Differential Performance Fuzzing of Configuration Options

Haesue Baik*, Chenyang Yang†, Vasudev Vikram†, Pooyan Jamshidi‡, Rohan Padhye†, Christian Kästner†
*University of Michigan, †Carnegie Mellon University, ‡University of South Carolina

***Abstract*—Highly-configurable software often includes performance-sensitive configuration options. There are performance expectations in different configurations, but these expectations may not hold due to inaccurate mental models, corner cases, or unanticipated interactions with other options. We propose differential performance fuzzing of configuration options, a fuzzing technique that uses differential performance feedback to automatically identify inputs that violate these expectations for specific configuration changes. By guiding fuzzing toward scenarios where a supposedly faster configuration performs worse, differential performance fuzzing reveals unexpected performance behavior effectively. In our preliminary evaluation, our method identified unexpected performance gains in configurations presumed slower for 4 configuration options in Closure, demonstrating the potential for detecting performance issues in real-world applications.**

I. Introduction

Performance problems are prevalent in software but are challenging to identify [1]. Configuration options in software systems make this even more challenging [2], as the performance of a software system may be (intentionally or accidentally) influenced by configuration decisions. Many configuration options are intended to drive performance tradeoffs, for example, lower rendering quality for faster rendering times [3] or invest more time in a compiler's optimization phase to produce faster programs [4].

Developers often have a mental model of how a configuration option will influence performance, and the expected input may be explicitly documented. For example, in the *grep* implementation in Unix4j [5], configuration option `-F` (short for `--fixedStrings`) enables fixed-string matching instead of regular expressions, which according to the official documentation [6] is "usually faster than the standard regexp version." However, such performance expectations do not necessarily hold in reality. They may be violated entirely or only hold for certain inputs; they may be violated due to unrealistic assumptions, due to implementation problems, or due to unexpected interactions with other options. Indeed, we found that enabling `-F` is actually slower on many inputs, due to a naïve slow implementation of fixed-string matching (Figure 1). Testing can potentially help reveal such inputs that *violate* the performance expectation, and therefore help developers identify potential performance bugs across configurations.

In this work, we propose **differential performance fuzzing of configuration options**, a method that enhances coverage-guided performance fuzzing [7] to automatically identify inputs that violate performance expectations of configuration changes. The key idea is to introduce a new performance feedback mechanism for fuzzing (Figure 2), that encourages fuzzers to explore inputs that trigger violations regarding expected *performance difference between two configurations*. For our example above, differential performance fuzzing will generate inputs and run them on two configurations, usually differing in one performance-sensitive option. If the fuzzer finds an input that violates the performance expectation (i.e., it runs faster on the supposedly slower configuration), the input will be added for the future fuzzing loop. This way, the fuzzer can efficiently identify inputs that violate performance expectations of configurations.

```
1  public final class Unix4j {
2    public static Unix4jCommandBuilder grep(
3      GrepOptions options,
4      String regexp,
5      java.io.File... files) {
6      return builder().grep(options, regexp, files);
7    }
8  }
9
10 public class GrepTest {
11   public void configTest() {
12     Unix4j.grep(input1, input2file);
13     // This should be faster than the regexp version.
14     Unix4j.grep(Grep.Options.F, input1, input2file);
15   }
16 }
```

Fig. 1. In Unix4j, grep has a configuration of `--F` (`--fixedStrings`). This configuration uses fixed-strings matching instead of regular expressions. In the official documentation, this is stated to be "usually faster than the standard regexp version." However, our fuzzer found that this configuration is actually slower on many inputs, due to a naïve slow implementation of fixed-strings matching.

We perform a preliminary evaluation of our method on four compilation configuration options of a popular open-sourced project, Closure [8] for Javascript optimization. The assumption is that additional compile-time optimization will make compilation time longer. From our evaluation, however, we found that these configuration options, though adding additional passes to the compiler, reduce compilation time for many inputs. This demonstrated that (1) violations of performance expectation do exist in practice, and (2) performance-feedback-guided fuzzer can efficiently identify such violations.

To summarize, this paper makes the following contributions:

- Observations of performance expectations of software configurations and their potential violations.
- A differential performance fuzzing algorithm that incorporates differential performance feedback across configurations to automatically identify violations of performance expectations.

