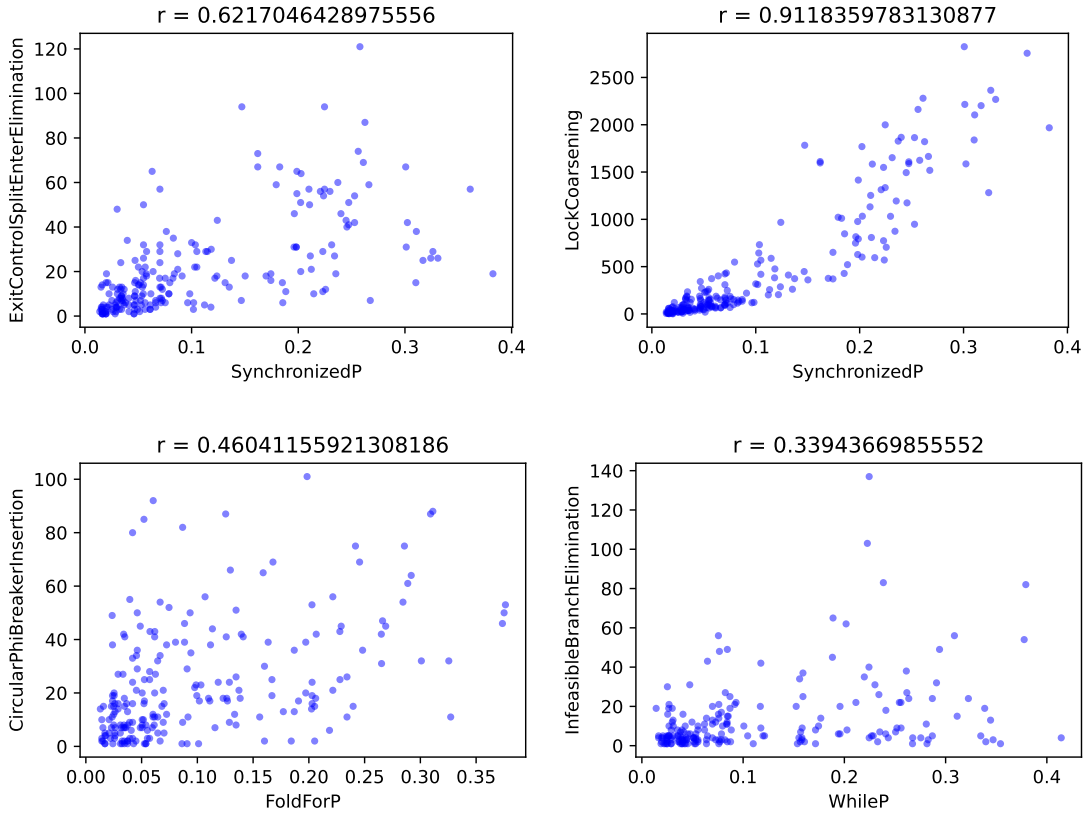


Figure 3: Correlations between different parameters and triggered optimizations.

More detailed visualizations of correlations are displayed in Figure 3. The strong relationship between `synchronized` blocks and lock optimizations can be seen once again, as well as between fold `for` loops and `BreakChainedPhis_CircularPhiBreakerInsertion`.

`while` loops might not have strong individual correlations, but many medium correlations, one of them is depicted in the bottom right subfigure.

Based on these experiments, we select the parameters described in Section 5.1.

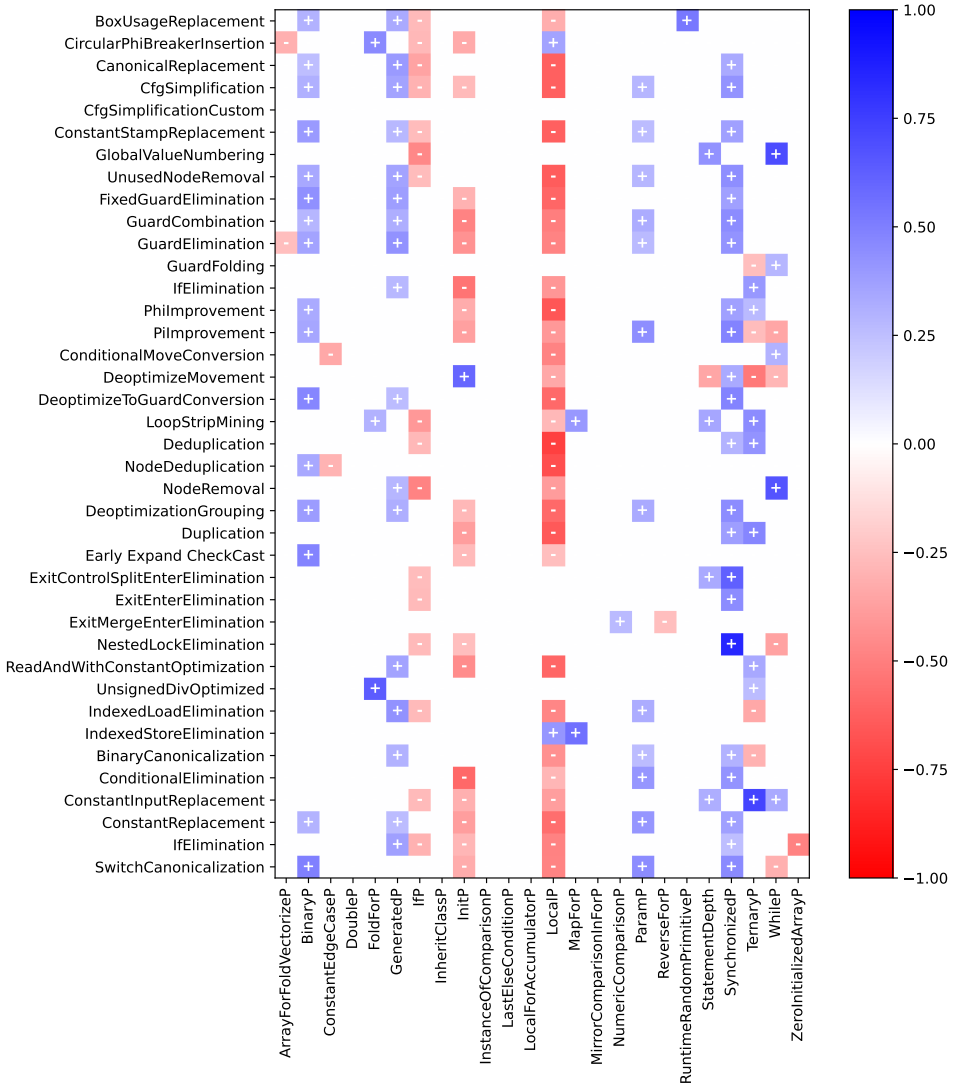


Figure 4: First half of the coefficients for every pair of parameter (x axis) and optimization (y axis). Values between -.25 and .25 have been set to 0 to decrease noise.

