This is a preprint of a paper that will appear in: 2021 IEEE 37th International Conference on Software Maintenance and Evolution (ICSME), Luxembourg City, Luxembourg (Virtual Event), 2021.

Test Case Reduction: A Framework, Benchmark, and Comparative Study

Patrick Kreutzer, Tom Kunze, Michael Philippsen
Programming Systems Group, Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany
patrick.kreutzer@fau.de, tom.kunze@fau.de, michael.philippsen@fau.de

***Abstract*—Given a program that triggers a bug in a compiler (or other kind of language processor), the goal of *test case reduction* is to cut away all code that is irrelevant for the bug, i.e., to generate a smaller program that still induces the bug. Research has proposed several language-agnostic reduction techniques that automatically reduce bug-inducing programs in arbitrary programming languages, but there is no large-scale, conclusive evaluation of these algorithms yet. Furthermore, the development of new algorithms is hampered by the unavailability of comparable implementations of previous techniques and of diverse test programs that trigger different bugs in real compilers.**

To close these gaps and to foster future research in this area, this paper makes three contributions: (1) A *framework* that includes efficient, fine-tuned implementations of 6 state-of-the-art reducers, (2) a diverse *benchmark* that comprises 321 fuzzer-generated programs in two programming languages that trigger 110 different bugs in real compilers, and (3) a *comparative study* that builds upon our framework and benchmark and compares the reduction techniques w.r.t. their effectiveness and efficiency. Our results show that there is no reduction technique yet that performs best across all test cases and languages.

Our framework and benchmark are available online and we provide the necessary scripts and tools to replicate our study.

***Index Terms*—test case reduction, compiler testing, benchmark**

I. Introduction

Test case reduction is an important component in any compiler developer's toolbox. A program that triggers a compiler bug (e.g., a crash or miscompilation) usually contains code that is irrelevant for the bug. This excessive code often makes it difficult to find the root cause of the bug. Reducing the program to a smaller one that still induces the bug thus aids developers in finding and fixing it [1].

Since manually reducing a failure-inducing program is cumbersome, research has proposed different reduction algorithms. While there are some approaches that are tailored to specific programming languages (see Sec. VI), recent research mainly focuses on *language-agnostic* techniques that claim to be applicable to programs in arbitrary languages. Unfortunately, there is no large-scale evaluation yet that compares these language-agnostic techniques in a thorough and conclusive way. Even worse, we found that research on test case reduction is unnecessarily hampered: (1) The available implementations of previous techniques use different implementation languages, input grammars, and representations of the input program on which they operate. This makes a fair comparison of different reducers difficult and is probably one of the main reasons why some reduction algorithms have never been compared with each other before. (2) There is no extensive collection of (still unreduced) programs that trigger bugs in real compilers. Code from bug trackers is ill-suited, since compiler developers typically ask for reduced programs in bug reports [2], [3]. Since this makes gathering a diverse set of suitable test programs a complex and tedious task, some reducers have only been evaluated with few test programs. For example, *HDD* [4] has been evaluated with only 2 XML files and 4 C programs (one of which is a synthetic example), and even the recent *Pardis* [5] has been evaluated with only 14 C programs.

To close these gaps and to foster future research in this area, this paper makes the following three main contributions: (1) We introduce a *framework* that includes efficient, fine-tuned implementations of 6 state-of-the-art test case reducers (*HDD* [4], *CoarseHDD* [6], *HDDr* [7], *GTR* [8], *Perses* [9], and *Pardis* [5]), as well as several variants thereof. The framework can easily be extended with new reducers and eases their evaluation against previous techniques. (2) We present a diverse *benchmark* consisting of 321 fuzzer-generated programs in two very different programming languages (C and SMT-LIB 2) that trigger 110 different bugs in 5 real-world compilers, as well as an automated execution environment to evaluate each test program with the respective compiler. This frees researchers from the burden of gathering diverse and meaningful test cases and considerably reduces the work to set up a testing environment. (3) We present the results of a *comparative study* that builds upon our framework and benchmark and that evaluates the reduction techniques. To the best of our knowledge, this is by far the most extensive comparison of test case reducers to date. Our results show that there is no reducer yet that works best across all test programs and languages and also offer interesting insight for future research. Furthermore, the considerable variations across the test programs emphasize the necessity of a diverse benchmark for a meaningful and reliable evaluation of test case reducers.

Our framework and benchmark are available online under an open-source license and we provide the necessary scripts and instructions to replicate the results of our study.

II. Test Case Reduction

This section formalizes test case reduction and introduces the terminology that we use in the remainder of this paper.

Let $\mathbb{P}$ denote the set of all possible programs and $|P|$ the size of a program $P \in \mathbb{P}$. Throughout this work, we measure a program's size by counting the number of tokens in it (also see

© 2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Sec. III-A); this number ignores whitespace and is therefore robust w.r.t. different formattings. A *test function* is a function $\text{TEST} : \mathbb{P} \mapsto \{✗, ✓\}$ that takes as input a program $P \in \mathbb{P}$ and returns ✗ if P triggers a bug in the compiler under test (and ✓ otherwise). The definition of the TEST function depends on the specific use case; for example, it may return ✗ if compiling P leads to a violated assertion in the compiler under test.

When given a TEST function and an original program P_O with $\text{TEST}(P_O) = ✗$ as input, the goal of a *test case reducer* R is to find a program P_R with $\text{TEST}(P_R) = ✗$ such that $|P_R| < |P_O|$, typically by removing or replacing parts of P_O. Sec. III explains how the algorithms perform this reduction.

In general, test case reducers do *not* try to find the smallest possible program that triggers the bug, since this is computationally prohibitive [10]. Instead, the different algorithms use different heuristics to reduce the input. As a result, they vary w.r.t. their effectiveness and efficiency. Since most algorithms do not give any guarantees regarding the minimality of their results, applying a reducer R in a fixpoint iteration oftentimes leads to even smaller results: from the original input P_O, R obtains a reduced program P_R, which is then fed into R again; this process is repeated until no more changes occur and the reduction returns the final result P_R^*. Note that, in general, P_R^* is still not the smallest possible program with $\text{TEST}(P_R^*) = ✗$; R just cannot reduce P_R^* any further. Throughout this paper, we write R^* to denote the fixpoint variant of a reducer R.

III. FRAMEWORK

This section describes our *framework* that contains efficient implementations of 6 state-of-the-art test case reducers and several variants thereof. Since all of them operate on the syntax tree of the input program, Sec. III-A first gives some technical details on how our framework parses the programs and how the resulting trees look like. Sec. III-B then introduces the two list reduction algorithms that the reducers employ under the hood to determine the tree nodes that can be removed. Finally, Sec. III-C gives high-level descriptions of the algorithms.

Our framework is implemented in *Java* and is fully self-contained. Along with a documentation that explains how to build, execute, and extend the framework, it can be found at:

https://github.com/FAU-Inf2/RedPEG

A. Input Parsing and Syntax Trees

State-of-the-art reducers operate on the syntax tree of the input program, which is constructed by parsing the program w.r.t. a grammar for the respective programming language. Our framework uses *parsing expression grammars* (PEGs) [11] as the grammar formalism to describe the language rules. Given a PEG as input, it dynamically builds an efficient *packrat parser* [12] at runtime that then parses the program. (Thus, the framework is fully self-contained and does not rely on external tools to translate a grammar to parser code first.)

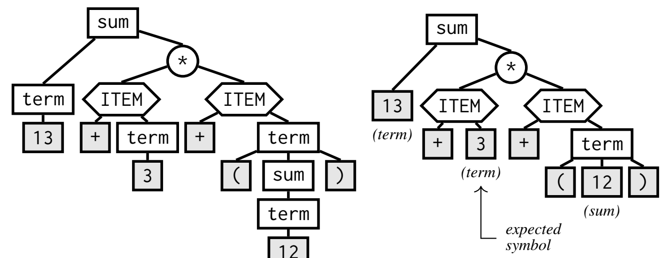
Fig. 1 shows an example PEG that describes (parenthesized) sums of natural numbers. Upper-case symbols represent *terminals* described by regular expressions (e.g., an ADD terminal consists of a single "+" character). These terminal definitions are used to split the input program into its *tokens*. Lower-case symbols represent *non-terminals* described by productions that may reference other terminals and non-terminals (e.g., a term is either a NUM or a sum enclosed in a pair of parentheses). Productions can apply *quantifiers* to describe possible repetitions of sub-rules: the *- and +-quantifiers allow any number of repetitions (but the +-quantifier requires at least one occurrence), the ?-quantifier describes optional occurrences.

```
sum: term ( ADD term )* ;
term: NUM | LPAREN sum RPAREN ;
NUM: '0' | [1-9][0-9]* ;
ADD: '+' ; LPAREN: '(' ; RPAREN: ')';
```

Fig. 1. PEG for (parenthesized) sums of natural numbers.