Algorithm 1 Differential configuration performance fuzzing

1: **procedure** DCPF(Program P, Set of inputs *seeds*, Budget T, Configurations *slow_config, fast_config*)
2: $corpus \leftarrow seeds$ ▷ Initialize saved inputs
3: **repeat** ▷ Fuzzing loop
4: $x \leftarrow \text{PICKINPUT}(corpus)$ ▷ Sample using heuristics
5: $x' \leftarrow \text{MUTATEINPUT}(x)$ ▷ Synthesize new input
6: **if** $\text{PERFORMANCE}(P, x', slow\_config) < \text{PERFORMANCE}(P, x', fast\_config)$ **then**
7: $corpus \leftarrow corpus \cup x'$ ▷ Violations found!
8: **if** $\text{COVERAGE}(P, x') \not\subseteq \bigcup_{x \in corpus} \text{COVERAGE}(P, x)$ **then**
9: $corpus \leftarrow corpus \cup x'$
10: **until** budget T
11: **return** *corpus* ▷ Final corpus

- A preliminary evaluation that demonstrates our method's ability to identify violations in real-world applications.

II. Background

Many software systems have hundreds or thousands of built-in configuration options to allow users to make individual tradeoff decisions, often involving performance and other qualities, such as output fidelity, formatting, and so forth [3, 4, 9, 10]. This allows developers to defer committing to specific tradeoffs and allows users to customize a system for their needs. In a nutshell, a system may have many configuration options, whether assigned as command-line parameters, in configuration files, or even at compile-time (e.g., #ifdef). A configuration is a specific assignment of values for each option [11].

Much past research has explored automatic performance optimization by tuning options (also known as algorithm selection) [9, 10, 12], automatic reconfiguration (e.g., in self-adaptive systems) [3, 13], or developer support to predict the performance influence of options to guide manual tradeoff decisions [4, 10, 14] – usually using various machine learning and optimization techniques to incrementally search for faster configurations or to build models of the expected performance influence of individual options [4, 15, 16]. Past work usually uses benchmarks for optimization or optimizes the workload in a running system, whereas our work aims to identify specific inputs that violate performance assumptions, either manually documented or learned using performance modeling techniques.

To find inputs with extreme or surprising performance behavior, fuzzing has been effective [17]. Performance fuzzing specifically searches for inputs to a function or program that maximizes its runtime, typically as a variation of coverage-guided fuzzing [7], leveraging information about program internals during the execution to drive input selection for subsequent fuzzing rounds. Performance fuzzers often find input that reveals bugs in implementations or provides insights about performance behavior in unexpected corner cases. In the former case, the implementation may be fixed, whereas in the latter case expectations and documentation may need to be adjusted (they could be considered documentation bugs [18]). However, traditional performance fuzzing focuses on a single program (in any specific configuration), not the influence of configuration options.

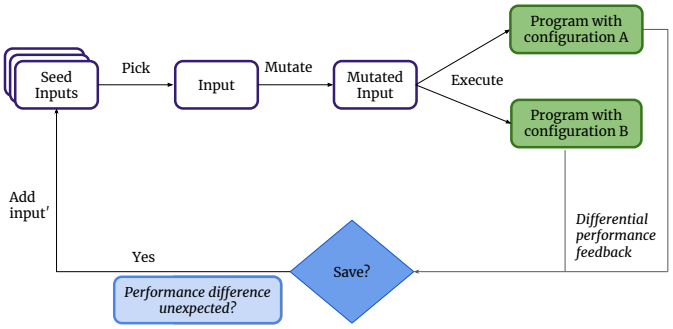

Fig. 2. Approach overview. Our performance-feedback-guided fuzzer will run each picked input on two configurations, with performance-sensitive options on and off. Identified inputs that violate performance expectation will be added to seed inputs for future selection and mutation.

Of course, performance optimization work to tune configurations can also be applied to configuration in fuzzers themselves [19], but that is orthogonal to our work. In principle, our work could be used to find configuration options in fuzzers that change the fuzzer's performance in unexpected ways.

Closest to our work is differential fuzzing [20], which runs two different versions of a program on the same input to maximize the difference in resource consumption such as identifying side-channel vulnerabilities, where resources can be execution time, consumed memory or response size. Compared to their work, we focus on different *configurations* of the same program and aim to identify violations on performance expectation, rather than maximizing resource consumption differences.