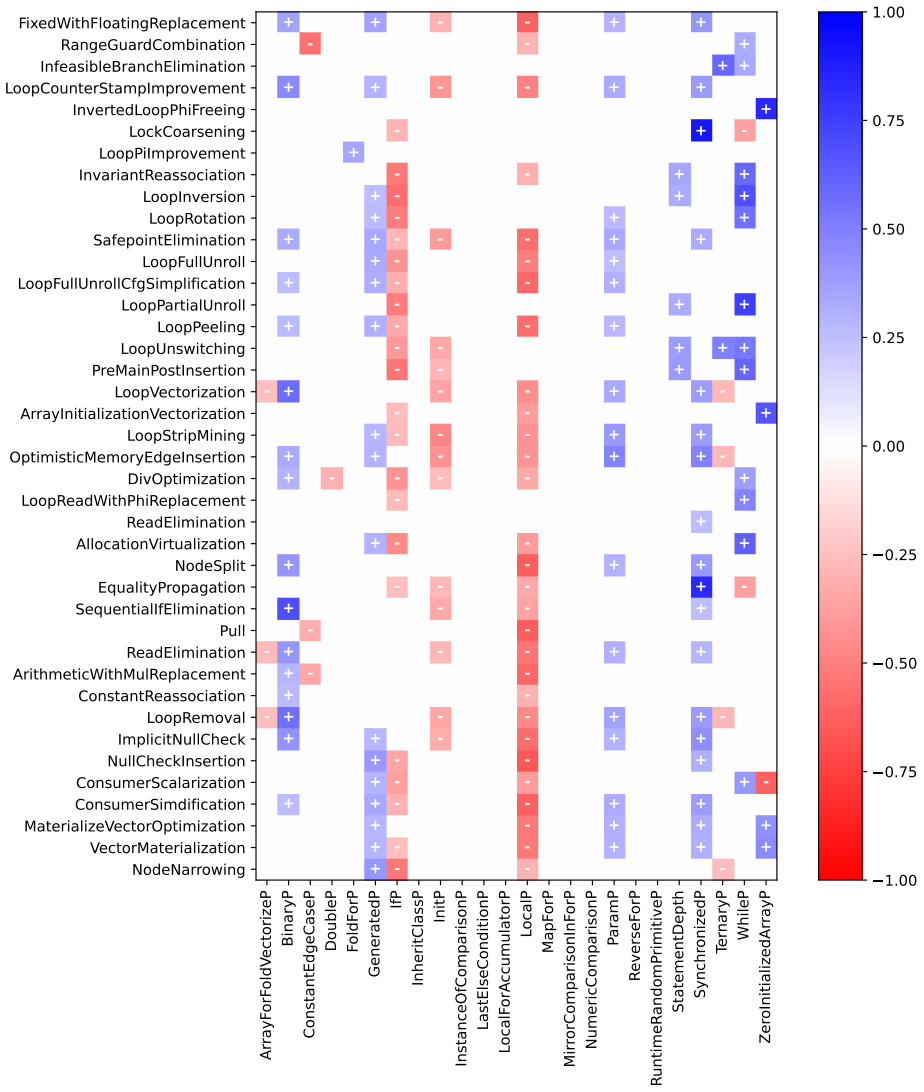


Figure 5: Second half of the correlation coefficients for every pair of parameter (x axis) and optimization (y axis). Values between -.25 and .25 have been set to 0 to decrease noise.

5 Implementation

Our goal in this thesis is to use the Graal compiler's optimization log to optimize the code generator's parameters to exercise more interesting parts of the compiler and ultimately find bugs. A key assumption we make is that no single parameter setting results in code that visits all interesting parts. Hence, we need a solution that works with multiple parameter settings to meet multiple goals at the same time. We propose to use a genetic algorithm (GA) to direct different vectors of parameters in different directions, some focusing on applying a certain optimization as often as possible while others trigger bugs.

In the following sections we will describe how we instantiate the general GA framework from Section 2.6 by explaining the implementations of the individual components in the context of the GraalVM fuzzer.

5.1 Component A: Invisible properties

Component A is the simplest component. In our case, the invisible properties of an individual are the values of a parameter vector we choose to optimize. Simplified parameter vectors can be seen in Table 1 on page 24. We divide all parameters into three categories:

Probability distributions are sets of parameters that represent the probabilities of different options the code generator can choose from. Examples are statement types or expression types. Their sum must stay constant to ensure that it does not exceed 1. What might be unusual is that the sum does not have to equal 1 as we allow partial probability distributions. The probabilities for the options that are not optimized stay constant, but still have to be accounted for when checking that the sum is below or equal to 1. So the sum of the partial distribution must be 1 minus the sum of the constant probabilities.

Independent probabilities are probabilities for binary decisions that do not depend on other parameters, e.g. whether a class implements `Cloneable`. Their value must be between 0 and 1, in our case the valid range is 0.05 to 0.95.

Discrete parameters are small integer numbers, e.g. the maximum statement nesting depth. The valid range must be specified for each discrete parameter individually.

5.2 Component B: Transforming invisible to visible properties

Component B is where the fuzzing takes place. It produces the visible properties, i.e. the optimization log as well as found bugs, using the invisible properties, i.e. the values of the parameter vector. We achieve this by executing a fuzzing run using GraalVM's existing fuzzing infrastructure (see Section 2.5) and setting the code generator's parameters accordingly.