During parsing, the parser builds a *syntax tree* that describes the structure of the input program w.r.t. to the PEG. Fig. 2(a) shows the syntax tree for the input "13+3+(12)" and the grammar in Fig. 1. Inner nodes represent non-terminals and leaves represent terminals (i.e., the program's tokens).

We designed our syntax trees in a way that supports all reducers. There are two noteworthy particularities: (a) To also cover the requirements of the more advanced reducers, our syntax trees contain *quantifier nodes* (e.g., the circular *-node in the example) that represent matches of quantified sub-rules. Each child of such a node is represented as a sub-tree rooted at an auxiliary ITEM node (hexagons). The reducers handle these nodes differently. For example, whereas *HDD* [4] (essentially) just ignores the quantifier nodes during reduction, our *Perses* [9] implementation uses these nodes to identify lists in the syntax tree. Furthermore, *Perses* assumes that each element of such a list is represented by a single sub-tree with a single root; the ITEM nodes ensure this, even if the quantified sub-rule consists of multiple terminals and/or non-terminals (as in our example). (b) We *compactify* the syntax trees in a preprocessing step by collapsing linear chains of regular nodes to a single node, see Fig. 2(b). To not lose the original information, we annotate the resulting nodes with the original non-terminal (called the *expected symbol* below). These linear chains may occur if a non-terminal A *subsumes* a terminal or non-terminal B, i.e., if A can directly or indirectly be expanded to a single B. Such subsumptions are quite prevalent



(a) Original syntax tree. (b) Compactified syntax tree.

Fig. 2. Syntax tree for "13+3+(12)" and the PEG in Fig. 1.

© 2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

in typical grammars (e.g., due to auxiliary non-terminals that model the precedence levels of arithmetic operators). In our example, `sum` subsumes both `term` and `NUM`, whereas `term` only subsumes `NUM`. The compactification is motivated by the results of Hodován et al. [13], who observed that reducing such chains with *HDD* may require many superfluous TESTs, which hampers the reducer's efficiency. In preliminary experiments, we have found this to be true for the other reducers as well.

B. List Reduction Algorithms

Most reducers from Sec. III-C repeatedly collect tree nodes into a list and then apply a *list reduction algorithm* to identify (and remove) the irrelevant ones. This section introduces the two algorithms that the reducers make use of, *DDMin* [10] and *OPDD* [14].[1] In our framework, any reducer can be combined with any list reduction algorithm to enable further experiments in the future. For example, this enables an easy evaluation of parallel [15] or advanced [16] versions of the list reduction algorithms. For the comparison in Sec. V, we always used the algorithm stated in the respective publication.

1) Delta Debugging (DDMin): DDMin by Zeller and Hildebrandt [10] successively removes contiguous chunks of list elements until the result is *1-minimal*, i.e., until the removal of any single element no longer triggers the bug (the removal of larger and/or non-contiguous chunks may still be possible).

Let L denote the list of tree nodes that should be reduced. For the sake of presentation, assume that *DDMin* is applied to the list of all leaf nodes in the tree from Fig. 2, in which case L initially consists of the seven elements $[\texttt{13}, \texttt{+}, \texttt{3}, \texttt{+}, \texttt{(}, \texttt{12}, \texttt{)}]$. In each step, *DDMin* splits L into n subsets Δ_i of roughly the same size and checks if a Δ_i or a complement ∇_i of a Δ_i still triggers the bug. *DDMin* starts with $n = 2$, but if no reduction is possible, it repeatedly increases the granularity to divide L into successively smaller chunks. In our example, if no reduction is possible for $n = 2$, n is set to 4 and $\Delta_1 = [\texttt{13}, \texttt{+}]$, $\Delta_2 = [\texttt{3}, \texttt{+}]$, $\Delta_3 = [\texttt{(}, \texttt{12}]$, $\Delta_4 = [\texttt{)}]$. If one of the subsets Δ_i still triggers the bug, Δ_i is handled recursively and the granularity is reset to $n = 2$ (*"reduce to subset"*). Otherwise, if a complement ∇_i triggers the bug, ∇_i is handled recursively (*"reduce to complement"*). In this case, n is decreased by 1 so that the same subsets are *revisited*. For example, after a reduction to $\nabla_2 = [\texttt{13}, \texttt{+}, \texttt{(}, \texttt{12}, \texttt{)}]$ the subsets Δ_1, Δ_3, and Δ_4 (as well as their new complements) are considered again. This is repeated until no more elements can be removed and the granularity cannot be increased any further. In the best case, this requires $\mathcal{O}(log\, n)$ TESTs for n elements, but in some cases *DDMin* requires up to $\mathcal{O}(n^2)$ checks.

Note that *DDMin* may TEST the same configuration multiple times [10]. Our implementation tracks which configurations have already been tested and avoids their re-evaluation. Also note that the *"reduce to subset"* step is not necessary to achieve 1-minimal results [15] (and in a later work, Zeller et al. omitted this step altogether [17]). Since the results of our preliminary experiments were inconclusive whether this step is beneficial or not, we used the original definition of *DDMin* for our study (but the framework includes all variants).

2) One Pass Delta Debugging (OPDD): After a successful *"reduce to complement"* step, *DDMin* revisits all previously considered subsets. Gharachorlu and Sumner [14] observed that this revisiting may be inefficient and not necessary in practice. They thus proposed a modified version of *DDMin* called *OPDD*. It shares the same best case runtime complexity as *DDMin*, but only requires $\mathcal{O}(n)$ TESTs in the worst case.

If L cannot be reduced to a single subset Δ_i, *OPDD* loops over all subsets and tries to remove them one after the other. For example, it first tries to remove the first subset Δ_1 (this is equivalent to testing ∇_1 in *DDMin*). If this still triggers the bug, it tries to also remove Δ_2 (instead of recursively handling ∇_1 like *DDMin*) and so on. The resulting list is then handled recursively with increased granularity.

Gharachorlu and Sumner showed that, under certain conditions, *OPDD* achieves 1-minimal results without *DDMin*'s revisiting. If these conditions are not satisfied, the result is not necessarily 1-minimal, but the authors showed empirically that the impacts on the reducer's effectiveness are negligible in practice (and they also showed that most test programs satisfy these conditions). Note that, even when *OPDD*'s result is not 1-minimal, it may still be smaller than that of *DDMin* (e.g., since different reductions in the beginning may enable more, or other, reductions later on).

C. Reduction Algorithms

This section gives high-level descriptions of the reduction algorithms in our framework; details can be found in the respective publications. All algorithms operate on the syntax tree of the input program, but they differ in the operations that they apply to the tree to generate new reduction candidates.

1) Hierarchical Delta Debugging (HDD): HDD by Misherghi and Su [4] reduces a program tree level by level, from top to bottom. After collecting all nodes of the current level into a list, it uses a list reduction (*DDMin* by default) to identify and remove the irrelevant ones (and their sub-trees). It then recursively considers the children of the remaining nodes.

Our *HDD* implementation skips all quantifier nodes (*HDD* does not handle lists in a special way), but explicitly tries to remove `ITEM` nodes. Thus, it considers the two `ITEM`s in Fig. 2(b) to be on the same level as the node "`13`" and uses a list reduction to determine which of these nodes are removable.

Deleting nodes often leads to syntactically invalid candidates, which TEST functions often reject (e.g., deleting the `term` node yields the invalid "`13+3+`"). To only generate syntactically valid candidates, the authors suggested to apply a (manually defined, language-specific) *tree manipulator* that replaces deleted nodes with small, syntactically valid replacements (e.g., it could replace the `term` "`(12)`" with a plain number). In a later work [18], Misherghi gave an algorithm to automatically compute such replacements for all symbols of a quantifier-free grammar; Hodován and Kiss [19] extended this computation for grammars with quantifiers. If a node is marked

[1] These algorithms can also be used as standalone reducers by splitting the input program into a list of characters, tokens, or lines. We omit these standalone versions, since they are inferior to the techniques from Sec. III-C.

© 2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

for deletion, we replace it with the minimal replacement for its *expected symbol*. ITEM nodes have an empty replacement since their parent is always a quantifier node; we can thus completely remove ITEMs without losing syntactic validity.[2]

We also employ another idea by Hodován et al. [13]: if a sub-tree already matches its minimal replacement, considering it for deletion is unnecessary. For example, deleting the "(" node from the tree in Fig. 2(b) has no effect, since it would be replaced with a "(" anyway. To reduce the number of TESTs, we filter out such "unremovable nodes."