III. Differential Configuration Performance Fuzzing

A. Algorithm

The core idea of our fuzzing algorithm is to use differential performance as additional feedback for a coverage-guided fuzzer, as presented in Algorithm 1. The algorithm makes three modifications to a standard coverage-guided fuzzing algorithm [7]: First, users need to choose two configurations with expected performance differences to test against (line 1). For each selected input, the algorithm runs it in two different configurations – usually differing in a single option, but larger

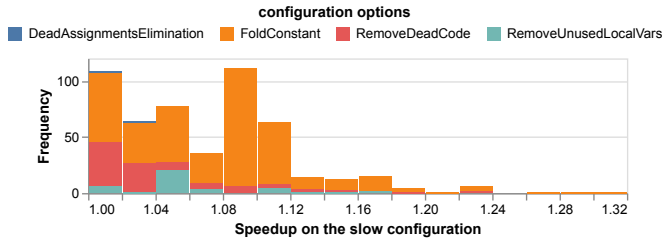

Fig. 3. We found a total of 513 inputs that violate performance expectation of configuration options. Half of the violations speed up the program by more than 1.06x, while 23% of inputs speed up the program by at least 1.1x. Most of these violations are identified for `foldConstants`, followed by `removeDeadCode`, `removeUnusedLocalVars`, and lastly `deadAssignmentElimination`, with only 3 violations identified.

configuration changes are possible too. One of the configurations is expected to have worse performance as indicated by the tester, based on their mental model or the option's documentation. The algorithm then compares the performance under both configurations and saves the input for future fuzzing if it violates the expected performance, that is, when the slower configuration turns out to be faster (line 7). The rest of the algorithm is the same as in standard coverage-guided fuzzing.

B. Implementation

We implemented our algorithm as an extension of the JQF library [21], a popular Java fuzzing framework. We track coverage information and use the number of bytecode instructions executed as an approximate measure of program performance, which linearly scales with the number of instructions. We chose this over CPU clock time because it can provide more fine-grained information. As pointed out by prior work on performance fuzzing [17], coverage information allows us to count beyond just the number of lines covered but also which lines are covered specifically (e.g., lines in a loop).

IV. Preliminary Evaluation

Our preliminary evaluation aims to understand:

- **RQ**: Can our differential configuration performance fuzzing algorithm identify inputs that violate performance expectations in real-world applications?

A. Experiment Setup

We selected a popular open-source project, Closure [8], for evaluating our method. From Closure's official documentation of its `compile` method, we identified a series of configuration options that apply different compile-time optimization techniques. Since they each perform additional work that may speed of the resulting program, we expect that the *compiler's performance* to compile a program is slower with additional optimization options. Specifically, we selected 4 options for our evaluation: `removeDeadCode`, `removeUnusedLocalVars`, `deadAssignmentElimination`, and `foldConstants`. For each option, we ran our fuzzer for 30 minutes, with the full `SIMPLE_OPTIMIZATIONS` as the slow configuration, which turns on all the above

```
1  for (;;
2       ((n_0).i_1)) {
3       continue;
4       (!("WS(hm>/"));
5       throw (this);
6       continue;
7  }
```

Fig. 4. An examplar generated fuzzing input with dead code (line 4-6) for Closure. With `removeDeadCode` on, Closure is expected to perform additional passes and hence slower for compilation. However, we found 98 inputs that violate this performance expectation. This is because of *interactions* between different passes: For some inputs with dead code, `removeDeadCode` can reduce time for future passes, as they only need to analyze a smaller AST, which reduces total compilation time.

configuration options along with many other options, and the full `SIMPLE_OPTIMIZATIONS` with the selected option turned off as the fast configuration. For all violations we identified, we compute the speedups of using the number of instructions executed.

B. Results

For all four configuration options tested, we successfully identified inputs that violate performance expectations. Overall, we found a total of 513 inputs that violate the performance expectation of the configuration options. Half of the violations speed up the compiler by more than 1.06x, while 23% of inputs speed up the program by at least 1.1x. Most of these violations are identified for `foldConstants`, followed by `removeDeadCode`, `removeUnusedLocalVars`, and lastly `deadAssignmentElimination`, with only 3 violations identified. This establishes the feasibility of identifying inputs that violate performance expectations using our method.