After a run, we have to take care of any failed or stuck tests. For once, we must back them up for further investigation, and also count them and store the resulting value alongside the optimization log. Some bugs that occur are already known or even fixed in the most recent version, so before backing up a failed test, we scan its error message for substrings that classify it as a known bug. This form of deduplication is simple, yet effective enough to prevent overfitting on these bugs if they are easy to trigger. Also, our bug analysts have to do less unnecessary work.

5.3 Component C: Fitness function

Component C calculates a score that represents how well a parameter vector performs. The fitness function should return a high result if the parameter vector outperforms its siblings in one or more of the following categories.

5.3.1 Optimization counts

We want to reward parameter vectors that apply 'rare' optimizations more often. In this thesis, 'rare' describes those N optimizations with the least number of occurrences across all runs of a generation of parameter vectors, where N is a tunable hyperparameter. Defining rare optimizations dynamically ensures that the GA does not overfit on any specific optimization, so others might become relevant if they become rarer after some generations.

For every rare optimization, we rank all parameter vectors according to how often they reported the optimization in the optimization log. Then, the best ten receive a score from 10 to 1 that is multiplied by a defined weight and added to their total fitness. Parameter vectors that did not result in an application of the optimization at all do not participate in the ranking, because else we would reward them for not contributing to a goal.

5.3.2 Bugs

(Unclassified) bugs are handled in a similar manner to optimization counts. We construct a ranking and reward the ten best parameter vectors that have a bug count greater than 0.

5.4 Component D: Selection function

After calculating the fitness, the GA selects parameter vectors randomly, where those with high fitness have a higher chance of being selected. Then, those vectors are mixed and mutated to form a new generation. As with most other components, there are different ways of implementing Component D. For this thesis, we choose tournament selection [20]. Tournament selection works by first picking N random parameter vectors from the current generation, where N is the tournament size. Within this tournament, the GA selects the parameter vector with the highest fitness for reproduction. Tournament selection has several benefits, most importantly, it is widely used and well understood. Furthermore, we can easily tune its 'selective pressure'. Selective pressure describes how biased the selection function is towards parameter vectors with a high fitness [20]. The tournament size directly influences this metric as a bigger (smaller) tournament results in a higher (lower) probability of high-fitness vectors being present in the tournament.

5.5 Component E: Crossover function

Crossover happens with a configurable probability, which is 70% by default. The GA selects two parameter vectors (parents) for reproduction, combining them to produce new parameter vectors (children). Some popular variants of crossover are one-point, k-point and uniform crossover [21]. One-point crossover, in our case, means that we cut both parents at the same position and produce two children: By concatenating the first part of parent A and the second part of parent B, we can construct the first child, for the other child we concatenate the other pieces. k-point crossover is an extension of this procedure which performs not one, but k cuts and concatenates the pieces, alternating between taking a piece from parent A and parent B. Figure 6 shows how 3-point crossover works.

If k is small compared to the size of the parameter vector, values that are close to each other have a high probability of being passed down to the same child. This is beneficial if close values influence each other. However, in our parameter vectors, each parameter's position is arbitrary. Therefore, we deem k-point crossover unsuitable for our purposes. Additionally, probability distributions cannot be mixed as the sum must stay constant. Hence, we choose uniform crossover to implement Component E: When two parameter vectors are selected for crossover, we construct a child by randomly picking

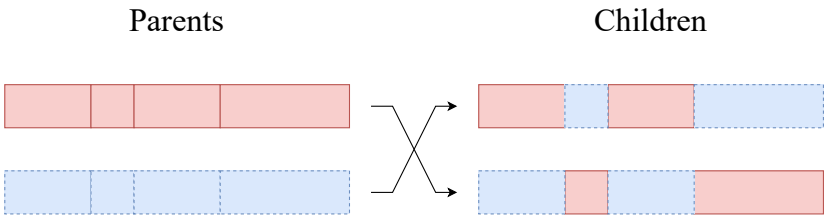


Figure 6: 3-point crossover of two parents resulting in two children.

each parameter from either parent. Probability distributions are treated as a single parameter in this case, so we pick the whole distribution from either parent. Table 1 on page 24 shows a concrete example of our uniform crossover implementation.

5.6 Component F: Mutation function

Crossover alone does not introduce new parameter values into a generation, but only mixes existing ones, limiting the optimization problem's search space. Mutation in GAs, similar to mutation in nature, causes a slight change of the invisible properties, i.e. the values of a parameter vector. Our implementation of Component F handles every category of parameters differently to adhere to their constraints. We perform two mutations of every probability distribution, five mutations of randomly selected independent probabilities, and with a 25% chance we mutate our single discrete parameter. See Table 1 on page 24 for an example mutation of a parameter vector generated by crossover.

Probability distributions must have a constant sum. To guarantee this, we mutate them by selecting one parameter which 'steals' some probability of another selected parameter while keeping their individual values between 0.05 and 0.95. With a 5% probability, we perform an extreme version of this mutation, where the selected parameter steals from 'all' other parameters, increasing its value drastically.

Independent probabilities are mutated by generating a uniformly distributed random value between -0.05 and 0.05 and adding it to the parameter's value as long as its result lies between 0.05 and 0.95.

Discrete parameters are increased or decreased by one, unless they would go out of their individually defined bounds.

Table 1: Uniform crossover of two (simplified) parameter vectors, followed by mutation. Bold parameters of the child are inherited from parent A, bold parameters of the mutated child are mutated.

Parameter	Parent A	Parent B	Child	Mutated Child
		Statement distribution		
`If`	0.0580	0.0810	0.0810	**0.1281**
`FoldFor`	0.1503	0.0557	0.0557	0.0557
`MapFor`	0.1039	0.0913	0.0913	0.0913
`While`	0.0522	0.1944	0.1944	0.1944
`Synchronized`	0.1310	0.0729	0.0729	**0.1472**
		Expression distribution		
`Init`	0.0584	0.1409	**0.0584**	0.0584
`Local`	0.0914	0.1986	**0.0914**	**0.1016**
`Param`	0.2024	0.0548	**0.2024**	0.2024
`Binary`	0.0548	0.0780	**0.0548**	0.0548
`Generated`	0.0737	0.0505	**0.0737**	0.0737
`Ternary`	0.1153	0.3191	**0.1153**	**0.1051**
		Independent probabilities		
`RuntimeRandomPrimitive`	0.3410	0.3166	0.3166	**0.3257**
`ConstantEdgeCase`	0.4568	0.5356	0.5356	0.5356
`InitInStaticBlock`	0.2548	0.2367	**0.2548**	0.2548
`InheritClass`	0.5162	0.5001	0.5001	**0.5152**
`Cloneable`	0.2074	0.2074	**0.2074**	**0.1957**
...				
		Discrete parameters		
`StatementDepth`	2	3	3	3