2) CoarseHDD: CoarseHDD by Hodován et al. [6] is an *HDD* variant that trades effectiveness for efficiency by only considering nodes with an empty replacement (i.e., nodes that can be deleted without losing syntactic validity). This focuses the work on parts where the most reduction is to be expected.

In addition to the removable ITEM[3] nodes, there may also be regular nodes that have an empty replacement. We can automatically infer from a given grammar which nodes can be safely deleted by computing the minimal replacement for each grammar symbol (as we do for *HDD*, see above).

3) HDDr: The *HDDr* algorithm by Kiss et al. [7] is another *HDD* variant. It is based on the observation that collecting *all* nodes of a tree level into a common list does not respect the syntactic structures of the underlying language, since such a list may contain nodes of unrelated program parts. This may hamper the list reduction and hence adversely affect the reducer's efficiency and effectiveness. *HDDr* thus restricts the collection of nodes to sibling nodes and handles groups of siblings on the same level one after another. For example, if the two ITEMs from Fig. 2(b) cannot be removed, the original *HDD* considers a single list containing all their children next. In contrast, *HDDr* handles the children of both ITEMs separately.

Two parameters control the tree traversal. The first one controls whether *HDDr* performs a depth-first or a breadth-first traversal, the second one determines if groups of siblings are handled from left to right or from right to left. Based on the published results, we use a depth-first, right-to-left traversal for our study (but our implementation supports all four variants).

Just like our *HDD* implementation, our *HDDr* implementation replaces removed nodes with their (automatically computed) minimal replacement and filters out unremovable nodes.

4) Generalized Tree Reduction (GTR): In contrast to the *HDD* variants, *GTR* by Herfert et al. [8] not only supports the deletion of nodes but can also apply arbitrary transformations to the input tree. These transformations are specified by means of *reduction templates*, of which *GTR* uses two different ones: the *Deletion* template simply tries to remove a node, whereas the *Substitute-by-child* template tries to replace a node with one of its (direct) children. To reduce an input tree, *GTR* processes its nodes level by level like *HDD*. After collecting all nodes of the current level into a list, it applies all templates one after another. For templates that return a single candidate per node (e.g., the *Deletion* template), *GTR* uses a list reduction algorithm (*DDMin* by default) to determine the set of nodes that should be transformed. For templates that may return more than one candidate per node (e.g., the *Substitute-by-child* template), *GTR* uses a greedy backtracking algorithm to decide which transformations to apply to which nodes.

In general, this process generates many reduction candidates, many of which are syntactically invalid and often rejected by the TEST function. To narrow the search space and to reduce the number of TEST evaluations, *GTR* uses a *filtering* mechanism. For this purpose, it analyzes a corpus of example programs to determine the possible contexts that each node can appear in. A node is only considered for deletion or substitution if the resulting context matches one of the previously determined ones.[4] To not depend on a corpus of example programs and to make sure that we do not dismiss a possible transformation, we follow a slightly different approach: before the actual reduction starts, we determine *all* possible contexts based on the given input grammar (the runtime is negligible, even for complex grammars); in addition, we handle *subsumed* symbols faithfully when checking a context. This is equivalent to using a "perfect" corpus that covers all possible contexts.

In the example in Fig. 1, sum subsumes NUM. Thus, all valid contexts for sum nodes are also valid contexts for NUM nodes. It is therefore syntactically safe to substitute the sum tree from Fig. 2(b) by its NUM child "13". Note that our version of the *Substitute-by-child* template skips all quantifier and ITEM nodes when searching for a node's children. This enables *GTR* to also substitute the sum node with, say, the NUM node "3".

5) Perses: Similar to *GTR*, *Perses* by Sun et al. [9] reduces a program tree by both deleting nodes and by replacing nodes with compatible descendants. Unlike *GTR*, it ensures that all reduction candidates are syntactically valid.[5]

Starting at the root node, *Perses* reduces the input tree node by node. For this purpose, it maintains a priority queue of nodes sorted by the number of terminals in their respective sub-tree. In each step, it pops the largest node from the queue.

If the current node is a quantified list, *Perses* applies a list reduction algorithm (*DDMin* by default) to reduce the list's elements, obeying the additional constraint imposed by +-quantified lists. All list elements that have not been removed are then added to the priority queue for further reduction.

If the current node is a regular non-terminal node, *Perses* uses a bounded breadth-first-search to determine syntactically compatible descendants that the current node could be replaced with. A descendant is compatible if the *expected symbol* of the current node *subsumes* the descendant's symbol (if the current node's parent is a quantifier node, *Perses* also considers compatible quantifier nodes as replacements; see the original paper for more details). In the example from Fig. 2(b), the nodes

[2] In general, the last ITEM of a +-quantified list cannot be deleted. Thus, if the last ITEM is to be removed, we replace it with its minimal replacement.

[3] In contrast to its original description, our *CoarseHDD* implementation keeps at least one ITEM of +-quantified lists to guarantee syntactic validity.

[4] Despite the filtering, *GTR* may produce syntactically invalid candidates, since it applies the filtering for each node individually but does not check the combination of transformations applied to all nodes of the current level.

[5] Perses claims to be the first reducer that guarantees syntactic validity. However, as described above, *HDD* and its variants also only generate syntactically valid programs if appropriate replacements for deleted nodes are provided (which can be deduced automatically from the underlying grammar).

© 2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

labeled "13", "3", and "term" are compatible descendants for the tree's root node. *Perses* then tries to replace the current node with all candidates one after another to find the smallest replacement that still triggers the bug (if any). If the current node can be replaced, its replacement is added to the priority queue; otherwise, the node's original children are added.

While *Perses* works with arbitrary grammars, its efficiency and effectiveness depend on its ability to identify lists in the syntax tree. The authors thus introduced an algorithm to automatically convert a grammar to its *Perses Normal Form* (*PNF*), which tries to introduce as many quantifiers to the grammar as possible. Unfortunately, we found—like others before us [20]—that the proposed algorithm cannot handle all grammars correctly. In addition, some of the transformation steps are incompatible with the semantics of PEGs that we use as the grammar formalism in our framework. We thus decided to leave out this automatic transformation and to manually add all quantifiers to the grammars used for our study instead.

6) Pardis: The *Pardis* reducer by Gharachorlu and Sumner [5] is a variant of *Perses*. The authors observed that *Perses* suffers from *priority inversion*. If a list contains many small elements, its total size may be larger than that of any other sub-tree currently contained in the priority queue. In this case, *Perses* determines the minimal configuration for the list's elements and, in doing so, may spend considerable effort on removing some of these small list elements (instead of first focusing the work on other, larger sub-trees).

Pardis tries to avoid this by adjusting how lists are handled during the reduction. Instead of reducing a list's elements with a list reduction algorithm, *Pardis* adds all elements to the priority queue individually to handle them one after another.[6] In each step, it pops the largest element from the queue and checks if removing it from the program still triggers the bug.[7] This way, *Pardis* always handles the largest sub-trees first. This may lead to a faster reduction at the beginning, which in turn may increase the overall performance.

Unlike *Perses*, *Pardis* does not try to replace regular nodes with compatible descendants. The authors stated that their own *Perses* implementation with replacements often reached a timeout and hence concluded that this replacement might be detrimental. We could not observe this with our own *Perses* implementation, but our *Pardis* implementation matches its original definition and does not handle regular nodes either.

In case of long lists, the *Pardis* strategy of handling only one node at a time becomes inefficient, since it requires one TEST per list element. In contrast, the list reduction algorithms from Sec. III-B try to remove multiple elements at once, which in general reduces the number of required checks. Gharachorlu and Sumner thus also introduced the *Pardis Hybrid* variant. It groups together all elements from the same list that have the exact same number of tokens. If more than one element is popped from the priority queue, *Pardis Hybrid* uses a list reduction algorithm (*OPDD* by default) to decide which of these elements can be removed. This variant requires fewer TESTs in some cases while still respecting the priorities. Our comparative study in Sec. V includes both variants of *Pardis*.

[6]Contrary to the descriptions in the paper, the available *Pardis* implementation first tries to completely remove a list before it handles its items individually. Since our *Perses* implementation effectively does the same, we adapted *Pardis* accordingly. A "*nullability pruning*" step [5] ensures that lists with one element are only checked once.

[7]As explained above, removing a list element is syntactically safe in most cases, but +-quantified lists have to be handled specially. The original paper does not consider this case explicitly, so we added respective checks ourselves.