Upon closer inspection of the generated inputs, we found that this happens because of *interactions* between different compilation options. This is exemplified in Figure 4: For some inputs with dead code, an additional pass from `removeDeadCode` can reduce time for future passes, as they only need to analyze a smaller abstract syntax tree, which reduces total compilation time. This explains the variances in how frequently the speedups occur across configurations (Figure 3), as `foldConstants` is likely to be applied more frequently on generated inputs.

V. Conclusion

This work proposes the idea of differential performance fuzzing of configuration options. Our preliminary evaluation demonstrated the feasibility of this approach. We envision there are several future directions that can be explored:

Identifying performance-sensitive configuration options. To perform differential performance fuzzing of configuration options, we assume the knowledge of such performance-sensitive configuration options, which is currently acquired by manually reading through official documentation. Future work can explore how to automatically identify such configuration options and their performance expectation from documentations using large language models [e.g., 22] to fully automate the fuzzing loop. With configuration options automatically identified, future work can further conduct a larger-scale study on other

open-sourced projects. In particular, it would be interesting to use our method to understand performance impact of options of a fuzzer [e.g., 19].

More efficient differential performance fuzzing algorithm. Our current fuzzing algorithm treats each configuration pair independently. However, there is a lot of information that can be reused across configuration pairs. Future work can explore how to reuse coverage information from one fuzzing run for another to improve efficiency.

ACKNOWLEDGMENT

The work was supported by NSF awards 2106853, 2107463, 2120955 and the NSF REU site *REU-SE* 2244348.

REFERENCES

[1] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 77–88, 2012.

[2] X. Han and T. Yu, "An empirical study on performance bugs for highly configurable software systems," in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2016, pp. 1–10.

[3] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic knobs for responsive power-aware computing," *ACM SIGARCH computer architecture news*, vol. 39, no. 1, pp. 199–212, 2011.

[4] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner, "Performance-influence models for highly configurable systems," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 284–294.

[5] "unix4j: An implementation of unix command line tools in java." [Online]. Available: https://github.com/tools4j/unix4j

[6] "Grepoptions." [Online]. Available: http://unix4j.org/javadoc

[7] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon, "Semantic fuzzing with zest," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 329–340.

[8] "closure-compiler: A javascript checker and optimizer." [Online]. Available: https://github.com/google/closure-compiler

[9] H. H. Hoos, "Automated algorithm configuration and parameter tuning," in *Autonomous search*. Springer, 2012, pp. 37–71.

[10] P. Jamshidi and G. Casale, "An uncertainty-aware approach to optimal configuration of stream processing systems," in *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2016, pp. 39–48.

[11] S. Apel, D. Batory, C. Kästner, and G. Saake, "Feature-oriented software product lines," 2013.

[12] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *Learning and Intelligent Optimization: 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers 5*. Springer, 2011, pp. 507–523.

[13] P. Jamshidi, M. Velez, C. Kästner, and N. Siegmund, "Learning to sample: Exploiting similarities across environments to learn performance models for configurable systems," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 71–82.

[14] M. Velez, P. Jamshidi, N. Siegmund, S. Apel, and C. Kästner, "On debugging the performance of configurable software systems: Developer needs and tailored tool support," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1571–1583.

[15] P. Jamshidi, N. Siegmund, M. Velez, C. Kästner, A. Patel, and Y. Agarwal, "Transfer learning for performance modeling of configurable systems: An exploratory analysis," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 497–508.

[16] M. Velez, P. Jamshidi, N. Siegmund, S. Apel, and C. Kästner, "White-box analysis over machine learning: Modeling performance of configurable systems," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1072–1084.

[17] C. Lemieux, R. Padhye, K. Sen, and D. Song, "Perffuzz: Automatically generating pathological inputs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 254–265.

[18] A. Rabkin and R. Katz, "Static extraction of program configuration options," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 131–140.

[19] Z. Zhang, G. Klees, E. Wang, M. Hicks, and S. Wei, "Fuzzing configurations of program options," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 2, pp. 1–21, 2023.

[20] S. Nilizadeh, Y. Noller, and C. S. Pasareanu, "Diffuzz: differential fuzzing for side-channel analysis," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 176–187.

[21] R. Padhye, C. Lemieux, and K. Sen, "Jqf: Coverage-guided property-based testing in java," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 398–401.

[22] Z. Wang, D. J. Kim, and T.-H. Chen, "Identifying performance-sensitive configurations in software systems through code analysis with llm agents," *arXiv preprint arXiv:2406.12806*, 2024.