6 Experiments and Results

We evaluate two versions of our implementation: (1) One with a high crossover probability (80%), (2) another with no crossover at all, i.e. all children are created by mutating one parent. For both versions, we use the seven least applied optimizations for ranking. We compare the results to two baselines: (1) Fuzzing runs using the current hard-coded configuration of the code generator and (2) fuzzing runs with random mutation of parameters similar to the experiments in Section 4. The experiment will be executed twice: Once with the execution of generated code turned off (i.e. the code is only compiled, not run), and once with the execution turned on. We want to show that both GA versions surpass the baselines in the number and diversity of bugs found as well as the number of applications of rare optimizations.

All experiments are executed on the same type of machine on an internal distributed computing cluster within Oracle. The machines each have 2 Intel E5-2690 18-core processors at 2.6 GHz and 512 GB DDR4-1600 memory. Of these 36 cores, we use 32 for fuzzing only. Feedback-directed fuzzing experiments run for 20 generations with 32 parameter vectors each, and every fuzzing run lasts one hour using eight parallel fuzzing workers, producing about 1500 to 2000 test cases per worker, depending on whether the code is run or only compiled. Similarly, we execute 640 (20 * 32) hour-long fuzzing runs for each baseline.

6.1 Results without code execution

6.1.1 Rare optimizations

We show some of the results regarding rare optimizations in Figure 7. It displays for each generation how often the average parameter vector applies an optimization. As there are no generations in the baseline implementations, we always group 32 fuzzing runs for comparison. We chose `BreakChainedPhis_CircularPhiBreakerInsertion` and `EnterpriseLockElimination_ExitEnterElimination` because they clearly show that both GA implementations have an upwards trend line, indicating that over time the average number of optimizations increases. In contrast, the trend lines of the baselines are flat. Whether the no-crossover GA implementation surpasses the high-crossover GA implementation or vice-versa cannot be determined from these results and is subject to further research. Appendix A contains figures for other rare optimizations to further prove that for most rare optimizations, we reached our goal.

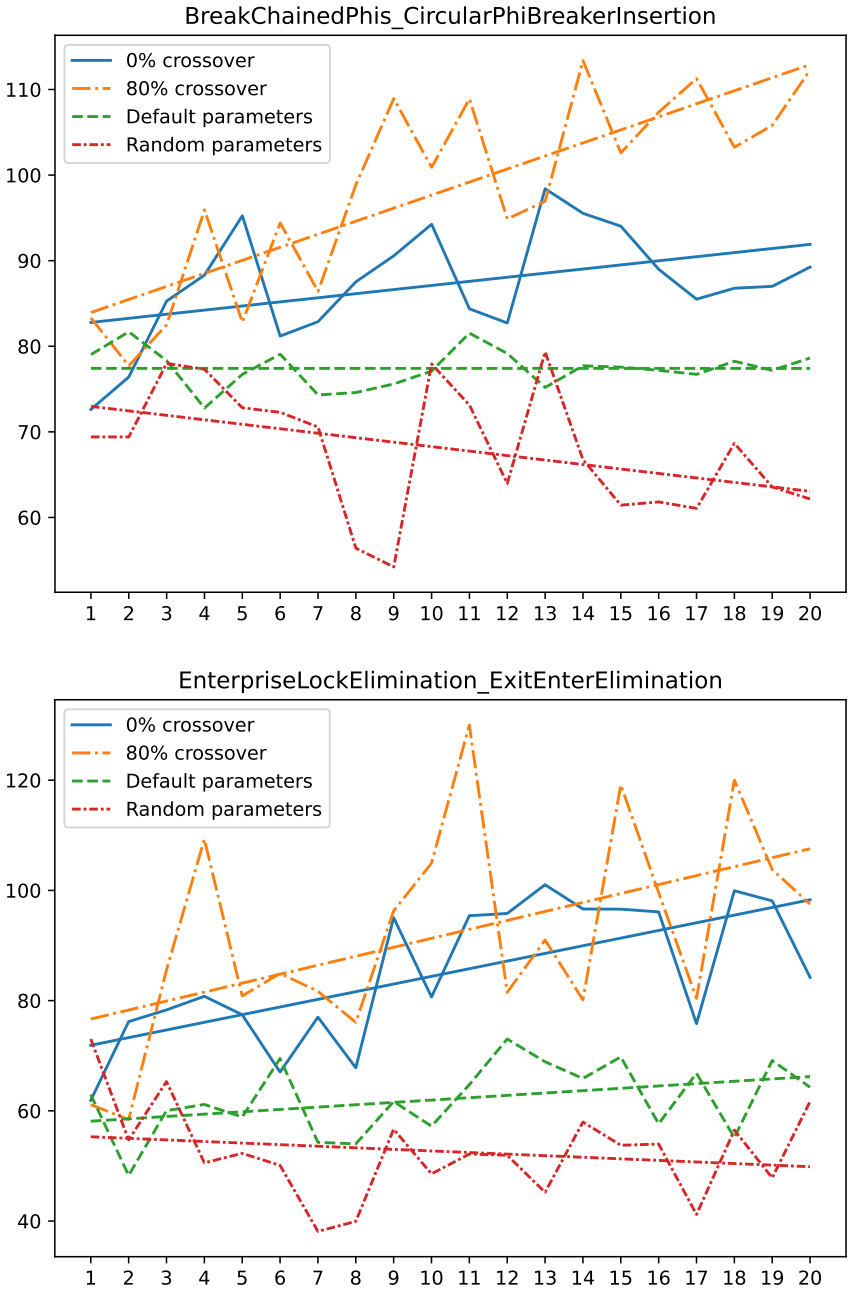


Figure 7: Average number of applications (y axis) during the compile-only run per generation (x axis) of two rare optimizations with their respective trend lines.