IV. BENCHMARK

This section describes our *benchmark* that contains 321 fuzzer-generated C and SMT-LIB 2 programs that trigger 110 real bugs in 5 different compilers. Sec. IV-A first describes the design of our benchmark; Sec. IV-B then gives some more detailed statistics. The benchmark itself is available online:

`https://github.com/FAU-Inf2/RedBench`

A. Benchmark Design

Language-agnostic reducers that claim to be broadly applicable should be evaluated with test programs that (1) trigger *different bugs* in real compilers, (2) cover *different languages and input formats* with diverse characteristics, and (3) cover a *large range of program sizes* (i.e., the programs should vary in their amount of excessive code that can be cut away).

Since bug databases of compilers often only contain reduced programs, collecting such programs manually is not an option. Luckily, the scientific literature has proposed several highly effective *fuzzing* techniques for the automatic generation of random test programs that uncover compiler bugs [1], [21], [22], [23], [24], [25]. In fact, reducers have been evaluated with fuzzer-generated programs before; for example, 13 of the 14 programs that the recent *Pardis* [5] has been evaluated with have been generated with the *Csmith* [22] C compiler fuzzer.

To ensure a diverse benchmark, we chose two very different languages. Besides the often used imperative language C, we chose the declarative language SMT-LIB 2 [26], which most state-of-the-art SMT solvers support as input language. As also evidenced by the results of our comparative study in Sec. V, the SMT-LIB 2 test cases complement the C test cases well: (1) The Lisp-like SMT-LIB 2 syntax is based on S-expressions and contains fewer quantified lists and optional parts. (2) SMT-LIB 2 is more strongly typed [23], which, in general, imposes more requirements on the reduction candidates to not be rejected by the TEST function. (3) The TEST evaluation times are much lower for SMT-LIB 2 (on average, 44 ms vs. 512 ms for successful reductions in our study) and much less dependent on the program size (Spearman correlation of -0.12 vs. 0.54); this may favor some reducers more than others.

To further diversify the benchmark, we used two different fuzzers per language that produce very different output. To generate the C test cases, we used *Csmith* [22] and *$*$Smith* [23]. While *Csmith* is a language-specific fuzzer that generates executable programs without undefined behavior, *$*$Smith* is a language-agnostic fuzzer that takes as input a specification describing the syntactical and semantic rules of the respective language. We used the C specification from the *$*$Smith* repository which generates compilable (but not necessarily

© 2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

executable) programs. These programs make heavy use of arithmetic expressions, while *Csmith* generates calls of wrapper functions to avoid the undefined behavior of overflows. Both fuzzers detect compiler crashes, but only *Csmith* can also detect miscompilations via *differential testing* [27], [22].

To generate the SMT-LIB 2 test cases, we used the three available language specifications for ∗*Smith* [23] (which cover different language features) as well as the language-specific *FuzzSMT* [28], [21]. While the ∗*Smith* test cases consist of (potentially many, but relatively small) commands contained in a top level list, the *FuzzSMT* test cases mainly consist of a single (but potentially very deeply nested) command. Both fuzzers can detect crashes in SMT solvers as well as cases in which a solver computes a wrong result (there is no undefined behavior in the SMT-LIB 2 fragments that we consider).

We let the fuzzers run for several weeks and initially collected all programs that triggered a bug in one of the compilers under test (see Sec. IV-B). We then manually analyzed and labeled these programs to distinguish between different bugs. To keep our benchmark diverse, we kept at most 4 different programs of varying size per bug and fuzzer (having some programs for the same bug is not a problem, since test case reduction is mostly concerned with the code parts that are *unrelated* to the bug). Finally, we augmented each program with metadata that precisely describes the respective bug, and used this data to automatically derive the TEST functions (this makes adding new test programs easy). To reduce the risk of *bug slippage* [29], [30] (i.e., cases in which a reduced program triggers a *different bug* than the original one), we made the TESTs as specific as possible (e.g., in case of a compiler crash we TEST for the exact error message and location). To keep the reductions of C miscompilation bugs free of undefined behavior, their TEST functions reject all candidates for which different compilers or the *Frama-C* [31] framework detect such behavior; for more details, please refer to Regehr et al. [32].

We found that some of the collected bugs were non-deterministic, which could impact the evaluation of test case reducers in an unfair and non-reproducible way. To filter out such cases, we applied the TEST functions multiple times to the original programs and checked for deviating results. We also removed all test cases for which we saw divergent reduction results between runs in our later experiments.

B. Benchmark Statistics

Tab. I contains statistics on our benchmark. In total, it consists of 321 failure-inducing test programs. The file size of the programs ranges from 0.9 KiB (few lines) to 910.5 KiB (several thousand lines). We deliberately decided to also include some very large test programs; if desired, these programs can be used to generate smaller variants (e.g., by applying a reducer and stopping prematurely at a certain program size). In total, the 321 programs trigger 110 different bugs in 19 different versions of 5 real-world compilers, see Tab. II. The collection contains 227 crashes and 23 miscompilations in GCC [33] and LLVM [34], as well as 48 crashes and 23 cases of wrong results in Yices [35], z3 [36], and CVC4 [37].

TABLE I
STATISTICS ON THE BENCHMARK (SIZE IN # TOKENS).

language	fuzzer	# progs.	min. size	med. size	max. size	tot. size
C	Csmith	122	245 (1.0 KiB)	37 597 (113.7 KiB)	133 092 (430.6 KiB)	5 116 916 (15.1 MiB)
	∗Smith	128	1 549 (3.0 KiB)	66 080 (128.9 KiB)	466 111 (910.5 KiB)	11 637 893 (22.2 MiB)
		250				16 754 809 (37.3 MiB)
SMT-LIB 2	FuzzSMT	26	483 (1.2 KiB)	1 962 (4.5 KiB)	22 189 (53.4 KiB)	112 075 (0.3 MiB)
	∗Smith	45	240 (0.9 KiB)	3 820 (12.2 KiB)	31 103 (99.7 KiB)	301 911 (1.0 MiB)
		71				413 986 (1.3 MiB)

Note that the benchmark programs also target several older compiler versions. These versions are typically more error-prone [22], [23], which makes it easier to find different bugs. We provide a Docker [38] environment to automatically build these versions. Setting up the testing environment should be as easy as typing "`make`" and waiting for the build to finish. This reduces the effort for users of our benchmark (and allows for an easy reproduction of our study).

V. COMPARATIVE STUDY

This sections presents the results of our *comparative study*. After describing our evaluation setup in Sec. V-A, we first evaluate the framework itself in Sec. V-B to show that our reducer implementations do not introduce unnecessary overhead. Sec. V-C then compares the efficiency and effectiveness of these reducers when applied to the benchmark programs. Finally, we discuss some possible threats to validity in Sec. V-D.

Note that, due to space constraints and since they are more relevant in practice [5], we only consider the fixpoint variants of the reducers here. The results for a single iteration of each reducer are available online in the benchmark repository.

A. Evaluation Setup

We follow the common methodology for the evaluation of test case reducers [4], [5], [6], [7], [8], [9] and evaluate both the *effectiveness* (📏) and the *efficiency* (⏱) of the algorithms. To quantify a reducer's effectiveness, we count the number of tokens in its results (fewer is better); to quantify a reducer's efficiency, we measure the wall time that its reductions require (the number of TESTs does not necessarily correlate with reduction time [9], [5]; the runtime is more relevant in practice).

TABLE II
BUGS TRIGGERED IN THE DIFFERENT COMPILERS UNDER TEST.

language	compiler	versions	# bugs	# progs.
C	GCC	4.0.0, 4.1.0, 4.2.0, 4.3.0, 4.4.0	47	134
	LLVM	1.9, 2.0, 2.1, 2.2	42	116
SMT-LIB 2	Yices	2.2.0, 2.3.0, 2.4.0, 2.6.0	7	22
	z3	4.4.0	2	7
	CVC4	1.4, 1.5, 1.6, 1.7, 1.8	12	42
			110	321

© 2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

In some cases, reducers produce the same reduction candidate multiple times; to not TEST the same candidate again, we use *test outcome caching* [13]. In preliminary experiments we found that, in total, all reducers (or at least their fixpoint variants) benefit from caching (but some do more than others).

All measurements were conducted sequentially on a single workstation equipped with 128 GB of RAM, a 500 GB SSD, and eight 2.4 GHz Intel Xeon CPUs running Debian 10. We executed the reducer implementations with OpenJDK 8 running in the benchmark Docker containers (ver. 18.09).

Due to the high total reduction times (see Sec. V-C), we only executed a single reduction run. But to lessen the effects of small runtime variations, we executed *five* runs for all reductions that took less than 60 s or for which the difference to another reducer was less than 20 s (1 605 of the 2 247 reductions); the reported runtimes are the median values.