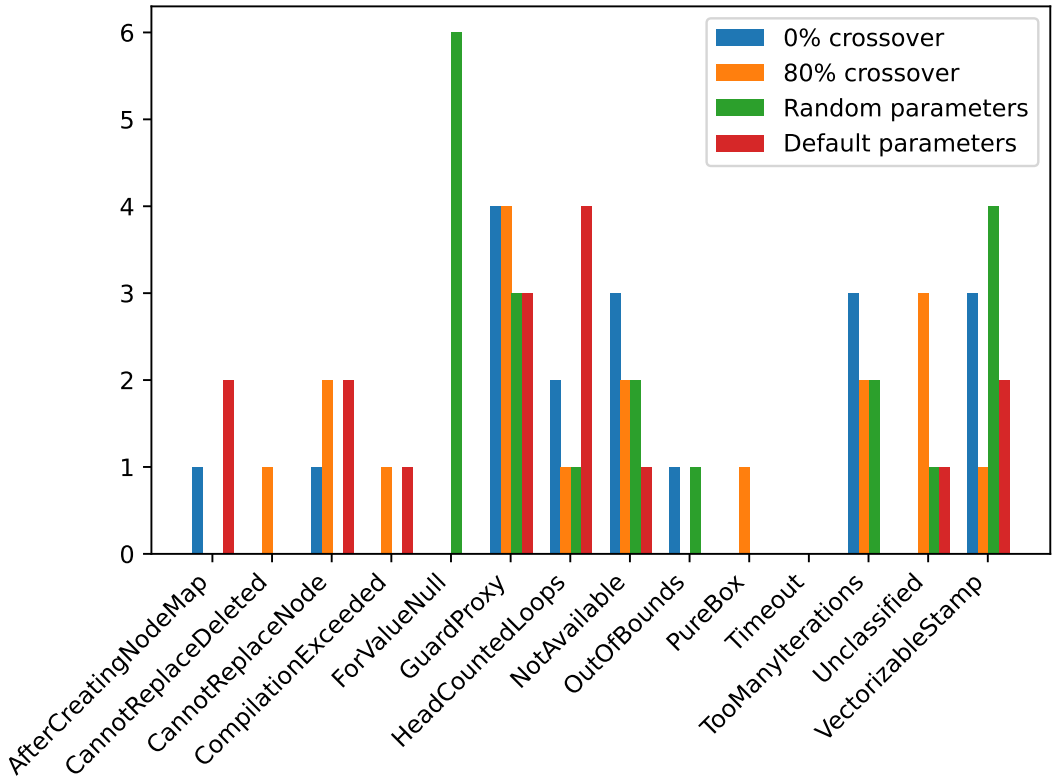


Figure 8: Bugs collected by each configuration during the compile-only run.

6.1.2 Found Bugs

In each fuzzing run, we kept all bugs that were not removed in the deduplication step. To avoid overfitting on bugs that are easy to trigger while still keeping all others for comparison, we only removed the most common bugs. Figure 8 shows all bugs that were triggered and not removed by each implementation.

Unfortunately, this result does not show a clear superiority of the GA implementations. The total number of bugs triggered is 18 for each GA implementation, 16 and 20 for the random and default baseline respectively, which is not an improvement. From a diversity perspective, however, we can see that the high-crossover implementation triggered 9 classified and 3 unclassified bugs, and further investigation showed that all unclassified bugs were different, resulting in 12 different bugs. In comparison to the other implementations, which all triggered either 7 or 8 different bugs, this result is promising.

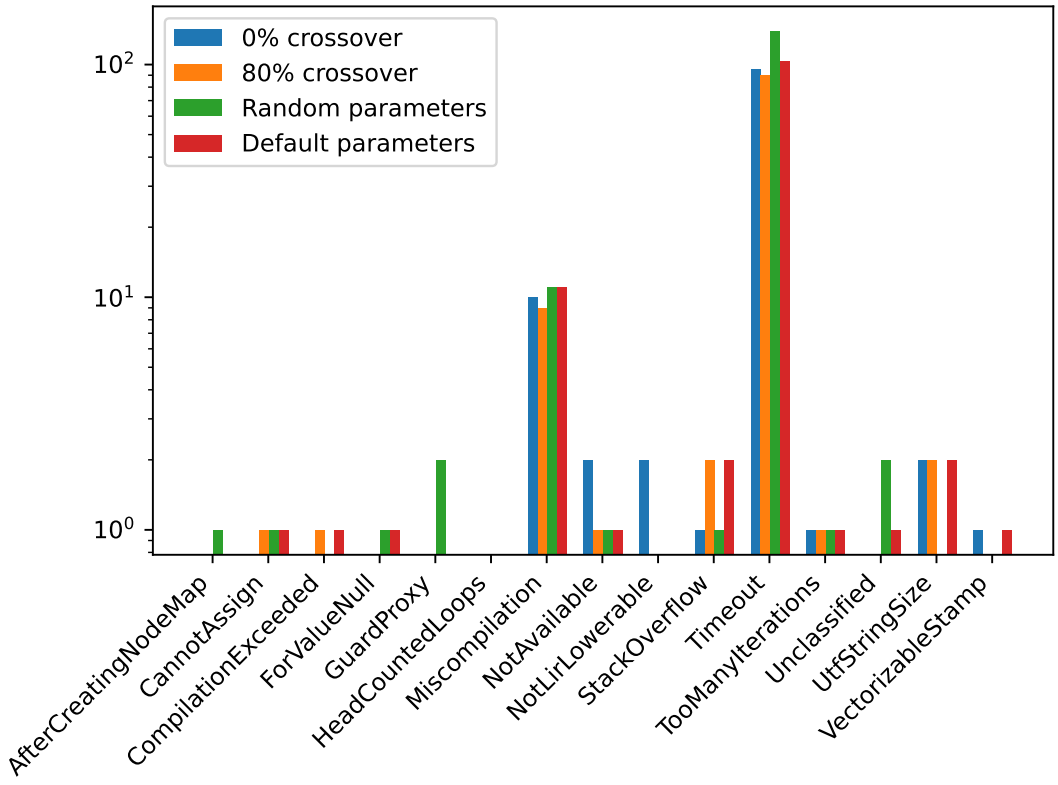


Figure 9: Bugs collected by each configuration during the run with code execution. The y axis is scaled logarithmically to make the smaller bars more visible.

6.2 Results with code execution

We expected the performance to be negatively impacted by allowing execution of the generated code, but the results were even worse than expected. Figure 9 shows that all implementations had about 100 timeouts (note the logarithmic y axis). This drastically reduces the number of test cases generated in a fuzzing run as workers can be blocked for most of the time. Also, the number of rare optimizations that were triggered did not increase as shown in Figure 10. We suspect that the timeout in addition to the general overhead of executing the code introduces too much statistical noise in an hour-long fuzzing run. Tackling this problem at its source is something that we will look into in future work as described in Section 7.1.

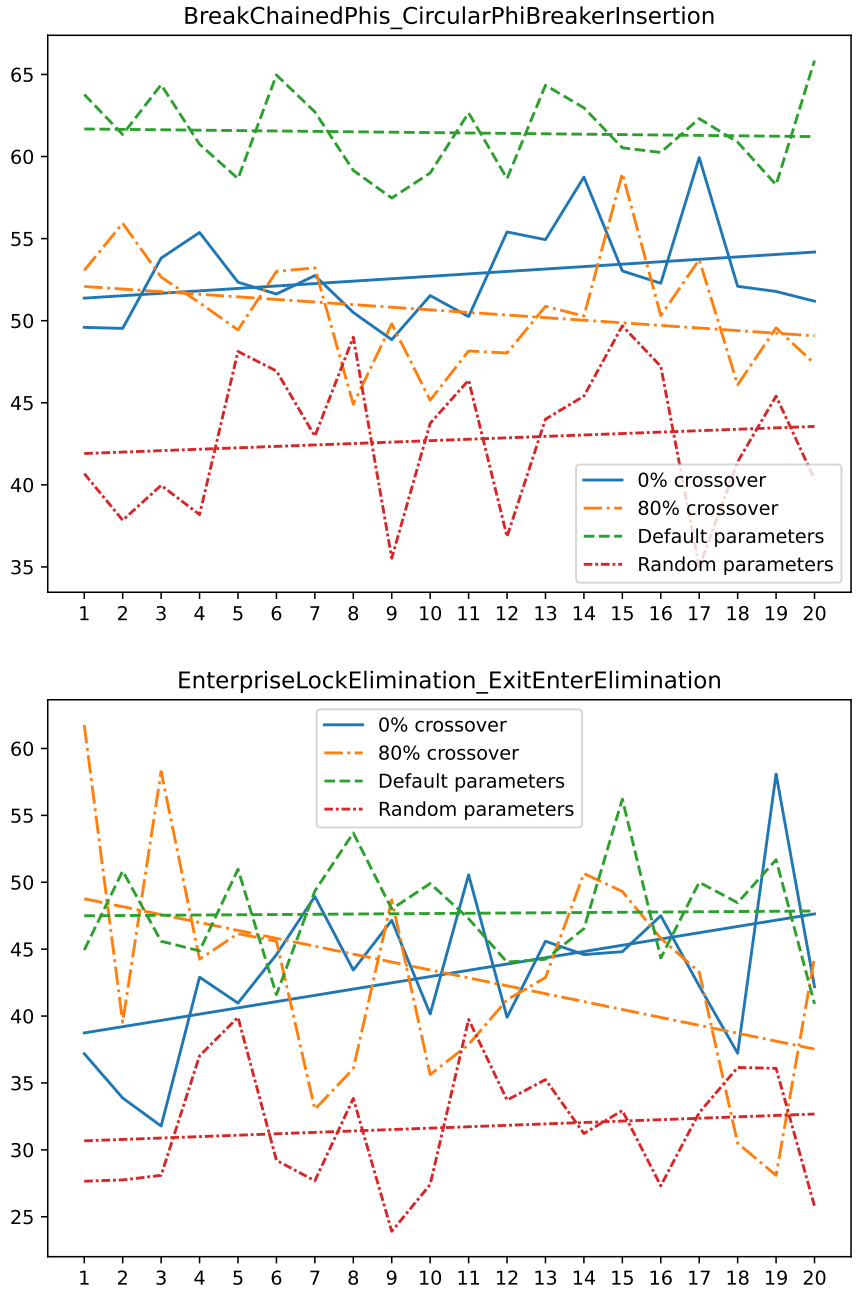


Figure 10: Average number of applications (y axis) during the run with code execution per generation (x axis) of two rare optimizations with their respective trend lines.

7 Future work

7.1 Increasing test case throughput

The best way to decrease statistical noise introduced when doing fuzzing runs in Component B is generating more tests. One possibility would be increasing the duration per run, but this also increases the time the GA takes to converge. Instead, we should focus on the most time-consuming phases of a fuzzing run.

7.1.1 Expression generator

The `ExperessionGenerator` is a good candidate to start. Some generated expressions must fulfill a certain property, e.g. they must contain a local variable. Currently, the generator handles this with a retry loop that generates expressions until one satisfies the property. Sometimes, the number of retries is bounded, so if no generated expression is satisfactory, the caller of the generator does something different. But in other cases, an expression *must* be generated, and if the property is too complex, the retry loop blocks a fuzzing worker for a long time, possibly until the end of the run. Such timeouts happen quite frequently and fixing them would increase throughput significantly.

7.1.2 Timeouts

Sometimes, the generated code contains segments that run nearly indefinitely. A prominent example is string concatenation in a deeply nested loop. The fuzzing infrastructure can detect such timeouts if every test case runs in a separate subprocess that gets killed if the test does not finish within a certain timeframe. But, this causes a lot of overhead. For this thesis, we decided to deactivate testing in a subprocess to increase test case throughput as long as we do not run the code (i.e., only test for crashing / stuck compilation bugs). When activating the execution of the code again, we observe test cases that block fuzzing workers until the end of the run, but the total number of test cases created is still higher (on average) than if we would execute every single test in a subprocess.

Some form of timeout detection without creating subprocesses would avoid blocked workers, increasing fuzzing throughput.

7.2 Automatic Deduplication

Currently, we scan a (crashing) bug's error message to find out if we have seen the same bug before. This information must be available before the start of the feedback-directed fuzzing campaign, and could even be refreshed at some point, e.g. between

generations. Nonetheless, an engineer must update the list of known bugs manually, which can introduce errors. If a bug is thought to be fixed, the entry might still be in the list, making rediscovery impossible.