B. Efficiency of our Implementations

The total time that a reducer requires to reduce an input program can be split into two parts: (1) the time spent in the TEST function to check which candidates still trigger the bug, (2) the time spent in the reducer itself to generate new candidates. While there is not much that we can do about the first part (the number of TEST evaluations and how the generated reduction candidates look like is dictated by the respective reduction algorithm), the second part also depends on the efficiency of the respective implementation: the more efficiently a reducer is implemented, the less time it spends for generating new candidates. To allow for a fair comparison, we made sure that all our implementations are efficient.

For each reducer, Fig. 3 shows a boxplot (whisker range = $1.5 \cdot$ interquartile range) for the fraction of the time that is spent in the TEST function (distribution over all benchmark programs). As can be seen, all reducers spend most of the time in the TEST function; the median fraction ranges from 93.78% (*GTR*∗) to 95.64% (*Pardis*∗). These high numbers not only suggest that our implementations of the different reducers are indeed efficient, but also that their efficiency is comparable. Some slight differences are expected and are due to the fact that some reducers are inherently more elaborate than others. For example, *GTR*∗ modifies the whole program tree over and over again and has to check after each candidate transformation if it is legal (Sec. III-C4), while *Pardis [Hyb.]*∗ simply traverses the tree and marks nodes as deleted (Sec. III-C6).

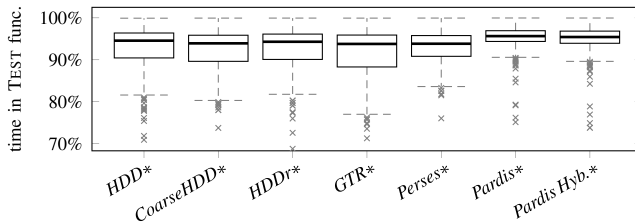


Fig. 3. Fraction of the time spent in the TEST function (distribution over all benchmark programs; some outliers not shown).

The results also provide another interesting insight: since the runtime is dominated by the TEST function in most cases anyway, it does not matter much if a reducer requires slightly more time to generate candidates; it is much more important that a reducer generates fewer and "better" candidates.

C. Comparison of the Reduction Algorithms

In this section, we present the results of the reducers when applied to the benchmark programs. These results not only show which reducers perform best, but also emphasize the necessity of a large, diverse benchmark like ours.

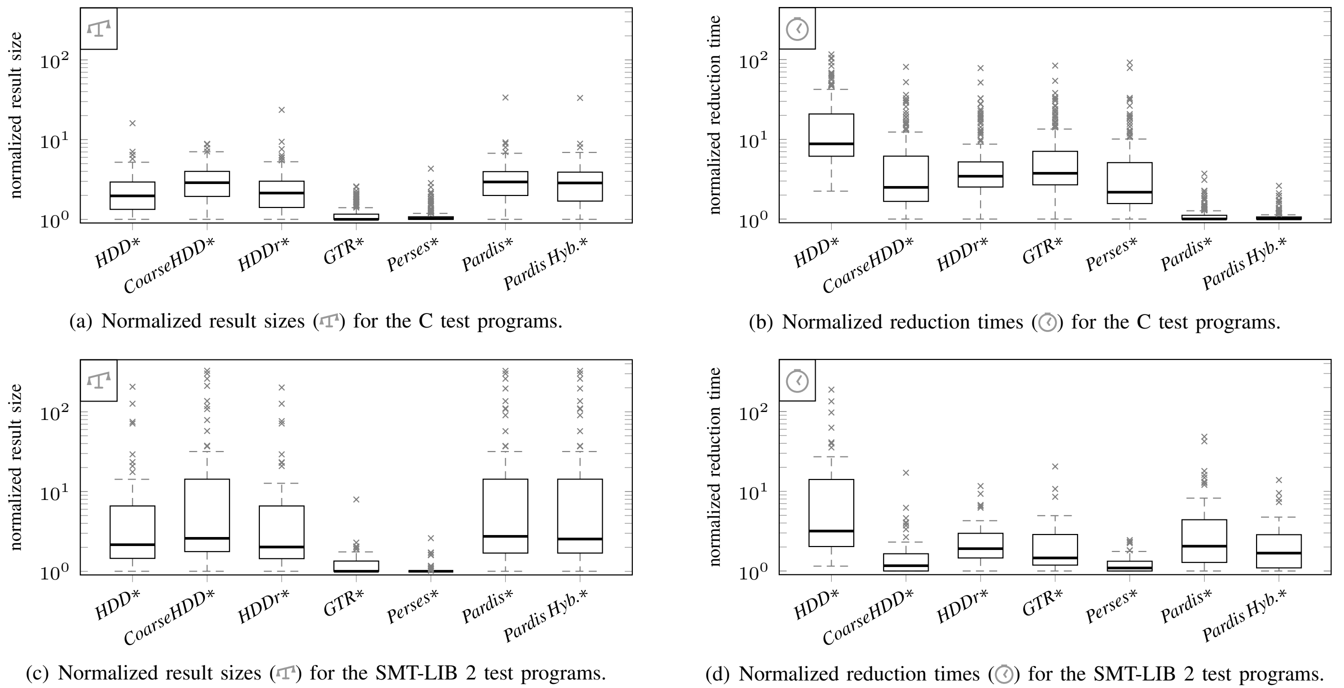
1) Accumulated Results: Due to space constraints, we cannot list all results individually (they are available online) but only report condensed results here. We begin with the accumulated results for all test programs, grouped by language. Tab. III shows how few of the original 16 754 809 (C) and 413 986 (SMT-LIB 2) tokens from Tab. I remain after the reduction with each reducer and how long these reductions took in total (lower is better in both cases). It also lists the *reduction speed*, i.e., the number of tokens that each reducer removed per minute (higher is better). To give an idea of how good the reducers really are, we also include a comparison with a ***virtual best*** reducer, i.e., the results we would get if we chose the most effective or most efficient reducer *per program*.

⚖) Regarding the effectiveness, there are three groups of reducers that can be clearly distinguished based on the results for both languages. The most tokens (C: >59k; SMT-LIB 2: >100k) remain with *CoarseHDD*∗ and *Pardis [Hyb.]*∗ (that only remove complete sub-trees). *HDD*∗ and *HDDr*∗ (that also replace sub-trees with minimal replacements) are more effective (C: ~ 45k; SMT-LIB 2: ~ 50k). *Perses*∗ and *GTR*∗ (that hoist sub-trees by replacing nodes with compatible descendants) are the most effective (C: <27k; SMT-LIB 2:

TABLE III
ACCUMULATED RESULTS FOR ALL TEST PROGRAMS.

		red. size (⚖) *(# tokens)*	red. time (🕓) *(min)*	red. speed *(# rem. tokens / min)*
C (250 test programs)	*HDD*∗	43 496	10 547.7	1 584.3
	CoarseHDD∗	59 032	3 363.6	4 963.4
	HDDr∗	47 528	2 891.3	5 778.2
	GTR∗	26 507	3 106.7	5 384.4
	Perses∗	24 965	2 487.4	6 725.6
	Pardis∗	63 085	764.6	21 828.9
	Pardis Hyb.∗	60 321	682.8	24 449.1
	virtual best	**22 842**	**610.5**	**-**
SMT-LIB 2 (71 test programs)	*HDD*∗	50 938	295.3	1 228.9
	CoarseHDD∗	102 545	17.0	18 259.0
	HDDr∗	51 547	30.8	11 753.0
	GTR∗	6 744	30.4	13 362.5
	Perses∗	5 207	10.6	38 210.7
	Pardis∗	101 622	23.4	13 308.1
	Pardis Hyb.∗	101 465	18.7	16 651.4
	virtual best	**5 022**	**8.6**	**-**

© 2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.



(a) Normalized result sizes (📏) for the C test programs.

(b) Normalized reduction times (⏱) for the C test programs.

(c) Normalized result sizes (📏) for the SMT-LIB 2 test programs.

(d) Normalized reduction times (⏱) for the SMT-LIB 2 test programs.

Fig. 4. Distributions of the result sizes (📏) and reduction times (⏱), normalized to the ***virtual best*** results per test program (note the logarithmic scales).

< 7k). Their results are close to the ***virtual best*** results per test case (C: ∼ 23k; SMT-LIB 2: ∼ 5k), which suggests that hoisting sub-trees is an effective reduction strategy. *Perses*∗ is the most effective reducer for both languages (1 542 fewer C tokens and 1 537 fewer SMT-LIB 2 tokens than *GTR*∗).

The differences are more pronounced for SMT-LIB 2 since, first, its grammar has fewer quantified lists and symbols with empty replacements (thus, *CoarseHDD*∗ and *Pardis [Hyb.]*∗ have fewer opportunities for reductions) and, second, some programs (in particular, the *FuzzSMT* [28], [21] ones) contain deeply nested expressions that can only be reduced effectively by hoisting sub-expressions, as *Perses*∗ and *GTR*∗ do.