Performing deduplication automatically would make the campaign more robust against overfitting on a bug that is easy to trigger, but not listed as known.

7.3 Grammar-based code generation with automatic parameter extraction

A topic we plan to research is grammar-based (Java) code generation. This new code generator creates random test cases using an attributed grammar instead of our current abstraction of the Java language. We hope that we can maintain and expand this grammar-based generator with less effort than the current code generator.

Also, the parameters of the code generator could be automatically extracted from the grammar: If a production has more than two alternatives, we create a probability distribution, for example. However, there probably would be more parameters than there are in the `LivenessDrivenCodeGenerator`, which results in a large search space when used for guided fuzzing with the implementation presented in this thesis. Hence, some form of parameter selection similarly to Section 4 would be required.

Liveness should also be guaranteed, of course.

8 Conclusion

In this thesis, we presented an implementation of feedback-directed fuzzing for the GraalVM compiler. Our goal was to use the information from the compiler's optimization log to mutate the parameters of the code generator to exercise rare optimizations more often while also finding bugs. First, we analyzed the code generator and looked at the different parameterized decisions it can make. Then, we showed the impact of certain parameters on the triggered optimizations of fuzzing runs and selected those with the greatest impact, positive as well as negative. We implemented feedback-directed fuzzing using a genetic algorithm to optimize the selected parameters based on the optimization log and the triggered bugs. To prove that the implementation outperforms the current non-directed fuzzing, we conducted experiments and analyzed their results, which showed an increase in the occurrences of rare optimizations over time if the code is only compiled, indicating that the genetic algorithm can converge towards this goal. Additionally, the number of unclassified bugs triggered was about 50% higher for the high-crossover compile-only run when compared to each of the other runs, but this result is not definitive as it might not be reproducible. The experiment where code execution was allowed did not show good results, highlighting the importance of avoiding timeouts during test creation and execution.

Personal thoughts. At first, we were skeptical because the 'distance' between the code generator's parameters and the optimization log is quite large, so it was not certain that the GA will actually meet our needs. Also, the statistical noise introduced by random code generation could have made it a lot harder, even for the compile-only runs. When the first positive results emerged, we were quite relieved to see that our approach has potential to be successful, which boosted our motivation. We are satisfied with the results presented in this thesis and hope that further research in this topic can enhance our implementation even further.

When we integrate this feedback-directed fuzzing into the daily fuzzing jobs and update to the latest version of GraalVM, we are certain that we can discover even more crashing, hanging or silent miscompilation bugs to ensure the Graal compiler's correctness and robustness.

References

[1] X. Leroy, "The CompCert C compiler." `https://compcert.org/compcert-C.html`, 2023. [Online; accessed 26-August-2024].

[2] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, "CakeML: a verified implementation of ML," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, (New York, NY, USA), p. 179–191, Association for Computing Machinery, 2014.

[3] S. Bauer, P. Cuoq, and J. Regehr, "Deniable backdoors using compiler bugs," *International Journal of PoC|| GTFO, 0x08*, pp. 7–9, 2015.

[4] B. David, "How a simple bug in ML compiler could be exploited for backdoors?." `https://arxiv.org/abs/1811.10851`, 2018. [Online; accessed 28-August-2024].

[5] Zalewski, "AFL Whitepaper." `https://lcamtuf.coredump.cx/afl/technical_details.txt`, 2016. [Online; accessed 26-August-2024].

[6] F. Schwarcz, F. Berlakovich, G. Barany, and H. Mössenböck, "LOOL: Low-Overhead, Optimization-Log-Guided Compiler Fuzzing (Registered Report)," in *Proceedings of the 3rd ACM International Fuzzing Workshop (FUZZING '24)*, FUZZING 2024, (New York, NY, USA), Association for Computing Machinery, 09 2024.

[7] "The Java Virtual Machine Specification." `https://docs.oracle.com/javase/specs/jvms/se20/html/`, 2023. [Online; accessed 04-June-2024].

[8] M. Curtin, "Write Once, Run Anywhere: Why It Matters." `http://web.interhack.com/publications/write-once-run-anywhere.pdf`, 04 1998. [Online; accessed 26-August-2024].

[9] J. Aycock, "A brief history of just-in-time," *ACM Comput. Surv.*, vol. 35, p. 97–113, jun 2003.

[10] "HotSpot Runtime Overview." `https://openjdk.org/groups/hotspot/docs/RuntimeOverview.html`, 2024. [Online; accessed 04-June-2024].

[11] G. Duboscq, L. Stadler, T. Wuerthinger, D. Simon, C. Wimmer, and H. Mössenböck, "Graal IR: An Extensible Declarative Intermediate Representation," 02 2013.

[12] C. Click and M. Paleczny, "A simple graph-based intermediate representation," *SIGPLAN Not.*, vol. 30, p. 35–49, mar 1995.

[13] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Commun. ACM*, vol. 33, p. 32–44, dec 1990.

[14] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The Art, Science, and Engineering of Fuzzing: A Survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2021.

[15] M. Aizatsky, K. Serebryany, O. Chang, A. Arya, and M. Whittaker, "Announcing OSS-Fuzz: Continuous fuzzing for open source software," *Google Testing Blog*, 2016.

[16] "IEEE Standard Glossary of Software Engineering Terminology," *IEEE Std 610.12-1990*, pp. 1–84, 1990.

[17] G. Barany, "Liveness-driven random program generation," in *Logic-Based Program Synthesis and Transformation - 27th International Symposium, LOPSTR 2017, Namur, Belgium, October 10-12, 2017, Revised Selected Papers* (F. Fioravanti and J. P. Gallagher, eds.), vol. 10855 of *Lecture Notes in Computer Science*, pp. 112–127, Springer, 2017.

[18] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. Cambridge, MA, USA: MIT Press, 1992.

[19] S. K. Debray, W. Evans, R. Muth, and B. De Sutter, "Compiler techniques for code compaction," *ACM Trans. Program. Lang. Syst.*, vol. 22, p. 378–415, mar 2000.

[20] B. L. Miller and D. E. Goldberg, "Genetic Algorithms, Tournament Selection, and the Effects of Noise," *Complex Syst.*, vol. 9, 1995.

[21] P. Kora and P. Yadlapalli, "Crossover Operators in Genetic Algorithms: A Review," *International Journal of Computer Applications*, vol. 162, pp. 34–36, 03 2017.

List of Code Snippets

List of Figures

List of Tables

A Rare optimizations

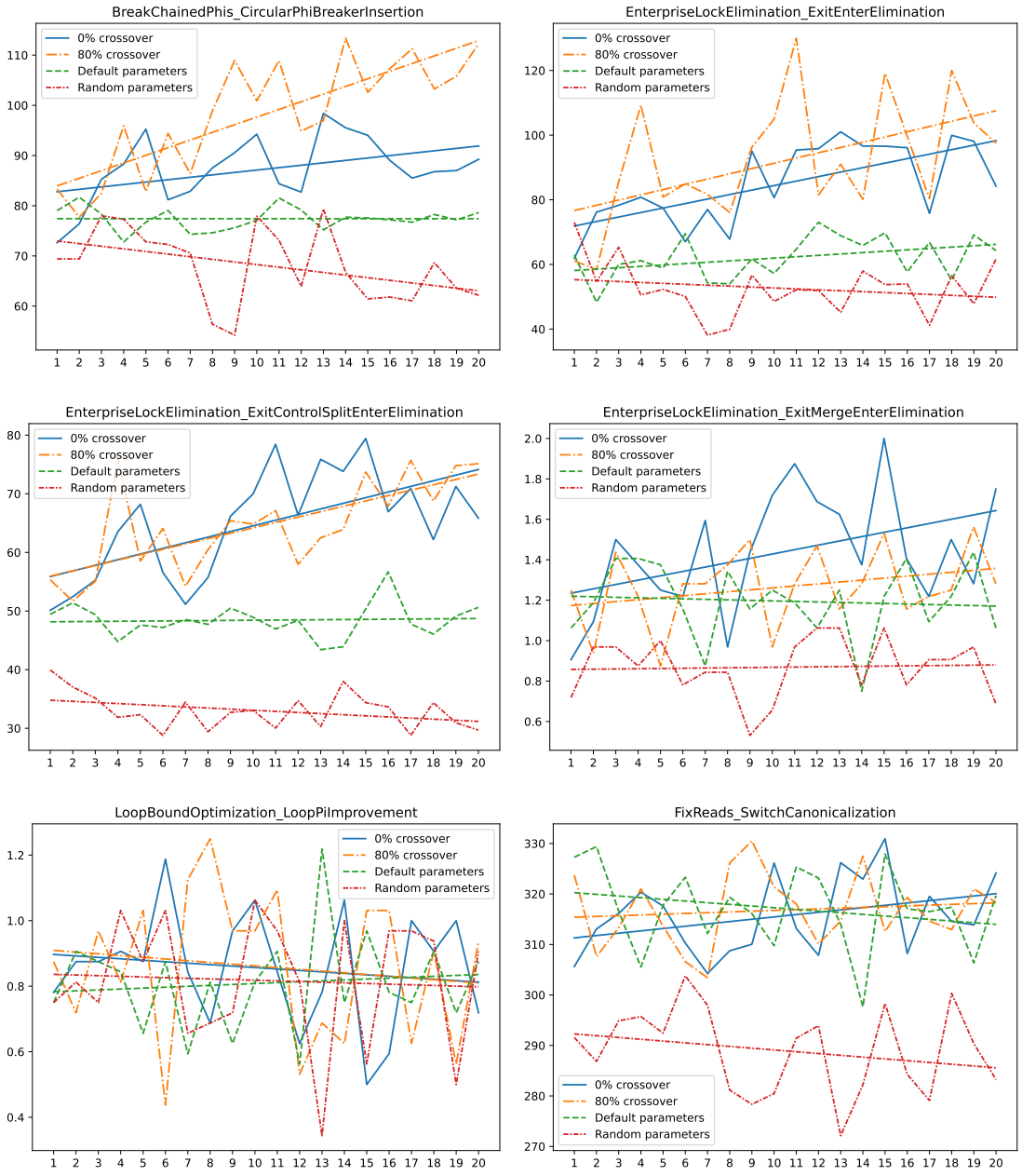


Figure 11: Average applications of rare optimizations over generations with their respective trend lines.

B Reported bugs

Table 2: Bugs that we discovered or rediscovered during our experiments. The issue number refers to the GraalVM team's internal issue tracker.

Issue	Symptom	Status
GR-52972	Compiler Crash	Open
GR-52973	Compiler Crash	Fixed
GR-53022	Compiler Crash	Fixed
GR-53084	Compiler Crash	Fixed
GR-53277	Compiler Crash	Open
GR-53279	Compiler Crash	Fixed
GR-53299	Compiler Crash	Fixed
GR-53301	Compiler Crash	Fixed
GR-53307	Compiler Nontermination	Fixed
GR-53329	Compiler Crash	Open
GR-53391	Compiler Crash	Fixed
GR-53404	Compiler Crash	Open
GR-53570	Compiler Crash	Fixed
GR-53598	Compiler Crash	Fixed
GR-53785	Compiler Nontermination	Open
GR-53893	Compiler Crash	Open
GR-53900	Compiler Crash	Fixed
GR-54218	Compiler Crash	Open
GR-54219	Compiler Crash	Open
GR-54304	Compiler Crash	Open
GR-54479	Compiler Crash	Open
GR-55993	Compiler Crash	Open
GR-55998	Compiler Crash	Open
GR-56000	Compiler Crash	Open
GR-56006	Compiler Crash	Fixed
GR-56528	Compiler Crash	Open
GR-57091	Compiler Crash	Fixed
GR-57100	Compiler Crash	Open
GR-57101	Compiler Crash	Open

GR-57118	Compiler Crash	Fixed
GR-57240	Compiler Crash	Fixed
GR-57244	Compiler Crash	Fixed
GR-57344	Compiler Crash	Open
GR-57583	Compiler Crash	Fixed
GR-57639	Miscompilation	Open
GR-57643	Miscompilation	Open
GR-57647	Miscompilation	Open
GR-57861	Compiler Crash	Fixed
GR-57901	Compiler Crash	Open
GR-57942	Compiler Crash	Open
GR-58003	Compiler Crash	Open
GR-58054	Miscompilation	Open
GR-58061	Miscompilation	Open