⏱) Regarding the efficiency, the results are less clear-cut and they also vary across the languages. For the C programs, the two *Pardis*∗ variants are by far the most efficient reducers and close to the ***virtual best*** results per test case (with *Pardis Hyb.*∗ being even a bit more efficient than *Pardis*∗); the other reducers are 3.6–15.4 times slower. For the SMT-LIB 2 test cases, *Perses*∗ is the most efficient reducer and the *Pardis*∗ variants are 1.8–2.2 times slower (other reducers: 1.6–27.6 times). The reason is that some SMT-LIB 2 syntax trees (in particular, the ∗*Smith* [23] ones) contain a very long top level list of commands. Since *Pardis*∗ tries to remove each of these list elements individually, it requires many TESTs. By removing multiple list elements with the exact same size at once, *Pardis Hyb.*∗ improves upon this, but still does not achieve the efficiency of *Perses*∗, which applies a list reduction to the complete list. The results show that, as a rule of thumb, the longer the lists are, the more important it is to use an efficient list reduction. As a corollary, efficient reducers for one language may be less efficient for other languages.

2) More Detailed Comparison: The accumulated results above gave a first overview, but the results vary considerably across the test programs. There are two reasons: (1) Due to their diversity, some test programs contain more complex bugs than others, evidenced by the result sizes and reduction times of the ***virtual best*** reducer that range from 12 to 3515 tokens and from 0.2 to 3155.0 seconds. (2) The reducers do not perform equally well for all test cases. This is evidenced by Fig. 4 that shows boxplots for the distribution of the result sizes and reduction times per reducer. Due to the aforementioned variations across the test programs, we normalized these values to the ***virtual best*** results per test case and also use logarithmic scales. As can be seen, the median values mostly correlate with the accumulated results from Tab. III. On the left-hand side of Fig. 4 the three effectiveness groups show up again, this time as boxplots of similar shapes; for example, the three least effective reducers have the widest boxplots and highest median values in Figs. 4(a) and 4(c). Likewise, the boxplots for *GTR*∗ and *Perses*∗ look similar. Note that there are clear outliers for all reducers. For example, even the overall most effective reducer *Perses*∗ struggles with one C program for which another (the ***virtual best***) reducer achieves a 4.32 times smaller result. Similarly, even the overall most efficient reducer for the C programs, *Pardis Hyb.*∗, takes 2.61 times as long as the ***virtual best*** reducer for one of them.

To deal with the threat that these variations could skew the accumulated results in Tab. III, we also present a pairwise comparison of the reducers. To the best of our knowledge, some of them have never been compared with each other before. For example, *Pardis*∗ has only been compared with a stripped-down version of *Perses*∗ without hoisting [5] and

© 2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

TABLE IV
PAIRWISE COMPARISON OF THE REDUCTION TECHNIQUES.

		HDD∗	*CoarseHDD∗*	*HDDr∗*	*GTR∗*	*Perses∗*	*Pardis∗*	*Pardis Hyb.∗*	***virtual best***
C (250 test programs)	*HDD∗*	-	226 (0.73) 1 (3.48)	163 (1.00) 3 (2.52)	37 (1.77) 1 (2.29)	32 (1.91) 5 (3.94)	220 (0.73) 1 (8.53)	219 (0.75) 0 (8.33)	19 (1.97) 0 (8.78)
	CoarseHDD∗	29 (1.37) 249 (0.29)	-	30 (1.29) 146 (0.82)	12 (2.62) 224 (0.71)	14 (2.74) 73 (1.19)	153 (1.00) 14 (2.29)	141 (1.00) 14 (2.33)	4 (2.88) 8 (2.50)
	HDDr∗	129 (1.00) 247 (0.40)	225 (0.78) 104 (1.22)	-	30 (1.97) 169 (0.87)	26 (2.03) 52 (1.33)	222 (0.79) 7 (3.33)	220 (0.81) 4 (3.28)	14 (2.14) 2 (3.45)
	GTR∗	218 (0.56) 249 (0.44)	243 (0.38) 26 (1.41)	225 (0.51) 81 (1.15)	-	147 (0.99) 26 (1.69)	241 (0.37) 6 (3.50)	238 (0.39) 7 (3.66)	135 (1.00) 2 (3.75)
	Perses∗	224 (0.52) 245 (0.25)	246 (0.37) 177 (0.84)	230 (0.49) 198 (0.75)	124 (1.01) 224 (0.59)	-	248 (0.37) 16 (2.08)	245 (0.38) 14 (2.13)	114 (1.02) 11 (2.17)
	Pardis∗	35 (1.37) 249 (0.12)	188 (1.00) 236 (0.44)	34 (1.26) 243 (0.30)	13 (2.67) 244 (0.29)	13 (2.69) 234 (0.48)	-	200 (1.00) 129 (1.00)	4 (2.94) 117 (1.00)
	Pardis Hyb.∗	37 (1.33) 250 (0.12)	208 (1.00) 236 (0.43)	36 (1.23) 246 (0.31)	17 (2.57) 243 (0.27)	17 (2.64) 236 (0.47)	237 (1.00) 122 (1.00)	-	4 (2.86) 111 (1.01)
SMT-LIB 2 (71 test programs)	*HDD∗*	-	67 (0.80) 2 (2.57)	58 (1.00) 4 (1.67)	3 (1.94) 2 (2.31)	3 (2.00) 0 (2.53)	63 (0.81) 29 (1.53)	61 (0.84) 19 (1.96)	2 (2.15) 0 (3.17)
	CoarseHDD∗	17 (1.25) 69 (0.39)	-	15 (1.25) 65 (0.65)	2 (2.55) 56 (0.82)	3 (2.42) 32 (1.01)	48 (1.00) 40 (0.79)	44 (1.00) 47 (0.94)	2 (2.59) 20 (1.17)
	HDDr∗	61 (1.00) 67 (0.60)	69 (0.80) 6 (1.54)	-	4 (1.90) 22 (1.16)	4 (1.99) 2 (1.61)	64 (0.81) 36 (0.99)	61 (0.83) 29 (1.25)	2 (2.02) 1 (1.91)
	GTR∗	70 (0.51) 69 (0.43)	71 (0.39) 15 (1.22)	69 (0.53) 49 (0.86)	-	41 (1.00) 13 (1.18)	70 (0.43) 37 (0.94)	68 (0.43) 30 (1.13)	39 (1.00) 3 (1.46)
	Perses∗	70 (0.50) 71 (0.39)	70 (0.41) 39 (0.99)	69 (0.50) 69 (0.62)	60 (1.00) 58 (0.85)	-	70 (0.41) 54 (0.49)	68 (0.42) 51 (0.65)	58 (1.00) 28 (1.09)
	Pardis∗	19 (1.23) 42 (0.65)	59 (1.00) 31 (1.27)	19 (1.23) 35 (1.01)	3 (2.34) 34 (1.06)	3 (2.42) 17 (2.05)	-	60 (1.00) 28 (1.19)	2 (2.74) 13 (2.05)
	Pardis Hyb.∗	22 (1.19) 52 (0.51)	65 (1.00) 24 (1.07)	22 (1.20) 42 (0.80)	5 (2.34) 41 (0.89)	5 (2.39) 20 (1.53)	68 (1.00) 43 (0.84)	-	4 (2.54) 6 (1.68)

The values in a row R and a column C denote how many test programs reducer R reduces at least as effectively or efficiently as reducer C. Higher is better for R. Values in parentheses denote the median ratio of their result sizes and reduction times. Lower is better for R. The ***virtual best*** column thus shows the number of cases for which R performs ***best***. ● appears in the discussion.

the two most effective reducers, *GTR∗* and *Perses∗*, have not been compared at all. Tab. IV holds the results.

The values in a row R and a column C denote how many test programs reducer R reduces at least as effectively or efficiently as reducer C (higher is better for R); the last column contains the number of cases for which R is the ***best*** (i.e., most effective or most efficient) reducer. While these values confirm the results from above, they also provide some additional insight. Although *Perses∗* is more effective in total (see the accumulated results in Tab. III), *GTR∗* is at least as effective for more than half of the programs (147 C and 41 SMT-LIB 2 programs). In particular, *GTR∗* produces the smallest results for 135 of the 250 C programs, 21 more than *Perses∗*. Similarly, while *Pardis Hyb.∗* is the overall most efficient reducer for the C programs as per Tab. III, *Pardis∗* is more efficient in 129 cases; it is also the fastest reducer for 117 of the C programs, 6 more than *Pardis Hyb.∗*. Once again, these results show the diversity of our benchmark, which is further stressed by the following two observations: (1) For all pairs of reducers R and C, there are non-zero values in Tab. IV, i.e., there are some C and/or SMT-LIB 2 programs for which R is on par with or outperforms C; this holds both for effectiveness and efficiency. (2) For every reducer R, there is at least one program for which R is the most effective or efficient one (non-zero values in the last column); the only exception is that there is no program for which *HDD∗* is fastest.

Since the accumulated results in Tab. III hide the distribution of the results across all test programs, we also computed the ratios of result sizes and reduction times for all test programs. Tab. IV holds the median values of these ratios as the parenthesized values. Note for example that, while *Perses∗* saves a total of 1 542 tokens compared to *GTR∗*, *GTR∗* is slightly more effective than *Perses∗* for the C programs in the median case: the median *GTR∗* results are only 0.99 times as large as the *Perses∗* results. On the other hand, *GTR∗* is also 1.69 times slower than *Perses∗* in the median case, which is consistent with the accumulated results in Tab. III.

To summarize, no reducer works best in all cases. While overall, *Perses∗* is the most effective reducer for both languages, *GTR∗* is better for many C programs. Regarding efficiency, *Perses∗* is best for SMT-LIB 2 and *Pardis Hyb.∗* for C. But since the results vary considerably and since there are cases for each reducer for which its effectiveness and/or efficiency degrades, reducers should always be evaluated with a diverse benchmark like ours.

D. Threats to Validity

In this section, we discuss some possible threats to validity.

Our reducer implementations may contain bugs and this may impact the results of our study. To mitigate this risk, we manually checked the executions for several test programs per reducer, including the examples described in their publications.

© 2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

We had to manually add the quantifiers that *Perses* and (even more so) the *Pardis* variants require for an effective reduction, see Sec. III-C5. Despite our best efforts, it is possible that we missed some sub-rules that could have been quantified.

As already explained in Sec. IV-A, non-deterministic bugs and imprecise TEST functions could impair the comparison in an unfair manner. To mitigate this risk we made the TESTs as specific as possible and discarded all problematic programs that we found. Similarly, the evaluation could be impaired if the same bug is triggered by multiple different locations in the input program. We manually checked a random sample of the reduction results, but have not observed such a case.

The results could differ both quantitatively and qualitatively, had we used "real" programs instead of fuzzer-generated ones. Our approach of using two different fuzzers per language and including programs of varying sizes can only mitigate this to a certain extent. However, reducers have been evaluated with fuzzer-generated programs before, see Sec. IV. Besides, reducers should also be able to handle such code well.

Finally, our benchmark is limited to functional compiler bugs, but some reducers could be better suited for non-functional bugs (e.g., *performance problems* [39] or *missed optimizations* [40]) than others. To close this gap, we consider extending our benchmark with such test cases in the future.

VI. RELATED WORK

This section gives an overview of the work closest to ours. To the best of our knowledge, our work presents the first implementation of renowned reducers in a common framework as well as the first extensive benchmark of (unreduced) test programs. The *Picire* [41] and *Picireny* [42], [19] frameworks include implementations of *DDMin* and the *HDD* variants, but they lack implementations of the newer, more advanced reducers. Most of the test programs that *Perses* has been evaluated with are available online [43], but the collection contains only 20 C programs (19 of which have been generated with *Csmith* [22]). These programs were also used for the original evaluation of *Pardis*, but the authors could only reproduce 14 of them. The *GTR* repository [44] contains 6 Python and 41 JavaScript test programs, but with a median file size of only 0.5 KiB most of these programs are quite small. With 250 C and 71 SMT-LIB 2 programs and program sizes from 0.9 KiB to 910.5 KiB, our benchmark is by far the most extensive and diverse one that we are aware of.

Recently, Vince et al. [45] experimented with various ways to extend *HDD* with a hoisting mechanism to increase its effectiveness. Since they plan to also extend other reducers accordingly (in particular, *CoarseHDD* and *HDDr*), we consider these extensions to be orthogonal to the algorithms considered in this work, but we plan to add such variants to our framework in the future (and to evaluate them with our benchmark).

Besides the language-agnostic reducers considered in this work, there are also several language-specific ones, e.g., *C-Reduce* [32] for C/C++, *ReduKtor* [46] for Kotlin, *SIMP* [47] for SQL and MDX, and *ddSMT* [48] for SMT-LIB 2. Such reducers have to be rewritten for each language, but by exploiting language-specific knowledge they often achieve even smaller results (it is believed [46] that a combination of language-agnostic and language-specific approaches yields the best results). For example, Sun et al. [9] reported that *C-Reduce* achieves almost 65 % smaller results than *Perses* (but it also takes 1.67 times as long). Unfortunately, these language-specific works do not offer benchmarks. One exception is *ReduKtor*, which in part has been evaluated with fuzzer-generated programs that are available online [49]. At the time of writing, this collection contains 808 Kotlin programs for 65 different bugs, but with a median file size of less than 0.4 KiB they are even smaller than the *GTR* programs.

Some reduction techniques are tightly coupled with a specific fuzzer. *GLFuzz* [50], *IFuzzer* [51], and *Nautilus* [52] include a mechanism to reduce failure-inducing test programs during or after their generation. The *Hypothesis* framework [53] for test case generation includes an *internal test case reducer* [54] that alters the random choices of the generator to produce small outputs. *Seq-Reduce* and *Fast-Reduce* [32] reduce *Csmith* [22] programs. All these approaches are valuable in their respective context, but they are not generally applicable and oftentimes less effective than dedicated reducers.

Finally, our work focuses on sequential reducers that generate and evaluate reduction candidates one after another. To fully utilize the computing power of modern machines, there are some first attempts at parallelizing the generation and evaluation of reduction candidates [15], [41], [55]. Our framework could be extended with parallel reducers in the future (and our benchmark could be used for their evaluation).

VII. CONCLUSION AND FUTURE WORK

Research has proposed different language-agnostic test case reducers that aid compiler developers in finding the root cause of a bug. Unfortunately, comparable implementations are hardly available and there are no appropriate test cases. To close these gaps, we presented (1) a *framework* with fine-tuned implementations of state-of-the-art techniques, (2) a diverse *benchmark* with 321 bug-inducing programs in two very different languages, and (3) a *comparative study* in which we evaluated the techniques w.r.t. their efficiency and effectiveness. *Perses*∗ and *GTR*∗ are very effective for both languages, which we attribute to their ability to hoist subtrees; *Perses*∗ is also very efficient for SMT-LIB 2, whereas the *Pardis*∗ variants are the most efficient reducers for C. But the results also show that no reducer works best in all cases, which emphasizes the need for a diverse benchmark like ours. As our framework and benchmark are available online, this eases the quantitative evaluation of future techniques.

When implementing and tuning the reducers we learnt that they often cut down the number of invalid candidates by generating only *syntactically* valid candidates. We see promising future work in also considering *semantic*, context-dependent properties (e.g., that variables have to be defined before they are used) to further reduce the number of such candidates and hence increase reducer efficiency.

© 2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

REFERENCES

[1] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A Survey of Compiler Testing," *ACM Computing Surveys*, vol. 53, no. 1, pp. 4:1–4:36, Feb. 2020.

[2] N.N., "A Guide to Testcase Reduction," Accessed on 2020-07-16, https://gcc.gnu.org/wiki/A_guide_to_testcase_reduction.

[3] ——, "How to submit an LLVM bug report," Accessed on 2020-07-16, https://llvm.org/docs/HowToSubmitABug.html.

[4] G. Misherghi and Z. Su, "HDD: Hierarchical Delta Debugging," in *ICSE'06: International Conference on Software Engineering*, Shanghai, China, May 2006, pp. 142–151.

[5] G. Gharachorlu and N. Sumner, "Pardis: Priority Aware Test Case Reduction," in *FASE'19: Fundamental Approaches to Software Engineering*, Prague, Czech Republic, Apr. 2019, pp. 409–426.

[6] R. Hodován, A. Kiss, and T. Gyimóthy, "Coarse Hierarchical Delta Debugging," in *ICSME'17: International Conference on Software Maintenance and Evolution*, Shanghai, China, Sep. 2017, pp. 194–203.

[7] A. Kiss, R. Hodován, and T. Gyimóthy, "HDDr: A Recursive Variant of the Hierarchical Delta Debugging Algorithm," in *A-TEST'18: Automating Test Case Design, Selection, and Evaluation*, Lake Buena Vista, FL, Nov. 2018, pp. 16–22.

[8] S. Herfert, J. Patra, and M. Pradel, "Automatically Reducing Tree-Structured Test Inputs," in *ASE'17: International Conference on Automated Software Engineering*, Urbana-Champaign, IL, Oct.–Nov. 2017, pp. 861–871.

[9] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su, "Perses: Syntax-Guided Program Reduction," in *ICSE'18: International Conference on Software Engineering*, Gothenburg, Sweden, May–June 2018, pp. 361–371.

[10] A. Zeller and R. Hildebrandt, "Simplifying and Isolating Failure-Inducing Input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, Feb. 2002.

[11] B. Ford, "Parsing Expression Grammars: A Recognition-Based Syntactic Foundation," in *POPL'04: Principles of Programming Languages*, Venice, Italy, Jan. 2004, pp. 111–122.

[12] ——, "Packrat Parsing: Simple, Powerful, Lazy, Linear Time," in *ICFP'02: International Conference on Functional Programming*, Pittsburgh, PA, Oct. 2002, pp. 36–47.

[13] R. Hodován, A. Kiss, and T. Gyimóthy, "Tree Preprocessing and Test Outcome Caching for Efficient Hierarchical Delta Debugging," in *AST'17: International Workshop on Automation of Software Testing*, Buenos Aires, Argentina, May 2017, pp. 23–29.

[14] G. Gharachorlu and N. Sumner, "Avoiding the Familiar to Speed Up Test Case Reduction," in *QRS'18: International Conference on Software Quality, Reliability and Security*, Lisbon, Portugal, July 2018, pp. 426–437.

[15] R. Hodován and A. Kiss, "Practical Improvements to the Minimizing Delta Debugging Algorithm," in *ICSOFT'16: International Joint Conference on Software Technologies*, Lisbon, Portugal, July 2016, pp. 241–248.

[16] A. Kiss, "Generalizing the Split Factor of the Minimizing Delta Debugging Algorithm," *IEEE Access*, vol. 8, pp. 219 837–219 846, Dec. 2020.

[17] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, "The Fuzzing Book," Available at https://www.fuzzingbook.org/, Accessed on 2020-06-10.

[18] G. Misherghi, "Hierarchical Delta Debugging," Master's thesis, University of California, Davis, June 2007.

[19] R. Hodován and A. Kiss, "Modernizing Hierarchical Delta Debugging," in *A-TEST'16: International Workshop on Automating Test Case Design, Selection, and Evaluation*, Seattle, WA, Nov. 2016, pp. 31–37.

[20] N. Sumner, "ANTLR2PNF: ANTLR4 to Perses Normal Form converter," Accessed on 2020-12-18, https://github.com/nsumner/antlr2pnf.

[21] R. Brummayer and A. Biere, "Fuzzing and Delta-Debugging SMT Solvers," in *SMT'09: International Workshop on Satisfiability Modulo Theories*, Montreal, Canada, Aug. 2009, pp. 1–5.

[22] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and Understanding Bugs in C Compilers," in *PLDI'11: Programming Language Design and Implementation*, San Jose, CA, June 2011, pp. 283–294.

[23] P. Kreutzer, S. Kraus, and M. Philippsen, "Language-Agnostic Generation of Compilable Test Programs," in *ICST'20: International Conference on Software Testing, Verification and Validation*, Porto, Portugal, Oct. 2020, pp. 39–50.

[24] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the Art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, 2018.

[25] M. Marcozzi, Q. Tang, A. F. Donaldson, and C. Cadar, "Compiler Fuzzing: How Much Does It Matter?" *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 155:1–155:29, Oct. 2019.

[26] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB Standard Version 2.6," The University of Iowa, Tech. Rep., July 2017.

[27] W. M. McKeeman, "Differential Testing for Software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[28] M. Soos, "FuzzSMT: SMT fuzzers by Robert Brummayer and Trevor Hansen," Accessed on 2021-04-19, https://github.com/msoos/fuzzsmt.

[29] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming Compiler Fuzzers," in *PLDI'13: Programming Language Design and Implementation*, Seattle, WA, June 2013, pp. 197–208.

[30] J. Holmes, A. Groce, and M. A. Alipour, "Mitigating (and Exploiting) Test Reduction Slippage," in *A-TEST'16: Automating Test Case Design, Selection, and Evaluation*, Seattle, WA, Nov. 2016, pp. 66–69.

[31] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-C: A software analysis perspective," *Formal Aspects of Computing*, vol. 27, no. 3, pp. 573–609, Jan. 2015.

[32] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-Case Reduction for C Compiler Bugs," in *PLDI'12: Programming Language Design and Implementation*, Beijing, China, June 2012, pp. 335–346.

[33] N.N., "GCC, the GNU Compiler Collection," Accessed on 2021-04-19, https://gcc.gnu.org/.

[34] ——, "The LLVM Compiler Infrastructure," Accessed on 2021-04-19, https://llvm.org/.

[35] B. Dutertre, "Yices 2.2," in *CAV'14: Computer Aided Verification*, ser. Springer LNCS vol. 8559, Vienna, Austria, July 2014, pp. 737–744.

[36] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *TACAS'08: Tools and Algorithms for the Construction and Analysis of Systems*, ser. Springer LNCS vol. 4963, Budapest, Hungary, Mar.–Apr. 2008, pp. 337–340.

[37] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *CAV'11: Computer Aided Verification*, ser. Springer LNCS vol. 6806, Snowbird, UT, July 2011, pp. 171–177.

[38] N.N., "Empowering App Development for Developers — Docker," Accessed on 2021-03-15, https://www.docker.com/.

[39] J. Jung, H. Hu, J. Arulraj, T. Kim, and W.-H. Kang, "APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems," *Proceedings of the VLDB Endowment*, vol. 13, no. 1, pp. 57–70, Sep. 2019.

[40] G. Barany, "Finding Missed Compiler Optimizations by Differential Testing," in *CC'18: Compiler Construction*, Vienna, Austria, Feb. 2018, pp. 82–92.

[41] R. Hodován, "Picire: Parallel Delta Debugging Framework," Accessed on 2020-09-28, https://github.com/renatahodovan/picire.

[42] ——, "Picireny: Hierarchical Delta Debugging Framework," Accessed on 2020-09-28, https://github.com/renatahodovan/picireny.

[43] C. Sun, "Perses C Benchmarks," Accessed on 2020-09-28, https://bitbucket.org/chengniansun/perses_c_benchmarks/.

[44] S. Herfert, "Generalized Reduction of Tree-Structured Test-Inputs," Accessed on 2021-04-09, https://github.com/sherfert/GTR.

[45] D. Vince, R. Hodován, D. Bársony, and Ákos Kiss, "Extending Hierarchical Delta Debugging with Hoisting," Apr. 2021, https://arxiv.org/abs/2104.03637.

[46] D. Stepanov, M. Akhin, and M. Belyaev, "ReduKtor: How We Stopped Worrying About Bugs in Kotlin Compiler," in *ASE'19: Automated Software Engineering*, San Diego, CA, Nov. 2019, pp. 317–326.

[47] N. Bruno, "Minimizing Database Repros using Language Grammars," in *EDBT'10: International Conference on Extending Database Technology*, Lausanne, Switzerland, Mar. 2010, pp. 382–393.

[48] A. Niemetz and A. Biere, "ddSMT: A Delta Debugger for the SMT-LIB v2 Format," in *SMT'13: International Workshop on Satisfiability Modulo Theories*, Helsinki, Finland, July 2013, pp. 36–45.

[49] M. Koltsov and M. Akhin, "Kotlin Fuzzer," Accessed on 2021-04-21, https://github.com/ItsLastDay/KotlinFuzzer/.

[50] A. F. Donaldson, H. Evrard, A. Lascu, and P. Thomson, "Automated Testing of Graphics Shader Compilers," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 93:1–93:29, Oct. 2017.

© 2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

[51] S. Veggalam, S. Rawat, I. Haller, and H. Bos, "IFuzzer: An Evolutionary Interpreter Fuzzer Using Genetic Programming," in *ESORICS'16: Computer Security*, ser. Springer LNCS vol. 9878, Heraklion, Greece, Sep. 2016, pp. 581–601.

[52] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Nautilus: Fishing for Deep Bugs with Grammars," in *NDSS'19: Network and Distributed System Security Symposium*, San Diego, CA, Feb. 2019.

[53] D. R. MacIver and Z. Hatfield-Dodds, "Hypothesis: A new approach to property-based testing," *Journal of Open Source Software*, vol. 4, no. 43, p. 1891, Nov. 2019.

[54] D. R. MacIver and A. F. Donaldson, "Test-Case Reduction via Test-Case Generation: Insights from the Hypothesis Reducer," in *ECOOP'20: European Conference on Object-Oriented Programming*, online, Nov. 2020, pp. 13:1—-13:27.

[55] T. Ormandy, "halfempty: A fast, parallel test case minimization tool," Accessed on 2020-09-29, https://github.com/googleprojectzero/halfempty.

© 2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.